Platform SDK Developer's Guide

Platform SDK 9.0.x

12/29/2021

Table of Contents

Welcome to the Developer's Guide!

This guide offers a collection of articles that will help you to get started with Platform SDK development.

For detailed information about the Platform SDKs, please refer to the Platform SDK API Reference for your specific release.

Getting Started

Learn about the Platform SDK architecture and how to begin creating your own custom applications.

- Introducing the Platform SDK
- Architecture and Design
- Event Handling
- More

Working Directly with Genesys Servers

Understand how Platform SDK allows your application to interact directly with the desired Genesys Server.

- Connecting to a Server
- Telephony (T-Server)
- More...

Using Application Blocks to Aid Development

Find out about the architecture of Genesys Co-browse.

- Application Template Application Block
- Warm Standby Application Block
- Configuration Object Model Application Block

Commonly Used Features

These features are not specific to a Genesys server, and can be used in most applications.

- Channel Encoding for String Values
- Logging Configuration
- Secure Connections Using TLS
- More...

Additional Resources

Additional articles to support Platform SDK developers.

Migration Overview

Legacy Support

Platform SDK API Reference

Introductory Topics

The following articles give information about common Platform SDK functionality and protocol usage that all developers should be aware of:

- Introducing the Platform SDK
- Architecture of the Platform SDK
- Connecting to a Server
- Configure Platform SDK Channel Encoding for String Values
- Using the Warm Standby Application Block
- Using the Application Template Application Block
- Using the Cluster Protocol Application Block
- Event Handling
- Setting Up Logging in Platform SDK
- Additional Logging Features
- Log4j2 Configuration with the Application Template Application Block

Introducing the Platform SDK

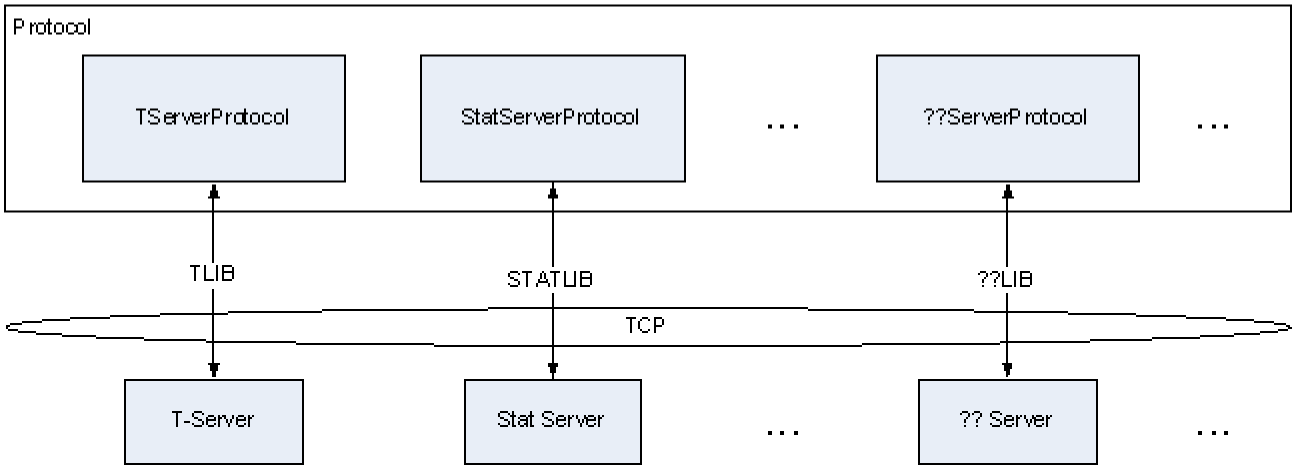
The Platform SDK exposes the protocols of Genesys servers as an API. This means you can write .NET and Java applications that communicate with these servers in their native protocols.

You can use the Platform SDK to do two main things:

- Establish and maintain a connection to each Genesys server used by your application
- Send and receive messages to and from each of these Genesys servers

In addition to enabling these two basic functions, the Platform SDK ships with *application blocks*, which have been built on top of the Platform SDK in order to provide simple yet high-performance ways to do things like configuring warm standby settings for your connections and working with configuration objects.

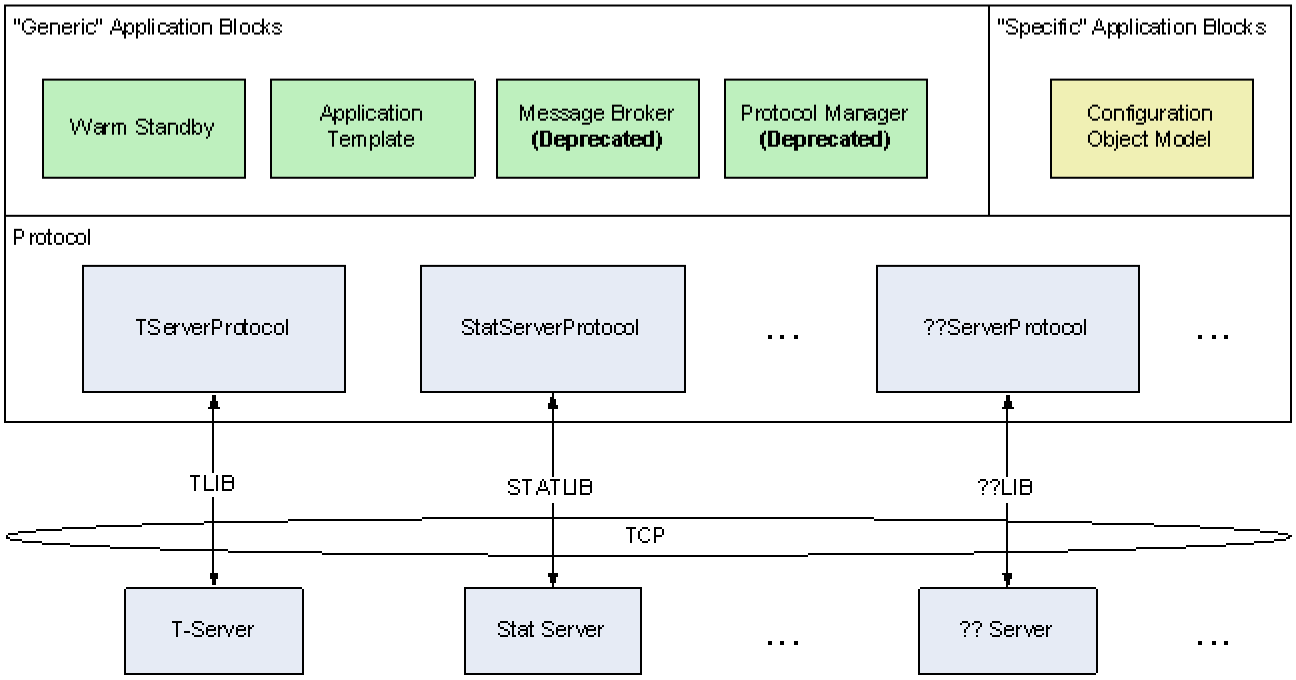
The following image shows the relationship between the Platform SDK protocol objects and the servers each of them connects with.



Each protocol object subclasses `ClientChannel`, which in turn subclasses `DuplexChannel` and implements the `Protocol` interface. This means they all share a common interface to the Genesys servers. The protocol objects communicate with the corresponding Genesys servers over a TCP connection, with each one using the native protocol of the server it connects with. For example, the `TServerProtocol` object communicates over TCP with a T-Server, using the TLIB protocol that is native to the T-Server.

As mentioned above, the Platform SDK also includes reusable production-quality application blocks that can be dropped into your code to provide simple yet high-performance ways to carry out important functions that are commonly needed by applications that communicate with Genesys servers.

As shown below, there are two main types of application blocks: generic and specific.



Generic application blocks provide functionality that is useful for a broad range of applications, such as configuring connection and warm standby settings. These application blocks are recommended for use in most development. Specific application blocks are only beneficial for certain types of applications. For example, the Configuration Object Model application block makes it easy to work with objects in the Genesys Configuration Layer and is only required when you are writing an application that requires this functionality.

Finally, the Platform SDK includes additional components designed to make development of custom applications easier. These components offer support for useful features such as customized logging or switch abstraction.

The Protocols

The Platform SDK is divided into separate “protocols.” Each component works with one or more of Genesys servers.

The following table shows the servers each of the Platform SDK protocols connects with, and gives the names of the native protocols that are used to communicate with each server.

Platform SDK Protocol Name	Genesys Servers	Native Protocols
Configuration Platform SDK	Configuration Server	CFGLIB
Contacts Platform SDK	Universal Contact Server	UCS Protocol
Management Platform SDK	• Message Server • Solution Control Server	• GMESSAGELIB • SCSLIB

Platform SDK Protocol Name	Genesys Servers	Native Protocols
	• Local Control Agent	• LCALIB
Open Media Platform SDK	Interaction Server	ITX, ESP
Outbound Contact Platform SDK	Outbound Contact Server	• CMLIB • OCS-Desktop Protocol
Routing Platform SDK	• Custom Server • Universal Routing Server	• Custom Server Protocol • Routing Server Protocol
Statistics Platform SDK	Stat Server	STATLIB
Voice Platform SDK	T-Servers	• TLIB • Preview Interaction Protocol
Web Media Platform SDK	• Chat Server • E-Mail Server Java • Callback Server	• MCR Chat Lib • MCR E-Mail Lib • MCR Callback Lib • ESP E-Mail Lib

Configuration Platform SDK

The Configuration Platform SDK enables you to build applications that use the services of the Genesys Configuration Server. This allows these applications to either query on objects in the Configuration Layer of your Genesys environment or to add, modify, and delete information about those objects, while taking advantage of an environment in which Configuration Server carries out several important administrative functions.

Contacts Platform SDK

The Contacts Platform SDK allows you to build applications that view, or interact with, the contact information for your contact center. This SDK accesses information directly from Universal Contact Server, allowing you to design applications that access contact information when dealing with multimedia interactions such as chat or email, for example.

Management Platform SDK

The Management Platform SDK enables you to write applications that interact with Message Server, Solution Control Server, and Local Control Agents.

Open Media Platform SDK

With the Open Media Platform SDK, you can build client applications that feed open media interactions into your Genesys environment, or applications that act as custom media servers to perform external service processing (ESP) on interactions that have already entered it.

Outbound Contact Platform SDK

The Outbound Contact Platform SDK can be used to build applications that allow you to manage outbound campaigns.

Routing Platform SDK

The Routing Platform SDK allows you to write Java and .NET applications that combine logic from your custom application with the router-based logic of URS, in order to solve many common interaction-related tasks.

Statistics Platform SDK

With the Statistics Platform SDK, you can build applications that use the services of Stat Server in order to solicit and monitor statistics from your Genesys environment.

Stat Server tracks information about customer interaction networks (contact center, enterprise-wide, or multi-enterprise telephony and computer networks). It also converts the data accumulated for directory numbers (DNs), agents, agent groups, and non-telephony–specific object types, such as email and chat sessions, into statistically useful information.

Voice Platform SDK

The Voice Platform SDK enables you to design applications that monitor and handle voice interactions from a traditional or IP-based telephony device.

Web Media Platform SDK

The Web Media Platform SDK can be used to build applications that interact with Chat Server, E-Mail Server Java, and Callback Server through a web server interface.

The Application Blocks

Important

These application blocks are reusable production-quality components, designed using industry best practices and provided with source code so they can be used "as is," extended, or tailored as necessary.

Please see the License Agreement for details.

Genesys application blocks are reusable production-quality components that provide specific functionality needed by a broad range of Genesys customers. They have been designed using industry best practices and provided with source code so they can be used “as is”, extended, or tailored if you need to. Please see the License Agreement for details.

Application Template Application Block

The Application Template Application Block provides a way to read configuration options for applications in Genesys Administrator and to configure Platform SDK protocols. It also allows standard connection settings (including ADDP or TLS details) to be retrieved from Configuration Server, and helps with common features like setting up WarmStandby or assigning message filters.

Configuration Object Model Application Block

The Configuration Object Model (COM) Application Block provides a consistent and intuitive object model for applications that need to work with Configuration Server objects. Use the COM Application Block when you need to create, update, or delete Configuration Layer Objects.

Warm Standby Application Block

You can use the Warm Standby Application Block to switch to a backup server in case your primary server fails, in cases where you do not need to guarantee the integrity of existing interactions.

Deprecated Content

The application blocks listed in this section are considered legacy products. Documentation is still provided for backwards compatibility, but new development should not use these application blocks.

Message Broker Application Block (deprecated)

The Message Broker Application Block makes it easy for your applications to handle events in an efficient way.

Deprecated with release 8.1.1. If you have existing applications that use the Message Broker Application Block, refer to the migration article for details on how to update your code. New applications should use the improved message handling capability now included in Platform SDK instead.

Protocol Manager Application Block (deprecated)

The Protocol Manager Application Block allows for simplified communication with more than one server. It takes care of opening and closing connections to many different servers, as well as handling the reconfiguration of high availability connections.

Deprecated with release 8.1.1. If you have existing applications that use the Protocol Manager Application Block, refer to the migration article for details on how to update your code. New applications should use the improved ability of Platform SDK to connect to servers instead.

The Components

Additional components are included to provide useful functionality for creating custom applications with the Platform SDK, even if that doesn't necessarily involve communicating with Genesys servers.

Platform SDK Log Library

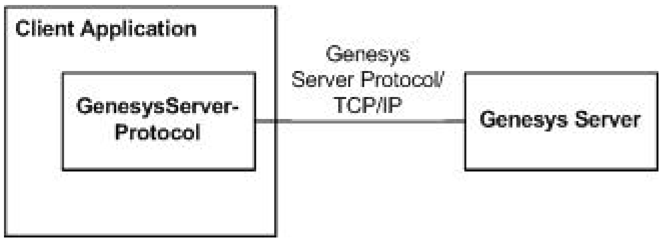
The Platform SDK Log Library presents an easy-to-use API for logging messages in custom-built applications.

Architecture of the Platform SDK

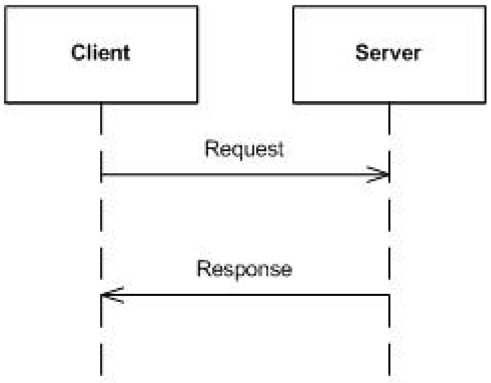
The Platform SDKs enable you to write client or server applications that use messages to communicate with Genesys servers.

Each SDK has one or more `Protocol` objects that you can use in your client applications to establish communication with the appropriate server. These objects use the native protocols of the Genesys servers they are designed to work with.

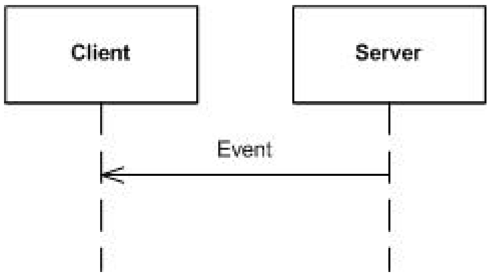
From a conceptual standpoint, your application's `Protocol` object, will be communicating directly with the appropriate server using the server's protocol running on TCP/IP, as shown below.



Once you have opened a connection to the server, you are ready to send and receive messages. The Platform SDK supports two message exchange patterns. In some cases, you will need to follow the Request/Response pattern. That is, you will send a message and wait for a response, as shown below.



At other times, following the Unsolicited Event pattern, you simply need to wait for unsolicited messages of a certain type.



The messages you send will be in the form of `Request` classes, such as `RequestAgentLogin` or `RequestAnswerCall`. The messages you receive, whether solicited or not, will be in the form of `Event` classes, such as `EventAck` or `EventDialing`.

As you can see, the architecture of the Platform SDKs is fairly simple — but you can use it to do some powerful things.

Connecting to a Server

Java

The applications you write with the Platform SDK need to communicate with one or more Genesys servers, so the first thing you need to do is create connections with these servers. You will have to reference libraries and add `import` statements to your project for each specific protocol you are working with. These steps are not explicitly described here because the files and packages required will vary depending on which protocols you plan to use.

> **Important**
> Starting with release 8.1.1, the Platform SDK uses Netty by default for the implementation of its transport layer. Therefore, your project will need to reference Netty as well.

Once you have connected to a server, you use that connection to exchange messages with the server. For details about sending and receiving messages to and from a server, refer to the event handling article.

Creating a Protocol Object

To connect to a Genesys server, you create an instance of the associated protocol class. As an example, this article will describe connecting to a Genesys T-Server using the `TServerProtocol` class. (For different applications, please use this API Reference to check protocol details for the specific server that you wish to connect to.)

In order to create a protocol object, you will first need to create an Endpoint object which acts as a container for generic connection parameters. An Endpoint object contains, at a minimum, a server name, the host name where the server is running, and the port on which the server is listening. The server name will appear in logs but does not affect protocol behavior; it may be any name that is significant to you.

```
Endpoint tserverEndpoint = new Endpoint("T-Server", TSERVER_HOST, TSERVER_PORT);
TServerProtocol tserverProtocol = new TServerProtocol(tserverEndpoint);
```

After creating your protocol object, you need to specify some connection parameters that are specific to that protocol. These parameters will differ depending on which server you are connecting to. Please check to the sections specific to the server that you wish to connect to for more information.

Once configuration is complete, you can open the connection to your server.

Working with a Connection Synchronously

The easiest way to open a connection to your server is to do it synchronously, which means that the method will block any additional processing until the server connection has either opened successfully or failed definitively. This is commonly used for non-interactive, batch applications. In this case, you can add code for using the protocol directly after opening. In the case of failure, the open method will throw an exception that should be caught and handled.

```
tserverProtocol.open();
// You can start sending requests here.
```

When you have finished communicating with your servers, you can close the connection. Similar to how a connection is opened, you close a synchronous connection by using the following method:

```
// Synchronous
tserverProtocol.close();
```

Working with a Connection Asynchronously

You may prefer to open a connection using asynchronous (non-blocking) methods. This is usually preferred for user-interactive applications, in order to avoid blocking the GUI thread so that the application does not appear "frozen" to the user.

Important

When using the asynchronous connections, make sure that your code waits for the `Opened` event to fire before attempting to send or receive messages. Otherwise you might be trying to use a connection that is not yet open.

There are two types of asynchronous open/close protocol operations available in Java:

1. Completion handler based asynchronous operations
2. Future based asynchronous operations

Completion Handler Based Asynchronous Operations

Instead of using a normal open or close method, you instead use the following `ClientChannel` API methods:

- `public <A> void openAsync(CompletionHandler<EventObject, A> handler, A attachment);`
- `public <A> void openAsync(long timeout, CompletionHandler<EventObject, A> handler, A attachment);`
- `public <A> void closeAsync(final CompletionHandler<ChannelClosedEvent, A> handler, final A attachment);`
- `public <A> void closeAsync(long timeout, final CompletionHandler<ChannelClosedEvent, A> handler, final A attachment);`

These methods allow you to conveniently specify timeout values, register an internal channel listener, and start the `beginOpen` or `beginClose` (as appropriate) operation. If the operation is completed (with or without errors) before the timeout occurs, then the timeout is canceled; otherwise the operation is assumed to have failed. In either case, the internal channel handler is unregistered and the handler notified of the results.

Example

```
CompletionHandler<EventObject, Object> openHandler = new CompletionHandler<EventObject,
Object>() {
            @Override
            public void completed(EventObject result, MyContext context) {
                ChannelOpenedEvent event = (ChannelOpenedEvent)result;
                UniversalContactServerProtocol ucs =
(UniversalContactServerProtocol)event.getSource();

                // TODO: do not lock current thread. Schedule some work asynchronously.
                context.executor.execute( new OpenCompletedTask(context, ucs) );
            }
            @Override
            public void failed(Throwable exc, MyContext context) {
                // TODO: do not lock current thread. Schedule some work asynchronously.
                context.executor.execute( new OpenFailedTask(context) );
            }
        };


CompletionHandler<ChannelClosedEvent, Object> closeHandler = new
CompletionHandler<ChannelClosedEvent, Object>() {
            @Override
            public void completed(ChannelClosedEvent event , MyContext context) {
                UniversalContactServerProtocol ucs =
(UniversalContactServerProtocol)event.getSource();

                // TODO: do not lock current thread. Schedule some work asynchronously.
                context.executor.execute( new CloseCompletedTask(context, ucs) );
            }

            @Override
            public void failed(Throwable exc, MyContext context) {
                // TODO: do not lock current thread. Schedule some work asynchronously.
                context.executor.execute( new CloseFailedTask(context) );
            }
        };

UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
ucs.setEndpoint( new Endpoint(HOST, PORT) );

MyContext context = new MyContext();

ucs.openAsync(15000, openHandler, context);


// ... in some other place

ucs.closeAsync(15000, closeHandler, context);
```

Future Based Asynchronous Operations

An alternative way to use asynchronous operations is to use the following Future-based `ClientChannel` API methods:

- `public Future<ChannelOpenedEvent> openAsync();`
- `public Future<ChannelOpenedEvent> openAsync(Long timeout);`
- `public Future<ChannelClosedEvent> closeAsync();`

Example

```
UniversalContactServerProtocol ucs1 = new UniversalContactServerProtocol(new Endpoint(HOST1,
PORT1));
UniversalContactServerProtocol ucs2 = new UniversalContactServerProtocol(new Endpoint(HOST2,
PORT2));

Future<ChannelOpenedEvent> fOpen1 =  ucs1.openAsync(15000L);
Future<ChannelOpenedEvent> fOpen2 =  ucs2.openAsync(15000L);

// TODO : do something

while (!(fOpen1.isDone() && fOpen2.isDone()) ) {
    // TODO : do something
    Thread.yield();
}


try {
    ChannelOpenedEvent ev1 = fOpen1.get();
    ChannelOpenedEvent ev2 = fOpen2.get();

   // TODO : something here with the opened protocols
}
catch(ExecutionException | InterruptedException ex) {

    // do something
}


 // ... in some other place
Future<ChannelClosedEvent> fClose1 = ucs1.closeAsync();
Future<ChannelClosedEvent> fClose2 = ucs2.closeAsync();

// TODO : do something

while (!(fClose1 .isDone() && fClose2 .isDone()) ) {
    // TODO : do something

    Thread.yield();
}
try {
    ChannelClosedEvent ev1 = fClose1.get();
    ChannelClosedEvent ev2 = fClose2.get();
    // TODO : something here
}
catch(ExecutionException | InterruptedException ex) {
    // do something
}
```

Configuring ADDP

The Advanced Disconnection Detection Protocol (ADDP) is a Genesys proprietary add-on to the TCP/IP stack. It implements a periodic poll when no actual activity occurs over a given connection. If a configurable timeout expires without a response from the opposite process, the connection is considered lost.

ADDP is enabled as part of the configuration process for a particular protocol connection instance, and can either be initialized before the connection is open or reconfigured on already opened connection.

> **Tip**
> Changing the configuration immediately after a connection is opened, or from the channel event handlers, is not recommended. Some connection configuration options (including ADDP) can be changed on the fly, however the channel configuration is not expected to change often or quickly - options are not treated as if they are dynamic values.

To enable ADDP, use the configuration options of your Endpoint object. Set the `UseAddp` property to `true` and configure the rest of the properties based on your desired performance.

Platform SDK connections have the following ADDP configuration options available:

- `protocol` - set the option value to addp to enable ADDP;
- `addp timeout` - specifies how often the client will send ADDP ping requests and wait for responses;
- `addp remote timeout` - specifies how often the server will send ADDP ping requests and wait for responses;
- `addp tracing enable` - used to enable logging of ADDP activities on both the client and server; can be set to "none", "local", "remote", "full" (or "both").

Here is an initialization code sample:

```
 PropertyConfiguration tserverConfig = new PropertyConfiguration();
 tserverConfig.setUseAddp(true);
 tserverConfig.setAddpServerTimeout(10);
 tserverConfig.setAddpClientTimeout(10);
 tserverConfig.setAddpTraceMode(AddpTraceMode.Both);

 Endpoint tserverEndpoint = new Endpoint("T-Server", TSERVER_HOST, TSERVER_PORT,
tserverConfig);
 TServerProtocol tserverProtocol = new TServerProtocol(tserverEndpoint);
```

or

```
 ...
 tserverConfig.setOption(AddpInterceptor.PROTOCOL_NAME_KEY, AddpInterceptor.NAME);
 tserverConfig.setOption(AddpInterceptor.TIMEOUT_KEY, "10");
 tserverConfig.setOption(AddpInterceptor.REMOTE_TIMEOUT_KEY, "11.5");
 tserverConfig.setOption(AddpInterceptor.TRACE_KEY, "full");
 ...
```

Note that timeout values are stored as strings and parsed as float values. So, it is ok to have:

```
 tserverConfig.setOption(AddpInterceptor.TIMEOUT_KEY, "10");
```

```
tserverConfig.setInteger(AddpInterceptor.TIMEOUT_KEY, 10);    // its the same value
tserverConfig.setOption(AddpInterceptor.TIMEOUT_KEY, "11.5"); // = is treated as 11500 ms
```

> **Tip**
> The minimum allowed value for ADDP timeouts is 1 (one second). If a timeout value is set to any value lower than 1, then a timeout of one second is used instead.

Also note that in `tserverConfig.setOption(AddpInterceptor.TRACE_KEY, "full")`, the `tserverConfig.setOption(...)` method accepts the following string values:

- "none" - no logging occurs
- "local" - ADDP activities are logged locally on the client side
- "remote" - a special initialization bit is sent in the ADDP initialization message to server side, asking the server to write its own ADDP tracing records to a server side log
- "full" - the equivalent of "local" + "remote"

Note that the comparison is case-insensitive for option values, so "FULL" == "Full" == "full". Unknown trace mode option values are treated as "none".

> **Tip**
> In release 8.1.0 of Platform SDK for Java, property handling logic was improved with truncation of the "CFGTM" prefix to automatically handle the Configuration Server protocol enumeration `CfgTraceMode.toString()`. So, if you use the latest Platform SDK 8.1.0 version for Java, writing `CfgTraceMode.CFGTMBoth.toString()` is acceptable, but earlier versions of Platform SDK for Java require that you translate the enumeration values to the corresponding string values.

Configuring IPv6 Connection

For backward compatibility with older/legacy Genesys servers and platforms, Platform SDK has disabled usage of IPv6 addresses by default. However, IPv6 usage may be explicitly enabled for a particular connection through specific connection configuration options.

There are two Platform SDK connection options to configure IPv6 usage:

1. enable-ipv6: possible values are:
    - '0' - support disabled (default)
    - '1' - support enabled;
2. ip-version: possible values are:
    - '4,6' - look for IPv4 addresses first (default)

- '6,4' - look for IPv6 addresses first.

```
 PropertyConfiguration tserverConfig = new PropertyConfiguration();
 tserverConfig.setIPv6Enabled(true);
 tserverConfig.setIPVersion(Connection.IP_VERSION_6_4);

 Endpoint tserverEndpoint = new Endpoint("T-Server", TSERVER_HOST, TSERVER_PORT,
tserverConfig);
 TServerProtocol tserverProtocol = new TServerProtocol(tserverEndpoint);
```

Note: In some environment configurations (Java 5, some Java 6 versions, or old versions of Windows) an `java.net.SocketException: Address family not supported by protocol` exception may occur. The problem is in the underlying JVM/OS platform related to NIO functionality.

This error may be resolved by:

1. Switching to Java 7+ version;
2. Switching to OIO usage instead of NIO with jvm system property:

```
-Dcom.genesyslab.platform.commons.connection.impl.netty.transport=OIO
```

or with Java method call (should be executed before any of Platform SDK protocols creation)

```
PsdkCustomization.setOption(PsdkOption.NettyTransportType, "OIO");
```

Configuring Client-Side Host/Port

Platform SDK allows client socket local host/port binding for Platform SDK connections.

```
 PropertyConfiguration tserverConfig = new PropertyConfiguration();
 tserverConfig.setLocalBindingPort(localPort);
 tserverConfig.setLocalBindingHost(localHost);

 Endpoint tserverEndpoint = new Endpoint("T-Server", TSERVER_HOST, TSERVER_PORT,
tserverConfig);
 TServerProtocol tserverProtocol = new TServerProtocol(tserverEndpoint);
```

Important

Be aware that client-side port binding may affect the restoration procedure for connections, leading to errors such as "port is in use". This error may also occur if the target hostname is resolved to several IP addresses and Platform SDK failed to connect to the first of them.

There is a special case regarding the usage of local port binding with TServer High Availiability (HA) connections. HA protocol connections hold two real connections behind the scenes: one to the primary and one to the backup. The system does not allow both connections to bind to the same local port, so to handle this situation an additional parameter is required for the backup connection local port binding:

```
 PropertyConfiguration tserverConfig = new PropertyConfiguration();
 tserverConfig.setLocalBindingPort(localPort);
 tserverConfig.setInteger(Connection.BACKUP_BIND_PORT_KEY, localPort2);

 Endpoint tserverEndpoint = new Endpoint("T-Server", TSERVER_HOST, TSERVER_PORT,
tserverConfig);
 TServerProtocol tserverProtocol = new TServerProtocol(tserverEndpoint);
```

Configuring Warm Standby

The WarmStandby Application Block will help you connect or reconnect to your Genesys servers. You will benefit by using the WarmStandby for every application that needs to maintain open connections to Genesys servers, whether you use hot standby or you are only connecting to a single server with no backup redundancy configured.

If you use hot standby, use the WarmStandby Application Block when retrying the connection to your primary or backup server until success, or for reconnecting after both the primary and backup servers are unavailable.

If you are connecting to a single server, use the WarmStandby Application Block to retry the first connection or to reconnect after that server has been unavailable. In this case, configure the `WarmStandbyService` to use the same Endpoint as primary and backup.

Activating the WarmStandby Application Block Service

To activate the WarmStandby Application Block, you create, configure and start a `WarmStandbyService` object. Two Endpoint objects must be defined: one with parameters for connecting to your primary server and one for connecting to your backup server. You must also remember to start the `WarmStandbyService` before opening the protocol.

```
 Endpoint tserverEndpoint = new Endpoint("T-Server", TSERVER_HOST, TSERVER_PORT,
tserverConfig);
 Endpoint tserverBackupEndpoint = new Endpoint("T-Server", TSERVER_BACKUP_HOST,
           TSERVER_BACKUP_PORT, tserverConfig);

 TServerProtocol tserverProtocol = new TServerProtocol(tserverEndpoint);

 WarmStandbyConfiguration warmStandbyConfig = new WarmStandbyConfiguration(tserverEndpoint,
            tserverBackupEndpoint);
 warmStandbyConfig.setTimeout(5000);
 warmStandbyConfig.setAttempts((short)2);

 WarmStandbyService warmStandby = new WarmStandbyService(tserverProtocol);
 warmStandby.applyConfiguration(warmStandbyConfig);
 warmStandby.start();

 tserverProtocol.beginOpen();
```

Stopping the WarmStandby Application Block Service

Stop the `WarmStandbyService` object when your application does not need to maintain the connection with the server any longer. This is typically done at the end of your program.

```
 warmStandby.stop();
```

```
tserverProtocol.close();
```

For more information about how the WarmStandby Application Block works, please refer to the WarmStandby Application Block documentation.

AsyncInvokers

AsyncInvokers are an important aspect of the Platform SDK protocols. They encapsulate the way a piece of code is executed. By using invokers, you can customize what thread executes protocol channel events and handles protocol events. You can also use a thread-pool for parsing protocol messages.

For GUI applications, you normally want most of the logic to happen in the context of the GUI thread. That will enable you to update GUI elements directly, and will simplify your code because you will not have to care about multi-threading.

For instance, if you are working with a Swing application, you can use the following `AsyncInvoker` implementation:

```
public class SwingInvoker implements AsyncInvoker {

        @Override
        public void invoke(Runnable target) {
                SwingUtilities.invokeLater(target);
        }

        @Override
        public void dispose() {}

}
```

Assigning a Protocol Invoker

The protocol invoker is in charge of executing channel events (such as channel closed and channel opened) and protocol events (received messages from the server). Usually, when developing a GUI application, you will want to use the GUI thread for handling all kinds of protocol events. By using the AsyncInvoker class described in the section before, you can assign a protocol invoker like this:

```
TServerProtocol tserverProtocol = new TServerProtocol(tserverEndpoint);
tserverProtocol.setInvoker(new SwingInvoker());
```

The protocol invoker is of utmost importance for your application. If you do not explicitly set an invoker, then a default internal Platform SDK thread is used, and you will need to use care with possible multi-threading issues.

.NET

The applications you write with the Platform SDK need to communicate with one or more Genesys servers, so the first thing you need to do is create connections with these servers. You will have to reference libraries and add using statements to your project for each specific protocol you are working with. These steps are not explicitly described here because the files and packages required will vary depending on which protocols you plan to use. Once you have connected to a server, you use that connection to exchange messages with the server. For details about sending and receiving messages to and from a server, refer to the Event Handling article.

Creating a Protocol Object

To connect to a Genesys server, you create an instance of the associated protocol class. As an example, this article will describe a connection to a Genesys T-Server using the TServerProtocol class. (For different applications, please use this API Reference to check protocol details for the specific server that you wish to connect to.)

In order to create a protocol object, you will first need to create an Endpoint object which acts as a container for generic connection parameters. An Endpoint object contains, at a minimum, a server name, the host name where the server is running, and the port on which the server is listening. The server name will appear in logs but does not affect protocol behavior; it may be any name that is significant to you.

```
var tserverEndpoint = new Endpoint("T-Server", TServerHost, TServerPort);
var tserverProtocol = new TServerProtocol(tserverEndpoint);
```

After creating your protocol object, you need to specify some connection parameters that are specific to that protocol. These parameters will differ depending on which server you are connecting to. Please check to the sections specific to the server that you wish to connect to for more information.

Once configuration is complete, you can open the connection to your server.

Working with a Connection Synchronously

The easiest way to open a connection to your server is to do it synchronously, which means that the method will block any additional processing until the server connection has either opened successfully or failed definitively. This is commonly used for non-interactive, batch applications. In this case, you can add code for using the protocol directly after opening. In the case of failure, the open method will throw an exception that should be caught and handled.

```
tserverProtocol.Open();
// You can start sending requests here.
```

When you have finished communicating with your servers, you can close the connection. Similar to how a connection is opened, you can also choose to close a connection either synchronously or asynchronously by using one of the following methods:

```
tserverProtocol.Close();
```

Or:

```
tserverProtocol.Dispose();
```

Working with a Connection Asynchronously

You may prefer to open a connection using asynchronous (non-blocking) methods. This is usually preferred for user-interactive applications, in order to avoid blocking the GUI thread so that the application does not appear "frozen" to the user.

Important

When using the asynchronous connections, make sure that your code waits for the `Opened` event to fire before attempting to send or receive messages. Otherwise you might be trying to use a connection that is not yet open.

There are two types of asynchronous open/close protocol operations:

1. Pure asynchronous operations
2. IAsyncResult based asynchronous operations

Pure Asynchronous Operations

These operation use channel timeout with the following `ClientChannel` API methods:

- `public void BeginOpen();`
- `public void BeginClose();`

Example

```
var ucs = new UniversalContactServerProtocol(new Endpoint(HOST, PORT)) {Timeout =
            TimeSpan.FromSeconds(20)};
var openedEvent = new ManualResetEvent(false);
ucs.Opened += (sender, args) => openedEvent.Set();
var closedEvent = new ManualResetEvent(false);
ucs.Closed += (sender, args) => closedEvent.Set();
ucs.BeginOpen();

// ... in some other place
if (!openedEvent.WaitOne(TimeSpan.FromSeconds(30))) // here might be any timeout value
{
  // channel has not been opened yet
}

// TODO: Work with the channel

ucs.BeginClose();

// ... in some other place
if (!closedEvent.WaitOne(TimeSpan.FromSeconds(30))) // here might be any timeout value
{
  // channel has not been closed yet
}
```

IAsyncResult Based Asynchronous Operations

Instead of using a normal open or close method, you instead use the following `ClientChannel` API methods:

- `public IAsyncResult BeginOpen(AsyncCallback callback, object state);`
- `public IAsyncResult BeginOpen(TimeSpan timeout, AsyncCallback callback, object state);`
- `public void EndOpen(IAsyncResult iAsyncResult);`
- `public IAsyncResult BeginClose(AsyncCallback callback, object state);`
- `public IAsyncResult BeginClose(TimeSpan timeout, AsyncCallback callback, object state);`
- `public void EndClose(IAsyncResult iAsyncResult);`

These methods allow you to conveniently specify timeout values, register an internal channel listener, and start the beginOpen or beginClose (as appropriate) operation. If the operation is completed (with or without errors) before the timeout occurs, then the timeout is canceled; otherwise the operation is assumed to have failed. In either case, the internal channel handler is unregistered and the handler notified of the results.

Additional information about `IAsyncResult` interface is available on MSDN.

Example: Using Callbacks

```
var ucs = new UniversalContactServerProtocol(new Endpoint(HOST, PORT)) { Timeout =
          TimeSpan.FromSeconds(20) };
var openedEvent = new ManualResetEvent(false);
var closedEvent = new ManualResetEvent(false);
ucs.BeginOpen(ar =>
{
      try
      {
        ucs.EndOpen(ar);
        openedEvent.Set();
        // TODO: notify if operation is successful
      }
      catch (Exception e)
      {
        // TODO: notify about the error
      }
}, ucs);


// ... in some other place
if (!openedEvent.WaitOne(TimeSpan.FromSeconds(30))) // here might be any timeout value
{
      // channel has not been opened yet
}


ucs.BeginClose(ar =>
{
      try
      {
        ucs.EndClose(ar);
        closedEvent.Set();

        // TODO: notify if operation is successful
      }
      catch (Exception e)
      {
        // TODO: notify about the error
      }
}, ucs);
```

```
  // ... in some other place
  if (!closedEvent.WaitOne(TimeSpan.FromSeconds(30))) // here might be any timeout value
  {
        // channel has not been closed yet
  }
```

Example: Using IAsyncResult

```
var ucs = new UniversalContactServerProtocol(new Endpoint(HOST, PORT)) { Timeout =
            TimeSpan.FromSeconds(20) };
var openResult = ucs.BeginOpen(null, null);
// ... in some other place
try
{
  ucs.EndOpen(openResult);
  // TODO: notify if operation is successful
}
catch (Exception e)
{
  // TODO: notify about the error
}


var closeResult = ucs.BeginClose(null, null);
// ... in some other place
try
{
  ucs.EndClose(closeResult);
  // TODO: notify if operation is successful
}
catch (Exception e)
{
  // TODO: notify about the error
}
```

Configuring ADDP

The Advanced Disconnection Detection Protocol (ADDP) is a Genesys proprietary add-on to the TCP/IP stack. It implements a periodic poll when no actual activity occurs over a given connection. If a configurable timeout expires without a response from the opposite process, the connection is considered lost.

To enable ADDP, use the configuration options of your Endpoint object. Set the UseAddp property to true and configure the rest of the properties based on your desired performance. For a description of all ADDP-related options, please refer to the API Reference.

```
var tserverConfig = new PropertyConfiguration();
tserverConfig.UseAddp = true;
tserverConfig.AddpServerTimeout = 10;
tserverConfig.AddpClientTimeout = 10;
tserverConfig.AddpTrace = "both";

var tserverEndpoint = new Endpoint("T-Server", TServerHost, TServerPort, tserverConfig);
var tserverProtocol = new TServerProtocol(tserverEndpoint);
```

Tip

The minimum allowed value for ADDP timeouts is 1 (one second). If a timeout value is set to any value lower than 1, then a timeout of one second is used instead.

Tip

In release 8.1.1 of Platform SDK for .NET, property handling logic was improved with truncation of the "CFGTM" prefix to automatically handle the Configuration Server protocol enumeration `CfgTraceMode.toString()`. So, if you use the latest Platform SDK 8.1.1 version for .NET, writing `CfgTraceMode.CFGTMBoth.toString()` is acceptable, but earlier versions of Platform SDK for .NET require that you translate the enumeration values to the corresponding string values.

Configuring IPv6 Connection

For backward compatibility with older/legacy Genesys servers and platforms, Platform SDK has disabled usage of IPv6 addresses by default. However, IPv6 usage may be explicitly enabled for a particular connection through specific connection configuration options.

There are two Platform SDK connection options to configure IPv6 usage:

1. enable-ipv6: possible values are:
   - 'false' - support disabled (default)
   - 'true' - support enabled
2. ip-version: possible values are:
   - '4,6' - look for IPv4 addresses first (default)
   - '6,4' - look for IPv6 addresses first

```
PropertyConfiguration configuration = new PropertyConfiguration();
configuration.SetOption(CommonConnection.EnableIPv6Key, "1");
configuration.SetOption(CommonConnection.IpVersionKey, "6,4");
var client = new TServerProtocol(new Endpoint(host, port, configuration));
```

or:

```
PropertyConfiguration configuration = new PropertyConfiguration
{
       IPv6Enabled = true,
       IPVersion = "6,4"
};
var client = new TServerProtocol(new Endpoint(host, port, configuration));
```

Configuring Client Side Host/Port

Platform SDK allows client socket local host/port binding for Platform SDK connections.

```
PropertyConfiguration configuration = new PropertyConfiguration
{
   LocalBindingHost = host,
   LocalBindingPort = port
};

var client = new TServerProtocol(new Endpoint(srvHost, srvPort, configuration));
```

Important

Be aware that client-side port binding may affect the restoration procedure for connections, leading to errors such as "port is in use". This error may also occur if the target hostname is resolved to several IP addresses and Platform SDK failed to connect to the first of them.

There is a special case regarding the usage of local port binding with T-Server High Availability (HA) connections. HA protocol connections hold two real connections behind the scenes: one to the primary and one to the backup. The system does not allow both connections to bind to the same local port, so to handle this situation an additional parameter is required for the backup connection local port binding:

```
PropertyConfiguration configuration = new PropertyConfiguration
{
   LocalBindingHost = host,
   LocalBindingPort = port,
   BackupLocalBindingHost = host,
   BackupLocalBindingPort = backupPort
};

var client = new TServerProtocol(new Endpoint(srvHost, srvPort, configuration));
```

Configuring Warm Standby

The Warm Standby Application Block will help you connect or reconnect to your Genesys servers. You will benefit by using the Warm Standby for every application that needs to maintain open connections to Genesys servers, whether you use hot standby or you are only connecting to a single server with no backup redundancy configured.

If you use hot standby, use the Warm Standby Application Block when retrying the connection to your primary or backup server until success, or for reconnecting after both the primary and backup servers are unavailable.

If you are connecting to a single server, use the Warm Standby Application Block to retry the first connection or to reconnect after that server has been unavailable. In this case, configure the `WarmStandbyService` to use the same `Endpoint` as primary and backup.

Activating the WarmStandby Application Block

To activate the Warm Standby Application Block, you create, configure and start a `WarmStandbyService` object. Two `Endpoint` objects must be defined: one with parameters for connecting to your primary server and one for connecting to your backup server. You must also remember to start the `WarmStandbyService` before opening the protocol.

```
var tserverEndpoint = new Endpoint("T-Server", TServerHost, TServerPort, tserverConfig);
var tserverBackupEndpoint = new Endpoint("T-Server", TServerBackupHost,
           TServerBackupPort, tserverConfig);

var tserverProtocol = new TServerProtocol(tserverEndpoint);

var warmStandbyConfig = new WarmStandbyConfiguration(tserverEndpoint, tserverBackupEndpoint);
warmStandbyConfig.Timeout = 5000;
warmStandbyConfig.Attempts = 2;

var warmStandby = new WarmStandbyService(tserverProtocol);
warmStandby.ApplyConfiguration(warmStandbyConfig);
warmStandby.Start();

tserverProtocol.Open();
```

Stopping the WarmStandby Application Block

Stop the `WarmStandbyService` object when your application does not need to maintain the connection with the server any longer. This is typically done at the end of your program.

```
warmStandby.Stop();
tserverProtocol.Dispose();
```

For more information about how the Warm Standby Application Block works, please refer to the Warm Standby Application Block documentation.

AsyncInvokers

AsyncInvokers are an important aspect of the Platform SDK protocols. They encapsulate the way a piece of code is executed. By using invokers, you can customize what thread executes protocol channel events and handles protocol events. You can also use a thread-pool for parsing protocol messages.

For GUI applications, you normally want most of the logic to happen in the context of the GUI thread. That will enable you to update GUI elements directly, and will simplify your code because you will not have to care about multi-threading.

For instance, if you are working with a Windows Forms or WPF application,, you can use the following `IAsyncInvoker` implementation:

```
public class SyncContextInvoker : IAsyncInvoker
{
        private readonly SynchronizationContext syncContext;

        public SyncContextInvoker()
        {
                this.syncContext = SynchronizationContext.Current;
        }

        public void Invoke(Delegate d, params object[] args)
        {
                syncContext.Post(s => { d.DynamicInvoke(args); }, null);
        }

        public void Invoke(WaitCallback callback, object state)
        {
                syncContext.Post(s => { callback(state); }, null);
        }

        public void Invoke(EventHandler handler, object sender, EventArgs args)
        {
                syncContext.Post(s => { handler(sender, args); }, null);
        }
}
```

The Protocol Invoker

The protocol invoker is in charge of executing channel events (such as channel closed and channel opened) and protocol events (received messages from the server). Usually, when developing a GUI application, you will want to use the GUI thread for handling all kinds of protocol events. By using the class implemented in the section before, you can assign a protocol invoker like this:

```
var tserverProtocol = new TServerProtocol(tserverEndpoint);
tserverProtocol.Invoker = new SyncContextInvoker();
```

The protocol invoker is of utmost importance for your application. If you do not explicitly set an invoker, then a default internal Platform SDK thread is used, and you will need to use care with multi-threading issues.

Advanced: Multithreaded Message Parsing

Tip

Please apply this section only if your application is suffering from performance problems because of large message parsing. You should identify the bottleneck using profiling techniques, and should measure the effect after making these changes by using the same profiling techniques.

Take into account that the technique described here can affect the correctness of your application, since concurrently parsing messages can affect the order in which those messages are received. So use this technique only selectively and in places where order of received messages is not relevant.

Every message you receive from a Genesys server is formatted in some way. Most Genesys servers use binary protocols, while some use XML-based protocols. When your application receives one of these messages, it parses the message and places it in the message queue for the appropriate protocol.

By default, the Platform SDK uses a single thread for parsing all messages. This parsing can be time-consuming in certain cases, and some applications may face performance issues. For example, some applications may receive lots of large binary-format messages, such as some of the statistics messages generated by Stat Server, while others might need to parse messages in non-binary formats, such as the XML format used to communicate with Genesys Multimedia (or e-Services) servers.

If message parsing becomes a bottleneck for your application, you can try to enable multi-threaded message parsing. This is done by setting the protocol connection invoker to an invoker that dispatches work to a pool of threads. One such invoker is provided out-of-the-box:

```
statServerProtocol.SetConnectionInvoker(DefaultInvoker.InvokerSingleton);
```

Configuring Platform SDK Channel Encoding for String Values

While sending string attributes/values to a server (or to the other side of any connection), Platform SDK packs strings to their binary representation. The binary representation depends on actual charset encoding, so it is important that this data will be unpacked with correct encoding when received on the other side of the connection.

Genesys protocols do not allow client and server sides to synchronize (that is, exchange) the encoding being used, so application developers may need to handle this configuration manually. (**Exception:** A Configuration Server 8.1.2+ deployment that is configured as UTF-8 multi-lingual can automatically synchronize UTF-8 encoding with Platform SDK 8.1.3 or later. For details, see Connecting Using UTF-8 Character Encoding.) The most common situation requiring this type of configuration occurs when a Genesys server and the application using Platform SDK to connect with that server have different localization settings, causing default encoding to be different on both sides.

There are two possible solutions for synchronizing the client side encoding with that of the server side:

1. (Java only) Change default jvm encoding with the jvm argument: `java -Dfile.encoding=...`
   This changes the charset encoding for the entire jvm, so will affect the main application and any Platform SDK connections to other servers. It may affect the client application relation with other components on the client host.
2. (Java only) Starting with Platform SDK 8.1.3, the new `com.genesyslab.platform.defaultcharset` system property can be used to set default charset encoding for Platform SDK connections without the need to change default encoding for whole jvm.
   Platform SDK checks this property once before opening the first connection, and if a value is specified then it will be used as the default encoding for all Platform SDK connections (instead of the value defined for the jvm).
3. Configure a particular Platform SDK connection to use the server side encoding with following connection configuration option (added in Platform SDK 8.0.1):

```
[Java]

protocol = ...;

PropertyConfiguration conf = new PropertyConfiguration();
conf.setOption(Connection.STR_ATTR_ENCODING_NAME_KEY, "windows-1252");

protocol.configure(conf);
protocol.open();
```

```
[C#]

protocol = ...;

PropertyConfiguration conf = new PropertyConfiguration();
conf.SetOption(CommonConnection.StringAttributeEncodingKey, "windows-1252");

protocol.Configure(conf);
protocol.Open();
```

Using the Warm Standby Application Block

Tip

- This application block is a reusable production-quality component. It has been designed using industry best practices and provided with source code so it can be used "as is," extended, or tailored if you need to.
  Please see the License Agreement for details.
- The Warm Standby Application Block described in this topic is a redesign first available with Platform SDK 8.5.101.06 for Java or 8.5.101.06 for .NET, that provides no backwards-compatibility for earlier releases. For information about earlier versions of the Warm Standby Application Block, please read the Legacy Warm Standby Application Block Description.

This article describes how developers can use the Warm Standby Application Block to maintain availability of connections between their applications and Genesys servers. It applies to all Server Deployment Modes, no matter if single-server mode, primary-backup mode, or cluster (N+1) mode.

The `WarmStandby` class is designed to handle the process of connecting (first-time connection) and reconnecting (in case of a connection failure) to the Genesys servers. `WarmStandby` maintains a pool of server addresses and sequentially tries to connect to a server until an attempt is successful or the pool has been exhausted.

`WarmStandby` raises events about its behavior that can be traced by client code.

Supported Server Deployment Modes

- **Single Server:** `WarmStandby` assists with reconnection attempts to the same server when the client gets disconnected.
- **Classical Genesys Primary-Backup:** `WarmStandby` assists with reconnection attempts to the same server, and failovers among the pair of servers. In some cases, failovers need to wait for a delay for the backup server to become active (for example, with Configuration Server).
- **Active N+1 Cluster:** `WarmStandby` assists with reconnections and failovers.

In addition, any of these configurations can be deployed in **Multiple Data Centers** mode. In this mode, the service is deployed across Data Centers, where every Data Center has a set of servers configured in any of the Server Deployment Modes above. `WarmStandby` enables the client to do Data Center failovers by allowing the pool of server addresses to be reconfigured. Triggering a Site failover (pool reconfiguration) is the responsibility of the client application, as it will depend on deployment-specifics.

Behavior

Behavior in the case of pool-exhaustion is defined programmatically by your client application, and depends on your needs. In some cases, you will want `WarmStandby` to stop trying to connect. This is the case for client applications that need an open connection to go on with the application logic. In other cases you will want to continue attempting to connect, for keeping service availability. For that you can programmatically activate automatic restore, in order for `WarmStandby` to continue connection attempts in the background. The same client application may need to use both approaches, in order to start up by using the stop-on-pool-exhaustion approach (for example: a user authentication step on startup), and then activate automatic restore as soon as all the startup logic is done.

`WarmStandby` defines a concrete background automatic restore strategy that follows these rules:

- When an established connection to a server breaks (disconnection), the same server is retried immediately once (reconnection), in order to recover from casual network or protocol failures. A random delay can be configured (`ReconnectionRandomDelayRange`), which will be useful for client applications with a large number of running instances (such as custom agent desktops) so that every client does not try to reconnect at the same time.
- If reconnection fails, the next server in the pool is tried (failover). A configurable delay (`BackupDelay`) can be applied before the failover, for cases where a passive server may need some time to become active (such as a Configuration Server running in backup mode).
- After all the servers in the configured pool are tried, the pool is retried again, after a configurable delay (`RetryDelay`). This repeats indefinitely until the client application programmatically decides to stop. `RetryDelay` is a list, so that a back-off strategy can be applied to retry delays. For example: "first wait 5 seconds, then 10 seconds, and then 30 seconds for all future attempts."

Summary

The following is a brief summary of the different ways that client applications should use `WarmStandby`:

- Batch (synchronous, non-interactive) client applications will just need to call `open()`, and possibly check if the connection is open during their execution by using the `isOpen()` method.
- GUI (asynchronous, interactive) client applications will normally want to connect, but only keep the connection open after some other conditions hold (for example: user authentication, other connections also open, etc). They will therefore call `openAsync()` and then `autoRestore()` when appropriate.
- Daemon (lengthy, non-interactive) client applications can just call `autoRestore()` or, if they need to process the result of the open operation, the can use `autoRestore(false)` and then `open()` or `openAsync()`.

Java

Creating

Before creating a new WarmStandby instance, you first create a protocol instance for the server you want to connect to. Every WarmStandby constructor requires a protocol instance as a parameter, as shown in the examples below.

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs);
```

or

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs, new Endpoint("host1", port1), new Endpoint("host2",
port2));
```

Important

Once the WarmStandby object is created, you can no longer use the open and close operations for that protocol or set the channel endpoint directly. These operations will now be handled using the WarmStandby object instead.

Configuring

The configuration for WarmStandby contains the following information:

- Endpoints: a list of endpoints which will be processed while trying to open the channel;
- Timeout: timeout for the channel opening operation;
- BackupDelay: interval between getting disconnected from a server and the first attempt to switch endpoints;
- RetryDelay: intervals between cycles for trying to reconnect to a server;
- ReconnectionRandomDelayRange: maximum value of additional random delay.

Initial configuration of WarmStandby occurs inside the instance constructor, but an external configuration can be applied whenever it is convenient for your application.

There are two ways you can update WarmStandby configuration:

1. directly updating specific configuration values in your WarmStandby instance
2. maintaining and updating a WSConfig object to hold configuration details, and then applying the entire configuration to your WarmStandby implementation

For simple applications where WarmStandby configuration typically does not change and you are connecting to a small number of Genesys servers, the first method may be easier. But if your application uses a more dynamic approach for the WarmStandby feature, or if you want to apply the same configuration details to multiple protocol objects, then using WSConfig to manage the configuration details can simplify your programming.

Updating Configuration Directly

You can use the getConfig() method to return and modify the current WarmStandby configuration details, as shown below.

```
ws.getConfig()
        .setEndpoints(new Endpoint("host1", port1), new Endpoint("host2", port2))
        .setBackupDelay(2000)
        .setReconnectionRandomDelayRange(5000)
        .setRetryDelay(100, 500, 5000)
        .setTimeout(10000);
```

Note that you only need to set fields you want updated using this method. For example, if you use a constructor that sets Endpoint details then the setEndpoints line could be ignored.

Using the WSConfig Object

You can also create a WSConfig object that holds configuration details. This object allows you to update and manage configuration settings, and only have them applied to the WarmStandby object(s) when you are ready by using setConfig.

```
WSConfig  cfg = new WSConfig()
        .setEndpoints(new Endpoint("host1", port1), new Endpoint("host2", port2))
        .setBackupDelay(2000)
        .setReconnectionRandomDelayRange(5000)
        .setRetryDelay(100, 500, 5000)
        .setTimeout(10000);

ws.setConfig(cfg);
```

or

```
WSConfig  cfg = new WSConfig();
cfg.setEndpoints(new Endpoint("host1", port1), new Endpoint("host2", port2))
cfg.setBackupDelay(2000)
cfg.setReconnectionRandomDelayRange(5000)
cfg.setRetryDelay(100, 500, 5000)
cfg.setTimeout(10000);

ws.setConfig(cfg);
```

Using WarmStandby

Opening a Protocol Without Reconnect

The following code shows how to make a single connection attempt. If this attempt is unsuccessful then WarmStandby finishes its work.

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs, new Endpoint("host1", port1), new Endpoint("host2",
port2));
try {
        ws.open();
}
catch (WSNoAvailableServersException ex) {
        // TODO: Handle exception
}
catch (WSCanceledException ex) {
        // TODO: Handle exception
}
```

or

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs, new Endpoint("host1", port1), new Endpoint("host2",
port2));
try {
    ws.open();
}
catch (WSException ex) {
    // TODO: Handle exception
}
```

Opening a Protocol with Reconnect

The following code leads to an endless cycle of connection attempts, with some delays between attempts, until a success is found or a manual break occurs.

In this scenario, if a channel gets disconnected then WarmStandy will initiate a new cycle of connections to server.

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs, new Endpoint("host1", port1), new Endpoint("host2",
port2));
ws.autoRestore();
```

`WarmStandby.autoRestore()` is a way to instruct the WarmStandby: "you now take care of keeping the connection available in the background".

Closing a WarmStandby Connection

To close an open WarmStandby connection, use the close() method.

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs, new Endpoint("host1", port1), new Endpoint("host2",
port2));
// TODO: do something
ws.close();
```

The close() method automatically cancels the requirement for any further attempts to re-establish a connection. Use the autoRestore(), or autoRestore(boolean), method to re-enable reconnection attempts.

Asynchronous Operations of WarmStandby

Code samples above are for synchronous operations, which block the current thread.

Asynchronous operations are similar to synchronous but don’t block the running thread. To use asynchronous operations, use the table below and replace any calls to synchronous methods with the asynchronous equivalents.

Synchronous method	Asynchronous equivalent
open()	openAsync().get()
close()	closeAsync().get()

Tip

You can also use the WarmStandby asynchronous open/close operations with the `Future` interface.

Using WarmStandby Event Handlers

WarmStandby contains four events you should use for notification of connection status.

Event name	Event description
ChannelOpened	Notifies that channel was opened successfully.
ChannelDisconnected	Notifies that channel was disconnected.
EndpointTriedUnsuccessfully	Notifies that the another connection attempt was unsuccessful.
AllEndpointsTriedUnsuccessfully	Notifies that the all connection attempts in the current cycle were unsuccessful.

Important

Using open() and openAsync().get() methods for your WarmStandby connection inside these event handlers will cause a thrown exception.

.NET

Creating

Before creating a new WarmStandby instance, you first create a protocol instance for the server you want to connect to. Every WarmStandby constructor requires a protocol instance as a parameter, as shown in the examples below.

```
var ws = new WarmStandby(new UniversalContactServerProtocol());
// any other child of ClientChannel may be used
```

Important

Once the WarmStandby object is created, you can no longer use the open and close operations for that protocol or set the channel endpoint directly. These operations will now be handled using the WarmStandby object instead.

Configuring

The configuration for WarmStandby contains the following information:

- Endpoints: a list of endpoints which will be processed while trying to open the channel;
- Timeout: timeout for the channel opening operation;
- BackupDelay: interval between getting disconnected from a server and the first attempt to switch endpoints;
- RetryDelay: intervals between cycles for trying to reconnect to a server;
- ReconnectionRandomDelayRange: maximum value of additional random delay.

Initial configuration of WarmStandby occurs inside the instance constructor, but you can also use the WSConfig object to hold a custom values that can be maintained and applied whenever it is convenient for your application. WSConfig allows your application to adjust the WarmStandby configuration details as needed, or allows you to apply the same configuration details to multiple protocol objects.

There are two ways to create a WSConfig object:

```
var cfg = new WSConfig
{
      Endpoints = new List<Endpoint>
      {
            new Endpoint("host1", port1),
            new Endpoint("host2", port2),
            new Endpoint("host3", port3)
      },
      BackupDelay = 2000,
      ReconnectionRandomDelayRange = 5000,
      RetryDelay = new []{100, 500, 5000},
      Timeout = 10000
};
```

or

```
cfg = new WSConfig();
cfg.SetEndpoints(new Endpoint("host1", port1),
      new Endpoint("host2", port2),
      new Endpoint("host3", port3));
cfg.SetRetryDelay(100, 500, 5000);
cfg.BackupDelay = 2000;
cfg.Timeout = 10000;
cfg.ReconnectionRandomDelayRange = 5000;
```

Then you can easily apply the configuration to an existing WarmStandby instance:

```
ws.Configuration = cfg;
```

Using WarmStandby

Opening a Protocol Without Reconnect

The following code shows how to make a single connection attempt. If this attempt is unsuccessful then WarmStandby finishes its work.

```
var ws = new WarmStandby(new UniversalContactServerProtocol ());
// any other child of ClientChannel may be used
var cfg = new WSConfig()
{
      Timeout = 1000,
      BackupDelay = 2000,
      ReconnectionRandomDelayRange = 3000
}.SetEndpoints(new Endpoint("host1", port1), new Endpoint("host2",
port2)).SetRetryDelay(1000, 2000);
ws.Configuration = cfg;
try
{
      ws.Open();
}
catch (Exception)
{
      // TODO: Handle exception
}
```

Opening a Protocol with Reconnect

The following code leads to an endless cycle of connection attempts, with some delays between attempts, until a success is found or a manual break occurs.

In this scenario, if a channel gets disconnected then WarmStandy will initiate a new cycle of connections to server.

```
var ws = new WarmStandby(new UniversalContactServerProtocol ());
// any other child of ClientChannel may be used
var cfg = new WSConfig()
{
      Timeout = 1000,
      BackupDelay = 2000,
      ReconnectionRandomDelayRange = 3000
}.SetEndpoints(new Endpoint("host1", port1), new Endpoint("host2",
port2)).SetRetryDelay(1000, 2000);
ws.Configuration = cfg;
ws.AutoRestore(true); // leads to opening of Warmstandby
```

Closing a WarmStandby Connection

To close an open WarmStandby connection, use the Close() method.

```
var ws = new WarmStandby(new UniversalContactServerProtocol ());
// any other child of ClientChannel may be used
// TODO: do something
try
{
      ws.Close();
}
catch (Exception)
{
        // TODO: Handle exception
}
```

The Close() method automatically cancels the requirement for any further attempts to re-establish a connection. Use the AutoRestore(), or AutoRestore(bool), method to re-enable reconnection attempts.

Asynchronous Operations of WarmStandby

Code samples above are for synchronous operations, which lock the current thread.

Asynchronous operations are similar to synchronous but don’t lock the running thread. To use asynchronous operations, use the table below and replace any calls to synchronous methods with the asynchronous equivalents.

Synchronous method	Asynchronous equivalent
Open()	EndOpen(BeginOpen(null,null))
Close()	EndClose(BeginClose(null,null))

Tip

WarmStandby .NET asynchronous open/close operations were designed using `IAsyncResult` interface.

Using WarmStandby Event Handlers

WarmStandby contains four events you should use for notification of connection status.

Event name	Event description
ChannelOpened	Notifies that channel was opened successfully.
ChannelDisconnected	Notifies that channel was disconnected.
EndpointTriedUnsuccessfully	Notifies that the another connection attempt was unsuccessful.
AllEndpointsTriedUnsuccessfully	Notifies that the all connection attempts in the current cycle were unsuccessful.

Important

Using the Open() or EndOpen(IAsyncResult) methods for your WarmStandby connection inside these event handlers will cause a thrown exception.

Using the Application Template Application Block

Important

This application block is a reusable production-quality component. It has been designed using industry best practices and provided with source code so it can be used "as is," extended, or tailored if you need to.

Please see the License Agreement for details.

Java

The Application Template Application Block provides a way to read configuration options for applications in Genesys Administrator and to configure Platform SDK protocols. It also allows standard connection settings (including ADDP or TLS details) to be retrieved from Configuration Server, and helps with common features like setting up WarmStandby or assigning message filters. Primary Application Template functionality includes:

- `ClientConfigurationHelper` sets up client connections and configures WarmStandby.
- `ServerConfigurationHelper` sets up server connections.
- `GConfigTlsPropertyReader` extracts TLS-related option values from configuration objects, and is intended to be used together with `com.genesyslab.platform.commons.connection.tls.TLSConfigurationParser`.
- `FilterConfigurationHelper` helps to bind message filters with protocol objects.
- `GFApplicationConfigurationManager` monitors the application configuration from Configuration Server and provides notification of any updates to options for your custom application, options of connected servers, or options of their host objects. If Log4j2 logging framework exists, then this component also enables Log4j2 configuration based on the application logging options in Configuration Manager. For more information, refer to the Additional Logging Features article.
- `ClusterClientConfigurationHelper` helps create and configure the Cluster Protocol Application Block.

Setting Up a Client Connection

Application Template helper creates `com.genesyslab.platform.Endpoint` instance with initialized configuration properties. Details about how to specify required options in Configuration server are available below. In order to retrieve specified options from Configuration Server, user should read `IGApplicationConfiguration` object where this properties are stored.

Sample:

```
//create Configuration Service
ConfServerProtocol confProtocol = new ConfServerProtocol(new Endpoint(host,port));
confProtocol.setUserName("...");
confProtocol.setClientName("...");
confProtocol.setClientApplicationType(CfgAppType.CFGSCE.ordinal());
IConfService confService = ConfServiceFactory.createConfService(confProtocol);
confProtocol.open();


//read your application options
String appName = "my-app-name";
CfgApplication cfgApplication = confService.retrieveObject(CfgApplication.class,
                new CfgApplicationQuery(appName));

GCOMApplicationConfiguration appConfiguration = new
GCOMApplicationConfiguration(cfgApplication);

//get particular connection definition
IGAppConnConfiguration connConfig = appConfiguration.getAppServer(CfgAppType.CFGStatServer);

//returns configured endpoint.
Endpoint epStatSrv = ClientConfigurationHelper.createEndpoint(appConfiguration,
    connConfig, connConfig.getTargetServerConfiguration());

//use protocol with configured endpoint
StatServerProtocol statProtocol = new StatServerProtocol(epStatSrv);
statProtocol.setClientName(clientName);
statProtocol.open();
```

Tip

Instructions on how to enable TLS using the Application Template Application Block are part of the TLS-specific documentation.

Configuring WarmStandby

Note: This section was updated for Platform SDK for Java release 8.5.102.02. For earlier releases, expand the "Legacy Content" text at the end of this section.

The Application Template helper `createWarmStandbyConfigEx()` allows you to create a configuration for the new implementation of the warm standby, as illustrated here:

```
String appName = "my-app-name"
CfgApplication cfgApplication = confService.retrieveObject(
           CfgApplication.class, new CfgApplicationQuery(appName));

GCOMApplicationConfiguration appConfiguration =
           new GCOMApplicationConfiguration(cfgApplication);

IGAppConnConfiguration connConfig = appConfiguration.getAppServer(CfgAppType.CFGStatServer);

//Helper method for new WarmStandby
```

```
WSConfig wsConfig = ClientConfigurationHelper.createWarmStandbyConfigEx(appConfiguration,
connConfig);

StatServerProtocol statProtocol = new StatServerProtocol();
statProtocol.setClientName(clientName);

WarmStandby warmStandby = new WarmStandby(statProtocol);
warmStandby.setConfig(wsConfig);
warmStandby.autoRestore();
```

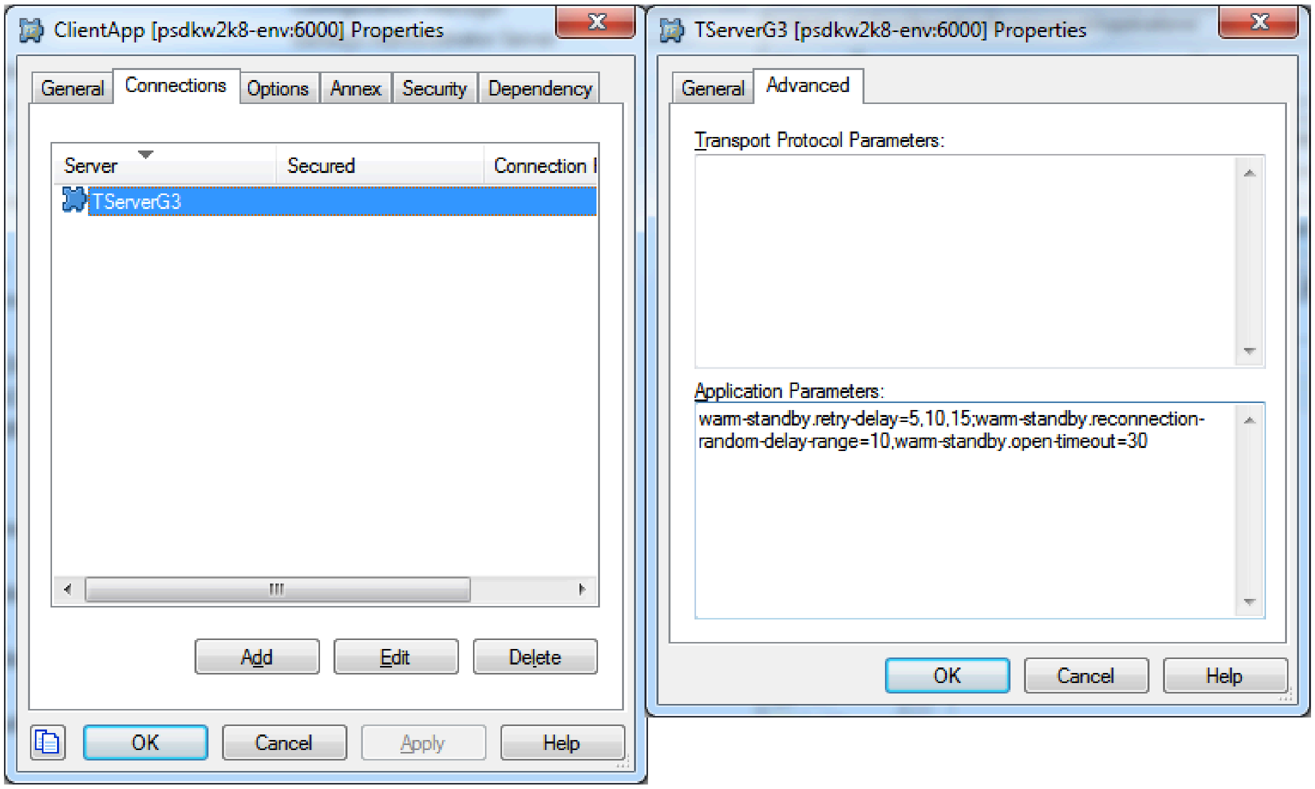
Configuration in Configuration Manager looks quite similar to the classic Warm Standby configuration with one difference: there are special timing parameters which are quite different than "Reconnection Timeout" and "Reconnection Attempts" and thus are specified apart from them. The "Reconnection Timeout" and "Reconnection Attempts" are not used in new Warm Standby Configuration.

Client Connection options:

Name	Values, in Seconds
warm-standby.retry-delay	5, 10, 15
warm-standby.reconnection-random-delay-range	10
warm-standby.open-timeout	30

See Warm Standby documentation and the API Reference guide for details about how these timing options customize Warm Standby behavior.

Options can be specified in Configuration Manager as shown below:



warm-standby.retry-delay=5,10,15;warm-standby.reconnection-random-delay-range=10,warm-standby.open-timeout=30

The last option is specified in Backup Server Applications on the Options tab. This characteristic of the backup server describes how much time is required for the server to step into primary mode.

Options Tab Section	Value, in Seconds
warm-standby	backup-delay=5

[+] Legacy Content

Configuring WarmStandby (Legacy Content)

Note: This section only applies to Platform SDK releases **prior** to:

- Java - 8.5.102.02
- .NET - 8.5.102.03

This section describes how to configure WarmStandby service with Application Template helpers. To find out how to use the WarmStandby service, see the corresponding Using the Warm Standby Application Block article.

Application Template helper method creates configuration for `com.genesyslab.platform.applicationblocks.warmstandby.WarmStandbyService`. The result includes parameters for the connection to primary and backup servers defined in the specified application configuration information.

Sample:

```
String appName = "my-app-name";
CfgApplication cfgApplication = confService.retrieveObject(
                   CfgApplication.class, new CfgApplicationQuery(appName));

GCOMApplicationConfiguration appConfiguration = new
GCOMApplicationConfiguration(cfgApplication);
IGAppConnConfiguration connConfig = appConfiguration.getAppServer(CfgAppType.CFGStatServer);

WarmStandbyConfiguration wsConfig =
ClientConfigurationHelper.createWarmStandbyConfig(appConfiguration, connConfig);

StatServerProtocol statProtocol = new StatServerProtocol(wsConfig.getActiveEndpoint());
statProtocol.setClientName(clientName);
WarmStandbyService wsService = new WarmStandbyService(statProtocol);

wsService.applyConfiguration(wsConfig);
wsService.start();
statProtocol.beginOpen();
```

Setting Up a Server Channel

Similar to the creation of the client connection, provide the `IGApplicationConfiguration` object to the helper class.

Sample:

```
Endpoint endpoint = ServerConfigurationHelper.createListeningEndpoint(appConfiguration,
appConfiguration.getPortInfo("default"));
ExternalServiceProtocolListener serverChannel = new ExternalServiceProtocolListener(endpoint);
```

This helper creates an Endpoint instance initialized with properties like the listening TCP port number, ADDP parameters, and so on.

Setting Up TLS

Instructions on how to enable TLS using the Application Template Application Block are part of the TLS-specific documentation. Please refer to that article for details about how to enable secure connections using the Application Template.

Also, see how to Configure TLS Parameters in Configuration Manager for a client or server channel.

Enabling Message Filtering

Using the Debug Log Level in Platform SDK protocol may affect Application performance due to the huge amount of log information output. It is possible to setup message filters for a protocol object, where the filter is configured in Genesys Administrator. This way, production applications will be able to provide appropriate log traces for troubleshooting without hurting performance with overly verbose logging.

See how to setup message filters for additional details.

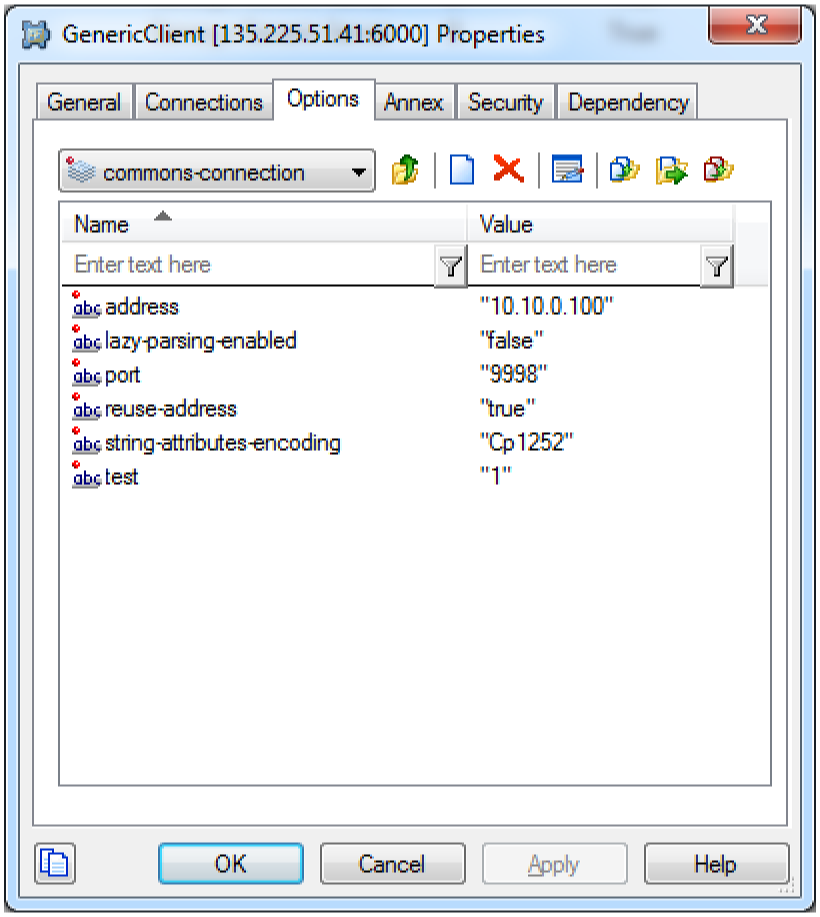
Defining Configuration Options in Genesys Administrator

Options can be specified in the `CfgApplication` object using Genesys Administrator. There are several possible option locations in the `CfgApplication` object:

- *Options* tab
- *Annex* tab
- *Connections* parameters
- *Port* parameters
- *Host* annex

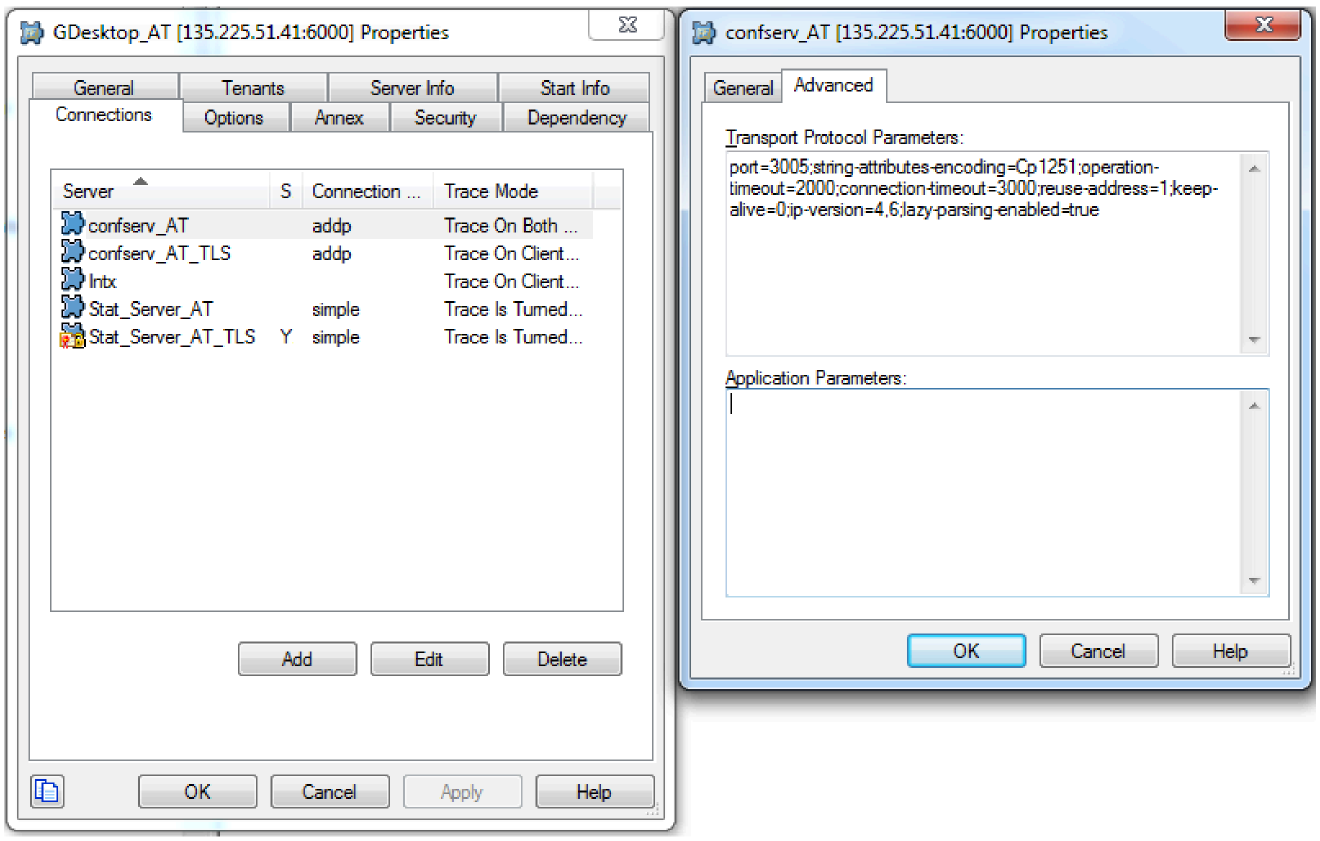
Common Options

Here is how options could be specified on the *Options* tab:



Name	Value
address	"10.10.0.100"
lazy-parsing-enabled	"false"
port	"9998"
reuse-address	"true"
string-attributes-encoding	"Cp1252"
test	"1"

Here is how options could be specified for a particular connection (using the *Connection* parameters):



Server	S	Connection ...	Trace Mode
confserv_AT		addp	Trace On Both ...
confserv_AT_TLS		addp	Trace On Client...
Intx			Trace On Client...
Stat_Server_AT		simple	Trace Is Turned...
Stat_Server_AT_TLS	Y	simple	Trace Is Turned...

Transport Protocol Parameters:

```
port=3005;string-attributes-encoding=Cp1251;operation-
timeout=2000;connection-timeout=3000;reuse-address=1;keep-
alive=0;ip-version=4,6;lazy-parsing-enabled=true
```

Application Parameters:

A complete options list is provided in the table below.

Section	Option	Description
commons-connection	string-attributes-encoding	Specifies encoding for string attributes.
	lazy-parsing-enabled	Boolean value. Enables or disables lazy parsing of properties, for which lazy parsing possibility is enabled in protocol. Currently used in Configuration Server protocol, enabled by default.
	address	Host bind option, specifies host from which connection should be made
	port	Port bind option, specifies port from which connection should be made
	backup-port	Port bind option, specifies port from which connection should be made (bind to) for backup server.
	operation-timeout	Integer value. Timeout for operations like stopReading and

Section	Option	Description
		resumeReading. Timeout is specified in milliseconds.
	connection-timeout	Integer value. Sets connection timeout option for the local socket to be opened by connection. Timeout is specified in milliseconds.
	reuse-address	Boolean value. Sets SO_REUSEADDR option for the local socket to be opened by connection
	keep-alive	Boolean value. Sets SO_KEEPALIVE option for the local socket to be opened by connection.
ucs-protocol	use-utf-for-responses	Boolean value. If set to false, UCSprotocol will add 'tkv.multibytes'='false' pair in Request KVlist of the message. It is false by default.
	use-utf-for-requests	Boolean value. If set to true, all string values of each KVlist will be packed as UtfStrings (in "UTF-16BE" encoding), instead of common strings. It is true by default.
webmedia-protocol	target-xml-version	Version of XML which is used to transport WebMedia protocol messages. See corresponding javax.xml.transform.OutputKeys.Version property for details.
	replace-illegal-unicode-chars	Boolean value. Enables or disables replacing of the illegal unicode chars in Webmedia XML messages.
	illegal-unicode-chars-replacement	String to replace illegal unicode chars

IPV6 Options

The IPV6 usage can be enabled with "enable-ipv6" option.

Section	Option	Description
common	enable-ipv6	Turns IPv6 support on/off.

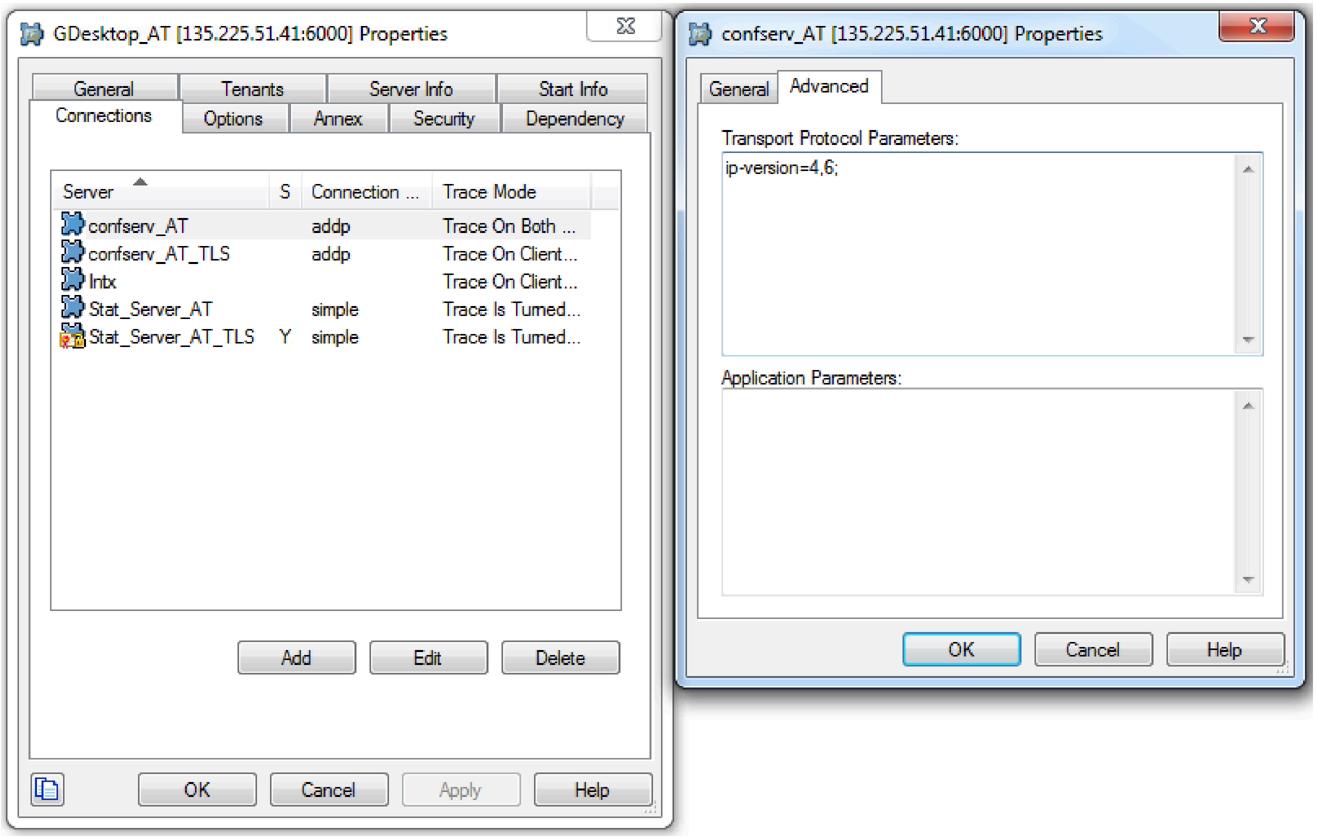
Section	Option	Description
		Possible values: 0 (default, implied) and 1. If set to 0, IPv6 support would be disabled, even if supported by OS/ platform.

The `ip-version` constant specifies the order in which connection attempts will be made to IPv6 and IPv4 addresses. Possible values include the following strings:

- "4,6" (default)
- "6,4"

This setting has no effect on the connection if `enable-ipv6` is set to 0; warning would be logged. Has no effect on server side; warning would be logged.

Option values should be specified within the connection transport properties:



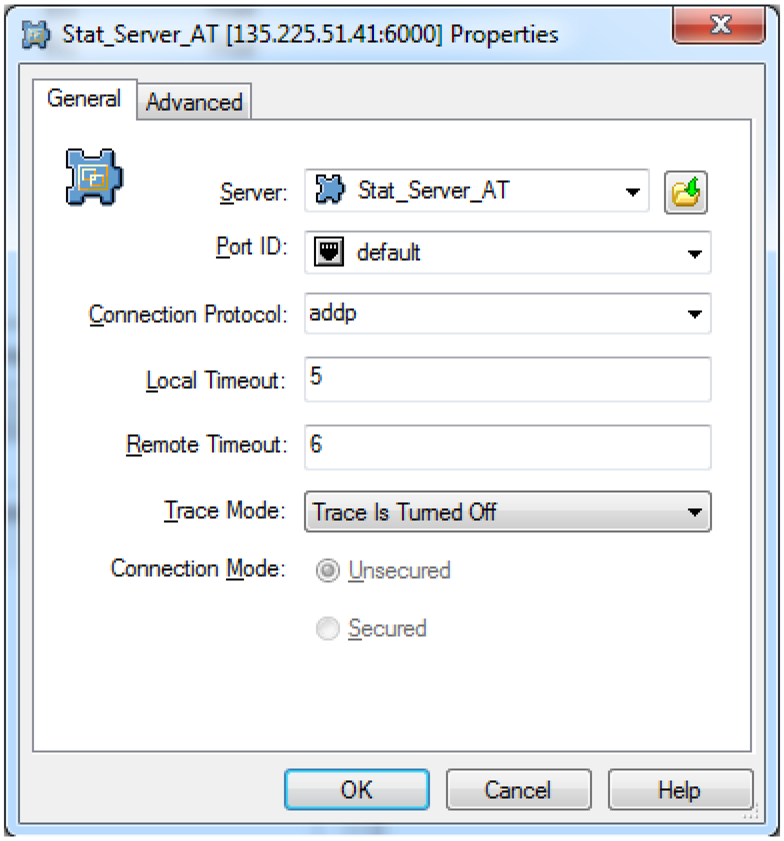
Server	S	Connection ...	Trace Mode
confserv_AT		addp	Trace On Both ...
confserv_AT_TLS		addp	Trace On Client...
Intx			Trace On Client...
Stat_Server_AT		simple	Trace Is Turned...
Stat_Server_AT_TLS	Y	simple	Trace Is Turned...

Transport Protocol Parameters:

```
ip-version=4,6;
```

ADDP Options

ADDP options, which can be handled with the Application Template helper, should be specified on the *Connections* tab.


Stat_Server_AT [135.225.51.41:6000] Properties

Server: Stat_Server_AT
Port ID: default
Connection Protocol: addp
Local Timeout: 5
Remote Timeout: 6
Trace Mode: Trace Is Turned Off
Connection Mode: [✓ Unsecured] Secured

TLS Options

See the article on Configuring TLS Parameters in Configuration Manager for more information about TLS options.

Cluster Connection Configuration Helpers

This section describes Application Template helper methods that support the configuration of cluster connections to Genesys server, including use cases and samples.

Cluster Connection Configuration Types

There are two types of Genesys server cluster configuration available in Configuration Server: client-aligned configuration, and node-aligned configuration.

Both types provide information about addresses of the target servers in a cluster, and connection options used to communicate with them. You must choose the proper helper to use for each type of cluster configuration.

Client-aligned configuration methods:

- `ClusterClientConfigurationHelper.createClusterProtocolEndpoints(IGApplicationConfiguration appConfig, CfgAppType serverType)`
- `ClusterClientConfigurationHelper.createClusterProtocolEndpoints(IGApplicationConfiguration appConfig, IGAppConnConfiguration clusterConn, CfgAppType serverType)`

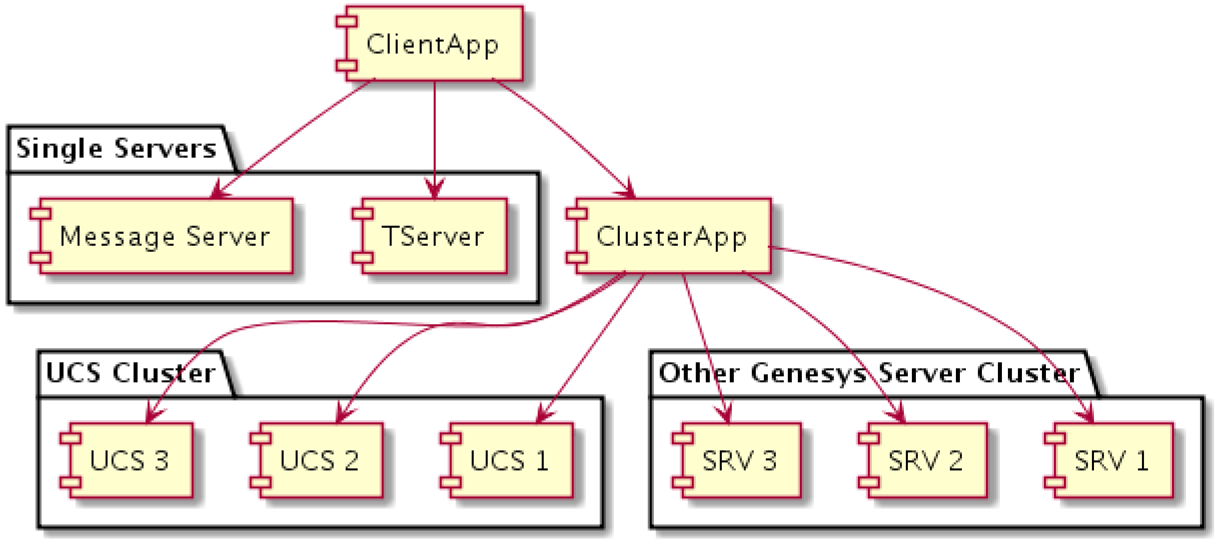
Node-aligned configuration methods:

- `ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(IConfService confService, IGApplicationConfiguration appConfig, CfgAppType serverType)`
- `ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(IConfService confService, IGApplicationConfiguration appConfig, IGAppConnConfiguration clusterConn, CfgAppType serverType)`

Client-Aligned Configuration Samples

Sample 1

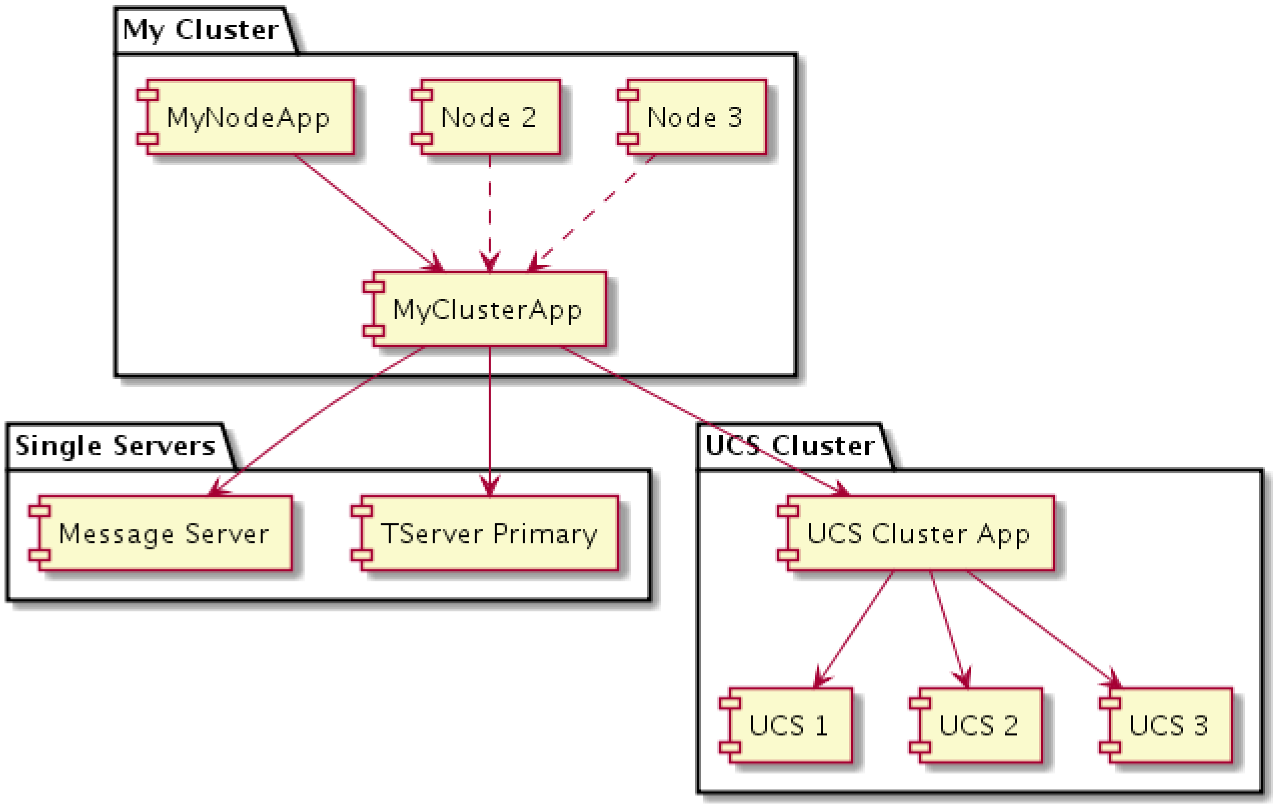
In the first sample scenario, a "ClientApp" Application is connected to the "ClusterApp'" virtual application (type=CFGApplicationCluster) that has connections to cluster nodes of one ore more clusters.



```
ClusterProtocol protocol ...
List<WSConfig> nodesList =
ClusterClientConfigurationHelper.createClusterProtocolEndpoints(ClientApp,
CfgAppType.CFGContactServer);
protocol.setNodes(nodesList);
```

Sample 2

In the second sample scenario, the "MyNodeApp" Application is a cluster node that is connected to the "MyClusterApp" virtual application. In this case, "MyClusterApp" is a shared store of connection configurations for all cluster nodes, which can have connections to other clusters like "UCS" as well as stand-alone servers.



```
ClusterProtocol protocol ...
List<WSConfig> nodesList =
ClusterClientConfigurationHelper.createClusterProtocolEndpoints(MyClusterApp,
clusterAppConnection, CfgAppType.CFGContactServer);
protocol.setNodes(nodesList);
```

Connection Options

You can use Genesys Administrator to define connection parameter options (such as ADDP, ip-version, strings encoding, or TCP socket options) that are used for all nodes in a cluster, or to define connection parameter for a particular node that will overriding common parameters.

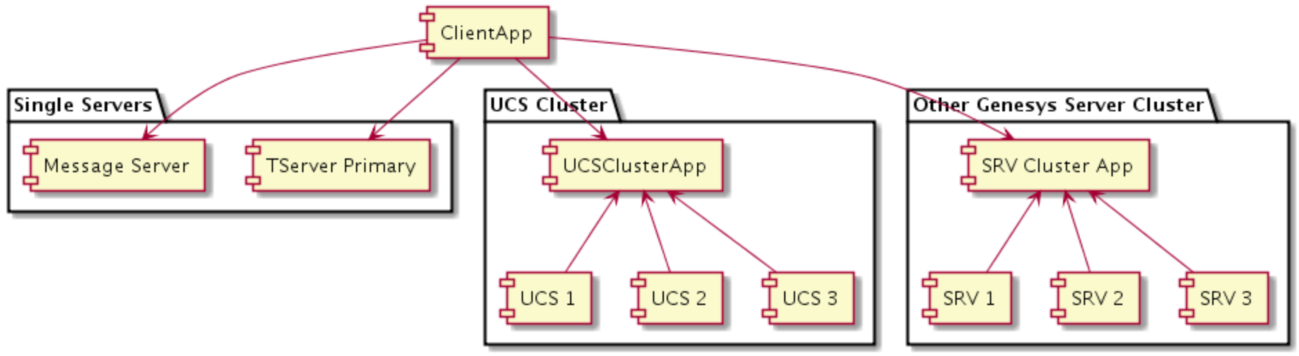
- Common Options: Specified in the connection from ClientApp to ClusterApp (or from OwnClusterApp if application is a node of some type of cluster).
- Node-Specific Options: can be specified in Configuration Manager in connection from ClusterApp to NodeApp.

See eServices Load Balancing Business Continuity for examples.

Node-Aligned Configuration

Sample 3

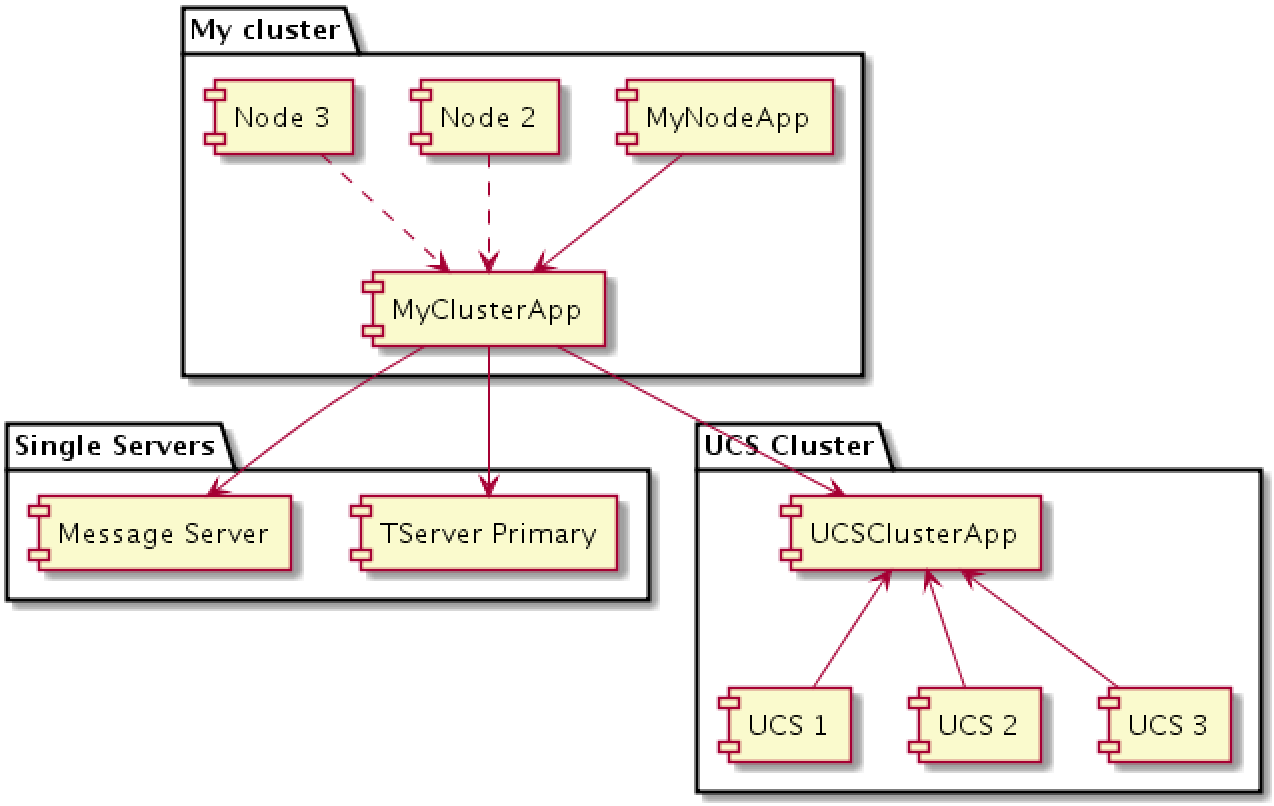
In the third sample scenario, the "ClientApp" Application is connected to the "UCSClusterApp" virtual application that groups UCS nodes. Unlike previous configurations, the cluster nodes in this scenario are also connected to the "UCSClusterApp" virtual application.



```
ClusterProtocol protocol ...
List<WSConfig> nodesList =
ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(confService, ClientApp,
CfgAppType.CFGContactServer);
protocol.setNodes(nodesList);
```

Sample 4

In the fourth sample scenario, the "MyNodeApp" Application is a cluster node. The "MyClusterApp" application has a connection to the UCS cluster ("UCSClusterApp" application). UCS nodes also have connections to "UCSClusterApp".



```
ClusterProtocol protocol ...
List<WSConfig> nodesList =
ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(confService, MyNodeApp,
clusterAppConnection, CfgAppType.CFGContactServer);
protocol.setNodes(nodesList);
```

Connection Options

You can use Genesys Administrator to define connection parameter options (such as ADDP, ip-version, strings encoding, or TCP socket options), but in this scenario this configuration is applied for all nodes in a cluster.

- Common Options: Specified in the connection from ClientApp to ClusterApp.

Client Connection Active Nodes Randomizer

Important

Dynamic updates to the cluster configuration are still possible with this approach. Your application should use the `shuffler.setNodes(nodes)` function to change the list of cluster nodes, instead of using Cluster Protocol methods directly.

The randomizer helper component supports cluster load balancing from client applications, by allowing your applications to work with a server cluster without creating live connections to all of its nodes. This also prevents many clients from being stuck on a single connection, to resolve issues such as a single server overloading on startup, and can be of critical importance in use cases where the number of clients could be in the tens of thousands.

This helper takes a list of cluster nodes, and performs periodical endpoint rotation. The existing Cluster Protocol functionality to update its nodes configuration dynamically..

For each new node that is added:

1. a new instance of the related PSDK protocol class is created
2. asynchronous open is started
3. after the node has connected to its server, the node is added to load balancer

For each node that is removed:

1. the removed node is immediately excluded from the load balancer
2. the node is scheduled to be closed after its protocol timeout delay (which allows responses to be delivered for in-progress requests)

The randomizer uses the `Collections.shuffle(nodes)` method to randomize the sequence of given nodes.When updating the protocol node configuration, the randomizer uses a round-robin rotation over a whole list of randomly-sorted cluster nodes, choosing a subset of given number of elements. This allows the protocol to avoid breaking all connections at a single moment, and to stay online during switchover.

The randomizer component also has an optional ability to attach truncated nodes' endpoints as backup endpoints for selected ones. This allows the internal WarmStandby service to quickly switchover from an unresponsive node.

Sample usage of this component could look like the following:

```
 final UcsClusterProtocol ucsNProtocol = new UcsClusterProtocolBuilder().build();
 List<WSConfig> nodes = ...;
 ClusterNodesShuffler shuffler = new ClusterNodesShuffler(ucsNProtocol, 2); // creates nodes
shuffler for 2 active connections
 shuffler.setUseBackups(true); // - lets the shuffler to use truncated nodes endpoints as
backup endpoints for the selected ones
 shuffler.setNodes(nodes); // - initializes shuffler with the whole list of cluster nodes

 ucsNProtocol.open();

 TimerActionTicket shufflerTimer = TimerFactory.getTimer().schedule(3000, 3000, shuffler); //
- schedules shuffling operation with 3 secs delay and 3 secs period

 // Do the business logic on the cluster protocol...
 // In case of update in the cluster configuration, application should use
'shuffler.setNodes(newNodes)'
 //     instead of ClusterProtocol's methods related to nodes configuration.

 // Shutdown:
 shufflerTimer.cancel();
 ucsNProtocol.close();
```

Code Samples

Simple Client Application Connecting to Any UCS Cluster

This sample checks the connection configuration for "WCC mode" UCS 9.0 cluster, then for "WDE mode" UCS cluster, and then for "legacy mode" connection to UCS (pri/bck) server.

```
// Take "my application configuration" from context, or read it in a way like this:
IGApplicationConfiguration myApp = new GCOMApplicationConfiguration(
        confService.retrieveObject(CfgApplication.class,
                new CfgApplicationQuery(myAppName)));

// For the first, try UCS 9 connection cluster:
List<WSConfig> conns = ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(
        confService, myApp, CfgAppType.CFGContactServer);
if (conns == null || conns.isEmpty()) {
    // If there is no UCS 9 cluster connected, then we try older UCS cluster, or simple UCS
connection(s):
    conns = ClusterClientConfigurationHelper.createClusterProtocolEndpoints(
            myApp, CfgAppType.CFGContactServer);
}

System.out.println("Connections: " + conns);
```

WCC-Based Cluster Node Application Connecting to Any UCS Cluster

This sample works in context of WCC.

```
// Take "my application configuration" from context, or read it in a way like this:
final IGApplicationConfiguration myApp = new GCOMApplicationConfiguration(
        confService.retrieveObject(CfgApplication.class,
                new CfgApplicationQuery(myAppName)));

IGApplicationConfiguration myClusterApp = null;
// if we do not have 'myClusterApp' from WCC context, we may take it by this way:
final List<IGAppConnConfiguration> clusters = GApplicationConfiguration
        .getAppServers(myApp.getAppServers(), CfgAppType.CFGApplicationCluster);
if (clusters != null) {
    if (clusters.size() == 1) {
        myClusterApp = clusters.get(0).getTargetServerConfiguration();
        log.infoFormat(
                "Application is recognized as a node of cluster ''{0}''",
                myClusterApp.getApplicationName());
    } else if (clusters.size() > 1) {
        log.error("Application has more than one application cluster connected"
                + " - its treated as a standalone app");
    }
}

// Select application cluster connection start point:
final IGApplicationConfiguration connSrc = (myClusterApp != null) ? myClusterApp : myApp;

// For the first, try UCS 9 connection cluster:
List<WSConfig> conns = ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(
        confService, connSrc, CfgAppType.CFGContactServer);
if (conns == null || conns.isEmpty()) {
    // If there is no UCS 9 cluster connected, then we try older UCS cluster, or simple UCS
connection(s):
    conns = ClusterClientConfigurationHelper.createClusterProtocolEndpoints(
            connSrc, CfgAppType.CFGContactServer);
}

System.out.println("Connections: " + conns);
```

Client Nodes Randomizer Usage Sample

Application code using the randomizer component may look like the following sample:

```
 final List<WSConfig> nodes = ...;
 final UcsClusterProtocol ucsNProtocol =
         new UcsClusterProtocolBuilder()
                 .build();
 ucsNProtocol.setTimeout(5000); // sets protocol timeout to 5 secs
 ucsNProtocol.setClientName("MyClientName");
 ucsNProtocol.setClientApplicationType("MyAppType");

 ClusterNodesShuffler shuffler = new ClusterNodesShuffler(ucsNProtocol, 2); // creates nodes
shuffler for 2 active connections
 TimerActionTicket shufflerTimer = null;

 try {
     shuffler.setNodes(nodes); // - initializes shuffler with the whole list of cluster nodes.
     ucsNProtocol.open();
     shufflerTimer = TimerFactory.getTimer().schedule(3000, 3000, shuffler); // - schedules
shuffling operation with 3 secs delay and 3 secs period

     // do the business logic on the cluster protocol...
     for (int i = 0; i < 200; i++) {
         EventGetVersion resp = (EventGetVersion)
ucsNProtocol.request(RequestGetVersion.create());
         System.err.println("Resp from: " + resp.getEndpoint());
         Thread.sleep(300);
     }
 } finally {
     if (shufflerTimer != null) {
         shufflerTimer.cancel();
         shufflerTimer = null;
     }
     ucsNProtocol.close();
 }
```

Handling Updates From Config Server

The `GFApplicationConfigurationManager` component monitors Config Server for updates and provides notifications about changes in applications.

You should register for updates at `GFApplicationConfigurationManager`.

```
GFApplicationConfigurationManager appManager = ...
appManager.register(new ClientConnEventListener());
appManager.init();
```

In the handle (`GFAppCfgEvent` event) method implementation, create a new connection configuration using one of the helpers mentioned above:

```
import com.genesyslab.platform.apptemplate.application;

public class ClientConnEventListener extends GFAppCfgEventListener {
    private UcsClusterProtocol ucsProtocol;
    @Override
    public void handle(GFAppCfgEvent event) {
        //get new application configuration
        IGApplicationConfiguration appconfig = event.getAppConfig();

        //create new connection configuration
        List<WSConfig> conns =
ClusterClientConfigurationHelper.createClusterProtocolEndpoints(appconfig,
CfgAppType.CFGContactServer);

        //apply connection configuration
        ucsProtocol.setNodes(conns);
    }
}
```

Important

WCC-based cluster configuration does not support an update handler at this time. Subscribing to updates in this case will lead to Config Server overloading, so customers are encouraged to make direct requests to Config Server to actualize the cluster configuration before opening `ClusterProtocol`.

.NET

The Application Template Application Block provides a way to read configuration options for applications in Genesys Administrator and to configure Platform SDK protocols. It also allows standard connection settings (including ADDP or TLS details) to be retrieved from Configuration Server, and helps with common features like setting up WarmStandby or assigning message filters. Primary Application Template functionality includes:

- `ClientConfigurationHelper` sets up client connections and configures WarmStandby.
- `ServerConfigurationHelper` sets up server connections.

Setting Up a Client Connection

Application Template helper creates an `Endpoint` instance with initialized configuration properties. Details about how to specify required options in Configuration server are available below. In order to retrieve specified options from Configuration Server, user should read `IGApplicationConfiguration` object where this properties are stored.

Sample:

```
//create Configuration Service
ConfServerProtocol confProtocol = new ConfServerProtocol(new Endpoint(host,port)){
  UserName = "...",
  ClientName = "...",
  ClientApplicationType= (int)cfgAppType,
  UserPassword = "..."
};
IConfService confService = ConfServiceFactory.CreateConfService(confProtocol);
confProtocol.Open();

//read your application options
var myApplicationName = "...";
var applicationConfiguration = new GCOMApplicationConfiguration(
      new CfgApplicationQuery(confService){ Name = myApplicationName }.ExecuteSingleResult()
);

CfgAppType myApplicationType = default(CfgAppType);
var applicationEndPoint = ClientConfigurationHelper.CreateEndpoint(applicationConfiguration,
      applicationConfiguration.GetAppServer(myApplicationType),
      applicationConfiguration.GetAppServer(myApplicationType).TargetServerConfiguration);

//use protocol with configured endpoint
StatServerProtocol statProtocol = new StatServerProtocol(applicationEndPoint);
statProtocol.ClientName = clientName;
statProtocol.Open();
```

Configuring WarmStandby

The Application Template helper `CreateWarmStandbyConfigEx()` allows you to create a configuration for the new implementation of the warm standby, as illustrated here:

```
var myApplicationName = "…";
CfgAppType myApplicationType = default(CfgAppType);
var applicationConfiguration = new GCOMApplicationConfiguration(
      new CfgApplicationQuery(confService){ Name = myApplicationName }.ExecuteSingleResult()
);
WSConfig warmStandbyConfig = ClientConfigurationHelper.CreateWarmStandbyConfigEx(
      applicationConfiguration,applicationConfiguration.GetAppServer(myApplicationType));
```

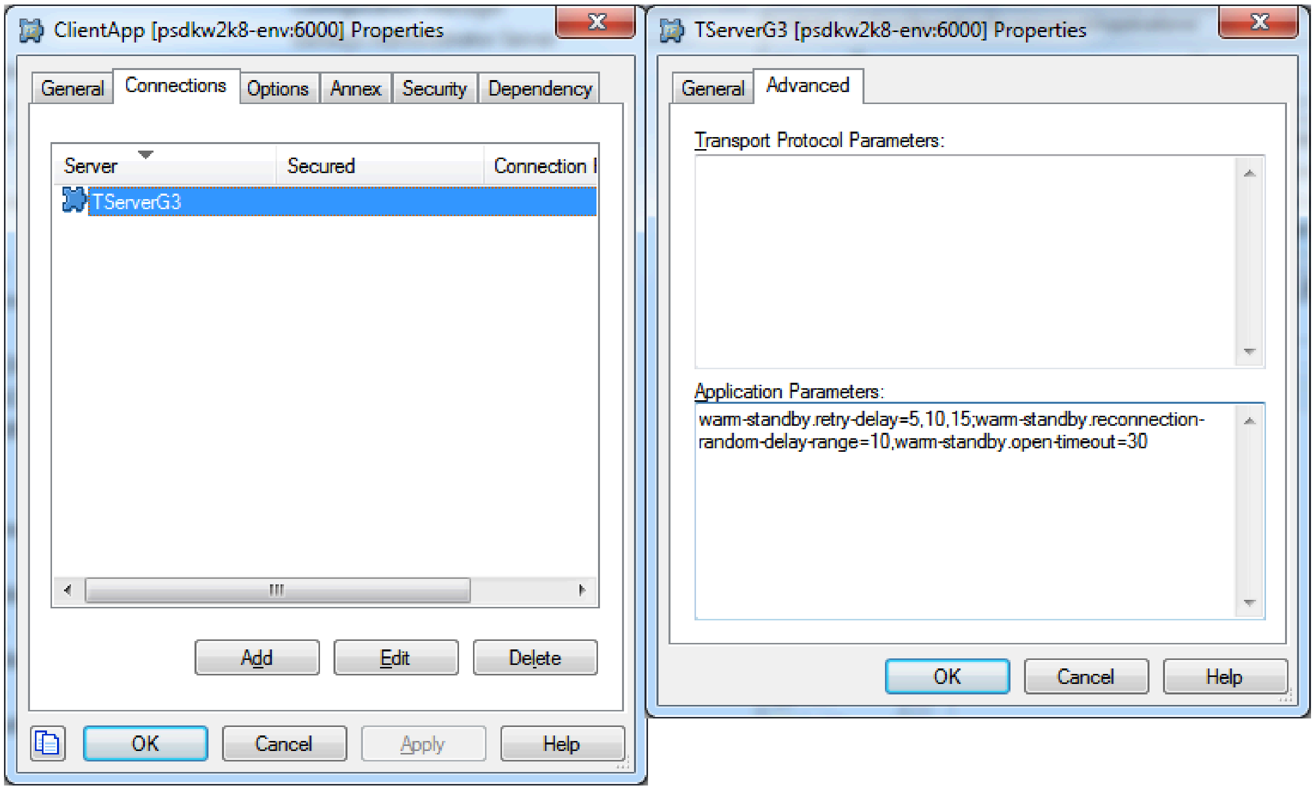
Configuration in Configuration Manager looks quite similar to the classic Warm Standby configuration with one difference: there are special timing parameters which are quite different than "Reconnection Timeout" and "Reconnection Attempts" and thus are specified apart from them. The "Reconnection Timeout" and "Reconnection Attempts" are not used in new Warm Standby Configuration.

Client Connection options:

Name	Values, in Seconds
warm-standby.retry-delay	5, 10, 15
warm-standby.reconnection-random-delay-range	10
warm-standby.open-timeout	30

See Warm Standby documentation and the API Reference guide for details about how these timing options customize Warm Standby behavior.

Options can be specified in Configuration Manager as shown below:



Application Parameters:

```
warm-standby.retry-delay=5,10,15;warm-standby.reconnection-random-delay-range=10,warm-standby.open-timeout=30
```

The last option is specified in Backup Server Applications on the Options tab. This characteristic of the backup server describes how much time is required for the server to step into primary mode.

Options Tab Section	Value, in Seconds
warm-standby	backup-delay=5

[+] Legacy Content

Configuring WarmStandby (Legacy Content)

This section describes how to configure WarmStandby service with Application Template helpers. To find out how to use the WarmStandby service, see the corresponding Using the Warm Standby Application Block article.

Application Template helper method creates configuration for `WarmStandbyService`. The result includes parameters for the connection to primary and backup servers defined in the specified application configuration information.

Sample:

```
var myApplicationName = "…";
CfgAppType myApplicationType = default(CfgAppType);
var applicationConfiguration = new GCOMApplicationConfiguration(
      new CfgApplicationQuery(confService){ Name = myApplicationName }.ExecuteSingleResult()
);
WarmStandbyConfiguration warmStandbyConfig =
ClientConfigurationHelper.CreateWarmStandbyConfig(
      applicationConfiguration,applicationConfiguration.GetAppServer(myApplicationType));
```

Setting Up a Server Channel

Similar to the creation of the client connection, provide the `IGApplicationConfiguration` object to the helper class.

Sample:

```
Endpoint endpoint = ServerConfigurationHelper.CreateListeningEndpoint(appConfiguration,
appConfiguration.PortInfo("default"));
ExternalServiceProtocolListener serverChannel = new ExternalServiceProtocolListener(endpoint);
```

This helper creates an Endpoint instance initialized with properties like the listening TCP port number, ADDP parameters, and so on.

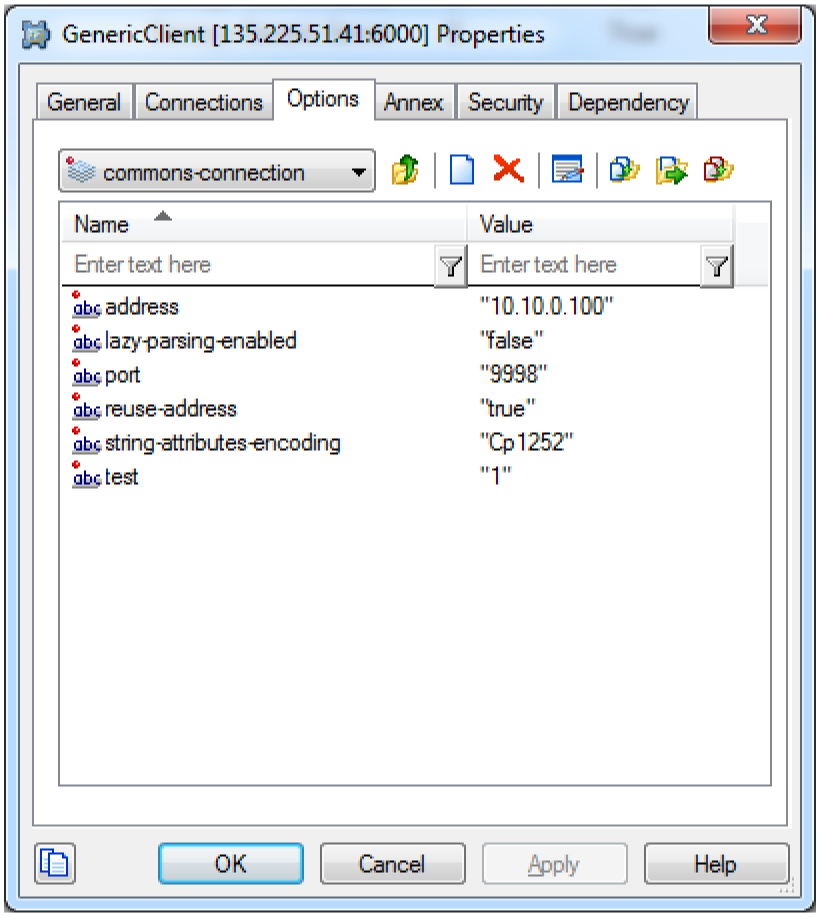
Defining Configuration Options in Genesys Administrator

Options can be specified in the `CfgApplication` object using Genesys Administrator. There are several possible option locations in the `CfgApplication` object:

- *Application* options tab
- *Connection* parameters
- *Port* parameters

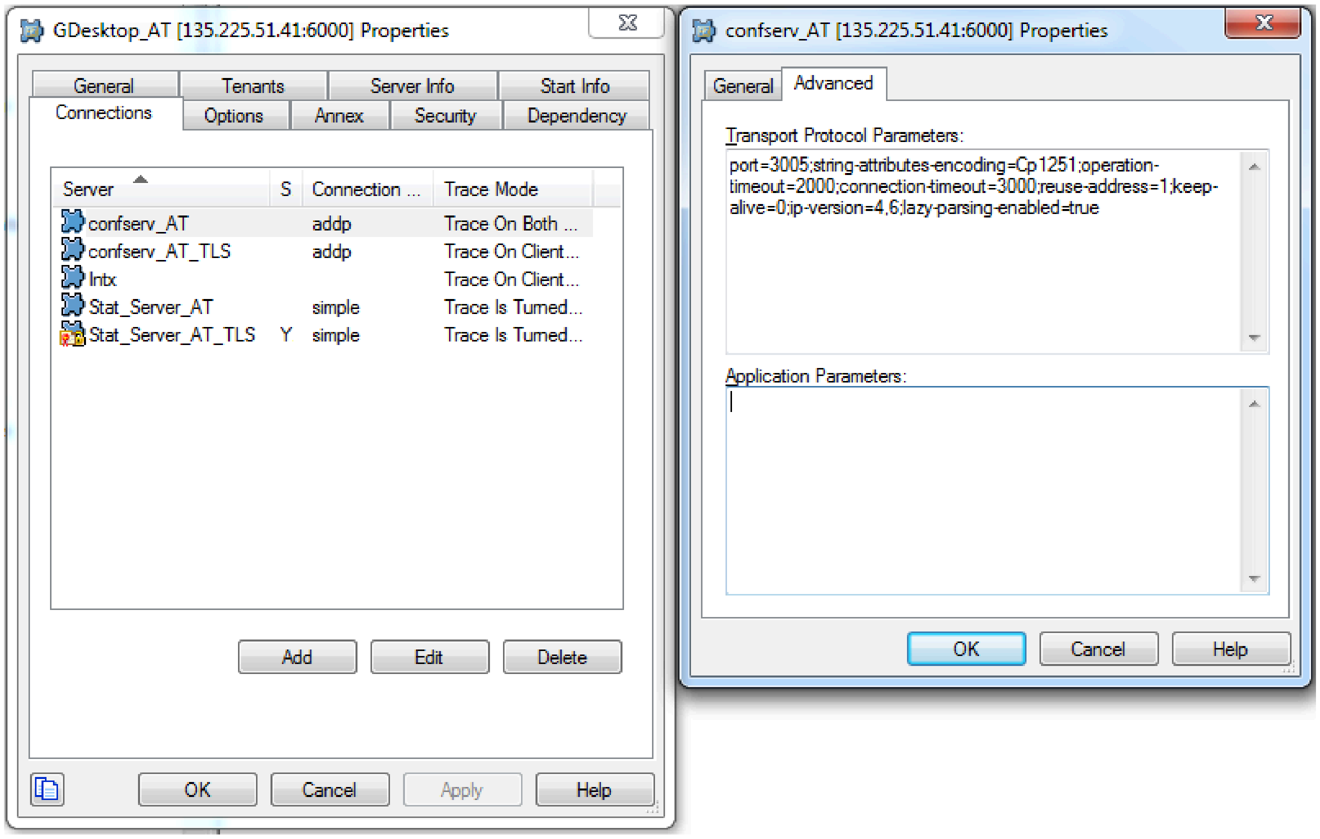
Common Options

Here is how options could be specified on the *Options* tab:



Name	Value
address	"10.10.0.100"
lazy-parsing-enabled	"false"
port	"9998"
reuse-address	"true"
string-attributes-encoding	"Cp1252"
test	"1"

Here is how options could be specified for a particular connection (using the *Connection* parameters):



Server	S	Connection ...	Trace Mode
confserv_AT		addp	Trace On Both ...
confserv_AT_TLS		addp	Trace On Client...
Intx			Trace On Client...
Stat_Server_AT		simple	Trace Is Turned...
Stat_Server_AT_TLS	Y	simple	Trace Is Turned...

Transport Protocol Parameters:

```
port=3005;string-attributes-encoding=Cp1251;operation-
timeout=2000;connection-timeout=3000;reuse-address=1;keep-
alive=0;ip-version=4,6;lazy-parsing-enabled=true
```

Application Parameters:

A complete options list is provided in the table below.

Section	Option	Description
commons-connection	string-attributes-encoding	Specifies encoding for string attributes.
	lazy-parsing-enabled	Boolean value. Enables or disables lazy parsing of properties, for which lazy parsing possibility is enabled in protocol. Currently used in Configuration Server protocol, enabled by default.
	address	Host bind option, specifies host from which connection should be made
	port	Port bind option, specifies port from which connection should be made
	backup-address	Host bind option, specifies host from which connection should be made for backup server.
	backup-port	Port bind option, specifies port from which connection should be

Section	Option	Description
		made (bind to) for backup server.
ucs-protocol	use-utf-for-responses	Boolean value. If set to false, UCSprotocol will add 'tkv.multibytes'='false' pair in Request KVlist of the message. It is false by default.
	use-utf-for-requests	Boolean value. If set to true, all string values of each KVlist will be packed as UtfStrings (in "UTF-16BE" encoding), instead of common strings. It is true by default.
webmedia-protocol	replace-illegal-unicode-chars	Boolean value. Enables or disables replacing of the illegal unicode chars in Webmedia XML messages.
	illegal-unicode-chars-replacement	String to replace illegal unicode chars

IPV6 Options

The IPV6 usage can be enabled with "enable-ipv6" option.

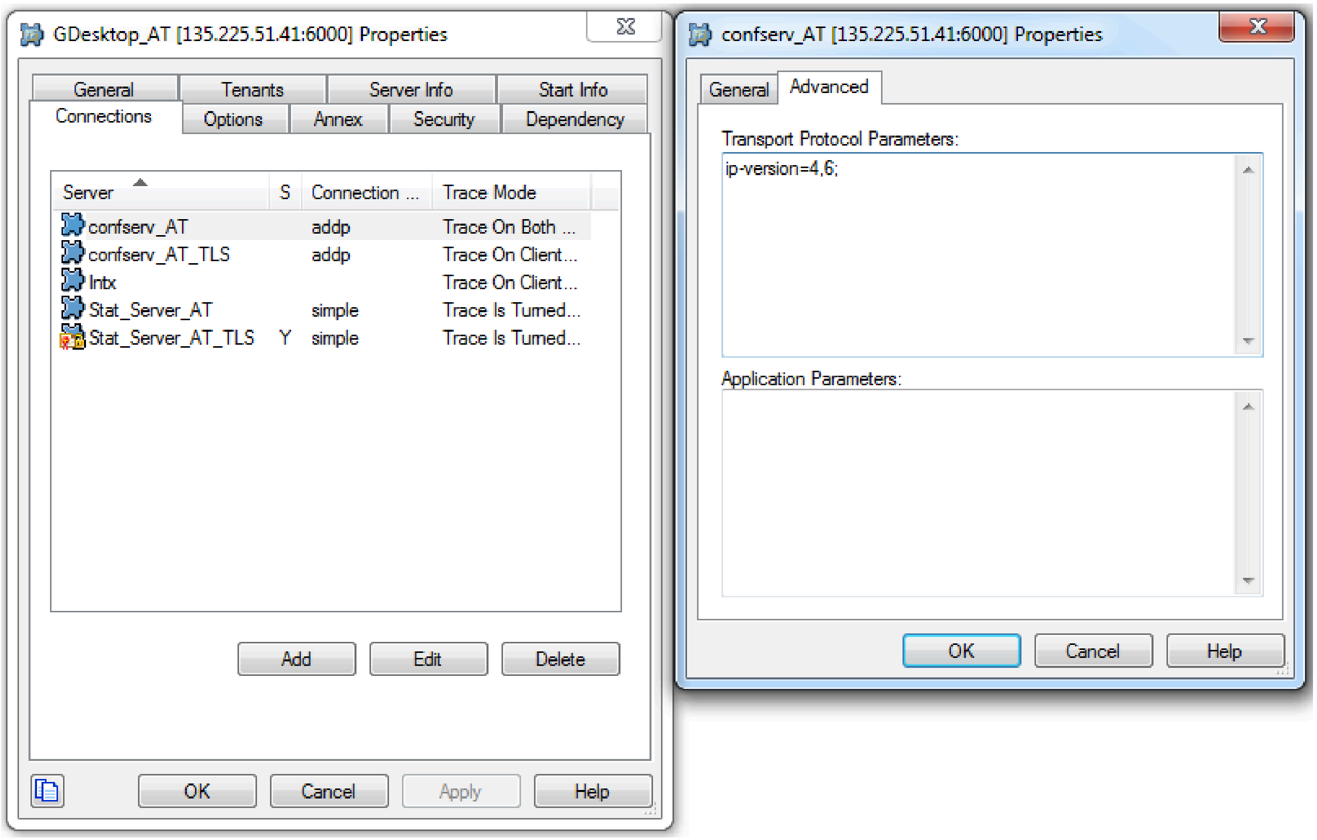
Section	Option	Description
common	enable-ipv6	Turns IPv6 support on/off. Possible values: 0 (default, implied) and 1. If set to 0, IPv6 support would be disabled, even if supported by OS/platform.

The `ip-version` constant specifies the order in which connection attempts will be made to IPv6 and IPv4 addresses. Possible values include the following strings:

- "4,6" (default)
- "6,4"

This setting has no effect on the connection if `enable-ipv6` is set to 0; warning would be logged. Has no effect on server side; warning would be logged.

Option values should be specified within the connection transport properties:



Server	S	Connection ...	Trace Mode
confserv_AT		addp	Trace On Both ...
confserv_AT_TLS		addp	Trace On Client...
Intx			Trace On Client...
Stat_Server_AT		simple	Trace Is Turned...
Stat_Server_AT_TLS	Y	simple	Trace Is Turned...

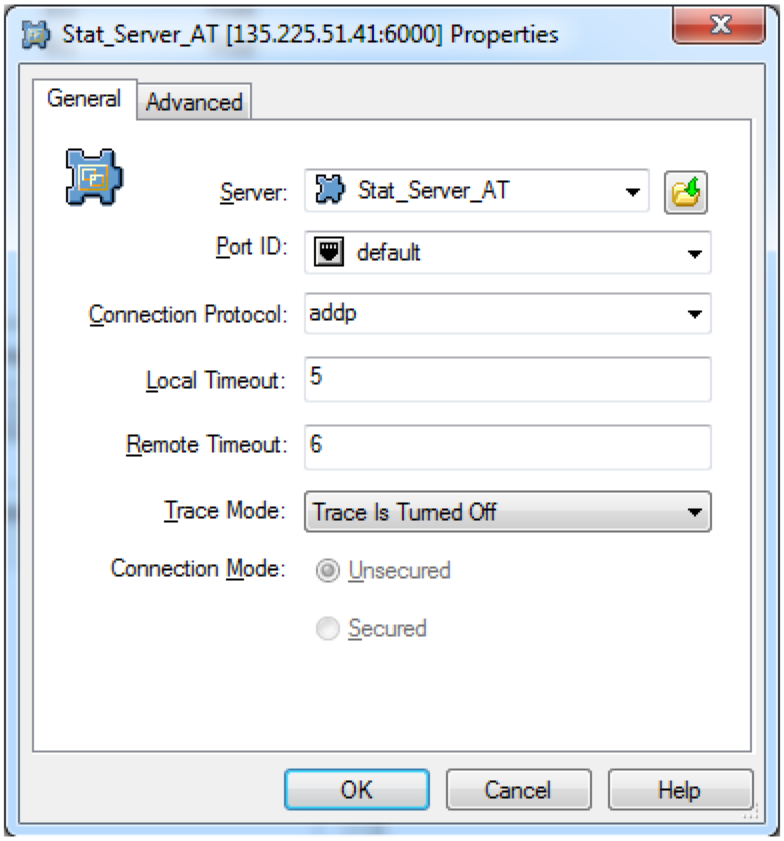
Transport Protocol Parameters:

```
ip-version=4,6;
```

Application Parameters:

ADDP Options

ADDP options, which can be handled with the Application Template helper, should be specified on the *Connections* tab.


Stat_Server_AT [135.225.51.41:6000] Properties

Server:	Stat_Server_AT
Port ID:	default
Connection Protocol:	addp
Local Timeout:	5
Remote Timeout:	6
Trace Mode:	Trace Is Turned Off
Connection Mode:	[✓ Unsecured] Secured

TLS Options

See the article on Configuring TLS Parameters in Configuration Manager for more information about TLS options.

Cluster Connection Configuration Helpers

This section describes Application Template helper methods that support the configuration of cluster connections to Genesys server, including use cases and samples.

Cluster Connection Configuration Types

There are two types of Genesys server cluster configuration available in Configuration Server: client-aligned configuration, and node-aligned configuration.

Both types provide information about addresses of the target servers in a cluster, and connection options used to communicate with them. You must choose the proper helper to use for each type of cluster configuration.

Client-aligned configuration methods:

- `ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(IGApplicationConfiguration appConfig, CfgAppType serverType)`
- `ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(IGApplicationConfiguration appConfig, IGAppConnConfiguration clusterConn, CfgAppType serverType)`

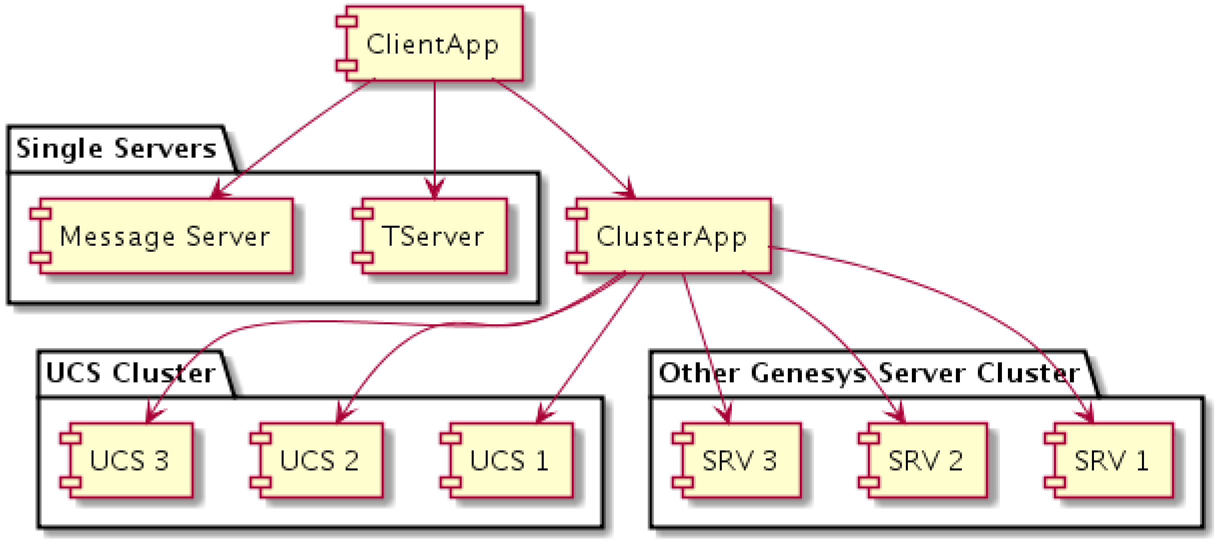
Node-aligned configuration methods:

- `ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(IConfService confService, IGApplicationConfiguration appConfig, CfgAppType serverType)`
- `ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(IConfService confService, IGApplicationConfiguration appConfig, IGAppConnConfiguration clusterConn, CfgAppType serverType)`

Client-Aligned Configuration Samples

Sample 1

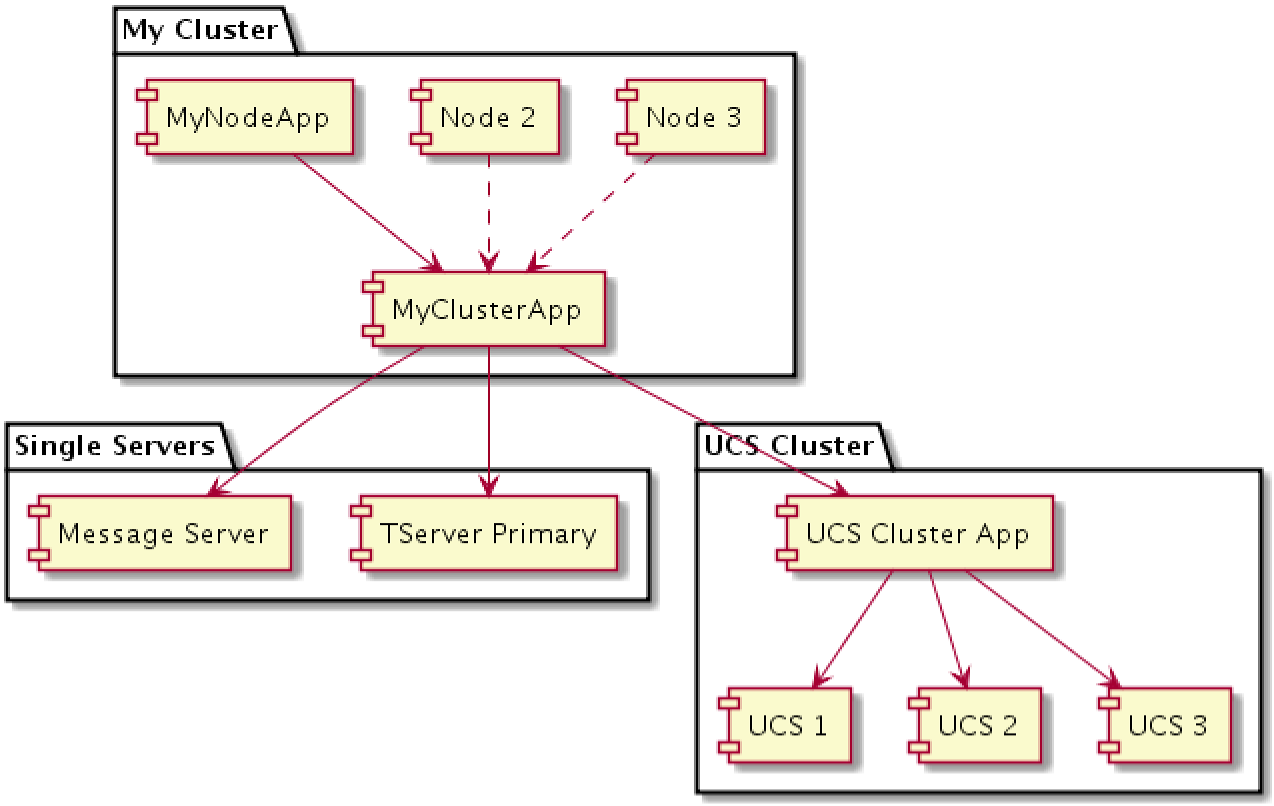
In the first sample scenario, a "ClientApp" Application is connected to the "ClusterApp'" virtual application (type=CFGApplicationCluster) that has connections to cluster nodes of one ore more clusters.



```
IClusterProtocol protocol ...
IList<WSConfig> nodesList =
ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(ClientApp,
CfgAppType.CFGContactServer);
protocol.SetNodes(nodesList);
```

Sample 2

In the second sample scenario, the "MyNodeApp" Application is a cluster node that is connected to the "MyClusterApp" virtual application. In this case, "MyClusterApp" is a shared store of connection configurations for all cluster nodes, which can have connections to other clusters like "UCS" as well as stand-alone servers.



```
IClusterProtocol protocol ...
IList<WSConfig> nodesList =
ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(MyClusterApp,
clusterAppConnection, CfgAppType.CFGContactServer);
protocol.SetNodes(nodesList);
```

Connection Options

You can use Genesys Administrator to define connection parameter options (such as ADDP, ip-version, strings encoding, or TCP socket options) that are used for all nodes in a cluster, or to define connection parameter for a particular node that will overriding common parameters.

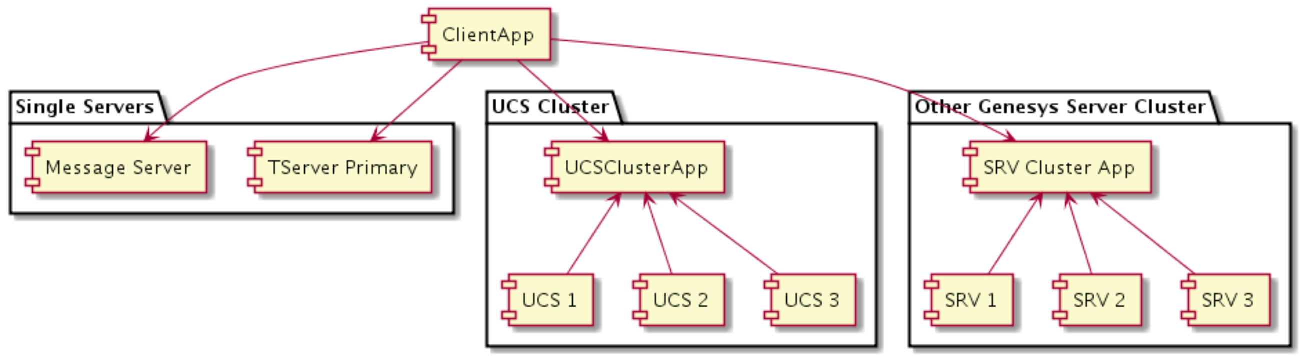
- Common Options: Specified in the connection from ClientApp to ClusterApp (or from OwnClusterApp if application is a node of some type of cluster).
- Node-Specific Options: can be specified in Configuration Manager in connection from ClusterApp to NodeApp.

See eServices Load Balancing Business Continuity for examples.

Node-Aligned Configuration

Sample 3

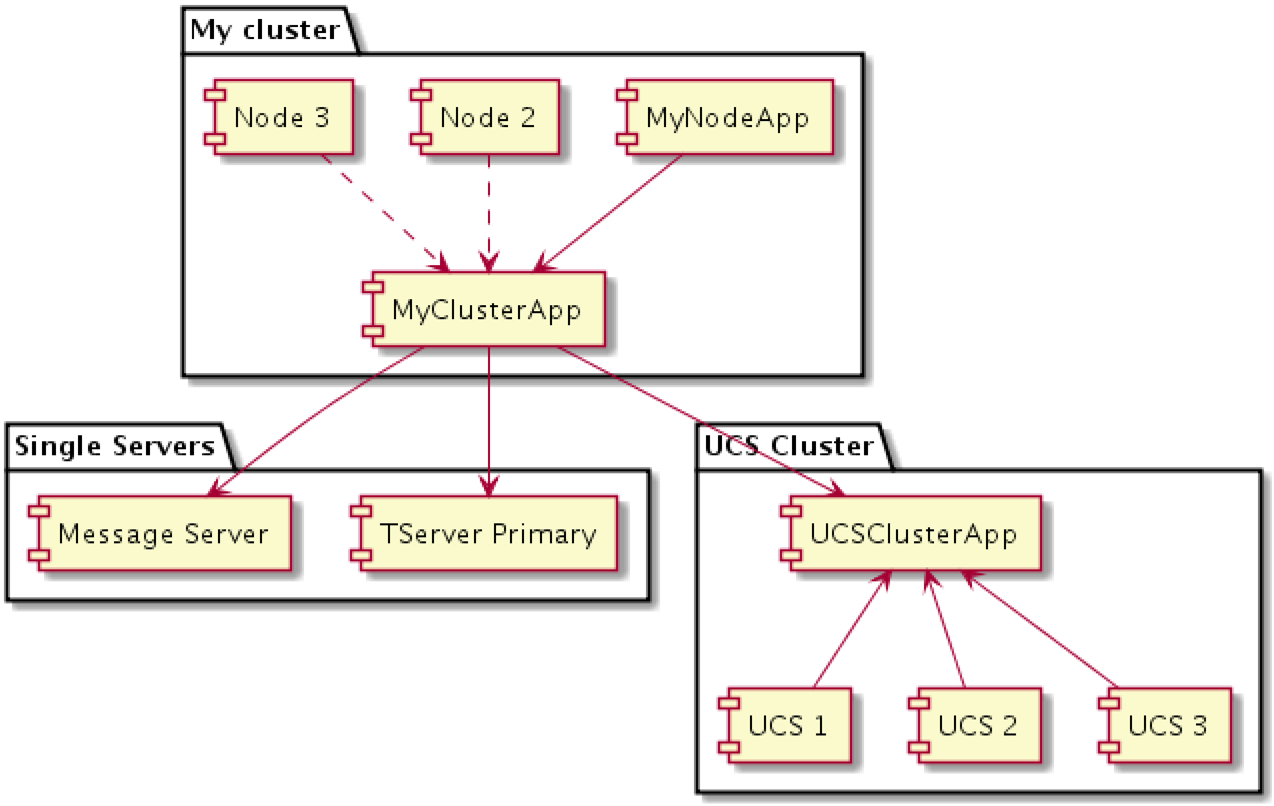
In the third sample scenario, the "ClientApp" Application is connected to the "UCSClusterApp" virtual application that groups UCS nodes. Unlike previous configurations, the cluster nodes in this scenario are also connected to the "UCSClusterApp" virtual application.



```
IClusterProtocol protocol ...
IList<WSConfig> nodesList =
ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(confService, ClientApp,
CfgAppType.CFGContactServer);
protocol.SetNodes(nodesList);
```

Sample 4

In the fourth sample scenario, the "MyNodeApp" Application is a cluster node. The "MyClusterApp" application has a connection to the UCS cluster ("UCSClusterApp" application). UCS nodes also have connections to "UCSClusterApp".



```
IClusterProtocol protocol ...
IList<WSConfig> nodesList =
ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(confService, MyNodeApp,
clusterAppConnection, CfgAppType.CFGContactServer);
protocol.SetNodes(nodesList);
```

Connection Options

You can use Genesys Administrator to define connection parameter options (such as ADDP, ip-version, strings encoding, or TCP socket options), but in this scenario this configuration is applied for all nodes in a cluster.

- Common Options: Specified in the connection from ClientApp to ClusterApp.

Client Connection Active Nodes Randomizer

Important

Dynamic updates to the cluster configuration are still possible with this approach. Your application should use the `shuffler.SetNodes(nodes);` function to change the list of cluster nodes, instead of using Cluster Protocol methods directly.

The randomizer helper component supports cluster load balancing from client applications, by allowing your applications to work with a server cluster without creating live connections to all of its nodes. This also prevents many clients from being stuck on a single connection, to resolve issues such as a single server overloading on startup, and can be of critical importance in use cases where the number of clients could be in the tens of thousands.

This helper takes a list of cluster nodes, and performs periodical endpoint rotation. The existing Cluster Protocol functionality to update its nodes configuration dynamically..

For each new node that is added:

1. a new instance of the related PSDK protocol class is created
2. asynchronous open is started
3. after the node has connected to its server, the node is added to load balancer

For each node that is removed:

1. the removed node is immediately excluded from the load balancer
2. the node is scheduled to be closed after its protocol timeout delay (which allows responses to be delivered for in-progress requests)

The randomizer uses the `Collections.shuffle(nodes)` method to randomize the sequence of given nodes.When updating the protocol node configuration, the randomizer uses a round-robin rotation over a whole list of randomly-sorted cluster nodes, choosing a subset of given number of elements. This allows the protocol to avoid breaking all connections at a single moment, and to stay online during switchover.

The randomizer component also has an optional ability to attach truncated nodes' endpoints as backup endpoints for selected ones. This allows the internal WarmStandby service to quickly switchover from an unresponsive node.

Sample usage of this component could look like the following:

```
IList<WSConfig> nodes = ...;
UcsClusterProtocol ucsNProtocol = new UcsClusterProtocolBuilder().Build();

var shuffler = new ClusterNodesShuffler(ucsNProtocol, 2); // creates nodes shuffler for 2
active connections
shuffler.UseBackups = true;// - lets the shuffler to use truncated nodes endpoints as backup
endpoints for the selected ones
try {
    shuffler.SetNodes(nodes); // - initializes shuffler with the whole list of cluster nodes.
    ucsNProtocol.Open();
    shuffler.StartTimer(3000, 3000); // - schedules shuffling operation with 3 secs delay and
3 secs period

    // Do the business logic on the cluster protocol...
    // In case of update in the cluster configuration, application should use
'shuffler.SetNodes(newNodes)'
    //     instead of ClusterProtocol's methods related to nodes configuration.
} finally {
    shuffler.StopTimer();
    ucsNProtocol.Close();
}
```

Code Samples

Simple Client Application Connecting to Any UCS Cluster

This sample checks the connection configuration for "WCC mode" UCS 9.0 cluster, then for "WDE mode" UCS cluster, and then for "legacy mode" connection to UCS (pri/bck) server.

```
 // Take "my application configuration" from context, or read it in a way like this:
ConfServerProtocol cfgProtocol=new ConfServerProtocol(new Endpoint(host, port));
IConfService confService=ConfServiceFactory.CreateConfService(cfgProtocol);
IGApplicationConfiguration myApp = new GCOMApplicationConfiguration(
  new CfgApplicationQuery(confService){Name = "myAppName"}.ExecuteSingleResult());

// For the first, try UCS 9 connection cluster:
IList<WSConfig> conns = ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(
        confService, myApp, CfgAppType.CFGContactServer);
if ((conns == null) || (conns.Count==0)) {
    // If there is no UCS 9 cluster connected, then we try older UCS cluster, or simple UCS
connection(s):
    conns = ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(
            myApp, CfgAppType.CFGContactServer);
}

Console.WriteLine("Connections: " + conns);
```

WCC-Based Cluster Node Application Connecting to Any UCS Cluster

This sample works in context of WCC.

```
var cfgProtocol=new ConfServerProtocol(new Endpoint(host, port));
var confService=ConfServiceFactory.CreateConfService(cfgProtocol);
// Take "my application configuration" from context, or read it in a way like this:
IGApplicationConfiguration myApp = new GCOMApplicationConfiguration(
        new CfgApplicationQuery(confService){Name = "myAppName"}.ExecuteSingleResult());

IGApplicationConfiguration myClusterApp = null;
// if we do not have 'myClusterApp' from WCC context, we may take it by this way:
IList<IGAppConnConfiguration> clusters =
GApplicationConfiguration.GetAppServers(myApp.AppServers, CfgAppType.CFGApplicationCluster);
if (clusters != null) {
    if (clusters.Count == 1) {
        myClusterApp = clusters[0].TargetServerConfiguration;
        log.InfoFormat("Application is recognized as a node of cluster
''{0}''",myClusterApp.ApplicationName);
    } else if (clusters.Count > 1) {
        log.Error("Application has more than one application cluster connected - its treated
as a standalone app");
    }
}

// Select application cluster connection start point:
IGApplicationConfiguration connSrc =  myClusterApp ?? myApp;

// For the first, try UCS 9 connection cluster:
IList<WSConfig> conns = ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(
        confService, connSrc, CfgAppType.CFGContactServer);
if (conns == null || conns.Count==0) {
    // If there is no UCS 9 cluster connected, then we try older UCS cluster, or simple UCS
connection(s):
    conns = ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(
            connSrc, CfgAppType.CFGContactServer);
}

Console.WriteLine("Connections: " + conns);
```

Client Nodes Randomizer Usage Sample

Application code using the randomizer component may look like the following sample:

```
 IList<WSConfig> nodes = ...;
 UcsClusterProtocol ucsNProtocol = new UcsClusterProtocolBuilder().Build();
 ucsNProtocol.Timeout = TimeSpan.FromSeconds(5); // sets protocol timeout to 5 secs
 ucsNProtocol.ClientName = "MyClientName";
 ucsNProtocol.ClientApplicationType = "MyAppType";

 var shuffler = new ClusterNodesShuffler(ucsNProtocol, 2); // creates nodes shuffler for 2
active connections

 try {
     shuffler.SetNodes(nodes); // - initializes shuffler with the whole list of cluster nodes.
     ucsNProtocol.Open();
     shuffler.StartTimer(3000, 3000); // - schedules shuffling operation with 3 secs delay
and 3 secs period

     // do the business logic on the cluster protocol...
     for (int i = 0; i < 200; i++)
     {
       var resp = ucsNProtocol.Request(RequestGetVersion.Create()) as EventGetVersion;
       if (resp!=null) Console.WriteLine("Resp from: " + resp.Endpoint);
       Thread.Sleep(300);
     }
 } finally {
     shuffler.StopTimer();
     ucsNProtocol.Close();
 }
```

Handling Updates From Config Server

The `GFApplicationConfigurationManager` component monitors Config Server for updates and provides notifications about changes in applications.

You should register for updates at `GFApplicationConfigurationManager`.

In the handle (`GFAppCfgEvent` event) method implementation, create a new connection configuration using one of the helpers mentioned above:

```
using Genesyslab.Platform.AppTemplate.Application;

UcsClusterProtocol ucsProtocol = ...;
GFApplicationConfigurationManager appManager = ...;
appManager.Register(@event =>
{
  //get new application configuration
  IGApplicationConfiguration appconfig = @event.AppConfig;

  //create protocol config
  var wsconfig = ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(appconfig,
CfgAppType.CFGContactServer);
```

```
  //apply new set of protocol endpoints
  ucsProtocol.SetNodes(wsconfig );

});
appManager.Init();
```

Important

WCC-based cluster configuration does not support an update handler at this time. Subscribing to updates in this case will lead to Config Server overloading, so customers are encouraged to make direct requests to Config Server to actualize the cluster configuration before opening `ClusterProtocol`.

Using the Cluster Protocol Application Block

Java

This Application Block is designed to be used with applications where a large number of requests should be spread between a configured set of UCS servers - or other Genesys servers - in a cluster, providing a type of high-availability (HA) connection to that cluster. In addition to this application block, the Cluster Protocol also includes a set of configuration helpers in the Application Template Application Block.

Tip

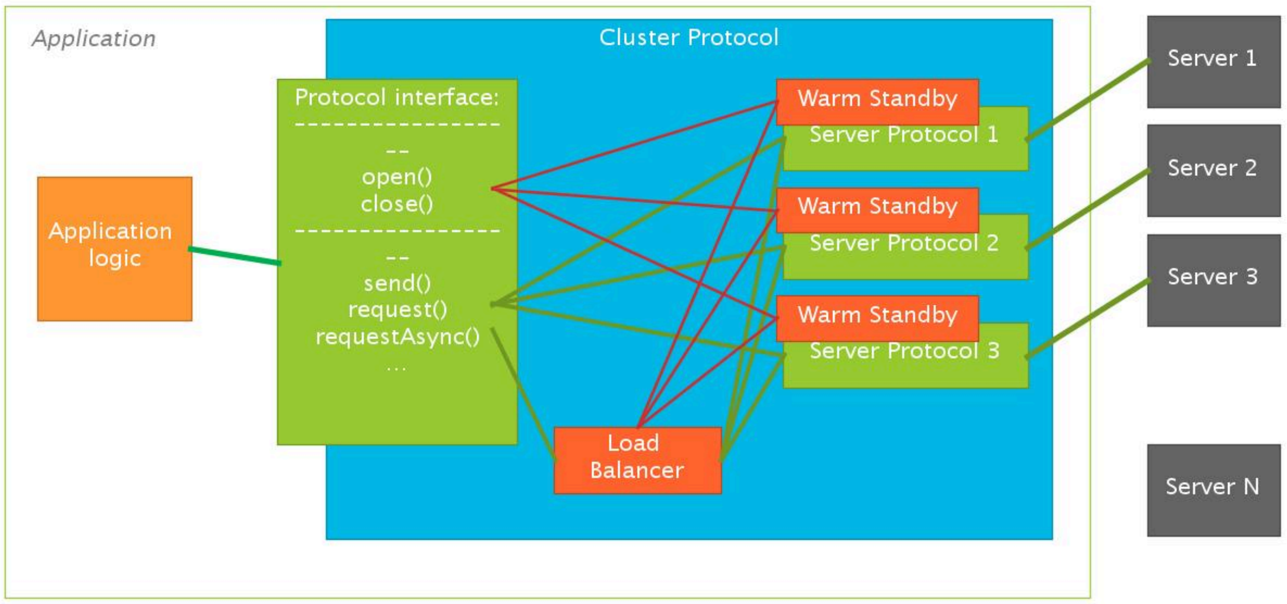
When the Cluster Protocol Application Block is connected to a cluster, it can be used in UCS9 N+1 server mode connected to all nodes, or in UCS9 N+1 client mode connected to one node. When the application block is connected to a single application that has a backup configured, it works in UCS8 mode Primary/Backup.

Architecture Overview

One of the simplest and most common uses of the Platform SDK interface is a protocol that interacts directly with a Genesys server using a set of standard protocol methods (such as Open, Close, Send, Request).

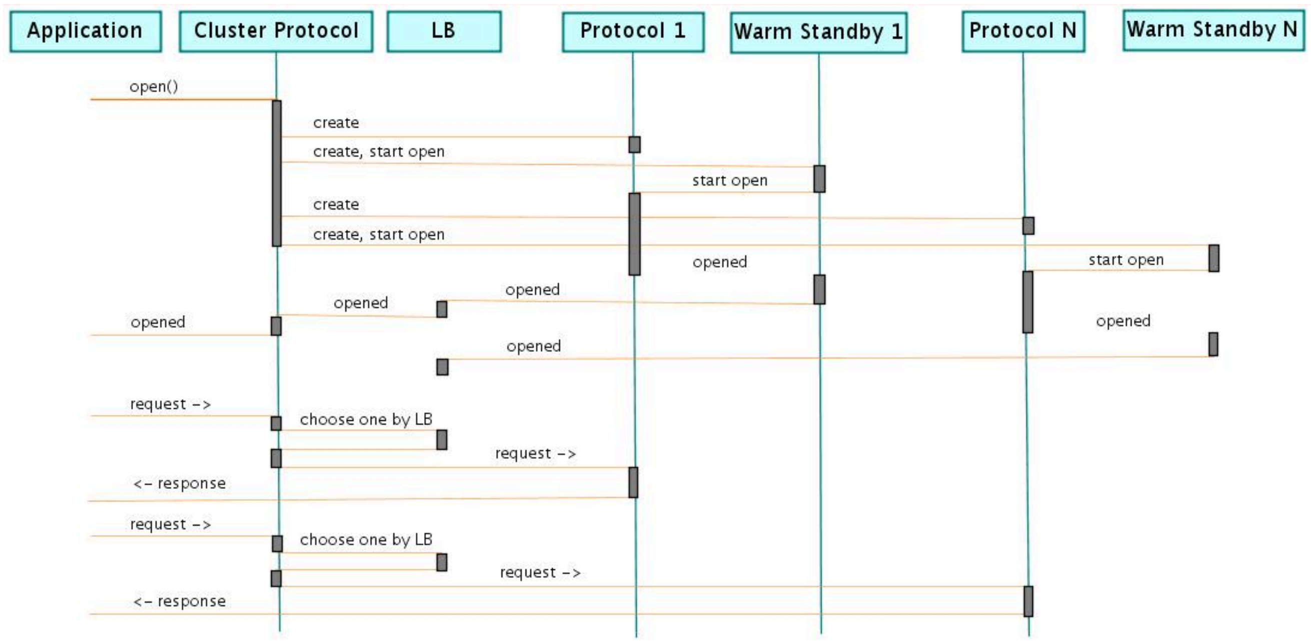
To provide a single working protocol with at least one backup, we use the Warm Standby Application Block. That application block intercepts control of the protocol interface and provides switch-over between multiple protocols, or protocol restoration, using a single Warm Standby endpoint.

The Cluster Protocol builds on this idea, allowing you to work simultaneously with a series of protocols and Warm Standby application blocks (each of which can represent one or many individual protocols) the same way that you would with a single, standard protocol. To configure the Cluster Protocol, you use a mixed list of protocol and Warm Standby endpoints gathered from Configuration Server.



The Cluster Protocol itself covers any scenarios where we need simultaneous connections through a load balancer to many servers, where each server may have one or many backups.

Sequence Diagram: Main Scenario When Using Cluster Protocol Application Block



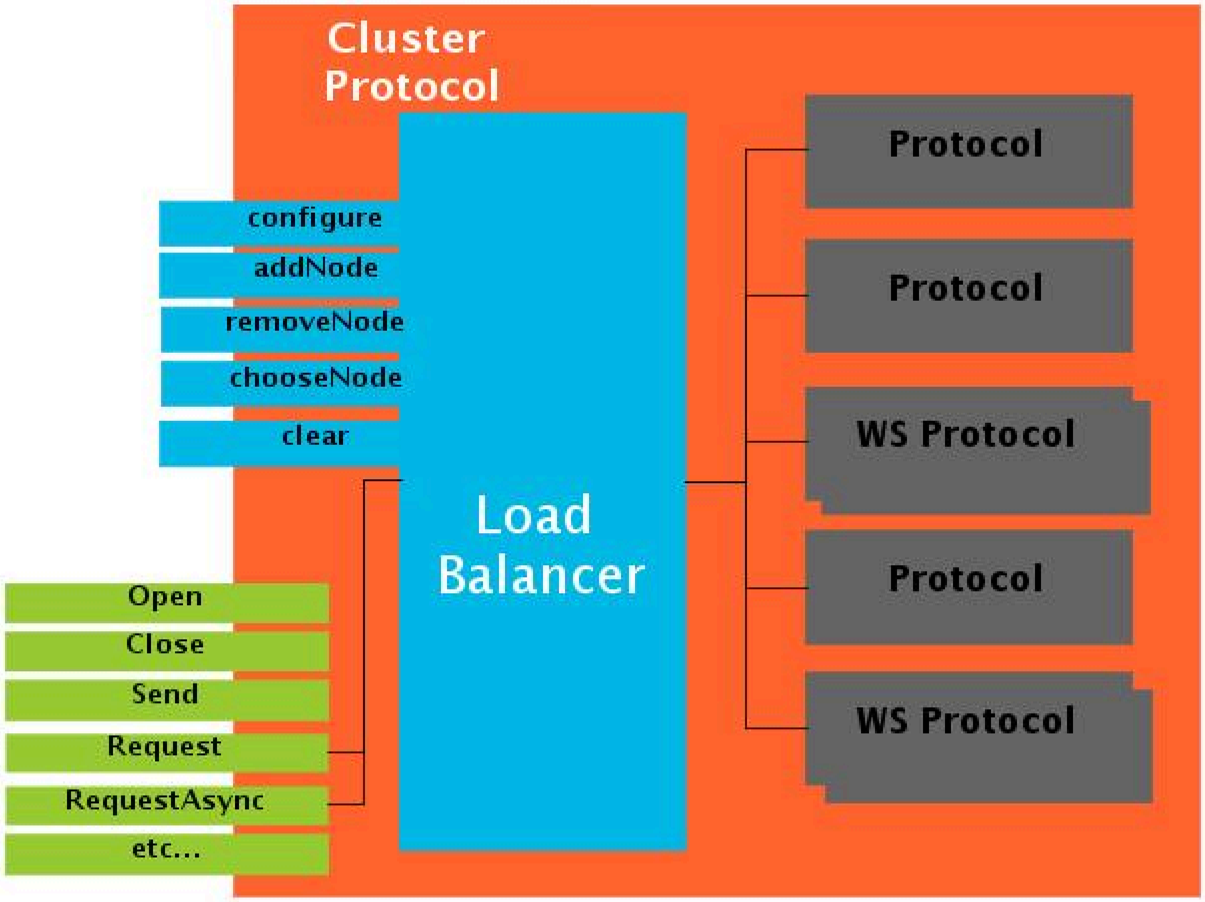
High Availability Options

The Cluster Protocol is able to control the state of protocol connections, and use the Warm Standby Application Block to keep them opened. Each cluster node may be initialized as a standard Platform SDK Endpoint, or using `WSConfig` (Warm Standby configuration) with one or more backup Endpoints added. A list of opened connections is tracked using a Load Balancer, which allows requests to be spread across connected protocols.

This gives your application the flexibility to configure one endpoint per connection ("server mode"), or to combine cluster endpoints in a single Warm Standby configuration and let the Cluster Protocol use a single connection to anyone of them ("client mode").

Load Balancing Options

The default Load Balancing strategy uses a "round robin" algorithm over connected servers for request forwarding. It is possible for you to create your own implementation of the Load Balancer interface, however, and provide it during creation of the Cluster Protocol instance.



Implementing a Custom LoadBalancer

```
final EspClusterProtocol haProtocol =
        new EspClusterProtocolBuilder()
                .withLoadBalancer(new MyLoadBalancer())
                .build();

protected class MyLoadBalancer implements ClusterProtocolLoadBalancer {
        @Override
        public void configure(final ConnectionConfiguration config) {
        }
        @Override
        public void addNode(final Protocol node) {
        }
        @Override
        public void removeNode(final Protocol node) {
        }
        @Override
        public Protocol chooseNode(final Message request) {
        }
        @Override
        public void clear() {
        }
}
```

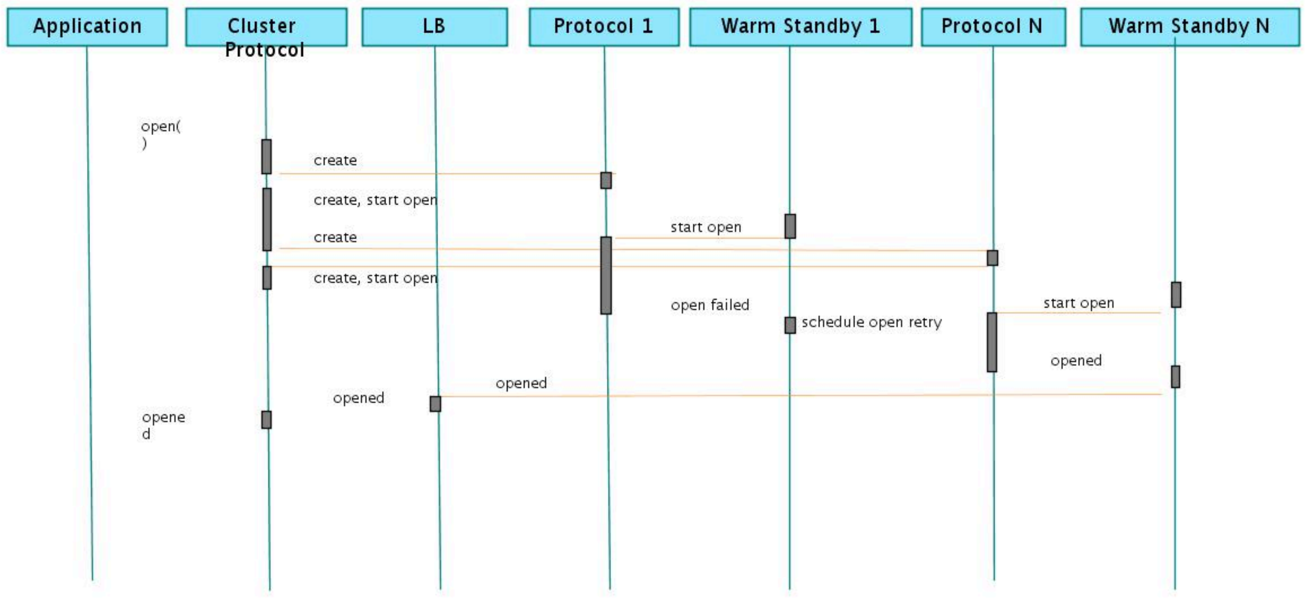
Disaster Recovery

Platform SDK does not inject any business functionality into connections, so the Cluster Protocol Application Block is able to provide disaster recovery by meeting the following requirements:

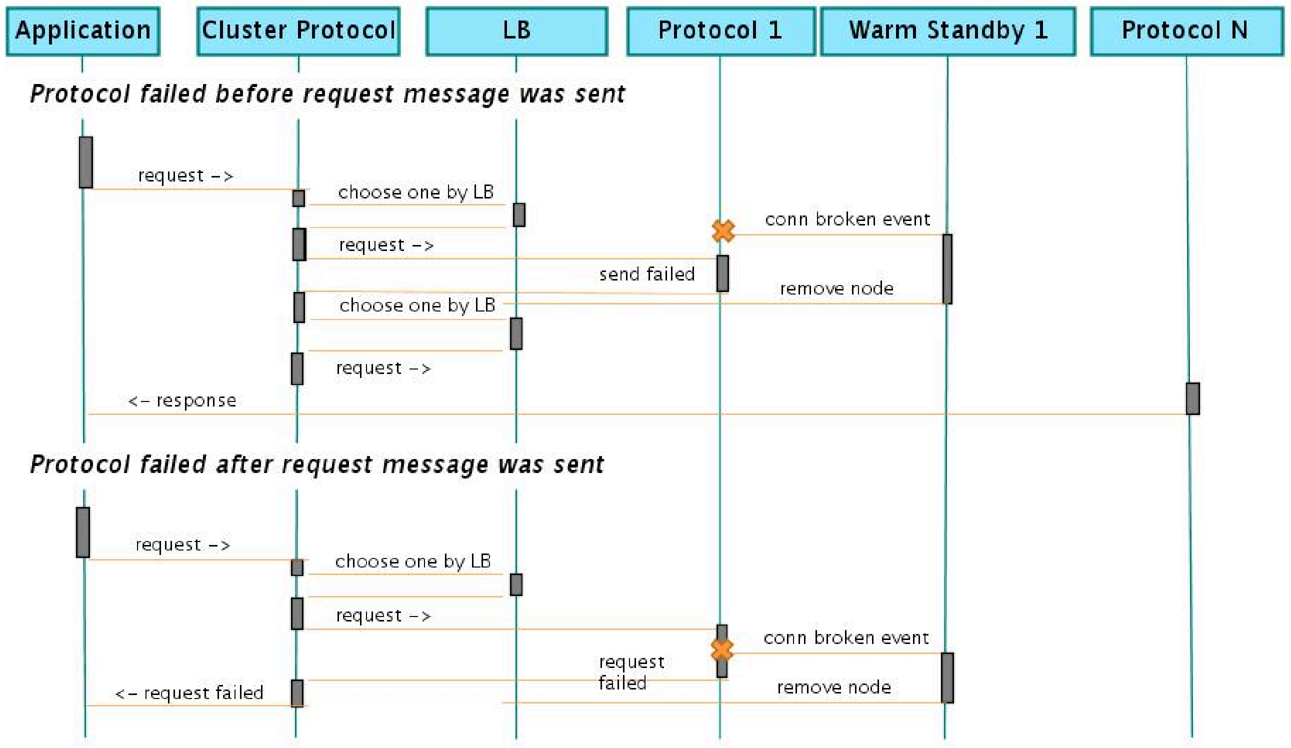
1. High availability maintains a list of active connections to ensure that requests are not sent to disconnected servers.
2. Any request that receives a `ChannelClosedOnSendException` or `ChannelClosedOnRequestException` response (because the connection was broken but not yet removed from the active list) is automatically resent to a different connection. Other exceptions are passed through to the client application to be handled manually.

There is no special configuration required to enable disaster recovery, but your application will need to include logic that handles generic IO exceptions or protocol timeout exceptions.

Sequence Diagram: Cluster Protocol with Node Connection Failure



Sequence Diagram: Cluster Protocol User Request Failure



How to Handle Lost Requests

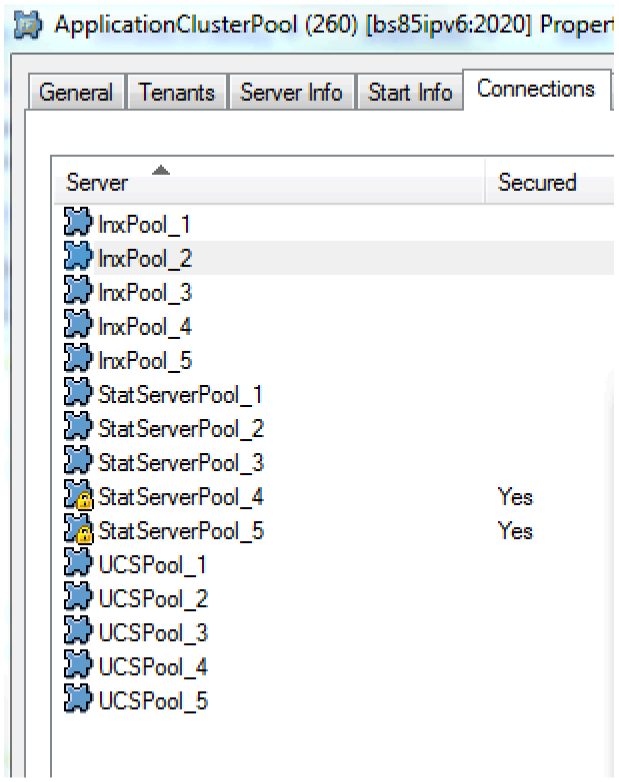
It is possible for a communication error to occur where your application sends a request but the connection is broken before any response is received. When using the Cluster Protocol you may not know if another node in the cluster handled the request. In this scenario your application won't know if the server was able to receive or process the request correctly.

To address this, your application should include business logic that handles exceptions or null returns for a Cluster Protocol request and acts appropriately based on the type of request. For example, a request to read data can be sent again without impact, while a request that modifies data on the server may require your application to check the server state before retrying or providing notification that the request was successful.

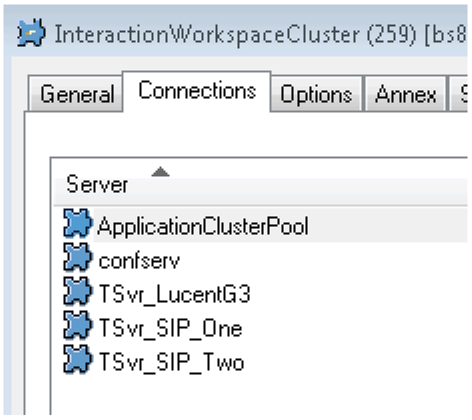
Configuration Options

Important
For this release, only the UCS cluster type is supported.

The Cluster Protocol Application Block supports configuration in Configuration Manager for the client application and cluster.



Server	Secured
InxPool_1	
InxPool_2	
InxPool_3	
InxPool_4	
InxPool_5	
StatServerPool_1	
StatServerPool_2	
StatServerPool_3	
StatServerPool_4	Yes
StatServerPool_5	Yes
UCSPool_1	
UCSPool_2	
UCSPool_3	
UCSPool_4	
UCSPool_5	



Server
ApplicationClusterPool
confserv
TSvr_LucentG3
TSvr_SIP_One
TSvr_SIP_Two

Dynamic Configuration Options Updates

Cluster Protocol objects support graceful, dynamic updates of the cluster configuration, allowing you to add or remove nodes on an *opened* and *actively used* cluster protocol.

Adding a new node automatically creates the new node protocol connection in the background and attempts an asynchronous opening. The load balancer is notified about the new node connection immediately after it is connected.

Removing a connected node from the cluster protocol causes the protocol to exclude that node from the load balancer, and then disconnect the node after the protocol timeout delay. This delay allows for delivery of responses on any requests that were already started.

Code Examples

Cluster Protocol Usage Example

```
UcsClusterProtocol ucsNProtocol =
        new UcsClusterProtocolBuilder()
                .build();
ucsNProtocol.setClientName("MyClientName");
ucsNProtocol.setClientApplicationType("MyAppType");
ucsNProtocol.setNodesEndpoints(
        new Endpoint("ucs1", UCS_1_HOST, UCS_1_PORT),
        new Endpoint("ucs2", UCS_2_HOST, UCS_2_PORT),
        new Endpoint("ucs3", UCS_3_HOST, UCS_3_PORT));
ucsNProtocol.open();

EventGetVersion resp1 = (EventGetVersion) ucsNProtocol.request(RequestGetVersion.create());
EventGetVersion resp2 = (EventGetVersion) ucsNProtocol.request(RequestGetVersion.create());
```

Configuration Helper Example

The `ClusterClientConfigurationHelper` class is designed to make it easier for your application to make use of this Application Block by performing the following steps:

1. Checks if client application is connected to cluster application
2. If cluster application detected, then creates endpoints for all cluster connections of the specify type
3. If cluster application not detected or no connections in cluster application found, then creates endpoints for all client application connections of the specify type (compatibility mode)
4. If connected server has backup application, endpoint will have classical Primary\Backup configuration
5. Supports specifying shared ADDP options, Transport and Application parameters in connection to cluster application. Those parameters can be overridden in connection to particular cluster node.

For more information, including an overview of the new Cluster Connection Configuration Helpers, see Using the Application Template Application Block.

.NET

This Application Block is designed to be used with applications where a large number of requests should be spread between a configured set of UCS servers - or other Genesys servers - in a cluster, providing a type of high-availability (HA) connection to that cluster. In addition to this application block, the Cluster Protocol also includes a set of configuration helpers in the Application Template Application Block.

Tip

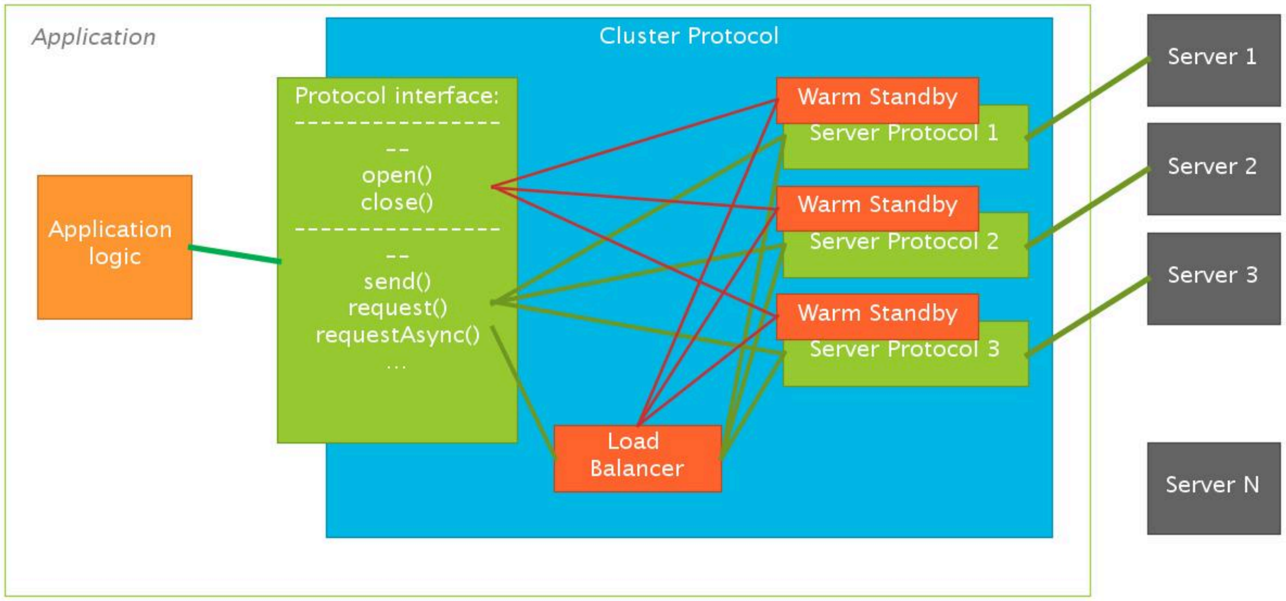
When the Cluster Protocol Application Block is connected to a cluster, it can be used in UCS9 N+1 server mode connected to all nodes, or in UCS9 N+1 client mode connected to one node. When the application block is connected to a single application that has a backup configured, it works in UCS8 mode Primary/Backup.

Architecture Overview

One of the simplest and most common uses of the Platform SDK interface is a protocol that interacts directly with a Genesys server using a set of standard protocol methods (such as Open, Close, Send, Request).

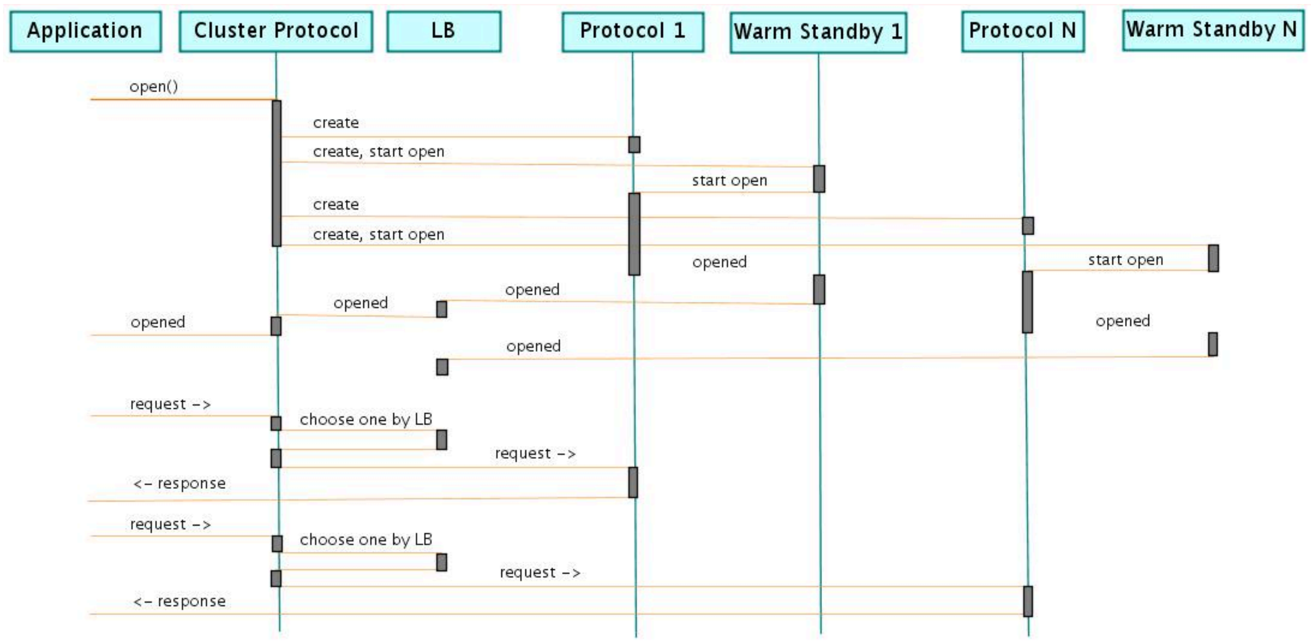
To provide a single working protocol with at least one backup, we use the Warm Standby Application Block. That application block intercepts control of the protocol interface and provides switch-over between multiple protocols, or protocol restoration, using a single Warm Standby endpoint.

The Cluster Protocol builds on this idea, allowing you to work simultaneously with a series of protocols and Warm Standby application blocks (each of which can represent one or many individual protocols) the same way that you would with a single, standard protocol. To configure the Cluster Protocol, you use a mixed list of protocol and Warm Standby endpoints gathered from Configuration Server.



The Cluster Protocol itself covers any scenarios where we need simultaneous connections through a load balancer to many servers, where each server may have one or many backups.

Sequence Diagram: Main Scenario When Using Cluster Protocol Application Block



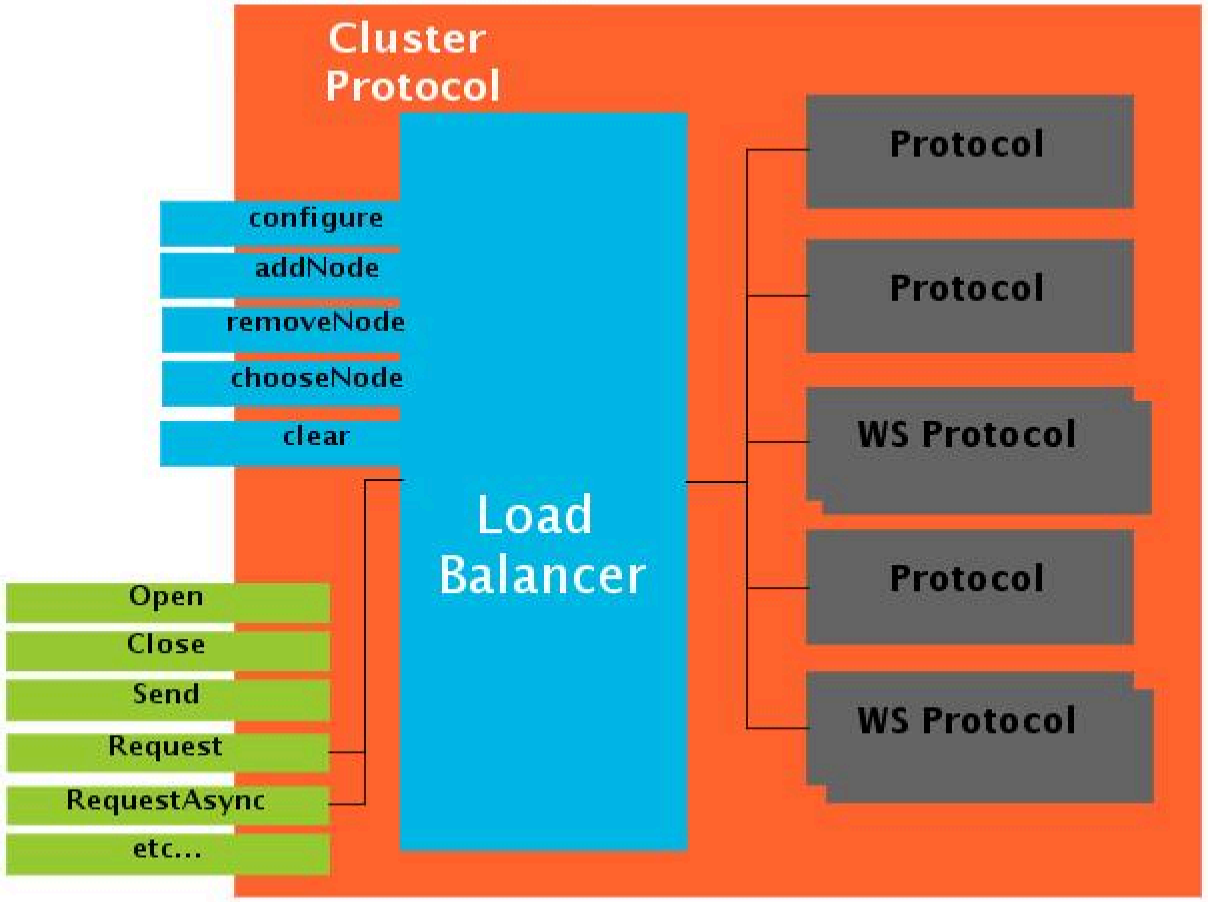
High Availability Options

The Cluster Protocol is able to control the state of protocol connections, and use the Warm Standby Application Block to keep them opened. Each cluster node may be initialized as a standard Platform SDK Endpoint, or using `WSConfig` (Warm Standby configuration) with one or more backup Endpoints added. A list of opened connections is tracked using a Load Balancer, which allows requests to be spread across connected protocols.

This gives your application the flexibility to configure one endpoint per connection ("server mode"), or to combine cluster endpoints in a single Warm Standby configuration and let the Cluster Protocol use a single connection to anyone of them ("client mode").

Load Balancing Options

The default Load Balancing strategy uses a "round robin" algorithm over connected servers for request forwarding. It is possible for you to create your own implementation of the Load Balancer interface, however, and provide it during creation of the Cluster Protocol instance.



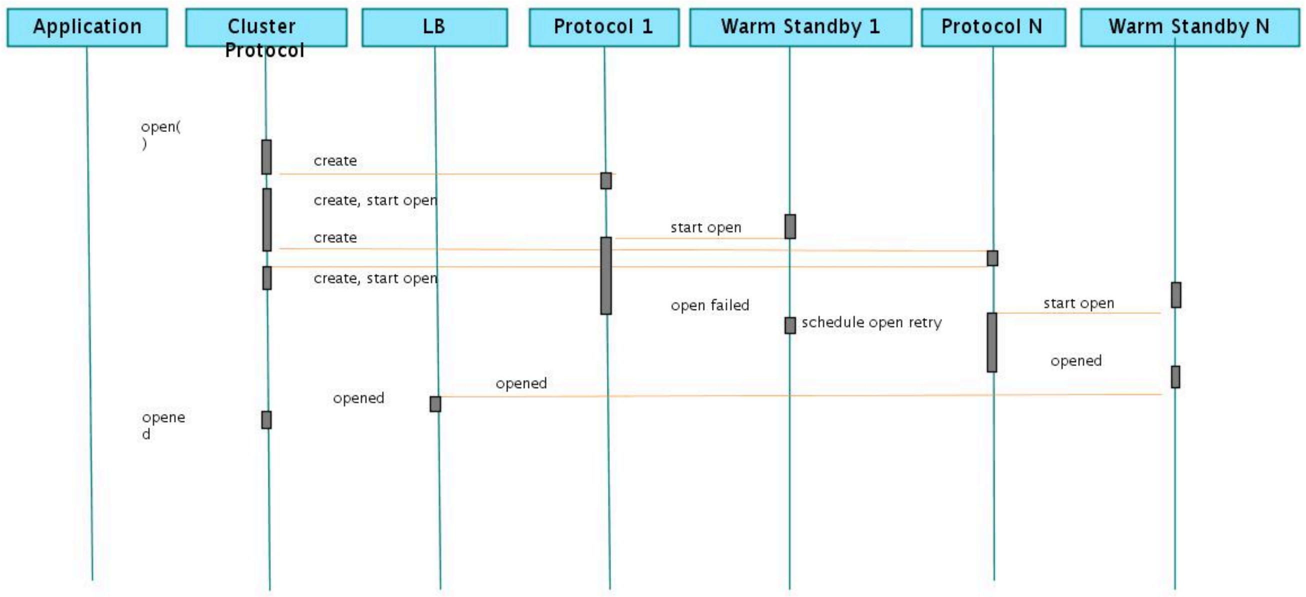
Disaster Recovery

Platform SDK does not inject any business functionality into connections, so the Cluster Protocol Application Block is able to provide disaster recovery by meeting the following requirements:

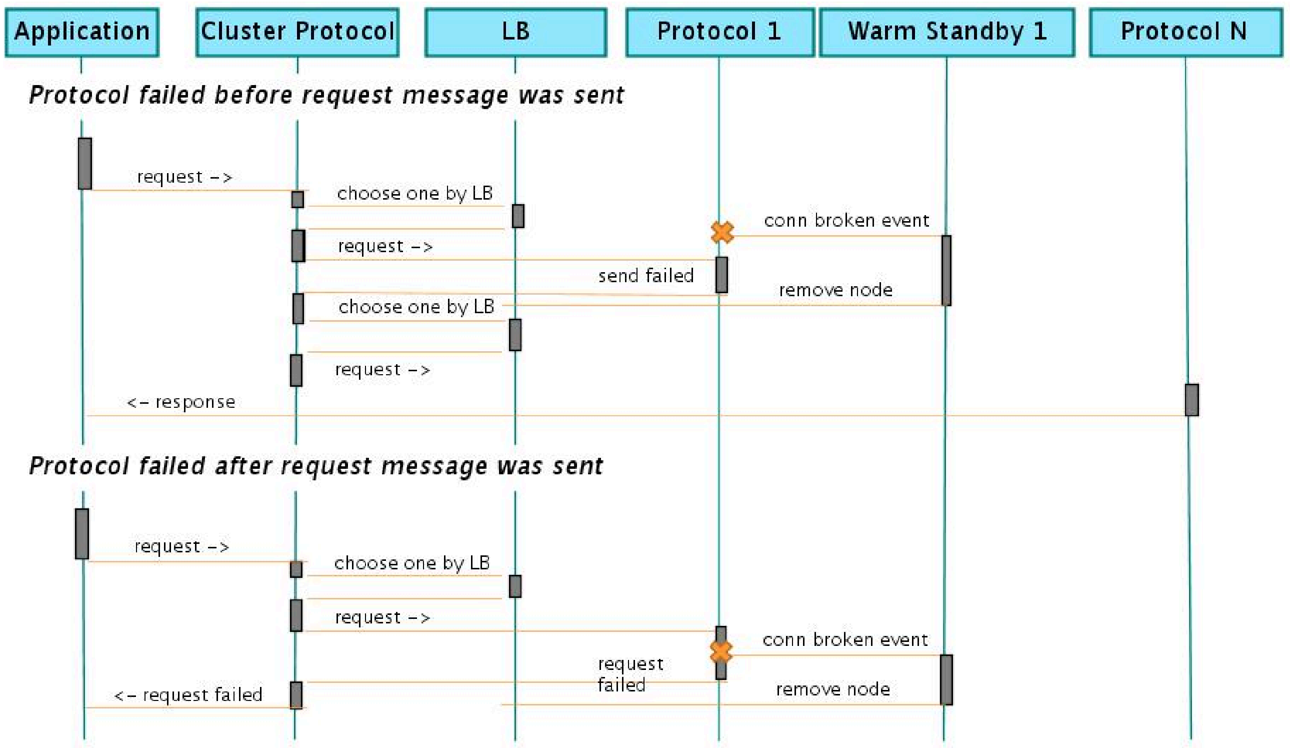
1. High availability maintains a list of active connections to ensure that requests are not sent to disconnected servers.
2. Any request that receives a `ChannelClosedOnSendException` or `ChannelClosedOnRequestException` response (because the connection was broken but not yet removed from the active list) is automatically resent to a different connection. Other exceptions are passed through to the client application to be handled manually.

There is no special configuration required to enable disaster recovery, but your application will need to include logic that handles generic IO exceptions or protocol timeout exceptions.

Sequence Diagram: Cluster Protocol with Node Connection Failure



Sequence Diagram: Cluster Protocol User Request Failure



How to Handle Lost Requests

It is possible for a communication error to occur where your application sends a request but the connection is broken before any response is received. When using the Cluster Protocol you may not know if another node in the cluster handled the request. In this scenario your application won't know if the server was able to receive or process the request correctly.

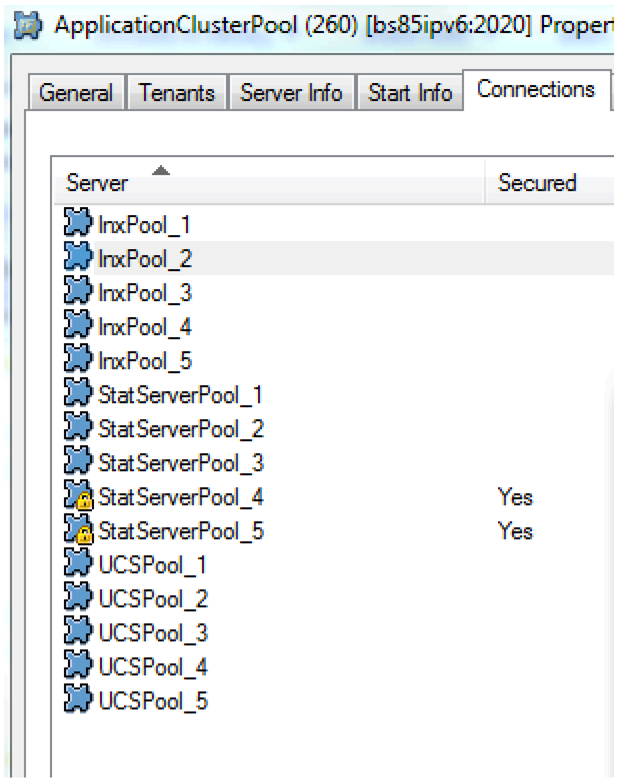
To address this, your application should include business logic that handles exceptions or null returns for a Cluster Protocol request and acts appropriately based on the type of request. For example, a request to read data can be sent again without impact, while a request that modifies data on the server may require your application to check the server state before retrying or providing notification that the request was successful.

Configuration Options

Important

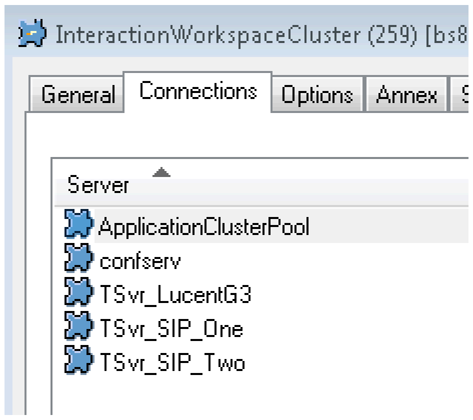
For this release, only the UCS cluster type is supported.

The Cluster Protocol Application Block supports configuration in Configuration Manager for the client application and cluster.


ApplicationClusterPool (260) [bs85ipv6:2020] Proper

General | Tenants | Server Info | Start Info | Connections

Server	Secured
InxPool_1	
InxPool_2	
InxPool_3	
InxPool_4	
InxPool_5	
StatServerPool_1	
StatServerPool_2	
StatServerPool_3	
StatServerPool_4	Yes
StatServerPool_5	Yes
UCSPool_1	
UCSPool_2	
UCSPool_3	
UCSPool_4	
UCSPool_5	


InteractionWorkspaceCluster (259) [bs8

General | Connections | Options | Annex

Server
ApplicationClusterPool
confserv
TSvr_LucentG3
TSvr_SIP_One
TSvr_SIP_Two

Dynamic Configuration Options Updates

Cluster Protocol objects support graceful, dynamic updates of the cluster configuration, allowing you to add or remove nodes on an *opened* and *actively used* cluster protocol.

Adding a new node automatically creates the new node protocol connection in the background and attempts an asynchronous opening. The load balancer is notified about the new node connection immediately after it is connected.

Removing a connected node from the cluster protocol causes the protocol to exclude that node from the load balancer, and then disconnect the node after the protocol timeout delay. This delay allows for delivery of responses on any requests that were already started.

Code Examples

Cluster Protocol Usage Example

```
var ucsNProtocol = new UcsClusterProtocolBuilder().Build();
ucsNProtocol.ClientName = "MyClientName";
ucsNProtocol.ClientApplicationType = "MyAppType";
ucsNProtocol.SetNodesEndpoints(
        new Endpoint("ucs1", UCS_1_HOST, UCS_1_PORT),
        new Endpoint("ucs2", UCS_2_HOST, UCS_2_PORT),
        new Endpoint("ucs3", UCS_3_HOST, UCS_3_PORT));
ucsNProtocol.Open();

EventGetVersion resp1 = (EventGetVersion)ucsNProtocol.Request(RequestGetVersion.Create());
EventGetVersion resp2 = (EventGetVersion)ucsNProtocol.Request(RequestGetVersion.Create());
```

Configuration Helper Class

The `ClusterClientConfigurationHelper` class is designed to make it easier for your application to make use of this Application Block by performing the following steps:

1. Checks if client application is connected to cluster application
2. If cluster application detected, then creates endpoints for all cluster connections of the specify type
3. If cluster application not detected or no connections in cluster application found, then creates endpoints for all client application connections of the specify type (compatibility mode)
4. If connected server has backup application, endpoint will have classical Primary\Backup configuration
5. Supports specifying shared ADDP options, Transport and Application parameters in connection to cluster application. Those parameters can be overridden in connection to particular cluster node.

For more information, including an overview of the new Cluster Connection Configuration Helpers, see Using the Application Template Application Block.

Event Handling

Java

Once you have connected to a server, much of the work for your application involves sending messages to that server and handling the events you receive from the server. This article describes how to send and receive messages from a server.

Messages: Overview of Events and Requests

Messages you send to a server are called requests, while messages you receive are called events. An event that is received from a server as the result of executing a request is called a response. In summary, messages can be classified by using the following taxonomy:

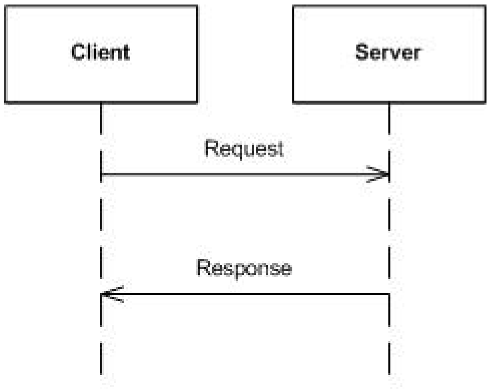
- Requests: sent to the server
- Events: received from the server
    - Responses: received as the result of a request
    - Unsolicited events: not a direct result of a request

Tip

On this page, we will use the more general term "message" instead of "event", in order to avoid confusion between protocol events and programming events.

For example, you may send a request to log in an agent or to gather statistics. You might also send a request to update a configuration object, or to shut down an application.

In each of these cases, the server will respond with an event message, as shown below.



You may also get unsolicited events from the server. That means receiving events that are not a response to a specific request. For example, `EventRinging` will notify you of a call ringing on an extension that you are currently monitoring.

Receiving Messages

With the Platform SDK, you can receive messages synchronously or asynchronously. It is important that you define the way your application will work in this aspect. In general, you will probably use only one type or the other in the same application.

Interactive applications normally use asynchronous message handling, because that will prevent the UI thread from being blocked, which could make the application appear "frozen" to a user. On the other hand, non-interactive batch applications commonly use synchronous response handling, as that allows writing easy code that performs step-by-step.

Receiving Messages Asynchronously

Most Platform SDK applications need to handle unsolicited events. This is particularly true for applications that monitor the status of contact center resources, such as extensions.

You receive server messages by implementing a `MessageHandler` that contains the event-handling logic:

```
[Java]

MessageHandler tserverMessageHandler = new MessageHandler() {
        @Override
        public void onMessage(Message message) {
                // your event-handling code goes here
        }
};
```

Then you set your implementation as the protocol `MessageHandler`.

```
[Java]

tserverProtocol.setMessageHandler(tserverMessageHandler);
```

Important

You need to know that your event-handling logic will be executed by using the protocol invoker. Please set the invoker appropriate for your application needs. For more information about the protocol invoker and how to set it, refer to Connecting to a Server.

Inside your event-handling code, you will want to execute different logic for different kinds of events. A typical way to do this is using a `switch` statement, based on the event identifier:

```
[Java]

switch (message.messageId()) {
        case EventAgentLogin.ID:
                OnEventAgentLogin(message);
                break;
        case EventAgentLogout.ID:
                OnEventAgentLogout(message);
                break;
}
```

Receiving Messages Synchronously

Some kinds of applications, such as batch applications, benefit from receiving messages synchronously. This means that received messages will queue up and be handled by the application on demand.

In order to receive messages this way, you simply **do not** set a protocol `MessageHandler` as described in the previous section.

Tip

For releases prior to Platform SDK 8.1.1, messages were received synchronously by default. Please note that 8.1.1 behavior is backwards-compatible, and pre-8.1.1 applications will continue to work as expected without any modification.

To receive a message synchronously, use the `Receive` method. This method blocks processing, waiting for the next message to be received before continuing. Take into account that the maximum time to wait is set by a configurable timeout value. If the timeout expires and no event is received, you will receive a `null` value.

```
[Java]

Message message = tserverProtocol.receive();
```

If you want to set your own timeout, you can use the `Receive` method overload that takes a timeout parameter. Otherwise, if you use `Receive` with no parameters, the protocol Timeout property will be used.

Sending Requests Asynchronously

This is the easiest way to send a message to a server. Suppose you have created and filled a request object, for example, a `RequestAgentLogin` message for Interaction Server:

```
[Java]

RequestAgentLogin loginRequest = RequestAgentLogin.create();
loginRequest.setTenantId(tenantId);
loginRequest.setAgentId(agentId);
loginRequest.setPlaceId(placeId);
```

You can then send it to the server using the following code:

```
[Java]

interactionServerProtocol.send(loginRequest);
```

This will result in your application receiving a response from the Interaction Server: either an `EventAck` or an `EventError` message. By using the Send method, you will ignore that response at the place where you make the request. You will get the response, like any other unsolicited event, using the techniques described in the *Receiving Messages* section.

Handling Responses

The understanding of how to send requests and receive events is all you need to communicate with Genesys servers. However, the Platform SDK also provides the ability to easily associate a response with the particular request that originated it.

Receiving a Response Synchronously

The easiest way to handle responses is with the `Request` method. This is a blocking method, as your application stops to wait for a response to come from the server. Using the same request example above:

```
[Java]

Message response = interactionServerProtocol.request(loginRequest);
if (response.messageId() == EventAck.ID) {
        EventAck eventAck = (EventAck)response;
        // continue here
}
else {
        // handle the error here
}
```

Notice that you will need to cast the message to a specific message type in order to access its attributes. If a request fails on the server side, you will typically receive an `EventError`.

Take into account that the `Request` method blocks until a message is received or a timeout occurs. If the timeout elapses and no response was received from the server, then a `null` value is received. The timeout parameter can be specified in the request method. If you do not use the timeout parameter then, then the protocol `Timeout` property is used.

The `Request` method will only return one message from the server. In the case that the server returns subsequent messages, apart from the first response, as a result of the requested operation, then you must process those messages separately as unsolicited events. Please make sure that your code handles all messages received from your servers.

When using the `Request` method, your application only receives the response to that request as a return value. The response will not be received as an unsolicited event as well. (You can change this behavior by using the CopyResponse protocol property, described below.)

Receiving a Response Asynchronously

For many applications, blocking your thread while waiting for a response to your request is not appropriate. For example GUI applications, where the GUI can appear "frozen" if the response takes too much time to be received. It can also be true for batch applications that may want to send multiple requests at the same time, while waiting for all responses concurrently. For these scenarios it is possible to receive responses asynchronously.

Receiving a Response Asynchronously Using a Callback

By using `requestAsync`, your thread will not block, and it will permit you to handle the response by using callback methods that will get called asynchronously.

First, you will need to implement a `CompletionHandler` which will contain the logic for handling the response to your request:

```
[Java]

private static final CompletionHandler loginResponseHandler = new  CompletionHandler() {

        @Override
        public void completed(Message message, Void notUsed) {
                // handle message here
        }

        @Override
        public void failed(Throwable exc, Void notUsed) {
                // handle error here
        }

};
```

Important
The `CompletionHandler` callback methods will be executed by the protocol invoker.

Then you can use the `CompletionHandler` as a parameter to the `requestAsync` method:

```
[Java]

interactionServerProtocol.requestAsync(loginRequest, null, loginResponseHandler);
```

Notice that in this example, the `attachment` parameter has not been used. If you are sharing the same `CompletionHandler` implementation for handling the responses to different requests then you may want to use an attachment to make it easy to differentiate among those requests.

Receiving the Response as a Future

Alternatively, you may want to handle responses using the same thread that did the request, but have the option to do something concurrently while waiting for the response. To accomplish this, use the `beginRequest` method.

As an example, you might perform two agent login requests concurrently: one for logging into the T-Server, and another for logging into Interaction Server.

```
[Java]

RequestFuture loginVoiceFuture = tserverProtocol.beginRequest(loginVoiceRequest);
RequestFuture loginMultimediaFuture =
interactionServerProtocol.beginRequest(loginMultimediaRequest);

Message loginVoiceResponse = loginVoiceFuture.get();
Message loginMultimediaResponse = loginMultimediaFuture.get();

// handle responses, both are available now
```

When using the `requestAsync` or `beginRequest` methods, you will **not** receive the response as an unsolicited event. (You can change this behavior by using the CopyResponse protocol property, described below).

CopyResponse

Previously it was stated that responses returned by request methods are not received as unsolicited events by default. This behavior can be modified by using the protocol `CopyResponse` property. The default value is `false`, but it can be set to `true` like this:

```
[Java]

tserverProtocol.setCopyResponse(true);
```

This is particularly useful for protocols which define events that can be both received unsolicited and as a response to a client request (such as `EventAgentLogin` defined by the T-Server protocol). By setting the `CopyResponse` property to `true`, you can execute your agent state change logic only when handling the message as an unsolicited event, and you do not need to include it when receiving the message as a response.

.NET

Once you have connected to a server, much of the work for your application will involves sending messages to that server and handling the events you receive from the server. This article describes how to send and receive messages from a server.

Messages: Overview of Events and Requests

Messages you send to a server are called requests, while messages you receive are called events. An event that is received from a server as the result of executing a request is called a response. In summary, messages can be classified by using the following taxonomy:

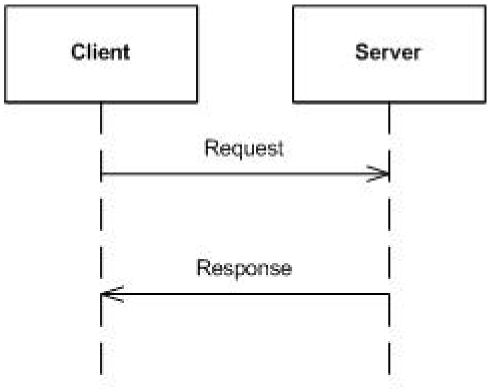
- Requests: sent to the server
- Events: received from the server
    - Responses: received as the result of a request
    - Unsolicited events: not a direct result of a request

Tip

On this page, we will use the more general term "message" instead of "event", in order to avoid confusion between protocol events and programming events.

For example, you may send a request to log in an agent or to gather statistics. You might also send a request to update a configuration object, or to shut down an application.

In each of these cases, the server will respond with an event message, as shown below.



You may also get unsolicited events from the server. That means receiving events that are not a response to a specific request. For example, `EventRinging` will notify you of a call ringing on an extension that you are currently monitoring.

Receiving Messages

With the Platform SDK, you can receive messages synchronously or asynchronously. It is important that you define the way your application will work in this aspect. In general, you will probably use only one type or the other in the same application.

Interactive applications normally use asynchronous message handling, because that will prevent the UI thread from being blocked, which could make the application appear "frozen" to a user. On the other hand, non-interactive batch applications commonly use synchronous response handling, as that allows writing easy code that performs step-by-step.

Receiving Messages Asynchronously

Most Platform SDK applications need to handle unsolicited events. This is particularly true for applications that monitor the status of contact center resources, such as extensions.

You receive server messages asynchronously by subscribing to the `Received` .NET event:

```
[C#]

tserverProtocol.Received += OnTServerMessageReceived;
```

Then you can implement your event-handling logic:

```
[C#]

void OnTServerMessageReceived(object sender, EventArgs e)
{
        IMessage message = ((MessageEventArgs)e).Message;
        // your event-handling code goes here
}
```

> **Important**
> You need to know that your event-handling logic will be executed by using the protocol invoker. Please set the invoker appropriate for your application needs. For more information about the protocol invoker and how to set it, refer to Connecting to a Server.

Inside your event-handling code, you will want to execute different logic for different kinds of events. A typical way to do this is using a `switch` statement, based on the event identifier:

```
[C#]

switch (message.Id)
{
        case EventAgentLogin.MessageId:
                OnEventAgentLogin(message);
                break;
        case EventAgentLogout.MessageId:
                OnEventAgentLogout(message);
                break;
}
```

Receiving Messages Synchronously

Some kinds of applications, such as batch applications, benefit from receiving messages synchronously. This means that received messages will queue up and be handled by the application on demand.

In order to receive messages this way, you simply do not subscribe to the `Received` .NET event as described in the previous section.

Tip

For releases prior to Platform SDK 8.1.1, messages were received synchronously by default. Please note that 8.1.1 behavior is backwards-compatible, and pre-8.1.1 applications will continue to work as expected without any modification.

To receive a message synchronously, use the `Receive` method. This method blocks processing, waiting for the next message to be received before continuing. Take into account that the maximum time to wait is set by a configurable timeout value. If the timeout expires and no event is received, you will receive a `null` value.

```
[C#]

IMessage message = tserverProtocol.Receive();
```

If you want to set your own timeout, you can use the `Receive` method overload that takes a timeout parameter. Otherwise, if you use Receive with no parameters, the protocol `Timeout` property will be used.

Sending Requests Asynchronously

This is the easiest way to send a message to a server. Suppose you have created and filled a request object, for example, a `RequestAgentLogin` message for Interaction Server:

```
[C#]

var loginRequest = RequestAgentLogin.Create();
loginRequest.TenantId = tenantId;
loginRequest.AgentId = agentId;
loginRequest.PlaceId = placeId;
```

Then you can send it to the server:

```
[C#]

interactionServerProtocol.Send(loginRequest);
```

This will result in your application receiving a response from the Interaction Server: either an `EventAck` or an `EventError` message. By using the Send method, you will ignore that response at the place where you make the request. You will get the response, like any other unsolicited event, using the techniques described in the *Receiving Messages* section.

Handling Responses

The understanding of how to send requests and receive events is all you need to communicate with Genesys servers. However, the Platform SDK also provides the ability to easily associate a response with the particular request that originated it.

Receiving a Response Synchronously

The easiest way to handle responses is with the `Request` method. This is a blocking method, as your application stops to wait for a response to come from the server. Using the same request example above:

```
[C#]

IMessage response = interactionServerProtocol.Request(loginRequest);
if (response.Id == EventAck.MessageId)
{
        var eventAck = (EventAck)response;
        // continue here
}
else
{
        // handle the error here
}
```

Notice that you will need to cast the message to a specific message type in order to access its attributes. If a request fails on the server side, you will typically receive an `EventError`.

Take into account that the `Request` method blocks until a message is received or a timeout occurs. If the timeout elapses and no response was received from the server, then a `null` value is received. The timeout parameter can be specified in the request method. If you do not use the `timeout` parameter then the protocol `Timeout` property is used.

The `request` method will only return one message from the server. In the case that the server returns subsequent messages, apart from the first response, as a result of the requested operation, then you must process those messages separately as unsolicited events. Please make sure that your code handles all messages received from your servers.

When using the `Request` method, your application only receives the response to that request as a return value. The response will not be received as an unsolicited event as well. (You can change this behavior by using the `CopyResponse` protocol property, described below).

Receiving a Response Asynchronously

For many applications, blocking your thread while waiting for a response to your request is not appropriate. For example GUI applications, where the GUI can appear "frozen" if the response takes too much time to be received. It can also be true for batch applications that may want to send multiple requests at the same time, while waiting for all responses concurrently. For these scenarios it is possible to receive responses asynchronously.

By using `BeginRequest`, your thread will not block, and it will permit you to handle the response the way that best suits your application. This method complies with .NET "Asynchronous Programming Model". You can find more information about the "Asynchronous Programming Model" in the Web.

For example, your application can handle responses asynchronously by using a callback, which is a piece of logic that executes asynchronously when the response is received. Define a callback method like this:

```
[C#]

void OnLoginResponseReceived(IAsyncResult result) {
        IMessage response = interactionServerProtocol.EndRequest(result);
        if (response.Id == EventAck.MessageId)
        {
                var eventAck = (EventAck)response;
                // continue here
        }
        else
        {
                // handle the error here
        }
}
```

Then you can submit your request using the callback method.

```
[C#]

interactionServerProtocol.BeginRequest(loginRequest, OnLoginResponseReceived, null);
```

As an alternative, you may want to do something concurrently, while waiting for the response. For example, you could perform two agent login requests concurrently: one for logging the agent into the T-Server, and another for logging the agent into Interaction Server.

```
[C#]

var resultLoginVoice = tserverProtocol.BeginRequest(loginVoiceRequest, null, null);
var resultLoginMultimedia = interactionServerProtocol.BeginRequest(loginMultimediaRequest,
null, null);

var loginVoiceResponse = tserverProtocol.EndRequest(resultLoginVoice);
var loginMultimediaResponse = interactionServerProtocol.EndRequest(resultLoginMultimedia);

// handle responses, both are available now
```

When using the `BeginRequest` method, your application receives the response to your request as the return value of `EndRequest`. You will not receive the response as an unsolicited event. (You can change this behavior by using the `CopyResponse` protocol property, described below).

CopyResponse

Previously it was stated that responses returned by request methods are not received as unsolicited events by default. This behavior can be modified by using the protocol `CopyResponse` property. The default value is false, but it can be set to true like this:

```
[C#]

tserverProtocol.CopyResponse = true;
```

This is particularly useful for protocols which define events that can be both received unsolicited and as a response to a client request (such as `EventAgentLogin` defined by the T-Server protocol). By setting the `CopyResponse` property to `true`, you can execute your agent state change logic only when handling the message as an unsolicited event, and you do not need to include it when receiving the message as a response.

Setting up Logging in Platform SDK

Java

Using the Built-In Logging Implementation

The Platform SDK Commons library provides adapters for the following implementations:

- com.genesyslab.platform.commons.log.SimpleLoggerFactoryImpl - redirect Platform SDK logs to System.out;
- com.genesyslab.platform.commons.log.JavaUtilLoggerFactoryImpl - redirect Platform SDK logs to Java common java.util.logging logging system;
- com.genesyslab.platform.commons.log.Log4JLoggerFactoryImpl - redirect Platform SDK logs to underlying Log4j 1.x;
- com.genesyslab.platform.commons.log.Log4J2LoggerFactoryImpl - redirect Platform SDK logs to underlying Log4J 2;
- com.genesyslab.platform.commons.log.Slf4JLoggerFactoryImpl - redirect Platform SDK logs to underlying Slf4j.

Note: Prior to release 8.5.102.02, the only log adapter available was for log4j v1.x and short names were not available.

By default, these log implementations are switched off but you can enable logging by using one of the methods described below.

1. In Your Application Code

The easiest way to set up Platform SDK logging in Java is in your code, by creating a factory instance for the log adapter of your choice and set it as the global logger factory for Platform SDK at the beginning of your program. An example using the log4j 1.x adapter is show here:

```
com.genesyslab.platform.commons.log.Log.setLoggerFactory(new Log4JLoggerFactoryImpl());
```

2. Using a Java System Variable

Using a Java system variable, by setting `com.genesyslab.platform.commons.log.loggerFactory` to the fully qualified name of the `ILoggerFactory` implementation class. For example, to set up log4j as the logging implementation you can start your application using the following command:

```
java -Dcom.genesyslab.platform.commons.log.loggerFactory=<log_type> <MyMainClass>
```

Where <log_type> is either a full-defined class names with packages, or one of the following short names:

- console - for SimpleLoggerFactoryImpl (to System.out);

- jul - for JavaUtilLoggerFactoryImpl;
- log4j - for the Log4J 1.x adaptor;
- log4j2 - for the Log4J 2 adaptor;
- slf4j - for the Slf4j adaptor;
- auto - with this value, Platform SDK Commons logging tries to detect available the logging system from the list of ['Log4j2', 'Slf4j', 'Log4j']; if no log system from the list is detected then the `JavaUtilLoggerFactoryImpl` adapter will be used.

3. Configuration in the Class Path

You can also configure logging using a `PlatformSDK.xml` Java properties file that is specified in your class path:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry
key="com.genesyslab.platform.commons.log.loggerFactory">com.genesyslab.platform.commons.log.Log4JLoggerFactoryI
</properties>
```

For more information, refer to details about the `PsdkCustomization` class in the API Reference Guide.

Providing a Custom Logging Implementation

If log4j does not fit your needs, it is also possible to provide your own implementation of logging.

In order to do that, you will need to complete the following steps:

1. Implement the `ILogger` interface, which contains the methods that the Platform SDK uses for logging messages, by extending the `AbstractLogger` class.
2. Implement the `ILoggerFactory` interface, which should create instances of your `ILogger` implementation.
3. Finally, set up your `ILoggerFactory` implementation as the global Platform SDK `LoggerFactory`, as described above.

Setting Up Internal Logging for Platform SDK

To use internal logging in Platform SDK, you have to set a logger implementation in Log class *before* making any other call to Platform SDK. There are two ways to accomplish this:

1. Set the `com.genesyslab.platform.commons.log.loggerFactory` system property to the fully qualified name of the factory class
2. Use the `Log.setLoggerFactory(...)` method

One of the log factories available in Platform SDK itself is `com.genesyslab.platform.commons.log.Log4JLoggerFactoryImpl` which uses log4j. You will have to setup log4j according to your needs, but a simple log4j configuration file is shown below as an example.

```
log4j.logger.com.genesyslab.platform=DEBUG, A1
```

```
log4j.appender.A1=org.apache.log4j.FileAppender
log4j.appender.A1.file=psdk.log
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %-25.25c %x - %m%n
```

The easiest way to set system property is to use `-D` switch when starting your application:

```
-Dcom.genesyslab.platform.commons.log.loggerFactory=com.genesyslab.platform.commons.log.Log4JLoggerFactoryImpl
```

Logging with AIL

In Interaction SDK (AIL) and Genesys Desktop applications, you can enable the Platform SDK logs by setting the option `log/psdk-debug = true`.

At startup, AIL calls: `Log.setLoggerFactory(new Log4JLoggerFactoryImpl());`

The default level of the logger `com.genesyslab.platform` is `WARN` (otherwise, applications would literally be overloaded with logs). The option is dynamically taken into account; it turns the logger level to `DEBUG` when set to true, and back to `WARN` when set to false.

Dedicated loggers

Platform SDK has several specialized loggers:

1. com.genesyslab.platform.ADDP
2. com.genesyslab.platformmessage.request
3. com.genesyslab.platformmessage.receive

Dedicated ADDP Logger

ADDP logs can be enabled using common Platform SDK log configuration.

```
log4j.logger.com.genesyslab.platform=INFO, A1
log4j.appender.A1=org.apache.log4j.FileAppender
log4j.appender.A1.file=psdk.log
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %-25.25c %x - %m%n
```

In addition, the `com.genesyslab.platform.ADDP` logger is controlled by the `addp-trace` option. If ADDP log is not required on INFO level, it can be disabled using the following option:

```
PropertyConfiguration config = new PropertyConfiguration();
config.setAddpTraceMode(AddpTraceMode.None);
```

or

```
config.setAddpTraceMode(AddpTraceMode.Remote);
```

The `addp-trace` option has no effect when DEBUG level is set. ADDP logs will be printed regardless of the option value.

Important

In Platform SDK 8.5.0, the second ADDP logger (`AddpIntreceptor`) was removed to avoid ADDP log duplication when `RootLogger` of the logging system is set to DEBUG level.

Instead of using second ADDP logger to print logs to another file, it is possible to specify additional appender.

A sample configuration is provided below:

```
log4j.logger.com.genesyslab.platform=WARN, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-d [%t] %-5p %-25.25c %x - %m%n
log4j.appender.A1.Threshold=WARN

//additional log file with addp traces.
log4j.logger.com.genesyslab.platform.ADDP=INFO, A2
log4j.appender.A2=org.apache.log4j.FileAppender
log4j.appender.A2.file=addp.log
log4j.appender.A2.append=false
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%-d [%t] %-5p %-25.25c %x - %m%n
```

Dedicated Request and Receive Loggers

A sample Log4j configuration is shown here:

```
log4j.logger.com.genesyslab.platformmessage.request=DEBUG, A1
log4j.logger.com.genesyslab.platformmessage.receive=DEBUG, A1
```

Important

In PSDK 8.5.0 version the `PSDK.DATA` logger was replaced with `com.genesyslab.platformmessage.request` and `com.genesyslab.platformmessage.receive` loggers.

These loggers allow printing complete message attribute values. By default, large attribute logs are truncated to avoid application performance impact:

```
'EventInfo' (2) attributes:
        VOID_DELTA_VALUE [bstr] =
           0x00 0x01 0xFF 0xFF 0x00 0x05 0x00 0x00 0x00 0x00
           0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
           0x09 0x00 0x00 0x00 0x05 0x00 0x00 0x00 0x00 0x00
           ... [output truncated, 362 bytes left out of 512]
```

However, in some cases a full data dump may be required in logs. There are three possible ways to do this, as shown below:

Important

To avoid log duplication when the logging system `RootLogger` is configured to DEBUG level, these loggers are disabled by default and can be activated with a system property. This system property affects both loggers.

1. Activate using system properties:

```
-Dcom.genesyslab.platform.trace-messages=true //for all protocols
-Dcom.genesyslab.platform.Reporting.StatServer.trace-messages=true //only for stat protocol
```

2. Activate from code:

```
//for all protocols
PsdkCustomization.setOption(PsdkOption.PsdkLoggerTraceMessages, "false");

//only for stat protocol
String protocolName = StatServerProtocolFactory.PROTOCOL_DESCRIPTION.toString();
PsdkCustomization.setOption(PsdkOption.PsdkLoggerTraceMessages, protocolName, "true");
```

These static options should be set once at the beginning of the program, before opening Platform SDK protocols.

3. Activate from `PlatformSDK.xml`:

```
 <?xml version="1.0" encoding="UTF-8"?>
 <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
 <properties>
   <entry key="com.genesyslab.platform.trace-messages">true</entry>
 </properties>
```

For details about the `PsdkCustomization` class, refer to the API Reference Guide.

.NET

Setting up Logging

For .NET development, the `EnableLogging` method allows logging to be easily set up for any classes that implement the `ILogEnabled` interface. This includes:

- All protocol classes: `TServerProtocol`, `StatServerProtocol`, etc.
- The `WarmStandbyService` class of the Warm Standby Application Block.

For example:

```
tserverProtocol.EnableLogging(new MyLoggerImpl());
```

Providing a Custom Logging Implementation

You can provide your custom logging functionality by implementing the `ILogger` interface. Samples of how to do this are provided in the following section.

Samples

You can download some samples of classes that implement the `ILogger` interface:

- AbstractLogger: This class can make it easier to implement a custom logger, by providing a default implementation of `ILogger` methods.
- TraceSourceLogger: A logger that uses the .NET TraceSource framework. It adapts the Platform SDK logger hierarchy to the non-hierarchical TraceSource configuration.
- Log4netLogger: A logger that uses the log4net libraries.

Additional Logging Features

Java

Application Configuration Manager Component

The Application Configuration Manager component is a new addition to the Application Template Application Block.

This component monitors the application configuration from Configuration Server and provides notification of any updates to options for your custom application, options of connected servers, or options of their host objects. It also checks the availability of Log4j2 logging framework and automatically enables Log4j2 configuration based on the application logging options in Configuration Manager.

The quickest way to get an application configured for logging in accordance to the application "log" section might look like the following example:

```
public class MyApplication {

  protected static final LmsEventLogger LOG =
LmsLoggerFactory.getLogger(MyApplication.class);
.....
    GFApplicationConfigurationManager appManager =
            GFApplicationConfigurationManager.newBuilder()
            .withCSEndpoint(new Endpoint("CS-primary", csHost1, csPort1))
            .withCSEndpoint(new Endpoint("CS-backup", csHost2, csPort2))
            .withClientId(clientType, clientName)
            .withUserId(csUsername, csPassword)
            .build();
    appManager.register(new GFAppCfgOptionsEventListener() {
        public void handle(final GFAppCfgEvent event) {
            Log.getLogger(getClass()).info(
                    "The application configuration options received: " + event);
            // Init or update own application options from 'event.getAppConfig()'
        }});
    appManager.init();

    // LmsEventLogger method usage:
    LOG.log(CommonLmsEnum.GCTI_APP_INIT_COMPLETED);
    // Common ILogger method usage:
    LOG.info("Some Log4j2 info message");
.....
    // Shutdown the configuration manager:
    appManager.done();
```

In this example, the builder for the manager creates and initializes an internal instance of `ConfService` and encapsulates the `WarmStandby` service to handle failures of the Configuration Server connection.

Created with these parameters, `ConfService` has:

- the default `ConfService` cache enabled;
- a `WarmStandby` service with default configuration for the two given Configuration Server endpoints;
- automatic Configuration Server subscription for notifications on the application and host object types;

If your application needs to have a custom `ConfService` instance with a specific configuration (for example, a customized cache) then it is possible to create an Application Configuration Manager on your pre-configured `ConfService` instance. In this case the manager does not take care of the service connection state or caching - it is up to your application to create and manage the `WarmStandby` service, Configuration Server subscriptions, and the `ConfService` cache.

An example of working with a custom `ConfService` instance is provided below:

```
GFApplicationConfigurationManager appManager =
        GFApplicationConfigurationManager.newBuilder()
        .withConfService(confService)
        .build();
appManager.register(new GFAppCfgOptionsEventListener() {
    public void handle(final GFAppCfgEvent event) {
        Log.getLogger(getClass()).info(
                "The application configuration options received: " + event.getAppConfig());
    }});
appManager.init();
```

Common Logging Interfaces Usage

Platform SDK for Java has its own interface for logging (using `com.genesyslab.platform.commons.log`) that includes the following classes/interfaces:

- `Log`,
- `ILoggerFactory`,
- `ILogger`

Platform SDK uses the `ILogger` interface to generate Platform SDK internal log messages, and custom applications are also able to use this logging interface as shown in below:

```
public class SomeUserClass {
    protected static final ILogger log = Log.getLogger(SomeUserClass.class);

    public void doSomething() {
        try {
            log.debug("doing something");
            // ...
        } catch (final Exception ex) {
            log.debug("exception while doing something", ex);
        }
    }
}
```

In this sample, the commons logging messages will go to the particular `ILogger` interface implementation.

Up to release 8.5.0 of Platform SDK, there were two implementations of the interface available: a silent "NullLogger" (default) and Log4j adapter.

Starting with release 8.5.1, the following additional implementations have been added:

- "simple" console printing implementation;
- java.util.logging adapter;
- Slf4j interface adapter;
- Log4j 2.x adapter.

Also it is possible to create a custom implementation of `ILogger` and `ILoggerFactory`, enable their usage, and get into some other logging system.

LMS Event Loggers and LMS files support

`LmsEventLogger` is an extension of the common Platform SDK `ILogger` interface that is used for logging Genesys LMS events to Message Server or for writing log files in the Genesys-specific format.

An example of simple `LmsEventLogger` usage is provided below:

```
class SampleClass {
    protected final static LmsEventLogger LOG = LmsLoggerFactory.getLogger(SampleClass.class);
    public void method() {
        // Use logger to generate event:
        LOG.log(LogCategory.Application, CommonLmsEnum.GCTI_LOAD_RESOURCE, "users.db", "no
such file");
            // => "Unable to load resource 'users.db', error code 'no such file'"

        // or, for event GCTI_CFG_APP[6053, STANDARD, "Configuration for application
obtained"]:
        LOG.log(CommonLmsEnum.GCTI_CFG_APP);
          // or
        LOG.log(6053); // => "Configuration for application obtained"

        // or "plain" logging methods:
        try {
            LOG.debug("Starting cache load...");
            // ... do something ...
        } catch (final Exception exception) {
            LOG.error("Failed to load cache", exception);
        }
.....
```

The Application Configuration Manager component included with Application Template Application Block is able to automatically configure and keep an updated `LmsMessageConveyor` with LMS files configuration.

However, if your custom application does not use the Application Configuration Manager component then it can configure LMS file usage in the following way:

```
public class SomeUserClass {
    protected static final LmsEventLogger LOG =
LmsLoggerFactory.getLogger(SomeUserClass.class);
```

```
    public void configureLogging() {
        // Create new instance of LMS conveyor:
        LmsMessageConveyor lmsConveyor = new LmsMessageConveyor();

        // Configure it:
        lmsConveyor.loadConfiguration(appConfig);
            // or:
            lmsConveyor.loadConfiguration("MyApp.lms");

        // Reinitialize the LmsLoggerFactory singleton with the new conveyor:
        LmsLoggerFactory.createInstance(lmsConveyor);
            // or
            LmsLoggerFactory.setLoggerFactoryImpl(Log.LOG_FACTORY_LOG4J2, lmsConveyor);
    }
.....
```

The LMS loggers also have several implementations in order to support different target logging frameworks including java.util.logging, log4j v1, slf4j, and log4j v2. You can enable specific target framework usage synchronously with the Platform SDK common logging.

The Application Template application block contains the LMS event loggers, a "common.lms" file, its correspondent `CommonLmsEnum` class, and an `LmsEnum` generator tool for generation of specific enumerations from the LMS files for your custom applications.

Using Custom LMS Files and Correspondent LmsEnums

1. Create a custom LMS file with the default localization context, for example, "MyApp.lms":

```
xxxxxxxx|LMS|1.0|MyApp.lms|8.5.100|*

 21001|STANDARD|MY_APP_START_EVENT|Application '%s' started the work
 21002|ALARM|MY_APP_DATABASE_LOST|App '%s' failed to connect to database '%s'
 .....
```

2. Generate the corresponding enumeration class using the Platform SDK generator:

```
%> java -cp apptemplate.jar
 com.genesyslab.platform.apptemplate.lmslogger.LmsEnumGenerator
             MyApp.lms MyAppLmsEnum custom.package.name
```

As a result there will be a `custom.package.name.MyAppLmsEnum` enumeration containing declarations from `MyApp.lms`.

3. (Optional) Create a customized version of the `MyApp.lms` file with localized messages.
4. (Optional) Load the enumeration(s) with message conveyor and initialize `LmsLoggerFactory` with it. The Application Template applcation block exposes annotations processor, which collects information about available `LmsEnums` in the application build time, so the default `LmsMessageConveyor()` constructor is able to register generated `LmsEnums` automatically. If it does not do this for some reason, and there is no acceptable way to let the annotation processor work for your use case, then it is possible to initialize `LmsMessageConveyor` explicitly:

```
LmsMessageConveyor lmsConveyor = new LmsMessageConveyor(CommonLmsEnum.class,
 MyAppLmsEnum.class);
 lmsConveyor.loadConfiguration("MyApp.lms"); // optional - if there is a localized LMS
 file

 // Reinitialize the LmsLoggerFactory singleton with the new conveyor:
 LmsLoggerFactory.createInstance(lmsConveyor);
     // or
     LmsLoggerFactory.setLoggerFactoryImpl(Log.LOG_FACTORY_LOG4J2, lmsConveyor);
```

5. Call the loggers

```
   lmsLog.log(MyAppLmsEnum.MY_APP_START_EVENT, "my application");
    lmsLog.log(LogCategory.Alarm, MyAppLmsEnum.MY_APP_DATABASE_LOST, "my application",
    "dbname");
```

Enabling Logging Framework Usage

The Platform SDK common loggers and LMS events loggers may be configured to direct logs to particular logging framework by using a system property at application startup, or at runtime with an explicit API call.

The system property name is `com.genesyslab.platform.commons.log.loggerFactory`. It may contain the FQCN of your particular implementation of the Platform SDK common loggers factory, or the alias name of a built-in implementation (such as "log4j2", "slf4j", etc). If you specify one of the alias names, this value is also used by the `LmsLoggerFactory` initialization logic, so that LMS events from `LmsEventLogger` will also be directed to the same logging framework.

In this case, the jvm option might look like:

```
-Dcom.genesyslab.platform.commons.log.loggerFactory=log4j2
```

This property is handled with the Platform SDK customization options, so you can also enable it by adding a PlatformSDK.xml file with the following contents to your application classpath:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <entry key="com.genesyslab.platform.commons.log.loggerFactory">log4j2</entry>
</properties>
```

Changing this property after initialization of the `LoggerFactory` has no effect. So if you need to enable logging or change the target framework at runtime, this should be done with an explicit API call. In this case the Platform SDK commons loggers and LMS events loggers need to be reconfigured separately:

```
// Platform SDK Commons loggers (re-)configuration:
Log.setLoggerFactory(Log.LOG_FACTORY_LOG4J2);

// AppTemplate LMS Events loggers:
LmsLoggerFactory.setLoggerFactoryImpl(Log.LOG_FACTORY_LOG4J2, lmsMessageConveyor);
```

Important

Be aware that re-initializing the LMS Loggers factory requires a reference to the actual `LmsMessageConveyor`. It is also possible to use "null", in which case the factory initialization method will try to reuse reference from the "current" `LmsLoggerFactory` or will create a default conveyor instance with support of the `CommonLmsEnum` events.

These calls are enough to reconfigure loggers which were created earlier; there is no need to recreate them.

Configuring Logging

Platform SDK does not write logs itself (that is, it's not about the legacy Logger Component). Instead Platform SDK is just able to send logs to the specified logging framework where they will be handled. Configuration of logging parameters such as log file names, log levels, and more may be done within that framework.

The Application Template application block contains parsing logic for Genesys Configuration Manager Application logging options properties, and Log4j2 configuration structures to allow automatic framework configuration. This helps applications to automatically start logging to the recommended Log4j2 framework (as discussed below).

It is also possible to create user defined configurators for some other framework. In this case your application may use the Application Configuration Manager to retrieve application configuration details from Configuration Server in the form of POJO structures, and apply those details to its custom logging framework.

Configuring Logging with Log4j2

Log4j2 is the recommended logging framework. The Application Template application block provides several options for the logging framework configuration.

Using the Application Configuration Manager allows automatic Log4j2 configuration and reconfiguration in accordance to the Genesys Configuration Manager application logging options.

Beside the common Genesys logging options, the Application Template application block also supports a custom "log4j2-config-profile" option, which allows you to create combined logging configuration as a merge of user defined loggers/appenders with ones created by the Configuration Manager application options. This is useful in cases where your application consists of several sub-systems, and is required to split logs from those sub-systems to different log files.

For example, in a Web Application it may be reasonable to separately write logs from Tomcat, Cassandra, Platform SDK, and the application itself. In this scenario, one straightforward way of logging configuration may be following:

1. Create a "startup" log4j2.xml configuration, which will be used before the application has retrieved information from configuration server. This configuration might contain declarations of the application-specific loggers and appenders. Appenders may be either "startup" or "permanent". The first ones have names starting with "PSDKAppTpl-".
2. When your application starts with enabled Log4j2 usage, it picks up and uses the startup configuration "as is". So, we have the application startup logs.
3. When your application has received the Genesys Configuration Manager Application options, it creates and applies `PsdkLog4j2Configuration` where `log4j2-config-profile = log4j2.xml`. It takes the log4j2 configuration as a base and replaces its startup appender(s) with new ones from the Configuration Manager Application logging options. This allows you to configure the application logging

parameters in accordance to Genesys Configuration Manager Application logging options, and gives you the ability to handle external (for example, Tomcat or Cassandra) logs separately if desired.

Log4j2 Compatibility

The following table describes compatibility and official testing status of Platform SDK with different versions of Log4j.

- full - this configuration is supported
- partial - Platform SDK commons logging redirect to Log4j2 is supported, but not Log4j2 configuration using the Application Template Application Block
- none - this configuration is not working

	Log4j 2.2	**Log4j 2.3**	**Log4j 2.4x**	**Log4j 2.5**	**Log4j 2.6x**	**Log4j 2.7**	**Log4j 2.8x**	**Log4j 2.9x**	**Log4j 2.10.0**	**Log4j 2.11.0**	**Log4j 2.11.1**
Platform SDK 8.5.x with Java 6	full (uses 2.2)	full	none	none	none	none	none	none	none	none	none
Platform SDK 8.5.x with Java 7+	full (uses 2.2)	full	full	full	partial	partial	partial	partial	partial	partial	partial
Platform SDK 9.0.000.x and 9.0.001.x with Java 8+	partial	partial	partial	partial	partial	full	full (uses 2.8.2)	partial	partial	partial	partial
Platform SDK 9.0.002.x and higher with Java 8+	partial	partial	partial	partial	partial	partial	partial	full	full (uses 2.10.0)	full	full

.NET

NLog Adapter for Platform SDK .NET

This adapter combines the benefits of NLog library for .NET - a wide variety of targets that cover most application requirements - with the the ability to write messages into Genesys Message Server.

Design Overview

The `PsdkLoggerFactory` class acts as a common entry point for Platform SDK for .NET logger systems, and can be used on-the-fly by changing the configuration of any existing application.

Starting with release 9.0.001.02, Platform SDK for .NET introduced a new `NLogLoggerFactory` that may be set as the default factory for implementing logging system functionality inside of Platform SDK. This new factory can also create detached logger instances to be used for general purposes.

`NLogLoggerFactory` produces logger instances that implement the Adapter pattern and include functionality to write messages to Genesys Message Server, as described by the `ILmsEventLogger` interface. This factory can create instances of the `ILmsEventLogger` interface using the `CreateLmsLogger` or `CreateNullLmsLogger` methods.

The `IGLoggerConfiguration` interface is used to describe the logger configuration. The `GAppLoggingOptions` helper class implements this interface, and should be given logger configuration details either from the `IGApplicationConfiguration` interface or directly from `KeyValueCollection` storage.

By assigning an `LmsMessageConveyor` instance to the factory when using the factory constructor, you can use your own LMS files with predefined message templates that replace the existing templates.

Usage Examples

Example 1: Configure File Target with Expiration

```
...
var logOptions = new KeyValueCollection()
{
  {"verbose", "standard"},
  {"message-format", "full"},
  {"time-format", "iso8601"},
  {"all", "FileTarget.log"},
  {"Segment","100Kb"},
  {"Buffering","true"},
  {"DeleteArchiveFiles","true"},
  {"Expire","1day"}
};
var config = new GAppLoggingOptions(logOptions, null);
var loggerFactory = new NLogLoggerFactory();
loggerFactory.Configure(config);
PsdkLoggerFactory.Factory = loggerFactory;
var logger = loggerFactory.CreateLogger("TestFileTarget");
logger.Debug("log message 1");
logger.Warn("log message 2");
logger.Info("log message 3");
logger.Error("log message 4");
logger.FatalError("log message 5");
...
```

Example 2: Create and Use Lms Event Logger

```
...
var logger = NLogLoggerFactory.Default.CreateLmsLogger(null); // use default factory
logger.Debug(CommonLmsMessages.GCTI_ADDP_NO_RESPONSE, "server","Msg", 20, 40);
...
```

Example 3: Using Lms logger with Custom Template and Logging Level Replacement

```
...
var logOptions = new KeyValueCollection()
  {
    {"verbose", "standard"},
    {"message-format", "full"},
    {"time-format", "iso8601"},
    {"all", "network"},
    {"message-server-timeout","20000"}
  };
var logExtOptions = new KeyValueCollection()
  {
    {"level-reassign-14005", "ALARM"},
  };
var config = new GAppLoggingOptions(logOptions, logExtOptions, null);
var loggerFactory = new NLogLoggerFactory();
loggerFactory.Configure(config);
var logger = loggerFactory.CreateLmsLogger("test");

var template = new LmsMessageTemplate(14005, LogLevel.Info, "TEST", "Test message: %s");
logger.Log(template,"12345");  // sends with LogLevel.Fatal
...
```

NLog Extensions

Although NLog has many predefined targets, you can also use additional targets introduced with Platform SDK by modifying the configuration file.

Examples of using these additional targets, by modifying either your NLog configuration file or your `app.config` file, are shown in the examples below:

Message Server Target

```
<configuration>
  <nlog>
    <targets>
      <target name="lms" type="MessageServer" port="2345" host="localhost" />
    </targets>
    <rules>
      <logger name="LmsLogger" minLevel="Warn" maxLevel="Fatal" writeTo="lms" />
    </rules>
  </nlog>
</configuration>
```

Attribute name	Description	Required/ Optional	Default value	Notes
name	Any name of logger to be used in code	Required		
type	target type	Required	must be 'MessageServer'	
host	host name of message server	Required		
port	port of message server	Required		must be integer value 1..65520
clientName	Client name field for Message server	Optional	null	used for handshake
clientHost	Client host field for Message server	Optional	null	used for handshake
clientType	Client type Id field for Message server	Optional	null	used for handshake, must be integer value
clientId	Client Id field for Message server	Optional	null	used for handshake, must be integer value
timeout	Timeout of MessageServerProtocol class	Optional	null	must be integer value
queueSize	Size of internal queue of messages to be sent on server	Optional	1024 (PsdkCustomization.LogFactory.MessageServerInitialQue parameter value)	must be integer value greater than 32

Memory Queue Target

```
<configuration>
  <nlog>
    <targets>
      <target name="mem" type="MemoryQueue" limit="8192"/>
    </targets>
    <rules>
      <logger name="MemLogger" minLevel="Warn" maxLevel="Fatal" writeTo="mem" />
    </rules>
  </nlog>
</configuration>
```

Attribute name	Description	Required/ Optional	Default value	Notes
name	Any name of logger to be used	Required		

Attribute name	Description	Required/ Optional	Default value	Notes
	in code			
type	target type	Required		must be 'MemoryQueue'
limit	Size of memory queue of log messages	Optional	4096 (PsdkCustomization.LogFactory.MemoryQueueSize parameter value)	

Log4j2 Configuration with the Application Template Application Block

Platform SDK Application Template Application Block contains Log4j2 plugin component for logging configuration.

Its goal is to allow application to configure logging in accordance to the Common Genesys Logging Options with Log4j2 library.

Enabling and configuring PSDK log4j2 logging

Using AppTemplate Configuration Manager

GFApplicationConfigurationManager, once initialized, automatically applies and listen for updates in the application logging configuration.

```
public class MyApplication {
    protected static final LmsEventLogger LOG =
LmsLoggerFactory.getLogger(MyApplication.class);

...
    GFApplicationConfigurationManager appManager =
            GFApplicationConfigurationManager.newBuilder()
            .withCSEndpoint(new Endpoint("CS-primary", csHost1, csPort1))
            .withCSEndpoint(new Endpoint("CS-backup", csHost2, csPort2))
            .withClientId(clientType, clientName)
            .withUserId(csUsername, csPassword)
            .build();
    appManager.register(new GFAppCfgOptionsEventListener() {
        public void handle(final GFAppCfgEvent event) {
            Log.getLogger(getClass()).info("The application configuration options received: "
+ event);
            // Init or update own application options from 'event.getAppConfig()'
        }});
    appManager.init();

    // LmsEventLogger method usage:
    LOG.log(CommonLmsEnum.GCTI_APP_INIT_COMPLETED);
    // Common ILogger method usage:
    LOG.info("Some Log4j2 info message");
...
```

For more details see Additional Logging Features.

Using AppTemplate API calls

If GApplicationConfigurationManager is not used, its still possible to configure logging in the same way by using AppTemplate API calls.

In package "com.genesyslab.platform.apptemplate.log4j2" AppTemplate provides classes named PsdkLog4j2Configuration and Log4j2Configurator.

Main cases are following:

1. Create PsdkLog4j2Configuration with its factory method "parse(KeyValueCollection)" and set it with the configurator (Log4j2Configurator.setConfig()). It uses optional config profile and given application Options "log" sections, and configures. Drawbacks: no "log-extended" Options section is taken into account, no Message Server appender from the application connections.
2. Obtain IGApplicationConfiguration and apply it with the configurator (Log4j2Configurator.applyLoggingConfig()). The application configuration may be explicitely filled using GApplicationConfiguration, or automatically initialized from ConfService using COMGApplicationConfiguration. This way includes initialization of "log-extended" Options sections and may add Message Server appender if its properly configured in the application configuration. Note: this configuration appliance method also switches PSDK loggers to send their logs to Log4j2 (if it was not enabled earlier).

More details on enabling of PSDK logging may be found in Setting up Logging in Platform SDK.

Switch PSDK logging to log4j2 with system properties

The following properties are supported:

- "-Dcom.genesyslab.platform.commons.log.loggerFactory=log4j2" to switch PSDK commons logging implementation to send logs to log4j2.
- "-Dcom.genesyslab.platform.apptemplate.lmslogger.factory=log4j2" to switch AppTemplate LmsEventLogger's to send logs and LMS events to log4j2.

AppTemplate Logging configuration profile

AppTemplate logging configuration profile is a usual Log4j2 xml configuration file. And it may be used as "startup configuration" before application is initialized and read its configuration from Configuration Server.

For details on Log4j2 configuration details see: http://logging.apache.org/log4j/2.x/manual/configuration.html.

To use "log4j2.xml" as the config profile, it should be referred with application CME Option:

```
[log]
"log4j2-config-profile"="log4j2.xml"
```

Example of startup logging configuration with logs formatted in the common Genesys LMS style:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration packages="com.genesyslab.platform.apptemplate.log4j2plugin">
  <Appenders>
    <GLogFile name="PSDKAppTpl-LogFile" fileName="startup.log">
      <GLmsLayout timeFormat="locale" messageFormat="full"/>
      <GLogRolloverStrategy expConfig="8 days"/>
    </GLogFile>
    <GLogFile name="PSDK" fileName="psdk.log">
```

```
      <GLmsLayout timeFormat="locale" messageFormat="full"/>
      <GLogRolloverStrategy expConfig="8 days"/>
    </GLogFile>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="PSDKAppTpl-LogFile"/>
    </Root>
    <Logger name="com.genesyslab.platform" level="debug">
      <AppenderRef ref="PSDK"/>
    </Logger>
  </Loggers>
</Configuration>
```

Logging profile feature: "startup" appenders

Appenders, which have "PSDKAppTpl-" prefix in their names are treated as "startup" appenders.

Those ones are actual when the config profile is used as a usual log4j2 configuration file at the application startup time. But, when application has started, loaded its application configuration options, and called AppTemplate logging configuration (referring this profile), the "startup" appenders are excluded from the new configuration.

Logging profile feature: separate subsystems logs

Specifics of the Genesys Logging Configuration Options is that log files may be generated and filtered by evel level, but not by logger or "source" name. I.e. generated by the "log" appenders are added to the "root" logger of logging configuration.

Logging configuration profile allows creation of dedicated appenders and custom loggers declaration for collecting of specific subsystem(s) logs separately from main application logs.

Example of such configuration is in the config profile sample above - appender named "PSDK", and logger named "com.genesyslab.platform". This file name is "hardcoded" in the config profile and can't be changed from application CME Options. But its still possible to change following parameters of the logger with "logger-<id>" option in "log-extended" section: "level" and "additivity" (see details bellow). By this way, its possible to change messages level of the PSDK log file, and to add or exclude PSDK messages from main application log files.

Useful Platform SDK loggers for optional configuration

PSDK root logger

Platform SDK loggers are named under PSDK base package - "com.genesyslab.platform". Usage of this logger name allows applications to separately configure, filter and record PSDK internal logs.

Protocol messages tracing loggers

Usually, PSDK generates debug logs reflecting Protocols communications messages in truncated form together with other debug messages.

Sometimes applications may need to have separated recordings of protocols communications messages dumps.

For this purpose PSDK contains dedicated loggers:

- "com.genesyslab.platformmessage.request",
- "com.genesyslab.platformmessage.receive",
- and their common name - "com.genesyslab.platformmessage".

By default, these loggers are disabled. Its possible to enable them with jvm system property "-Dcom.genesyslab.platform.trace-messages=true".

Important

This system property is "branched" for support of values definitions for different protocol types. For example, "-Dcom.genesyslab.platform.Configuration.ConfServer.trace-messages=true" or "-Dcom.genesyslab.platform.Reporting.StatServer.trace-messages=true".

These values enable tracing for Config Server and Stat Server protocols only. "Branched" options names are constructed by insertion of a string representation of specific ProtocolDecription value (see DuplexChannel.getProtocolDescription().toString()).

It's possible to provide several different declarations for different protocols at a same time. It's also possible to enable tracing for the common property ("enable by default"), and disable for some specific protocols.

ADDP logger

There is a dedicated specially named logger for ADDP traces - "com.genesyslab.platform.ADDP".

It's for possible adjustment of ADDP traces logging filtering.

Application CME Logging Options

The AppTemplate logging configuration is based on AppTemplate Application Configuration structure as a snapshot representation of a COM AB CfgApplication structure.

Following values are taken into account: "log" and "log-extended" sections of the applications' "Options", and connected Message Server CME application.

Options section "log"

Most of this section options are inherited from the Genesys Common Log Options descriptions, as listed in the Framework 8.5 Configuration Options Reference Manual.

verbose

Specifies if log output is created, and if so, the minimum level of log events generated. Log event levels, starting with the highest priority level, are Standard, Interaction, Trace, and Debug.

Default Value: *all*

Valid Values:

- *all* - All log events (that is, log events of the Standard, Trace, Interaction, and Debug levels) are generated.
- *debug* - The same as all.
- *trace* - Log events of Trace level and higher (that is, log events of Standard, Interaction, and Trace levels) are generated, but log events of the Debug level are not generated.
- *interaction* - Log events of Interaction level and higher (that is, log events of Standard and Interaction levels) are generated, but log events of Trace and Debug levels are not generated.
- *standard* - Log events of Standard level are generated, but log events of Interaction, Trace, and Debug levels are not generated.
- *none* - No log output is produced.

Changes Take Effect: Immediately

Additional Information: Note: For definitions of the Standard, Interaction, Trace, and Debug log levels, refer to the Framework Management Layer User's Guide or Framework Genesys Administrator Help.

To configure log outputs, set log level options ("all", "alarm", "standard", "interaction", "trace", and/or "debug") to the desired types of log output ("stdout", "stderr", "network", "memory", and/or [filename], for log file output).

You can use:

- One log level option to specify different log outputs.
- One log output type for different log levels.
- Several log output types simultaneously, to log events of the same or different log levels.

You must separate the log output types by a comma when you are configuring more than one output for the same log level.

all

Specifies the outputs to which an application sends all log events. The log output types must be separated by a comma when more than one output is configured.

Default Value: No default value.

Valid Values: Log output types:

- *stdout* - Log events are sent to the Standard output (stdout).
- *stderr* - Log events are sent to the Standard error output (stderr).
- *network* - Log events are sent to Message Server, which can reside anywhere on the network. Message Server stores the log events in the Log Database. Setting the all log level option to the network output enables an application to send log events of the *Standard*, *Interaction*, and *Trace* levels to Message Server. *Debug*-level log events are neither sent to Message

Server nor stored in the Log Database.

- [filename] - Log events are stored in a file with the specified name. If a path is not specified, the file is created in the application's working directory.

Changes Take Effect: Immediately

Additional Information: For example: `all = stdout, logfile`

Note1: The log output options are activated according to the setting of the Verbose configuration option.

Note2: To ease the troubleshooting process, consider using unique names for log files that different applications generate.

Warning! Directing log output to the console (by using "stdout", "stderr" settings) can affect application performance. Avoid using these log output settings in a production environment.

alarm

Specifies the outputs to which an application sends the log events of the *Alarm* level. The log output types must be separated by a comma when more than one output is configured.

Default Value: No default value.

Valid Values: Log output types:

- *stdout* - Log events are sent to the Standard output (stdout).
- *stderr* - Log events are sent to the Standard error output (stderr).
- *network* - Log events are sent to Message Server, which can reside anywhere on the network. Message Server stores the log events in the Log Database.
- [filename] - Log events are stored in a file with the specified name. If a path is not specified, the file is created in the application's working directory.

Changes Take Effect: Immediately

Additional Information: For example: `alarm = stderr, network`

Note1: The log output options are activated according to the setting of the Verbose configuration option.

Note2: To ease the troubleshooting process, consider using unique names for log files that different applications generate.

Warning! Directing log output to the console (by using "stdout", "stderr" settings) can affect application performance. Avoid using these log output settings in a production environment.

standard

Specifies the outputs to which an application sends the log events of the Standard level. The log output types must be separated by a comma when more than one output is configured.

Default Value: No default value.

Valid Values: Log output types:

- *stdout* - Log events are sent to the Standard output (stdout).
- *stderr* - Log events are sent to the Standard error output (stderr).
- *network* - Log events are sent to Message Server, which can reside anywhere on the network.

Message Server stores the log events in the Log Database.

- [filename] - Log events are stored in a file with the specified name. If a path is not specified, the file is created in the application's working directory.

Changes Take Effect: Immediately

Additional Information: For example: `standard = stdout, logfile`

Note1: The log output options are activated according to the setting of the Verbose configuration option.

Note2: To ease the troubleshooting process, consider using unique names for log files that different applications generate.

Warning! Directing log output to the console (by using "stdout", "stderr" settings) can affect application performance. Avoid using these log output settings in a production environment.

interaction

Specifies the outputs to which an application sends the log events of the Interaction level and higher (that is, log events of the Standard and Interaction levels). The log outputs must be separated by a comma when more than one output is configured.

Default Value: No default value.

Valid Values: Log output types:

- *stdout* - Log events are sent to the Standard output (stdout).
- *stderr* - Log events are sent to the Standard error output (stderr).
- *network* - Log events are sent to Message Server, which can reside anywhere on the network. Message Server stores the log events in the Log Database.
- [filename] - Log events are stored in a file with the specified name. If a path is not specified, the file is created in the application's working directory.

Changes Take Effect: Immediately

Additional Information: For example: `interaction = stdout, logfile`

Note1: The log output options are activated according to the setting of the Verbose configuration option.

Note2: To ease the troubleshooting process, consider using unique names for log files that different applications generate.

Warning! Directing log output to the console (by using "stdout", "stderr" settings) can affect application performance. Avoid using these log output settings in a production environment.

trace

Specifies the outputs to which an application sends the log events of the *Trace* level and higher (that is, log events of the *Standard*, *Interaction*, and *Trace* levels). The log outputs must be separated by a comma when more than one output is configured.

Default Value: No default value.

Valid Values: Log output types:

- *stdout* - Log events are sent to the Standard output (stdout).
- *stderr* - Log events are sent to the Standard error output (stderr).

- *network* - Log events are sent to Message Server, which can reside anywhere on the network. Message Server stores the log events in the Log Database. Setting the *all* log level option to the *network* output enables an application to send log events of the *Standard*, *Interaction*, and *Trace* levels to Message Server. *Debug*-level log events are neither sent to Message Server nor stored in the Log Database.
- [filename] - Log events are stored in a file with the specified name. If a path is not specified, the file is created in the application’s working directory.

Changes Take Effect: Immediately

Additional Information: For example: `trace = stdout, logfile`

Note1: The log output options are activated according to the setting of the Verbose configuration option.

Note2: To ease the troubleshooting process, consider using unique names for log files that different applications generate.

Warning! Directing log output to the console (by using "stdout", "stderr" settings) can affect application performance. Avoid using these log output settings in a production environment.

debug

Specifies the outputs to which an application sends the log events of the Debug level and higher (that is, log events of the *Standard*, *Interaction*, *Trace*, and *Debug* levels). The log output types must be separated by a comma when more than one output is configured.

Default Value: No default value.

Valid Values: Log output types:

- *stdout* - Log events are sent to the Standard output (stdout).
- *stderr* - Log events are sent to the Standard error output (stderr).
- *network* - Log events are sent to Message Server, which can reside anywhere on the network. Message Server stores the log events in the Log Database.
- [filename] - Log events are stored in a file with the specified name. If a path is not specified, the file is created in the application’s working directory.

Changes Take Effect: Immediately

Additional Information: For example: `debug = stdout, logfile`

Note1: The log output options are activated according to the setting of the Verbose configuration option.

Note2: To ease the troubleshooting process, consider using unique names for log files that different applications generate.

Note3: Debug-level log events are never sent to Message Server or stored in the Log Database.

Warning! Directing log output to the console (by using "stdout", "stderr" settings) can affect application performance. Avoid using these log output settings in a production environment.

segment

Specifies whether there is a segmentation limit for a log file. If there is, sets the mode of measurement, along with the maximum size. If the current log segment exceeds the size set by this option, the file is closed and a new one is created. This option is ignored if log output is not configured to be sent to a log file.

Default Value: 100 MB

Valid Values:

- false - No segmentation is allowed.
- <number>[KB] - Sets the maximum segment size, in kilobytes. The minimum segment size is 100 KB.
- <number> MB - Sets the maximum segment size, in megabytes.
- <number> hr - Sets the number of hours for the segment to stay open. The minimum number is 1 hour.

Changes Take Effect: Immediately

expire

Determines whether log files expire. If they do, sets the measurement for determining when they expire, along with the maximum number of files (segments) or days before the files are removed. This option is ignored if log output is not configured to be sent to a log file.

Default Value: 10

Valid Values:

- false - No expiration; all generated segments are stored.
- <number>[file] - Sets the maximum number of log files to store. Specify a number from 1–1000.
- <number> day - Sets the maximum number of days before log files are deleted. Specify a number from 1–100.

Changes Take Effect: Immediately

Additional Information: Note: If an option’s value is set incorrectly (out of the range of valid values), it will be automatically reset to 10.

compress-method

Platform SDK AppTemplate AB specific property to specify method that will be used for archiving log files.

The log option name is case insensitive, and can be "CompressMethod", "compress-method", or "compress_method".

Default Value: none

Valid Values:

- none - No archiving; all generated log files are stored "as-is".
- gzip - GZip archiving is to be used for "historical" log files.
- zip - Zip archiving is to be used for "historical" log files.
- zip<digit> - Zip archiving with given compression level is to be used for "historical" log files.

Changes Take Effect: Immediately
Introduced in release: 9.0.003.01

time-convert

Specifies the system in which an application calculates the log record time when generating a log file. The time is converted from the time in seconds since "00:00:00 UTC, January 1, 1970".

The log option name is case insensitive, and can be: "TimeConvert", "time-convert", or "time_convert".

Default Value: local

Valid Values:

- local - The time of log record generation is expressed as a local time, based on the time zone and any seasonal adjustments. Time zone information of the application’s host computer is used.
- utc - The time of log record generation is expressed as Coordinated Universal Time (UTC).

Changes Take Effect: Immediately

time-format

Specifies how to represent, in a log file, the time when an application generates log records. A log record’s time field in the ISO 8601 format looks like this: "2001-07-24T04:58:10.123".

The log option name is case insensitive, and can be: "TimeFormat", "time-format", or "time_format".

Default Value: time

Valid Values:

- time - The time string is formatted according to "HH:MM:SS.sss" (hours, minutes, seconds, and milliseconds) format.
- locale - The time string is formatted according to the system’s locale.
- iso8601 - The date in the time string is formatted according to the ISO 8601 format. Fractional seconds are given in milliseconds.

Changes Take Effect: Immediately

message-format

Specifies the format of log record headers that an application uses when writing logs in the log file. Using compressed log record headers improves application performance and reduces the log file's size. With the value set to short:

- A header of the log file or the log file segment contains information about the application (such as the application name, application type, host type, and time zone), whereas single log records within the file or segment omit this information.

- A log message priority is abbreviated to Std, Int, Trc, or Dbg, for Standard, Interaction, Trace, or Debug messages, respectively.
- The message ID does not contain the prefix GCTI or the application type ID.

The log option name is case insensitive, and can be: "MessageFormat", "message-format", or "message_format".

Default Value: medium

Valid Values:

- short - An application uses compressed headers when writing log records in its log file.
- medium - An application uses medium size headers when writing log records in its log file.
- full - An application uses complete headers when writing log records in its log file.
- shortcsv - An application uses compressed headers with comma delimiter when writing log records in its log file. This value is only valid for Platform SDK Application Template-based applications.
- shorttsv - An application uses compressed headers with tab char delimiter when writing log records in its log file. This value is only valid for Platform SDK Application Template-based applications.
- shortdsv - An application uses compressed headers with message-header-delimiter delimiter when writing log records in its log file. This value is only valid for Platform SDK Application Template-based applications.
- custom - An application uses custom log messages format, which is separately defined in option custom-message-format or output-pattern. This value is only valid for Platform SDK Application Template-based applications.

Changes Take Effect: Immediately

Additional Information: A log record in the full format looks like this:

```
2002-05-07T18:11:38.196 Standard localhost cfg_dbserver GCTI-00-05060 Application started
```

A log record in the short format looks like this:

```
2002-05-07T18:15:33.952 Std 05060 Application started
```

Note: Whether the full, short, or any other format is used, time is printed in the format specified by the "time-format" option.

use-native-levels

Platform SDK AppTemplate AB specific option.

It enables support of native log levels (like "Error", "Warn", etc) to be used for non-LMS messages in common LMS events formats instead of LMS levels like "Standard", "Interaction", etc.

The log option name is case insensitive, and can be: "UseNativeLevels", "use-native-levels", or "use_native_levels".

Note: This option is experimental and its value procession may get changed.

Introduced in release: 9.0.002.09

message-header-delimiter

Platform SDK AppTemplate AB specific property as a parameter for "shortdsv" message format ("message-format").

The log option name is case insensitive, and can be: "MessageHeaderDelimiter", "message-header-delimiter", or "message_header_delimiter".

Default Value: |

custom-message-format

Platform SDK AppTemplate AB specific option.

Value of this option is used as a log message pattern if "message-format" option value is equal to "custom".

In comparison with "output-pattern", this option provides predefined messages prefix containing timestamp (by "time-format"/"time-convert"), and the LMS-style log level.

The log option name is case insensitive, and can be: "CustomMessageFormat", "custom-message-format", or "custom_message_format".

Note: This option is experimental and its value procession may get changed.

Introduced in release: 9.0.002.01

output-pattern

Platform SDK AppTemplate AB specific option.

Value of this option is used as a log message pattern if "message-format" option value is equal to "custom".

In comparison with "custom-message-format", this option does not provide predefined messages prefix like a timestamp with log level.

The log option name is case insensitive, and can be: "OutputPattern", "output-pattern", or "output_pattern".

Introduced in release: 9.0.002.05

file-encoding

Platform SDK AppTemplate AB specific property for configuration of the log files encoding.

The log option name is case insensitive, and can be: "FileEncoding", "file-encoding", or "file_encoding".

Default Value: UTF-8

file-header-provider

Platform SDK AppTemplate AB specific property for customization of the log files header content.

The log option name is case insensitive, and can be: "FileHeaderProvider", "file-header-provider", or "file_header_provider".

Additional Information: If this option is not specified, then the provider is taken with SPI declaration for the *com.genesyslab.platform.apptemplate.log4j2plugin.FileHeaderProvider* interface.

enable-thread

Specifies whether to enable or disable the logging thread. If set to true (the logging thread is enabled), the logs are stored in an internal queue to be written to the specified output by a dedicated logging thread. This setting also enables the log throttling feature, which allows the verbose level to be dynamically reduced when a logging performance issue is detected.

The log option name is case insensitive, and can be: "EnableThread", "enable-thread", or "enable_thread".

Default Value: false

Valid Values: true, false

Additional Information: Refer to the Framework 8.5 Management Framework User’s Guide for more information about the log throttling feature.

If this option is set to false (the logging thread is disabled), each log is written directly to the outputs by the thread that initiated the log request. This setting also disables the log throttling feature.

Note: Platform SDK AppTemplate AB does not support the log throttling feature.

enable-location-for-thread

Platform SDK AppTemplate AB specific option to enable the call location information passing to the Log4j2 logging thread, which was enabled with option "enable-thread".

The log option name is case insensitive, and can be: "EnableLocationForThread", "enable-location-for-thread", or "enable_location_for_thread".

Additional Information: If one of the layouts is configured with a location-related attribute like HTML locationInfo, or one of the patterns %C or $class, %F or %file, %l or %location, %L or %line, %M or %method, Log4j will take a snapshot of the stack, and walk the stack trace to find the location information.

This is an expensive operation: 1.3 - 5 times slower for synchronous loggers. Synchronous loggers wait as long as possible before they take this stack snapshot. If no location is required, the snapshot will never be taken.

However, asynchronous loggers need to make this decision before passing the log message to another thread; the location information will be lost after that point. The performance impact of taking a stack trace snapshot is even higher for asynchronous loggers: logging with location is 4 - 20 times slower than without location. For this reason, asynchronous loggers do not include location information by default.

log4j2-config-profile

Platform SDK AppTemplate AB specific option.

It defines a base structure and may contain some predefined startup or permanent appenders, and initial custom loggers configurations.

The log option name is case insensitive, and can be: "Log4j2ConfigProfile", "log4j2-config-profile", or "log4j2_config_profile".

default-logdir

Platform SDK AppTemplate AB specific option.

Default root directory for the log files. If specified, it is applied to file names defined in options like "standard", "interaction", "debug", etc.

It's used if log file name is not absolute path, and is not started from "./", or "../".

Note: This option value may be overridden with jvm system property "*default-logdir*".

The log option name is case insensitive, and can be: "DefaultLogdir", "default-logdir", or "default_logdir".

Introduced in release: 9.0.005.00

msgsrv-intMsgsLevel

Platform SDK AppTemplate AB specific property to set log messages filter on Message Server Appender for Platform SDK internal events.

This value should not be lower than INFO level to not cause unlimited recursion/avalanche.

The log option name is case insensitive, and can be: "msgsrv-intMsgsLevel", "msgsrv_intMsgsLevel".

Default Value: info

Valid Values: info, warn, error, fatal, off

Changes Take Effect: Immediately

Introduced in release: 9.0.005.02

message-file

Specifies the file name for application-specific log events. The name must be valid for the operating system on which the application is running. The option value can also contain the absolute path to the application-specific *.lms file. Otherwise, an application looks for the file in its working directory.

The log option name is case insensitive, and can be: "MessageFile", "message-file", or "message_file".

Default Value: As specified by a particular application

Valid Values: Any valid message file ("<filename>.lms")

Changes Take Effect: Immediately, if an application cannot find its "*.lms" file at startup

Additional Information: For example:

```
message-file = my-app.lms
```

Warning! An application that does not find its *.lms file at startup cannot generate application-specific log events and send them to Message Server.

event-log-host

Platform SDK AppTemplate AB specific property to let user applications be able to override the applications' host name in log files and message server events.

It is used by the AppTemplate Log4j2 logging configuration functions in PsdkLog4j2Configuration and Log4j2Configurator.

The log option name (case insensitive): "EventLogHost", "event-log-host", or "event_log_host".

Changes Take Effect: Immediately

Additional Information: For example:

```
event-log-host = node-1-virtual-host
```

Options section "log-extended"

Some of this section options are inherited from the Genesys Common Log Options descriptions, as listed in the Framework 8.5 Configuration Options Reference Manual.

level-reassign-disable

When this option is set to true, the original (default) log level of all log events in the [log-extended] section are restored. This option is useful when you want to use the default levels, but not delete the customization statements.

Default Value: false

Valid Values: true, false

Changes Take Effect: Immediately

level-reassign-<eventID>

Specifies a log level for log event <eventID> that is different than its default level, or disables log event <eventID> completely. If no value is specified, the log event retains its default level.

Default Value: Default value of log event <eventID>

Changes Take Effect: Immediately

Additional Information: This option is useful when you want to customize the log level for selected log events.

These options can be deactivated with the *level-reassign-disable* option.

See Genesys Common Log Options descriptions section of Framework 8.5 Configuration Options Reference Manual for more details on this option.

Example: this example shows customized log level settings, subject to the following log configuration:

```
[log]
verbose     = interaction
all         = stderr
interaction = log_file
standard    = network
```

Before the log levels of the log are changed:

- Log event 1020, with default level standard, is output to stderr and log_file, and sent to Message Server.
- Log event 2020, with default level standard, is output to stderr and log_file, and sent to Message Server.
- Log event 3020, with default level trace, is not generated.
- Log event 4020, with default level debug, is not generated.

Extended log configuration section:

```
[log-extended]
level-reassign-1020 = none
level-reassign-2020 = interaction
level-reassign-3020 = interaction
level-reassign-4020 = standard
```

After the log levels are changed:

- Log event 1020 is disabled and not logged.
- Log event 2020 is output to stderr and log_file.
- Log event 3020 is output to stderr and log_file.
- Log event 4020 is output to stderr and log_file, and sent to Message Server.

logger-<id>

Platform SDK specific extended logging configuration option for applying of custom properties on loggers configuration.

Changes Take Effect: Immediately

Additional Information: The "log-extended" section of CME Application Options may contain multiple declarations of loggers customization records.

For example:

```
[log-extended]
logger-ID1 = "logger1.name: level=debug, additivity=false"
logger-ID2 = "logger2.name: level=error"
```

"ID1" and "ID2" are just unique identifiers of the logger declarations in the options section.

"logger1.name" and "logger2.name" are loggers names in a common log4j sense of loggers naming convention.

These values will be applied to the internally generated log4j2 configuration in the following way (the declarations will be created, or adjusted, if already exist):

```
  <Loggers>
    <Logger name="logger1.name" level="debug" additivity="false" />
    <Logger name="logger2.name" level="error" />
  </Loggers>
```

It's included in GAppLogExtOptions, which may contain list of such descriptions, and is a part of *GAppLoggingOptions*. The custom logger declaration supports following (optional) properties:

- level - the Level to be associated with the Logger;
- additivity ("true"/"false") - true if the logger should be additive, false otherwise;
- includeLocation ("true"/"false") - whether location should be passed downstream.

Advanced Platform SDK Topics

Advanced Platform SDK Topics

The following articles provide details about advanced Platform SDK features you may want to take advantage of:

- Using Kerberos Authentication in Platform SDK
- Secure Connections Using TLS
    - Quick Start
    - Using the Platform SDK Commons Library
    - Using the Application Template Application Block
    - Configuring TLS Parameters in Configuration Manager
    - Using and Configuring Security Providers
    - OpenSSL Configuration File
    - Use Cases
    - Using and Configuring TLS Protocol Versions
- Lazy Parsing of Message Attributes
- Log Filtering
- Hide or Tag Sensitive Data in Logs
- Profiling and Performance Services
- IPv6 Resolution
- Managing Protocol Configuration
- Friendly Reaction to Unsupported Messages
- Creating Custom Protocols
- JSON Support in Java
- Working with Custom Servers
- Bidirectional Messaging
- Hide Message Attributes in Logs
- Resources Releasing in an Application Container
- Transport Layer Substitution

Using Kerberos Authentication in Platform SDK

Java

Introduction

Platform SDK supports using Kerberos authentication with Configuration Server. Platform SDK can independently obtain a Kerberos ticket or use Kerberos ticket provided by user. Each case requires an individual approach.

Using Service Principal Name

Service Principal Name (SPN) is a unique identifier of service which in couples with user’s credentials can uniquely identify access to requested service. To use the `ServicePrincipalName` user have to assign it using `setSPN` method of a channel `Endpoint`.

Microsoft-specific Note: SPN has to be registered in Active Directory using utility `setspn.exe`. See Microsoft technet documentation. User has to have the required access rights to execute this utility's commands.

Code example: Connect CS using SPN

```
ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint(host, port).setSPN(spn));
protocol.setClientName(clientName);
protocol.setClientApplicationType(clientType);

protocol.open();
```

Usage of Independently Acquired Ticket

If user has a ticket as byte array data, Platform SDK can use it too. In this case user has to assign ticket acquirer to the protocol instance.

Code example: Connect to CS using raw data GSS Kerberos ticket

```
ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint(host, port));
protocol.setClientName(clientName);
protocol.setClientApplicationType(clientType);
RawDataTicketAcquirer ticketAcquirer = new RawDataTicketAcquirer(ticketBytes);
```

```
protocol.setTicketAcquirer(ticketAcquirer);

protocol.Open();
```

The previous example applies only for tickets compatible with GSS API (RFC 2743). Configuration Server also supports pure Kerberos tickets without a GSS envelope, as obtained by using the MIT Kerberos library for instance.

In this case please use the second constructor of `RawDataTicketAcquirer`:

```
RawDataTicketAcquirer(byte[] arguments, bool isGSSTicket)
```

If `isGSSTicket` is false, then a registration message is created with another attribute specially designed for this goal.

Code example: Connect to Configuration Server Using Raw Data Pure Kerberos Ticket

```
boolean isGSSTicket = false;
ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint(host, port));
protocol.setClientName(clientName);
protocol.setClientApplicationType(clientType);
RawDataTicketAcquirer ticketAcquirer = new RawDataTicketAcquirer(ticketBytes, isGSSTicket);
protocol.setTicketAcquirer(ticketAcquirer);

protocol.Open();
```

Notes for Windows

Kerberos authorization as current logined user must be enabled manually in few steps:

1. set registry key "AllowTGTSessionKey"=dword:00000001 in HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\Kerberos\Parameters
2. update JCE policy from oracle site
   - jre1.7/lib/security <- http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html
   - jre1.8/lib/security <- http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html
3. Time on kdc,server and client machines must be seconds synchronized
4. krb5.conf can be placed in any folder but you must specify its location using system property "java.security.krb5.conf"

.NET

Introduction

Platform SDK supports using Kerberos authentication with Configuration Server. Platform SDK can independently obtain a Kerberos ticket or use Kerberos ticket provided by user. Each case requires an individual approach.

Using Service Principal Name

Service Principal Name (SPN) is a unique identifier of service which in couples with user's credentials can uniquely identify access to requested service. To use the SPN user have to assign field `ServicePrincipalName` of `AbstractChannel.Endpoint`.

Microsoft-specific Note: SPN has to be registered in Active Directory using utility `setspn.exe`. See Microsoft technet documentation. User has to have the required access rights to execute this utility's commands.

Code example: Connect CS using SPN

```
var protocol = new ConfServerProtocol(new Endpoint(host, port) { ServicePrincipalName = spn })
{
  ClientApplicationType = clientApp,
  ClientName = clientName
};

protocol.Open();
```

Usage of Independently Acquired Ticket

If user has a ticket as byte array data Platform SDK can use it too. In this case user has to assign ticket acquirer to the protocol instance.

Code example: Connect to CS using raw data GSS Kerberos ticket

```
var protocol = new ConfServerProtocol(new Endpoint(host, port))
{
  ClientApplicationType = clientApp,
  ClientName = clientName,
  KerberosTicketAcquirer = new RawDataTicketAcquirer(rawTicketData)
};

protocol.Open();
```

The previous example applies only for tickets compatible with GSS API (RFC 2743). Configuration Server also supports pure Kerberos tickets without a GSS envelope, as obtained by using the MIT Kerberos library for instance.

In this case please use the second constructor of `RawDataTicketAcquirer`:

```
RawDataTicketAcquirer(byte[] arguments, bool isGSSTicket)
```

If `isGSSTicket` is false, then a registration message is created with another attribute specially designed for this goal.

Code example: Connect to Configuration Server Using Raw Data Pure Kerberos Ticket

```
var isGSSTicket = false;
var protocol = new ConfServerProtocol(new Endpoint(host, port))
{
  ClientApplicationType = clientApp,
  ClientName = clientName,
  KerberosTicketAcquirer = new RawDataTicketAcquirer(rawTicketData, isGSSTicket)
};

protocol.Open();
```

Secure connections using TLS

Java

This page provides an introduction to creating and configuring Transport Layer Security (TLS) for your Platform SDK connections, as introduced in release 8.1.1.

Introduction to TLS

This page provides an overview of the TLS implementation provided in the 8.1.1 release of Platform SDK. It introduces Platform SDK users to TLS concepts and then provides links to expanded articles and examples that describe implementation details.

Before working with TLS to create secure connections, you should have a basic awareness of how public key cryptography works.

Certificates

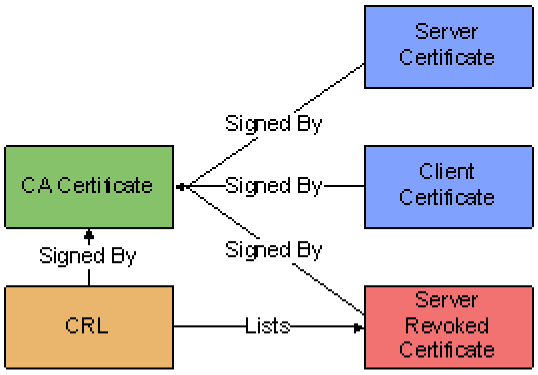
Transport Layer Security (TLS) technology uses public key cryptography, where the key required to encrypt and decrypt information is divided into two parts: a public key and a private key. These parts are reciprocal in the sense that data encrypted using a private key can be decrypted with the public key and vice versa, but cannot be decrypted using the same key that was used for encryption.

There is an X.509 standard for public key (certificate) format, and public-key cryptography standards (PKCS) that define format for private key (PKCS#8) and related data structures.

Certificate Authority (CA)

In the context of TLS, a CA is an entity that is trusted by both sides of network connection. Each CA has a public X.509 certificate and owns a related private key that kept secret. A CA can generate and sign certificates for other parties using its private key, and then that CA certificate can be used by the parties to validate their certificates. A CA can also issue public Certificate Revocation Lists (CRLs), which are also used by parties for certificate validation.

The relation between certificates and CRL can be depicted like this:



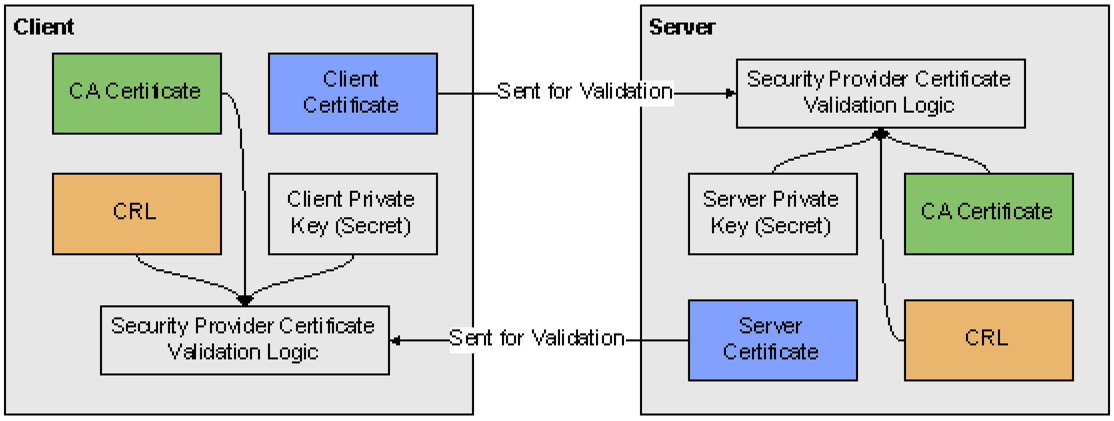
Certificate Usage

To create a secure connection, each party must have a copy of:

- a public CA certificate
- a CRL issued by the CA
- their own public certificate (with a corresponding private key)

When a network connection is established, the client initiates a TLS handshake process during which the parties exchange their public certificates, prove that they own corresponding private keys, create a shared session encryption key, and negotiate which cipher suite will be used.

Placement and exchange of certificate data is shown on the following diagram:


Client
CA Certificate
Client Certificate
CRL
Client Private Key (Secret)
Security Provider Certificate Validation Logic
Sent for Validation
Server
Security Provider Certificate Validation Logic
Server Private Key (Secret)
CA Certificate
Server Certificate
CRL
Sent for Validation

TLS only requires that servers send their certificates, but the client certificates can also be exchanged depending on server settings. Cases where the client certificates are demanded by the server are called “Mutual TLS”, as both sides send their certificates.

If all certificates pass validation and the ciphers are negotiated successfully, then a TLS connection is established and higher-level protocols may proceed.

Implementing and Configuring TLS

Genesys strongly recommends reading all TLS in Platform SDK articles in order to get understanding of how TLS works in general and how it is supported in Platform SDK. A Quick Start page is provided for reference, but the specific implementation details and expanded information provided in other pages will help you to better understand how to provide TLS support in your applications. Once you have an understanding of how TLS is implemented, you can use the Use Case guide to quickly find code snippets or relevant links for common tasks.

There are two main ways to implement TLS in your Platform SDK code:

1. Use the Platform SDK Commons Library to specify TLS settings directly when creating endpoints
2. Use the Application Template Application Block to read connection parameters inside configuration objects retrieved from Configuration Server, then use those parameters to configure TLS settings.

Note: If using the Application Template Application Block, you will need to configure TLS Parameters in Configuration Manager before the application is tested.

Recommendations are also provided for the configuration and use of security providers. The security providers discussed on that page have been tested within the described configurations, and worked reliably.

Migrating TLS Support From Platform SDK Release 8.1.0

This section outlines migration information that may be needed for applications that were developed using the TLS implementation provided with the 8.1.0 release of Platform SDK.

Platform SDK 8.1.0 had the following connection configuration parameters for TLS:

- *Connection.TLS_KEY*
- *Connection.SSL_KEYSTORE_PATH_KEY*
- *Connection.SSL_KEYSTORE_PASS*

The *TLS_KEY* parameter is the equivalent of *enableTls* flag in the current release, while the other parameters specified the location and password for the Java keystore file containing certificates that were used by the application to authenticate itself. TLS configuration code looked like this:

```
ConnectionConfiguration connConf = new KeyValueConfiguration(new KeyValueCollection());
connConf.setOption(Connection.TLS_KEY, "1");
connConf.setOption(Connection.SSL_KEYSTORE_PATH_KEY, "c:/certificates/client-certs.keystore");
connConf.setOption(Connection.SSL_KEYSTORE_PASS, "pa$$w0rd");
```

In Platform SDK 8.1.1, this code can be translated to the following:

```
boolean tlsEnabled = true;
// By default, PSDK 8.1.0 trusted any certificate
TrustManager trustManager = TrustManagerHelper.createTrustEveryoneTrustManager();
// Keystore entries may be protected with individual password,
// but usually, these passwords are the same as keystore password
KeyManager keyManager = KeyManagerHelper.createJKSKeyManager(
    "c:/certificates/client-certs.keystore", "pa$$w0rd".toCharArray(),
"pa$$w0rd".toCharArray());
SSLContext sslContext = SSLContextHelper.createSSLContext(keyManager, trustManager);
```

In most cases, certificates from other parties will need to be validated. Assuming there is a separate keystore file with a CA certificate, this can be achieved with the following code:

```
TrustManager trustManager = TrustManagerHelper.createJKSTrustManager(
     "c:/certificates/CA-cert.keystore", "pa$$w0rd".toCharArray(), null, null);
```

Please note that different keystore files are used for the *KeyManager* and *TrustManager* objects. For more information, see Using the Platform SDK Commons Library.

Known Issues

For more details about the known issues listed here, refer to Using and Configuring Security Providers.

- Java 7:
  - CRL files without extension section cannot be loaded: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7166885.
    Note: Although the bug is marked as "Will not fix", it seems to be fixed since Java 7 update 7.
  - CRLs located in WCS are ignored, please use CRLs as files.
- MSCAPI: MSCAPI does not have a documented way to programmatically set passwords to the private key stored in WCS. Regardless of the password returned by `CallbackHandler`, if the private key is protected with a confirmation prompt or password prompt then the user will be shown an OS-specific popup dialog.

.NET

This page provides an introduction to creating and configuring Transport Layer Security (TLS) for your Platform SDK connections, as introduced in release 8.1.1.

Introduction to TLS

This page provides an overview of the TLS implementation provided in the 8.1.1 release of Platform SDK. It introduces Platform SDK users to TLS concepts and then provides links to expanded articles and examples that describe implementation details.

Before working with TLS to create secure connections, you should have a basic awareness of how public key cryptography works.

Certificates

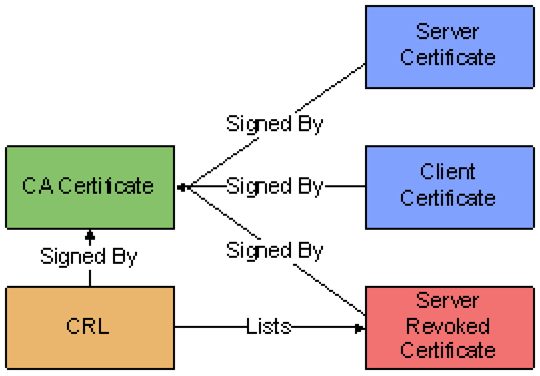
Transport Layer Security (TLS) technology uses public key cryptography, where the key required to encrypt and decrypt information is divided into two parts: a public key and a private key. These parts are reciprocal in the sense that data encrypted using a private key can be decrypted with the public key and vice versa, but cannot be decrypted using the same key that was used for encryption.

There is an X.509 standard for public key (certificate) format, and public-key cryptography standards (PKCS) that define format for private key (PKCS#8) and related data structures.

Certificate Authority (CA)

In the context of TLS, a CA is an entity that is trusted by both sides of network connection. Each CA has a public X.509 certificate and owns a related private key that kept secret. A CA can generate and sign certificates for other parties using its private key, and then that CA certificate can be used by the parties to validate their certificates. A CA can also issue public Certificate Revocation Lists (CRLs), which are also used by parties for certificate validation.

The relation between certificates and CRL can be depicted like this:



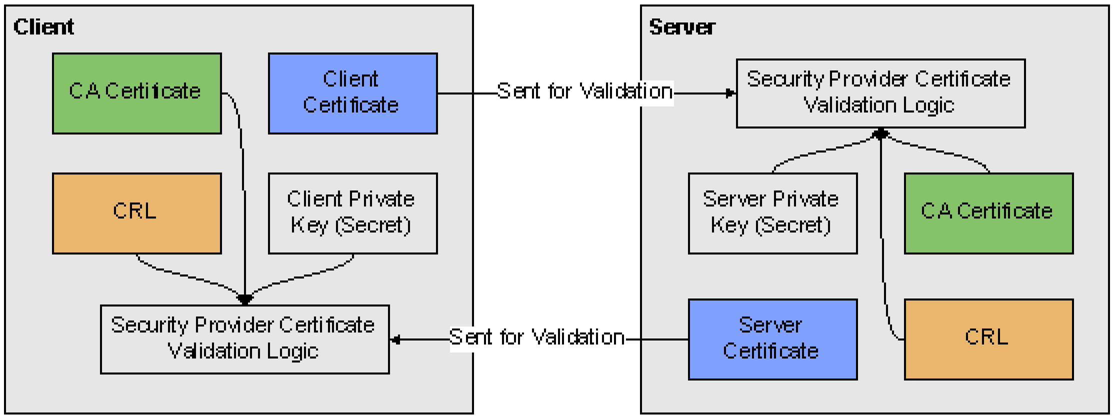
Certificate Usage

To create a secure connection, each party must have a copy of:

- a public CA certificate
- a CRL issued by the CA
- their own public certificate (with a corresponding private key)

When a network connection is established, the client initiates a TLS handshake process during which the parties exchange their public certificates, prove that they own corresponding private keys, create a shared session encryption key, and negotiate which cipher suite will be used.

Placement and exchange of certificate data is shown on the following diagram:


Client
CA Certificate
Client Certificate
CRL
Client Private Key (Secret)
Security Provider Certificate Validation Logic
Sent for Validation
Server
Security Provider Certificate Validation Logic
Server Private Key (Secret)
CA Certificate
Server Certificate
CRL
Sent for Validation

TLS only requires that servers send their certificates, but the client certificates can also be exchanged depending on server settings. Cases where the client certificates are demanded by the server are called “Mutual TLS”, as both sides send their certificates.

If all certificates pass validation and the ciphers are negotiated successfully, then a TLS connection is established and higher-level protocols may proceed.

Migrating TLS Support From Platform SDK Release 8.1.0

This section outlines migration information that may be needed for applications that were developed using the TLS implementation provided with the 8.1.0 release of Platform SDK.

There were no significant changes to interfaces for the .NET version of Platform SDK 8.1.1, so the same code would work for 8.1.0 and later releases:

```
KeyValueConfiguration config = new KeyValueConfiguration(new KeyValueCollection());
config.TlsEnabled = true;
config.TlsCertificate = "29 3f 0d d9 65 a1 a9 92 dd 1c 8c 2a e7 20 74 06 c5 ba 0f 10";
Endpoint ep = new Endpoint(AppName, Host, Port, config);
```

Quick Start

Platform SDK for Java

Understanding Port Modes

TLS is configured differently depending on target port mode:

- default - Default mode ports do not use or understand TLS protocol.
- upgrade - Upgrade mode ports allow unsecured connections to be made, switching to TLS mode only after TLS settings are retrieved from Configuration Server.
- secure - Secure mode ports require TLS to be started immediately, before sending any requests to server.

Connecting to Default Mode Ports

Default mode is supported for all protocols; no specific configuration is needed for it to work.

Example:

```
Endpoint cfgServerEndpoint = new Endpoint(appName, cfgHost, cfgPort);
ConfServerProtocol protocol = new ConfServerProtocol(cfgServerEndpoint);
protocol.setClientName(appName);
protocol.setClientApplicationType(appType);
protocol.setUserName(username);
protocol.setUserPassword(password);
protocol.open();
```

It is also OK to specify explicit null parameters for the connection configuration and TLS parameters:

```
// Explicit null ConnectionConfiguration
Endpoint cfgServerEndpoint = new Endpoint(appName, cfgHost, cfgPort, null);

// Explicit null ConnectionConfiguration and TLS parameters
Endpoint cfgServerEndpoint = new Endpoint(appName, cfgHost, cfgPort, null, false, null, null);
```

Connecting to Upgrade Mode Ports

TLS upgrade mode is supported only for Configuration Protocol, since the TLS settings for connecting clients must be retrieved from Configuration Server. No specific options are required; the TLS upgrade logic works by default.

If a user has provided custom settings, then those settings are used if the TLS parameters received from Configuration Server are empty. The only requirement that the *tlsEnabled* parameter in the Endpoint constructor is **not** to true, otherwise the client side starts TLS immediately and the connection would fail because an upgrade mode port expects the connection to be unsecured initially.

```
// Setting tlsEnabled to true would cause failure when connecting to upgrade port:
Endpoint cfgServerEndpoint = new Endpoint(appName, cfgHost, cfgPort,
    connConf, true, sslContext, sslOptions);
```

Connecting to Secure Mode Port

Secure mode is supported for all protocols. TLS configuration objects/properties must be specified before the connection is opened, and the *tlsEnabled* parameter must be set to true. Secure port mode expects the client to start TLS negotiation immediately after connecting, otherwise the connection fails.

Example:

```
boolean tlsEnabled = true;
// Here, the minimal TLS configuration is used, see the following section for details
TrustManager trustManager = TrustManagerHelper.createTrustEveryoneTrustManager();
KeyManager keyManager = KeyManagerHelper.createEmptyKeyManager();
SSLContext sslContext = SSLContextHelper.createSSLContext(keyManager, trustManager);
ConnectionConfiguration connConf = new KeyValueConfiguration(new KeyValueCollection());
Endpoint cfgServerEndpoint = new Endpoint(appName, cfgHost, cfgPort,
    connConf, tlsEnabled, sslContext, sslOptions);
ConfServerProtocol protocol = new ConfServerProtocol(cfgServerEndpoint);
protocol.setClientName(appName);
protocol.setClientApplicationType(appType);
protocol.setUserName(username);
protocol.setUserPassword(password);
protocol.open();
```

TLS Minimal Configuration

Frequently, there is a need to quickly set up code for working TLS connections, dealing with detailed TLS configuration later. The minimal configuration settings described below do exactly that.

The following code creates an *SSLContext* object that can be used to configure a connection to a secure port or to configure a secure server socket. This code uses *EmptyKeyManager* which indicates that the party opening connection/socket would not have any certificate to authenticate itself, and *TrustEveryoneTrustManager* which trusts any certificate presented by the other party - even expired or revoked certificates.

```
boolean tlsEnabled = true;
TrustManager trustManager = TrustManagerHelper.createTrustEveryoneTrustManager();
KeyManager keyManager = KeyManagerHelper.createEmptyKeyManager();
SSLContext sslContext = SSLContextHelper.createSSLContext(keyManager, trustManager);
```

Note: Connections using this configuration would have a working encryption layer, but they are not secure because they can neither authenticate themselves nor validate credentials provided by the other party.

Note: If a server uses mutual TLS mode, then it requires the client to present a certificate. Minimal configuration does not have certificates, so in this case the TLS negotiation would fail.

Platform SDK for .NET

Understanding Port Modes

TLS is configured differently depending on target port mode:

- default - Default mode ports do not use or understand TLS protocol.
- upgrade - Upgrade mode ports allow unsecured connections to be made, switching to TLS mode only after TLS settings are retrieved from Configuration Server.
- secure - Secure mode ports require TLS to be started immediately, before sending any requests to server.

Connecting to Default Mode Ports

Default mode is supported for all protocols; no specific configuration is needed for it to work.

Example:

```
Endpoint cfgServerEndpoint = new Endpoint(appName, cfgHost, cfgPort);
ConfServerProtocol protocol = new ConfServerProtocol(cfgServerEndpoint);
protocol.ClientName = appName;
protocol.ClientApplicationType = appType;
protocol.UserName = username;
protocol.UserPassword = password;
protocol.Open();
```

It is also OK to specify explicit null parameters for the connection configuration:

```
// Explicit null IConnectionConfiguration
Endpoint cfgServerEndpoint = new Endpoint(appName, cfgHost, cfgPort, null);
```

Connecting to Upgrade Mode Ports

TLS upgrade mode is supported only for Configuration Protocol, since the TLS settings for connecting clients must be retrieved from Configuration Server. No specific options are required; the TLS upgrade logic works by default.

If a user has provided custom settings, then those settings are used if the TLS parameters received from Configuration Server are empty. The only requirement that the *TlsEnabled* parameter in the connection configuration is **not** to true, otherwise the client side starts TLS immediately and the connection would fail because an upgrade mode port expects the connection to be unsecured initially.

```
// Setting TlsEnabled to true would cause failure when connecting to upgrade port:
KeyValueConfiguration connConf = new KeyValueConfiguration(new KeyValueCollection());
connConf.TlsEnabled = true;
Endpoint cfgServerEndpoint = new Endpoint(appName, cfgHost, cfgPort, connConf);
```

Connecting to Secure Mode Port

Secure mode is supported for all protocols. TLS configuration objects/properties must be specified before the connection is opened, and the *TlsEnabled* parameter must be set to true. Secure port mode expects the client to start TLS negotiation immediately after connecting, otherwise the connection fails.

Example:

```
boolean tlsEnabled = true;
// Here, the minimal TLS configuration is used, see the following section for details
KeyValueConfiguration connConf = new KeyValueConfiguration(new KeyValueCollection());
connConf.TlsEnabled = true;
Endpoint cfgServerEndpoint = new Endpoint(appName, cfgHost, cfgPort, connConf);
ConfServerProtocol protocol = new ConfServerProtocol(cfgServerEndpoint);
protocol.ClientName = appName;
protocol.ClientApplicationType = appType;
protocol.UserName = username;
protocol.UserPassword = password;
protocol.Open();
```

TLS Minimal Configuration

Frequently, there is a need to quickly set up code for working TLS connections, dealing with detailed TLS configuration later. The minimal configuration settings described below do exactly that.

Platform SDK for .Net requires less configuration, because it always uses the MSCAPI security provider and Windows Certificate Services (WCS) by default. The following code would trust all certificates located in the WCS Trusted Root Certificates folder for the current user account.

```
KeyValueConfiguration config = new KeyValueConfiguration(new KeyValueCollection());
config.TlsEnabled = true;
Endpoint ep = new Endpoint(appName, cfgHost, cfgPort, config);
```

Note: If a server uses mutual TLS mode, then it requires clients to present a certificate. Minimal configuration does not have certificates, so in this case the TLS negotiation would fail.

TLS and the Platform SDK Commons Library

Platform SDK for Java

Important
The contents of this page only apply to Java implementations.

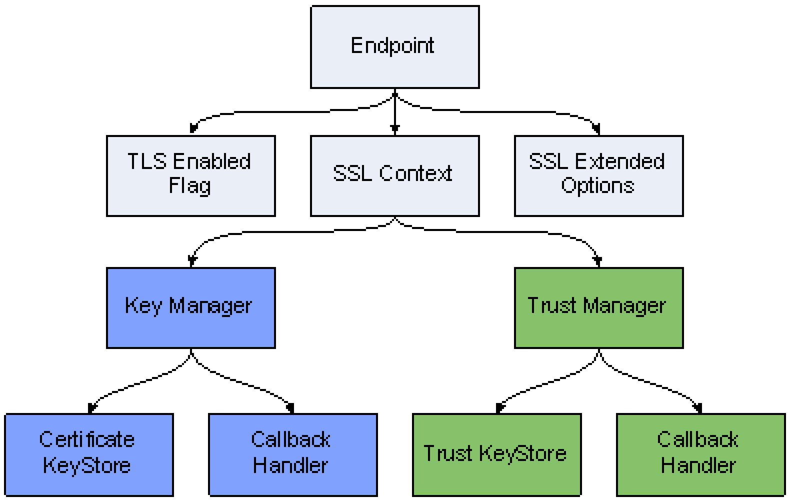
Using the Platform SDK Commons Library to Configure TLS

Starting with Platform SDK 8.1.1, the only way to configure connections is by using `Endpoint` objects, which contain all parameters related to the endpoint connection—including TLS parameters that indicate whether TLS is enabled and provide details about the SSL context and extended options.

Tip
In earlier releases, Platform SDK provided three ways to configure connections:

- using `ConnectionConfiguration` objects passed to `Protocol` constructors
- setting parameters in the protocol context
- adding a textual parameter representation to the URL query

The following diagrams show interdependencies among the Platform SDK objects used to establish network connections and support TLS.



TLS Configuration Objects Containment Hierarchy

This page outlines each step required to create supporting objects for a TLS-enabled `Endpoint`.

Callback Handlers

In many cases, certificate or key storage is password-protected. This means that Platform SDK will need the password to access storage. The Java `CallbackHandler` interface offers a flexible way to pass this type of credential data:

```
package javax.security.auth.callback;
...
public interface CallbackHandler {
    void handle(Callback[] callbacks)
        throws java.io.IOException, UnsupportedCallbackException;
}
```

The `handle()` method accepts credential requests in the form of `Callback` objects that have appropriate setter methods. The most common callback implementation is `PasswordCallback`. User code may use a GUI to ask the end user to:

- enter a password
- retrieve a password from a file, pipe, network, and so on

Here is an example of a `CallbackHandler` delegating password retrieval to a GUI:

```
CallbackHandler callbackHandler = new CallbackHandler() {
    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        for (Callback c : callbacks) {
            if (c instanceof PasswordCallback) {
                PasswordCallback p = (PasswordCallback) c;
                p.setPassword(gui.getKeyStorePassword());
            }
        }
    }
};
```

When No Password is Required

In some cases, certificate storage does not need a password. The API may still dictate that a `CallbackHandler` be provided however, so the Platform SDK includes a predefined class that can be used as a "dummy" `CallbackHandler` for this scenario:

```
com.genesyslab.platform.commons.connection.tls.DummyPasswordCallbackHandler
```

Here is an example of using this dummy class:

```
CallbackHandler callbackHandler = new DummyPasswordCallbackHandler();
```

Key Managers

Java provides a `KeyManager` interface. This interface defines functionality that can be used to load and contain certificates or keys, or to select appropriate certificates or keys.

Classes based on the `KeyManager` interface are used by Java TLS support to retrieve certificates that will be sent over the network to a remote party for validation. They are also used to retrieve the corresponding private keys. On the client side, `KeyManager` classes retrieve client certificates or keys; on the server side they retrieve server certificates or keys.

The Platform SDK Commons library has a helper class, `KeyManagerHelper`, which makes it easy to create key managers using several types of key stores and security providers. The built-in key manager types are:

- **PEM** — reads certificate/key pairs from X.509 PEM files.
- **MSCAPI** — uses the Microsoft CryptoAPI and Windows certificate services to retrieve certificate/key pairs.
- **PKCS11** — delegates to an external security provider plugged in via the PKCS#11 interface, for example, Mozilla NSS.
- **JKS** — retrieves a certificate/key pair from a Java Keystore file.
- **Empty** — does not retrieve anything. This type is for use as a dummy key manager. For example, clients that do not have certificates can use it.

Here are some examples of key manager creation:

```
// From PEM file
X509ExtendedKeyManager km = KeyManagerHelper.createPEMKeyManager(
        "c:/cert/client-cert.pem", "c:/cert/client-cert-key.pem");

// From MSCAPI
CallbackHandler cbh = new DummyPasswordCallbackHandler();
// Whitespace characters are allowed anywhere inside the string
String certThumbprint =
        "4A 3F E5 08 48 3A 00 71 8E E6 C1 34 56 A4 48 34 55 49 D9 0E";
X509ExtendedKeyManager km = KeyManagerHelper.createMSCAPIKeyManager(
        cbh, certThumbprint);

// From PKCS11
// This provider does not allow customization of Key Manager
// This is required for FIPS-140 certification
// Dummy callback handler will not work, must use strong password
CallbackHandler passCallback = ...;
```

```
X509ExtendedKeyManager km = KeyManagerHelper.createPKCS11KeyManager(
        passCallback);

// From JKS
// JKS key store does not allow callback usage (bug in Java?)
// Individual entries in JKS key store can be password-protected
char[] keyStorePass = "keyStorePass".toCharArray();
char[] entryPass = "entryPass".toCharArray();
X509ExtendedKeyManager km = KeyManagerHelper.createJKSKeyManager(
        "c:/cert/client-cert.jks", keyStorePass, entryPass);

// Empty key manager
// Using KeyManagerHelper class
X509ExtendedKeyManager km1 = KeyManagerHelper.createEmptyKeyManager();
// Direct creation
X509ExtendedKeyManager km2 = new EmptyX509ExtendedKeyManager();
```

Trust Managers

A Trust Manager is an entity that decides which certificates from a remote party are to be trusted. It performs certificate validation, checks the expiration date, matches the host name, checks the certificate against a CRL list, and builds and validates the chain of trust. The chain of trust starts from a certificate trusted by both sides (for example, a CA certificate) and continues with second-level certificates signed by CA, then possibly with third-level certificates signed by second-level authorities and so on. Chain length can vary, but Platform SDK was designed to explicitly support two-level chains consisting of a CA certificate and a leaf certificate signed by CA.

Trust manager instances are created based on storage that contains trusted certificates. The number of trusted certificates can vary depending on the type of trust manager being used. With PEM files, the storage contains only a single CA certificate; other provider types can have larger sets of trusted certificates.

The Platform SDK Commons library has a helper class, `TrustManagerHelper`, which makes it easy to create trust managers that use several types of certificate stores and security providers, and which can accept additional parameters that affect certificate validation. Built-in trust manager types are:

- **PEM** — Reads a CA certificate from an X.509 PEM file.
- **MSCAPI** — Uses the Microsoft CryptoAPI and Windows certificate services to retrieve CA certificates and validate certificates.
- **PKCS11** — Delegates certificate validation to an external security provider plugged in via the PKCS#11 interface, for example, Mozilla NSS.
- **JKS** — Retrieves a CA certificate from a Java Keystore file and uses Java built-in validation logic.
- **Default** — Uses trusted certificates shipped with or configured in Java Runtime and Java built-in validation logic.
- **TrustEveryone** — Trusts any certificates. Can be used on the server side when you do not expect any certificates from clients, or during testing.

Here are some examples of trust manager creation (with generic `crlPath` and `expectedHostName` parameters defined in the first example):

```
// Generic parameters for trust manager examples
String crlPath = "c:/cert/ca-crl.pem";
String expectedHostName = "serverhost";
// From PEM file
```

```
X509TrustManager tm = TrustManagerHelper.createPEMTrustManager(
        "c:/cert/ca.pem", crlPath, expectedHostName);

// From MSCAPI
// CRL is loaded from PEM file (Platform SDK supports only file-base CRLs)
// Concrete CA is not specified, all certificates from WCS Trusted Root are used
CallbackHandler cbh = new DummyPasswordCallbackHandler();
X509TrustManager tm = TrustManagerHelper.createMSCAPITrustManager(
        cbh, crlPath, expectedHostName);

// From PKCS#11
// This provider implementation in Java does not allow custom host name check,
// but CRL can still be used
X509TrustManager tm = TrustManagerHelper.createPKCS11TrustManager(
        cbh, crlPath);

// From JKS
// JKS key store does not allow callback usage (bug in Java?)
// Certificate-only entries cannot have passwords in JKS key store
// CRL and host name check are supported
char[] keyStorePass = "keyStorePass".toCharArray();
X509ExtendedKeyManager km = KeyManagerHelper.createJKSTrustManager(
        "c:/cert/ca-cert.jks", keyStorePass, crlPath, expectedHostName);

// From Java built-in trusted certificates
// This one does not support CRL and host name check
X509ExtendedKeyManager km = KeyManagerHelper.createDefaultTrustManager();

// Trust Everyone
X509ExtendedKeyManager km =
        KeyManagerHelper.createTrustEveryoneTrustManager();
```

SSLContext and SSLExtendedOptions

An `SSLContext` instance serves as a container for all SSL and TLS parameters and objects and also as a factory for `SSLEngine` instances.

`SSLEngine` instances contain logic that deals directly with TLS handshaking, negotiation, and data encryption and decryption. `SSLEngine` instances are not reusable and must be created anew for each connection. This is a good reason for requiring users to provide an `SSLContext` instance rather than an instance of `SSLEngine`. `SSLEngine` instances are created by the Platform SDK connection layer and are not exposed to user code.

Only some of the parameters for `SSLEngine` can be pre-set in `SSLContext`. However, the `SSLExtendedOptions` class may be used to collect additional parameters.

`SSLExtendedOptions` currently contains two parameters:

- the "mutual TLS" flag
- a list of enabled cipher suites

The mutual TLS flag is used only by server applications. When the flag is turned on, the server will require connecting clients to send their certificates for validation. The connections of any clients that do not send certificates will fail.

The list of enabled cipher suites contains the names of all cipher suites that will be used as filters for `SSLEngine`. As a result, only ciphers that are supported by `SSLEngine` and that are contained in the enabled cipher suites list will be enabled for use.

Platform SDK includes the `SSLContextHelper` helper class to support one-line creation of `SSLContext` and `SSLExtendedOptions` instances.

Here are some examples:

```
// Creating SSLContext
KeyManager km = ...;
TrustManager tm = ...;
SSLContext sslContext = SSLContextHelper.createSSLContext(km, tm);

String[] cipherList = new String[] {
        "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA",
        "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA",
        "TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA"};
// Can be single String with space-separated suite names
String cipherNames = "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA " +
        "TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA " +
        "TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA";
boolean mutualTLS = false;

// Creating SSLExtendedOptions directly
SSLExtendedOptions sslOpts1 =
        new SSLExtendedOptions(mutualTLS, cipherList);
SSLExtendedOptions sslOpts2 =
        new SSLExtendedOptions(mutualTLS, cipherNames);

// Create SSLExtendedOptions using the helper class:
SSLExtendedOptions sslOpts3 =
        SSLContextHelper.createSSLExtendedOptions(mutualTLS, cipherList);
SSLExtendedOptions sslOpts4 =
        SSLContextHelper.createSSLExtendedOptions(mutualTLS, cipherNames);
```

Endpoints

Now that supporting objects have been created and configured, you are ready to create an `Endpoint`.

The connection configuration parameters of an `Endpoint` are read-only—they cannot be changed after the `Endpoint` is created. This configuration information is then used by `Protocol` instances, the warm standby service, the connection layer and the TLS layer.

A sample `Endpoint` configuration is shown below:

```
ConnectionConfiguration connConf = ...;
SSLContext sslContext = ...;
SSLExtendedOptions sslOpts = ...;
tlsEnabled = true;
// Specifying host name and port.
Endpoint ep1 = new Endpoint("Server-1", "serverhost", 9090, connConf,
        tlsEnabled, sslContext, sslOpts);
// Specifying URI. Query part is still supported.
String uri = "tcp://Server-1@serverhost:9090/" +
        "?protocol=addp&addp-remote-timeout=5&addp-trace=remote";
Endpoint ep2 = new Endpoint("Server-1", uri, connConf,
        tlsEnabled, sslContext, sslOpts);
```

Note: Configuration parameters can be set directly in a `Protocol` instance context, but will be overwritten and lost under the following conditions:

- a new `Endpoint` is set up
- the protocol is forced to reconnect
- a warm standby switchover occurs

Configuring TLS for Client Connections

Using the information above, you are now ready to configure actual client connections.

Example:

```
// Get TLS configuration objects for connection
String clientName = "ClientApp";
String host = "serverhost";
int port = 9000;
SSLContext sslContext = ...; // Assume it is created
SSLExtendedOptions sslOptions = ...; // Assume it is created
boolean tlsEnabled = true;

ConnectionConfiguration connConf = new KeyValueConfiguration(new KeyValueCollection());
Endpoint epTSrv = new Endpoint(
        clientName, host, port, connConf, tlsEnabled, sslContext, sslOptions);

TServerProtocol tsProtocol = new TServerProtocol(epTSrv);
tsProtocol.setClientName(clientName);
tsProtocol.open();
```

Configuring TLS for Servers

Using the information above, you are now ready to configure actual server connections.

```
String serverName = "ServerApp";
String host = "serverhost";
int port = 9000;
SSLContext sslContext = ...; // Assume it is created
SSLExtendedOptions sslOptions = ...; // Assume it is created
boolean tlsEnabled = true;

ConnectionConfiguration connConf = new KeyValueConfiguration(new KeyValueCollection());
Endpoint epTSrv = new Endpoint(
        serverName, host, port, connConf, tlsEnabled, sslContext, sslOptions);

ExternalServiceProtocolListener serverChannel =
        new ExternalServiceProtocolListener(endpoint);
```

Parameter-based TLS Configuration

Platform SDK has a way to create TLS objects based on a set of parameters in a more declarative fashion rather than creating them programmatically. This feature was initially developed as a part of Application Template to configure TLS based on parameters from Configuration objects and then was generalized to use different parameter sources and moved to Commons. Currently this mechanism supports only three providers: PEM, MSCAPI and PKCS#11. Usage sequence is the following:

1. Prepare a source of TLS parameters and parse it using TLSConfigurationParser resulting in TLSConfiguration instance.

2. Customize TLSConfiguration.
    1. Add callback handlers.
    2. Clients: set expected host name.
3. Create SSLContext and SSLExtendedOptions from TLSConfiguration.

This section continues with step-by-step examples and ends with a more detailed review of helper classes.

Parsing TLS Parameters

Platform SDK Commons has a few helper classes that make it easier to extract TLS parameters from a properties files, command-line arguments, etc.: TLSConfiguration and TLSConfigurationParser. TLSConfiguration is a container for parsed TLS parameters and TLSConfigurationParser provides a general parsing method and several overloaded shortcut methods for specific cases.

Examples:

```
// Using KVList as a parameters source
KVList tlsProps = new KeyValueCollection();
tlsProps.addObject("tls", "1");
tlsProps.addObject("certificate", "client-cert.pem");
TLSConfiguration tlsConfClient =
        TLSConfigurationParser.parseClientTlsConfiguration(tlsProps);
TLSConfiguration tlsConfServer =
        TLSConfigurationParser.parseServerTlsConfiguration(tlsProps);

// Using Map as a parameters source
Map<String, String> tlsProps = new HashMap<String, String>();
tlsProps.put("tls", "1");
tlsProps.put("certificate", "client-cert.pem");
TLSConfiguration tlsConfClient =
        TLSConfigurationParser.parseClientTlsConfiguration(tlsProps);
TLSConfiguration tlsConfServer =
        TLSConfigurationParser.parseServerTlsConfiguration(tlsProps);

// Using Properties as a parameters source
Properties tlsProps = new Properties();
tlsProps.load(new FileInputStream("tls.properties"));
TLSConfiguration tlsConfClient =
        TLSConfigurationParser.parseClientTlsConfiguration(tlsProps);
TLSConfiguration tlsConfServer =
        TLSConfigurationParser.parseServerTlsConfiguration(tlsProps);

// Using String as a parameters source
// Format corresponds to Transport Parameters as they appear in Configuration Manager
String tlsProps = "tls=1;certificate=client-cert.pem"; // No spaces around ";"
TLSConfiguration tlsConfClient =
        TLSConfigurationParser.parseClientTlsConfiguration(tlsProps);
TLSConfiguration tlsConfServer =
        TLSConfigurationParser.parseServerTlsConfiguration(tlsProps);
```

Customizing TLS Configuration

When TLSConfiguration is prepared, it may still need some customization. Callback handlers for password retrieval, for example, cannot be configured in parameters and must be set explicitly. They should be set always, even if not used, because some security providers require them.

Specifying expected host name is not very straightforward and some aspects should be considered. When configuring TLS on client side, expected host names are in most cases different for primary and for backup connections. Though, on some virtualized environments, they can be the same. Users may choose to use IP addresses instead of DNS host names, or use DNS names with wildcards. Either way, expected host name must match one of names specified in server's certificate and in extreme cases it may not relate to actual host name at all. To account for these cases, setting expected host name is not automated in Platform SDK and left for user code. Example code below shows how to set this value to actual host name of target server.

According to X.509 specification, certificate may contain not just host name or IP address, but also URI or email address. Platform SDK supports only host names and IP addresses, but host name may use wildcard: a star symbol, "*", can be used instead of any one level of domain name.

Examples:

```
TLSConfiguration tlsConfiguration = ...;

// Applicable to both clients and servers
// Passwords are not used, so set dummies:
tlsConfiguration.setKeyStoreCallbackHandler(
        new DummyPasswordCallbackHandler());
tlsConfiguration.setTrustStoreCallbackHandler(
        new DummyPasswordCallbackHandler());

// In case some real password is needed:
tlsConfiguration.setKeyStoreCallbackHandler(new CallbackHandler() {
    public void handle(Callback[] callbacks) {
        char[] password = new char[] {
                'p', 'a', 's', 's', 'w', 'o', 'r', 'd'};
            for (Callback c : callbacks) {
                if (c instanceof PasswordCallback) {
                    ((PasswordCallback) c).setPassword(password);
                }
            }
        }
    }
);

// Expected host name may contain exact host name, ...
tlsConfiguration.setExpectedHostname("someserver.ourdomain.com");
// wildcard host name, ...
tlsConfiguration.setExpectedHostname("*.ourdomain.com");
tlsConfiguration.setExpectedHostname("someserver.*.com");

// IPv4 address, ...
tlsConfiguration.setExpectedHostname("192.168.1.1");
// IPv6 address.
tlsConfiguration.setExpectedHostname("fe80::ffff:ffff:fffd");
```

Creating SSLContext

Platform SDK Commons has helper class – TLSConfigurationHelper, which creates SSLContext and SSLExtendOptions based on TLSConfiguration object. TLSConfigurationHelper has two methods:

```
public static SSLContext createSslContext(TLSConfiguration config);
```

and

```
static SSLExtendedOptions createSslExtendedOptions(TLSConfiguration config);
```

Method createSSLContext() determines security provider type if it is not set explicitly, creates necessary key store objects, key manager, trust manager, and finally wraps it all into SSLContext.

Method createSSLExtendedOptions() does not contain any logic, it just creates new SSLExtendedOptions with the exact parameters taken from TLSConfiguration.

Usage of both methods is shown in code sample below.

Example:

```
// TLS preparation section follows
KVList tlsProps = new KeyValueCollection();
tlsProps.addObject("tls", "1");
tlsProps.addObject("certificate", "client-cert.pem");
TLSConfiguration tlsConf =
        TLSConfigurationParser. parseClientTlsConfiguration(tlsProps);

boolean tlsEnabled = true;

SSLContext sslContext =
        TLSConfigurationHelper.createSslContext(tlsConfiguration);
SSLExtendedOptions sslOptions =
        TLSConfigurationHelper.createSslExtendedOptions(tlsConfiguration);

// The same as above, using shortcut methods:
sslContext = tlsConfiguration.createSslContext();
sslOptions = tlsConfiguration.createSslExtendedOptions();

Endpoint ep = new Endpoint(appName, host, port, null, tlsEnabled, sslContext, sslOptions);
```

TLSConfiguration Class

TLSConfiguration class is used as intermediate container to keep stronger-typed TLS parameters extracted from a parameter source. It contains the following:

Properties

TLSConfiguration Properties List

Name	Type	Description
tlsEnabled	boolean	Correspond to TLS parameters in Configuration; please see the list of TLS Parameters in Configuration Manager for details.
provider	String	
certificate	String	
certificateKey	String	
trustedCaCertificate	String	
mutual	boolean	
crl	String	
targetNameCheckEnabled	boolean	
cipherList	String	
fips140Enabled	boolean	
clientMode	boolean	Should be set to true for client-side of connection and false for server-side.

Name	Type	Description
		`TLSConfigurationParser` specialized methods set it automatically.
expectedHostname	String	Host name to check against, used when `targetNameCheckEnabled` is turned on. Typically is used by client side and assigned to the host/domain part of target URL.
keyStoreCallbackHandler	CallbackHandler	Please see Callback Handlers for details.
trustStoreCallbackHandler	CallbackHandler	
version	String	Defines security protocol version (all previous version will be accepted by default) Note: By default, the following client-side TLS Protocol versions are enabled in Java: • Java 6, Java 7 - TLSv1 • Java 8 - TLSv1.2
enabledProtocols	String	Limits supported security protocol version list (space separated list)
secProtocol	String Possible values are "SSLv23", "SSLv3", "TLSv1", "TLSv11", "TLSv12". Example: "sec-protocol=TLSv1"	Defines security protocol version (all previous version won't be accepted); available in Platform SDK from release 8.5.102.
keyStoreEntryCallbackHandler	CallbackHandler	It is used only for JKS provider. If it isn't assigned, then `trustStoreCallbackHandler` will be used as `keyStoreEntryCallbackHandler`.

Methods

TLSConfiguration Methods List

Signature	Description
SSLContext createSslContext()	A shortcut for `TLSConfigurationHelper.createSslContext` method. Creates and configures `SSLContext` object based on the properties values.
SSLExtendedOptions createSslExtendedOptions()	A shortcut for `TLSConfigurationHelper.createSslExtendedOptions` method. Creates `SSLExtendedOptions` object based on the properties values.

Constants

The following constants define supported values for a provider property:

- String TLS_PROVIDER_PEM_FILE;
- String TLS_PROVIDER_PKCS11;
- String TLS_PROVIDER_MSCAPI;

TLSConfigurationParser Class

TLSConfigurationParser class has methods that extract TLS parameters from different sources and create TLSConfiguration instance containing the parameters. It uses interface PropertyReader and several classes implementing this interface to read TLS parameters.

Methods

TLSConfiguration Methods List

Signature	Description
public static TLSConfiguration parseTlsConfiguration(final PropertyReader prop, final boolean clientMode)	This is the main and most generic method. It reads all possible TLS parameters (parameter names and possible values are detailed in the list of TLS Parameters in Configuration Manager), converts them and assigns them to `TLSConfiguration` properties.
public static TLSConfiguration parseServerTlsConfiguration(KVList kvl)	These methods provide shortcuts to parse TLS configuration from different source types.
public static TLSConfiguration parseClientTlsConfiguration(KVList kvl)	
public static TLSConfiguration parseServerTlsConfiguration(Map<String, String> map)	
public static TLSConfiguration parseClientTlsConfiguration(Map<String, String> map)	
public static TLSConfiguration parseServerTlsConfiguration(Properties prop)	
public static TLSConfiguration parseClientTlsConfiguration(Properties prop)	
public static TLSConfiguration parseServerTlsConfiguration(String transportParams)	
public static TLSConfiguration parseClientTlsConfiguration(String transportParams)	

Interface PropertyReader and Implementing Classes

Interface PropertyReader contains just one method:

```
String getProperty(String key)
```

Here, key argument contains name of parameter to extract. Implementing classes contain code that actually extract and return value corresponding to the key. Currently there are five implementations:

1. GConfigTlsPropertyReader - This class belongs to Application Template and is used to extract TLS parameters from a set of related Configuration objects. It cannot be included to Commons library since it would cause circular references between the Commons and Application Template.
2. KVListPropertyReader - Extracts String value from a KVList instance.
3. MapPropertyReader - Extracts value from a Map<String, String> instance.
4. PropertiesReader - Extracts value from a Properties instance.
5. TransportParamsPropertyReader - Parses transport parameters as they appear in Configuration Manager, for example:

```
“tls=1;certificate=c:/cert/cert.pem;mutual=1”.
```

TLS and the Application Template Application Block

Platform SDK for Java

> **Warning**
> If the certificate or CA certificate are not set, then using the `tls` option flag in a client application will create an ***unauthenticated connection*** with encryption only. To ensure TLS authentication and certificate validation are performed, you must first configure those parameters correctly.

Introduction

Instead of using the Platform SDK Commons Library to configure TLS connections with hard-coded values, you can use the Platform SDK Application Template Application Block to retrieve configuration objects from Configuration Server which contain parameters that are used to configure your TLS settings.

The steps do accomplish this are as follows:

1. Parse a configuration object.
2. Create a `TLSConfiguration` object for the configuration object.
3. Customize your `TLSConfiguration` object:
    - Add callback handlers.
    - For clients, set the expected host names for primary and backup servers.
4. Create `SSLContext` and `SSLExtendedOptions` objects based on your `TLSConfiguration` object.
5. Use your `SSLContext` and `SSLExtendedOptions` objects to create `Endpoints` and/or `WarmStandbyConfiguration` objects.
6. Use your `Endpoints` and/or `WarmStandbyConfiguration` objects to create `Protocol` instances.

The sections below describe these steps in more detail.

If you plan on using this method to configure TLS settings, be sure that related application objects in Configuration Manager have been configured with TLS parameters.

If you aren't familiar with TLS configuration settings then please read Using the Platform SDK

Commons Library to gain a better understanding of what is required.

Parsing Configuration Objects

The Platform SDK Application Template has a helper class, `GConfigTlsPropertyReader`, which makes it easy to extract TLS parameters from Configuration Server. When used in conjunction with `TLSConfigurationParser`, `TLSConfigurationHelper`, `ClientConfigurationHelper` and `ServerConfigurationHelper` classes, all of the connection-related options found in Configuration Server are covered. They also provide other useful functionality.

`TLSConfigurationParser` has two constructors:

```
public GConfigTlsPropertyReader(
        IGApplicationConfiguration appConfig,
        IGApplicationConfiguration.IGPortInfo portConfig);
```

and

```
public GConfigTlsPropertyReader(
        IGApplicationConfiguration appConfig,
        IGApplicationConfiguration.IGAppConnConfiguration connConfig);
```

The first one is used for server-side connections while the second is for client-side connections.

For example:

```
// Client side
// Prepare configuration objects
String clientAppName = "<my-app-name>";
CfgAppType targetServerType = CfgAppType.CFGTServer;
CfgApplication cfgApplication = confService.retrieveObject(
        CfgApplication.class, new CfgApplicationQuery(clientAppName));
GCOMApplicationConfiguration appConfiguration =
        new GCOMApplicationConfiguration(cfgApplication);
IGApplicationConfiguration.IGAppConnConfiguration connConfig =
        appConfiguration.getAppServer(targetServerType);

// Parse TLS parameters
PropertyReader reader = new GConfigTlsPropertyReader(appConfiguration, connConfig);
TLSConfiguration tlsConfiguration =
        TLSConfigurationParser.parseTlsConfiguration(reader, true);
// At this point, tlsConfiguration contains TLS parameters read from
// configuration objects

// Server side
// Prepare configuration objects
String serverAppName = "<my-app-name>";
String portID = "secure";
CfgApplication cfgApplication = confService.retrieveObject(
        CfgApplication.class, new CfgApplicationQuery(serverAppName));
GCOMApplicationConfiguration appConfiguration =
        new GCOMApplicationConfiguration(cfgApplication);
IGApplicationConfiguration.IGPortInfo portConfig =
        appConfiguration.getPortInfo(portID);

// Parse TLS parameters
PropertyReader reader = new GConfigTlsPropertyReader(appConfiguration, portConfig);
```

```
TLSConfiguration tlsConfiguration =
        TLSConfigurationParser.parseTlsConfiguration(reader, false);
```

Customizing TLS Configuration

When Configuration objects are used as a source of TLS parameters, they can also provide values for expected host names.

Examples:

```
TLSConfiguration tlsConfiguration = ...;

// Client side
// Prepare configuration objects
String clientAppName = "<my-app-name>";
CfgAppType targetServerType = CfgAppType.CFGTServer;
CfgApplication cfgApplication = confService.retrieveObject(
        CfgApplication.class, new CfgApplicationQuery(clientAppName));
GCOMApplicationConfiguration appConfiguration =
        new GCOMApplicationConfiguration(cfgApplication);
IGApplicationConfiguration.IGAppConnConfiguration connConfig =
        appConfiguration.getAppServer(targetServerType);

// TLS-specific part
IGApplicationConfiguration.IGServerInfo primaryServer =
        connConfig.getTargetServerConfiguration().getServerInfo();
IGApplicationConfiguration.IGServerInfo backupServer =
        primaryServer.getBackup().getServerInfo();

tlsConfiguration.setExpectedHostname(primaryServer.getHost().getName());
// Or:
// tlsConfiguration.setExpectedHostname(backupServer.getHost().getName());
```

Creating SSLContext Objects

`SSLContext` and `SSLExtendedOptions` are created either using `TLSConfigurationHelper` or with `TLSConfiguration` shortcut methods:

Examples:

```
SSLContext sslContext =
        TLSConfigurationHelper.createSslContext(tlsConfiguration);
SSLExtendedOptions sslOptions =
        TLSConfigurationHelper.createSslExtendedOptions(tlsConfiguration);

// The same as above, using shortcut methods:
sslContext = tlsConfiguration.createSslContext();
sslOptions = tlsConfiguration.createSslExtendedOptions();
```

Configuring TLS for Client Connections

Platform SDK has a helper class, `ClientConfigurationHelper`, that makes it easier to prepare connections for client applications. This class has the following methods:

```
public static Endpoint createEndpoint(
        IGApplicationConfiguration appConfig,
        IGAppConnConfiguration connConfig,
        IGApplicationConfiguration targetServerConfig);

public static Endpoint createEndpoint(
        IGApplicationConfiguration appConfig,
        IGAppConnConfiguration connConfig,
        IGApplicationConfiguration targetServerConfig,
        boolean tlsEnabled,
        SSLContext sslContext,
        SSLExtendedOptions sslOptions);

public static WarmStandbyConfiguration createWarmStandbyConfig(
        IGApplicationConfiguration appConfig,
        IGAppConnConfiguration connConfig);

public static WarmStandbyConfiguration createWarmStandbyConfig(
        IGApplicationConfiguration appConfig,
        IGAppConnConfiguration connConfig,
        boolean primaryTLSEnabled,
        SSLContext primarySSLContext,
        SSLExtendedOptions primarySSLOptions,
        boolean backupTLSEnabled,
        SSLContext backupSSLContext,
        SSLExtendedOptions backupSSLOptions);
```

Two of these methods simply accept TLS-specific parameters and pass them through to the `Endpoint` and `WarmStandbyConfiguration` instances being created. A code sample using the `createEndpoint()` method is shown here:

```
String clientAppName = "<my-app-name>";
CfgAppType targetServerType = CfgAppType.CFGTServer;
CfgApplication cfgApplication = confService.retrieveObject(
        CfgApplication.class, new CfgApplicationQuery(clientAppName));

GCOMApplicationConfiguration appConfiguration =
        new GCOMApplicationConfiguration(cfgApplication);

IGAppConnConfiguration connConfig =
        appConfiguration.getAppServer(targetServerType);

// TLS preparation section follows
PropertyReader reader = new GConfigTlsPropertyReader(appConfiguration, connConfig);
TLSConfiguration tlsConfiguration =
        TLSConfigurationParser.parseTlsConfiguration(reader, true);

// TLS customization code goes here...
// As an example, host name verification is turned on
IGApplicationConfiguration.IGServerInfo targetServer =
        connConfig.getTargetServerConfiguration().getServerInfo();
tlsConfiguration.setExpectedHostname(targetServer.getHost().getName());

// Get TLS configuration objects for connection
SSLContext sslContext = tlsConfiguration.createSslContext();
SSLExtendedOptions sslOptions = tlsConfiguration.createSslExtendedOptions();
```

```
boolean tlsEnabled = tlsConfiguration.isTlsEnabled();
// TLS preparation section ends

Endpoint epTSrv = ClientConfigurationHelper.createEndpoint(
        appConfiguration, connConfig,
        connConfig.getTargetServerConfiguration(),
        tlsEnabled, sslContext, sslOptions);

TServerProtocol tsProtocol = new TServerProtocol(epTSrv);
tsProtocol.setClientName(clientName);
tsProtocol.open();
```

Configuring Warm Standby

In cases when the target server has a backup in warm standby mode, configuration requires a little extra effort, as shown in the following code sample.

Note: Configuring TLS for primary and backup servers in Warm Standby mode has some specifics that may not be obvious. Primary and backup servers typically share the same settings. Thus, when a server is selected as a backup for another server (the primary server), Configuration Manager copies settings from the primary server to the backup server to make them the same. This is also true of TLS settings, and the same `TLSConfiguration` object can be used to configure both the primary and backup connections. On the other hand, primary and backup servers usually reside on different hosts. This means that if a hostname check is used, each of these servers must have different `expectedHostname` parameter values. This is not hard to do, as the following code sample demonstrates, but it is not always obvious.

```
String clientAppName = "<my-app-name>";
CfgAppType targetServerType = CfgAppType.CFGStatServer;
CfgApplication cfgApplication = confService.retrieveObject(
        CfgApplication.class, new CfgApplicationQuery(appName));

GCOMApplicationConfiguration appConfiguration =
        new GCOMApplicationConfiguration(cfgApplication);

IGAppConnConfiguration connConfig =
        appConfiguration.getAppServer(targetServerType);

// TLS preparation section follows
PropertyReader reader = new GConfigTlsPropertyReader(appConfiguration, connConfig);
TLSConfiguration tlsConfiguration =
        TLSConfigurationParser.parseTlsConfiguration(reader, true);

IGApplicationConfiguration.IGServerInfo primaryServer =
        connConfig.getTargetServerConfiguration().getServerInfo();
IGApplicationConfiguration.IGServerInfo backupServer =
        primaryServer.getBackup().getServerInfo();

// Configure TLS for Primary
tlsConfiguration.setExpectedHostname(primaryServer.getHost().getName());
SSLContext primarySslContext = tlsConfiguration.createSslContext();
SSLExtendedOptions primarySslOptions = tlsConfiguration.createSslExtendedOptions();
boolean primaryTlsEnabled = tlsConfiguration.isTlsEnabled();

// Configure TLS for Backup
tlsConfiguration.setExpectedHostname(backupServer.getHost().getName());
SSLContext backupSslContext = tlsConfiguration.createSslContext();
SSLExtendedOptions backupSslOptions = tlsConfiguration.createSslExtendedOptions();
boolean backupTlsEnabled = tlsConfiguration.isTlsEnabled();
// TLS preparation section ends
```

```
WarmStandbyConfiguration wsConfig =
        ClientConfigurationHelper.createWarmStandbyConfig(
                appConfiguration, connConfig,
                primaryTlsEnabled, primarySslContext, primarySslOptions,
                backupTlsEnabled, backupSslContext, backupSslOptions);

StatServerProtocol statProtocol =
        new StatServerProtocol(wsConfig.getActiveEndpoint());
statProtocol.setClientName(clientName);

WarmStandbyService wsService = new WarmStandbyService(statProtocol);
wsService.applyConfiguration(wsConfig);
wsService.start();
statProtocol.beginOpen();
```

Configuring TLS for Servers

Platform SDK has a helper class, `ServerConfigurationHelper`, that makes it easier to prepare listening sockets for server applications. This class has the following methods:

```
public static Endpoint createListeningEndpoint(
        IGApplicationConfiguration application,
        IGApplicationConfiguration.IGPortInfo portInfo);

public static Endpoint createListeningEndpoint(
        IGApplicationConfiguration application,
        IGApplicationConfiguration.IGPortInfo portInfo,
        boolean tlsEnabled,
        SSLContext sslContext,
        SSLExtendedOptions sslOptions);
```

The overloaded version of the `createListeningEndpoint()` method accepts TLS parameters and passes them through to the `Endpoint` object that is being created. The following code sample shows how this is done:

```
String serverAppName = "<my-app-name>";
String portID = "secure";
CfgApplication cfgApplication = confService.retrieveObject(
        CfgApplication.class, new CfgApplicationQuery(appName));
GCOMApplicationConfiguration appConfig =
        new GCOMApplicationConfiguration(cfgApplication);
IGApplicationConfiguration.IGPortInfo portConfig =
        appConfig.getPortInfo(portID);

// TLS preparation section follows
PropertyReader reader = new GConfigTlsPropertyReader(appConfiguration, portConfig);
TLSConfiguration tlsConfiguration =
        TLSConfigurationParser.parseTlsConfiguration(reader, false);

// TLS customization code goes here...
// As an example, mutual TLS mode is turned on
tlsConfiguration.setMutual(true);

// Get TLS configuration objects for connection
SSLContext sslContext = tlsConfiguration.createSslContext();
SSLExtendedOptions sslOptions = tlsConfiguration.createSslExtendedOptions();
boolean tlsEnabled = tlsConfiguration.isTlsEnabled();
// TLS preparation section ends
```

```
Endpoint endpoint = ServerConfigurationHelper.createListeningEndpoint(
        appConfig, portConfig,
        tlsEnabled, sslContext, sslOptions);
ExternalServiceProtocolListener serverChannel =
        new ExternalServiceProtocolListener(endpoint);
...
```

Platform SDK for .NET

Tip

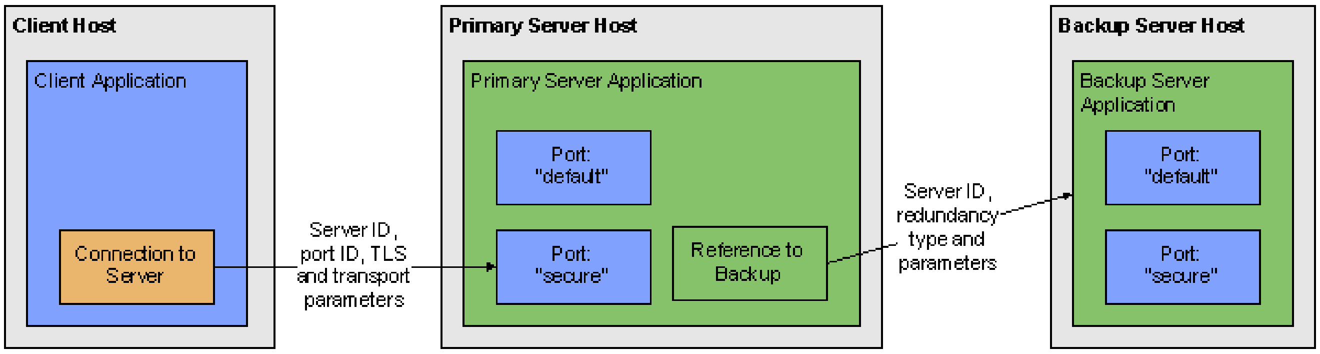
The contents of this page only apply to Java implementations.

Configuring TLS Parameters in Configuration Manager

Introduction

As described earlier, the Platform SDK Application Template Application Block allows both client and server applications to read TLS parameters from configuration objects. This page describes how to set TLS parameters correctly in those configuration objects.

Configuration objects that will be used, and their relations, are shown in the diagram below:



To edit TLS-related parameters for these objects, you will need to have access to the Annex tab in Configuration Manager.

Precedence of Configuration Objects

Platform SDK uses different sets of configuration objects to configure client- and server-side TLS settings. For TLS parameters, these objects are searched from the most specific object to the most general one. Parameters found in specific objects take precedence over those in more general objects.

Note: This search occurs independently for each supported TLS parameter.

Location of specific TLS parameters can differ for each object, but is detailed in the appropriate section on this page.

Configuration Object Precedence

Application type	Configuration Objects Used, in Order of Precedence
Client	1. Connection from the client application to the server.

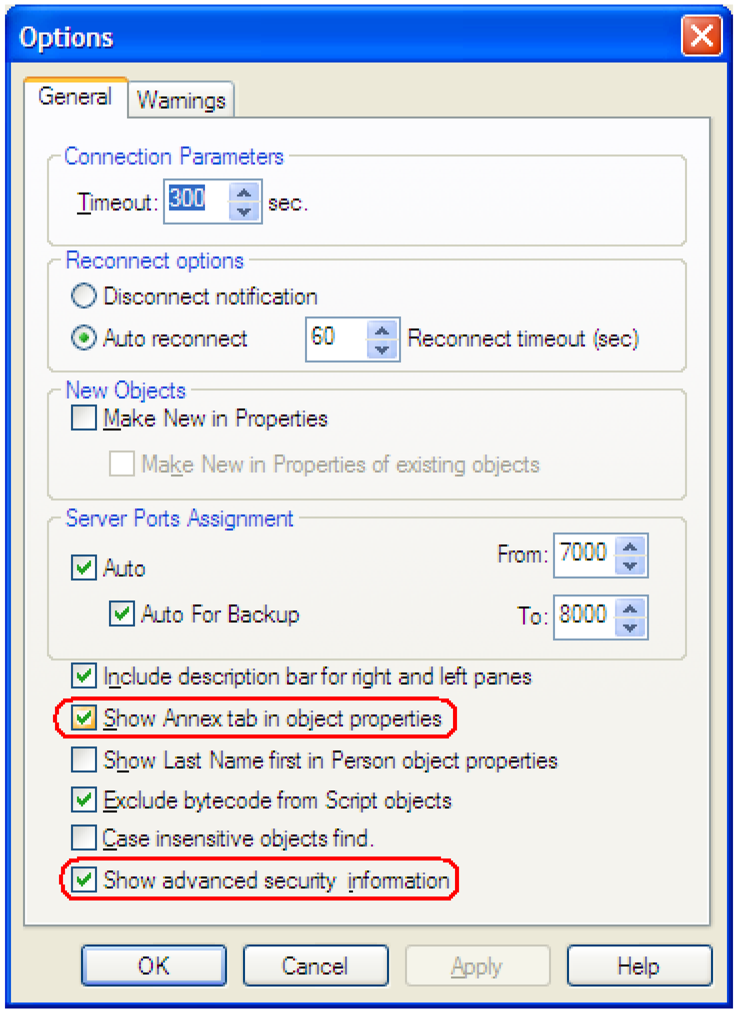
Application type	Configuration Objects Used, in Order of Precedence
	2. Application of the client. 3. Host where client application resides. 4. Port of the target server that client connects to.[1]
Server	1. Port of the server application. 2. Application of the server. 3. Host where the server application resides.

1. If the `tls` parameter is not set to 1 in both the client Application and Connection objects, then the client application will look to the Port object for the target server to determine if TLS should be turned on. Configuration Manager does not automatically add the `tls=1` parameter to Connection Transport parameters when it is linked to a server's secure Port. This is the only case when a client application considers settings in the server's configuration objects.

Displaying the Annex Tab in Configuration Manager

By default, Configuration Manager does not show Annex tab in Object Properties windows. This tab can contain TLS parameters for Host and Application objects.

To show the Annex tab, select *View > Options...* from the main menu and ensure the *Show Annex tab in object properties* and *Show Advanced Security Information* options are selected.



Connection Parameters

Timeout: 300 sec.

Reconnect options

Disconnect notification [✓ Auto reconnect] 60 Reconnect timeout (sec)

New Objects

☐ Make New in Properties

☐ Make New in Properties of existing objects

Server Ports Assignment

☑ Auto From: 7000

☑ Auto For Backup To: 8000

☑ Include description bar for right and left panes

☑ Show Annex tab in object properties

☐ Show Last Name first in Person object properties

☑ Exclude bytecode from Script objects

☐ Case insensitive objects find.

☑ Show advanced security information

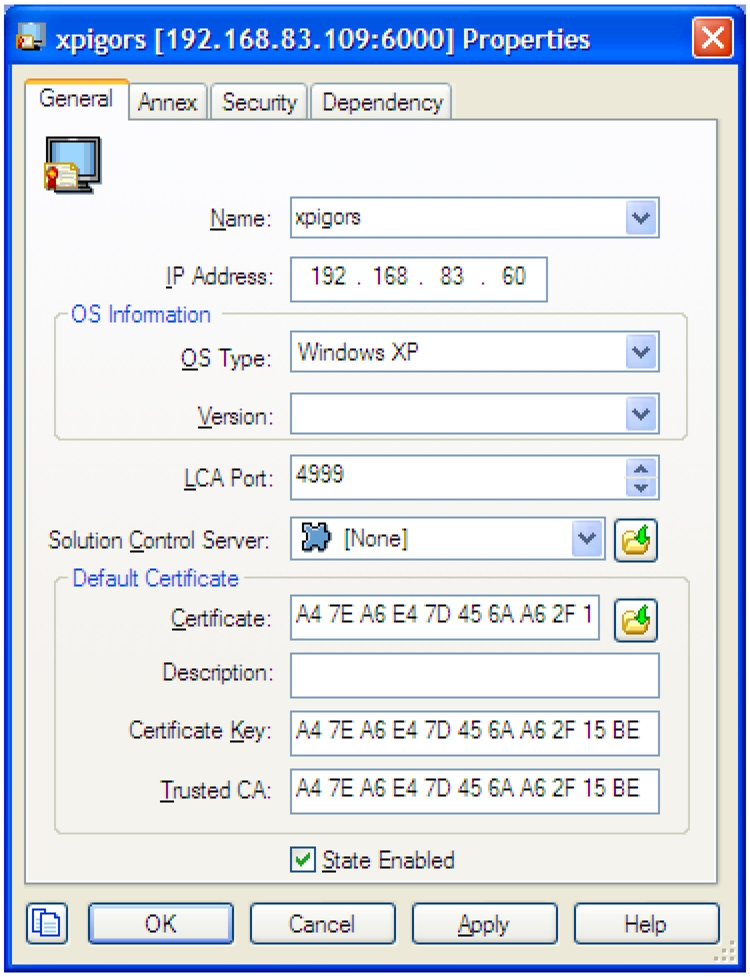
Application Objects

Host Object

The properties window for a Host object includes most common TLS parameters on the General tab:

- Certificate
- Certificate Key
- Trusted CA

These fields allow copy/paste operations, so they can be set manually by copying and pasting the "Thumbprint" field values from certificates in Windows Certificate Services (WCS) into the related field in Configuration Manager.



xpigors [192.168.83.109:6000] Properties

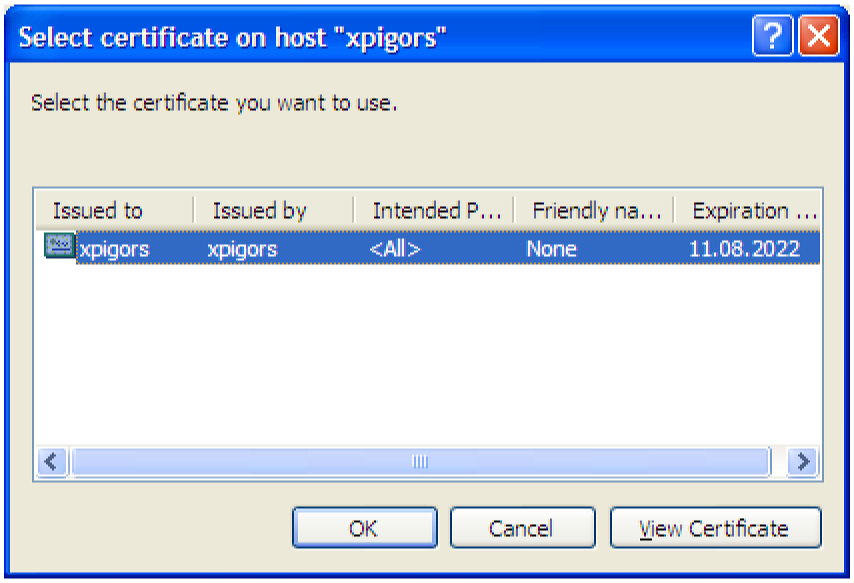
OS Information

Default Certificate

Name:	xpigors
IP Address:	192 . 168 . 83 . 60
OS Type:	Windows XP
Version:	
LCA Port:	4999
Solution Control Server:	[None]
Certificate:	A4 7E A6 E4 7D 45 6A A6 2F 1
Description:	
Certificate Key:	A4 7E A6 E4 7D 45 6A A6 2F 15 BE
Trusted CA:	A4 7E A6 E4 7D 45 6A A6 2F 15 BE

[✓ State Enabled]

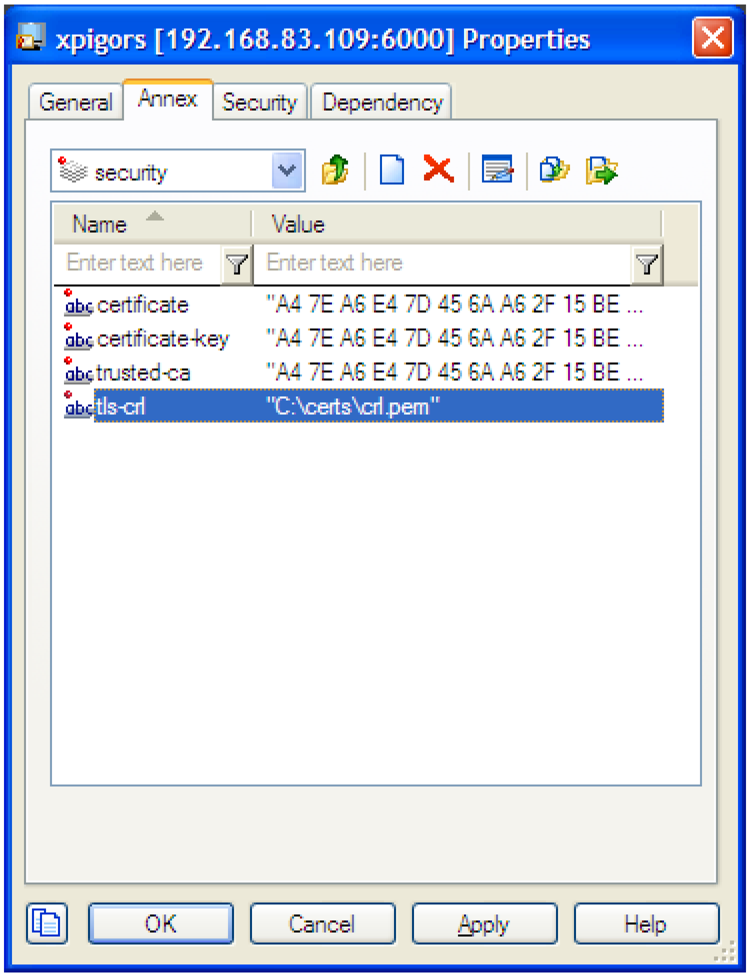
To select a certificate, use the button next to *Certificate* field. This opens the *Select certificate* window, displaying a list of certificates installed in WCS under the Local Computer account for the local machine.


Select certificate on host "xpigors"

Select the certificate you want to use.

Issued to	Issued by	Intended P...	Friendly na...	Expiration ...
xpigors	xpigors	<All>	None	11.08.2022

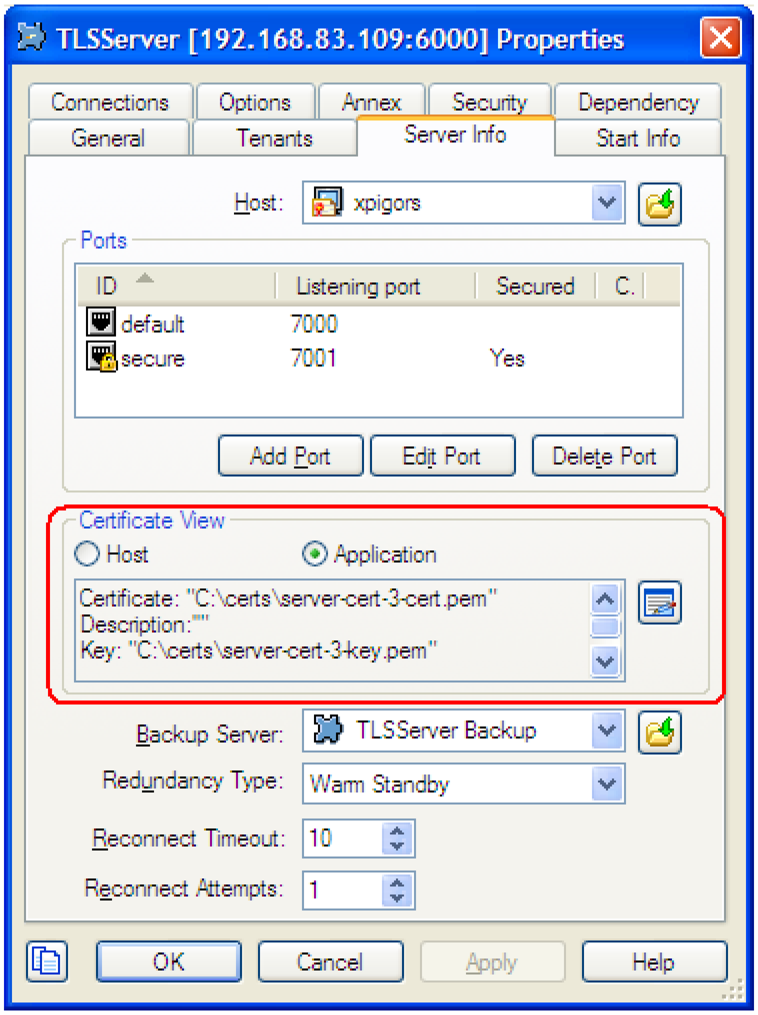
The Annex tab contains a security section that holds TLS settings for this object. Any change made to TLS-related fields on the General tab are mirrored between the Annex tab automatically. You can also specify additional TLS parameters here that aren't reflected on the General tab.



Name	Value
certificate	"A4 7E A6 E4 7D 45 6A A6 2F 15 BE ...
certificate-key	"A4 7E A6 E4 7D 45 6A A6 2F 15 BE ...
trusted-ca	"A4 7E A6 E4 7D 45 6A A6 2F 15 BE ...
tls-crl	"C:\certs\crl.pem"

Server Application Object

For the server Application object, TLS-related fields are located on the *Server Info* tab of the properties window. Note the *Certificate View* controls group, where the server can be set to use Host TLS parameters (generally recommended for Genesys Framework) or application-specific ones.



Host: xpigors

Ports

ID	Listening port	Secured	C.
default	7000		
secure	7001	Yes	

Certificate View

Host [✓ Application]

Certificate: "C:\certs\server-cert-3-cert.pem"
Description:""
Key: "C:\certs\server-cert-3-key.pem"

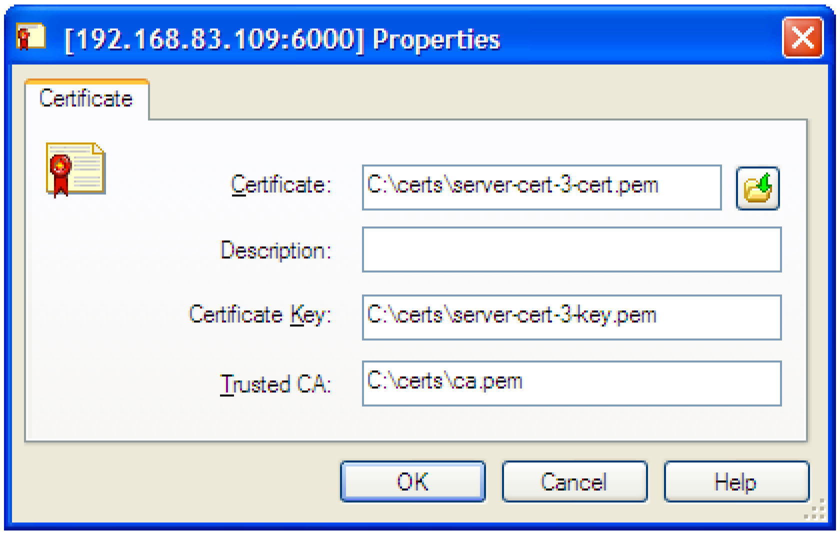
Backup Server: TLSServer Backup

Redundancy Type: Warm Standby

Reconnect Timeout: 10

Reconnect Attempts: 1

If using application-specific TLS parameters, use the button next to the certificate information field to open a certificate selection window where you can choose from a list of certificates installed for the Local Computer account or manually enter certificate information:



Certificate: C:\certs\server-cert-3-cert.pem

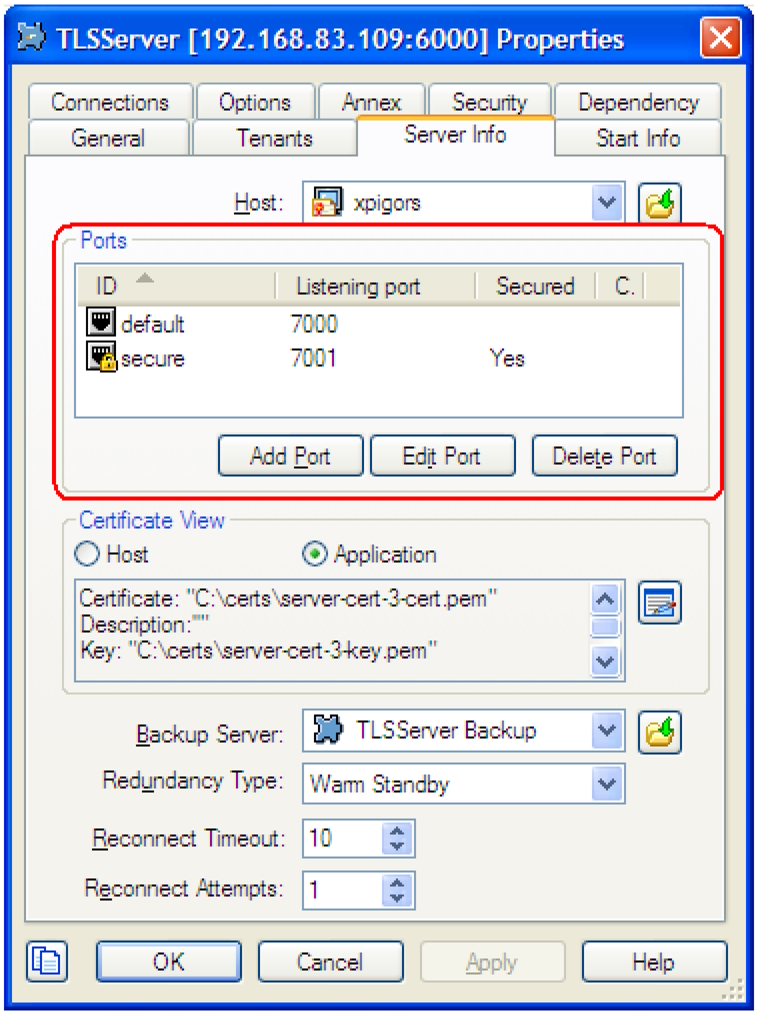
Description:

Certificate Key: C:\certs\server-cert-3-key.pem

Trusted CA: C:\certs\ca.pem

Port Object

For port objects, TLS-related fields are located on the *Server Info* tab of the properties window. You can see here whether a port is secured (TLS-enabled) or not, and have the option to edit existing ports to update TLS parameters or to add new ports.



Host: xpigors

Ports

ID	Listening port	Secured	C.
default	7000		
secure	7001	Yes	

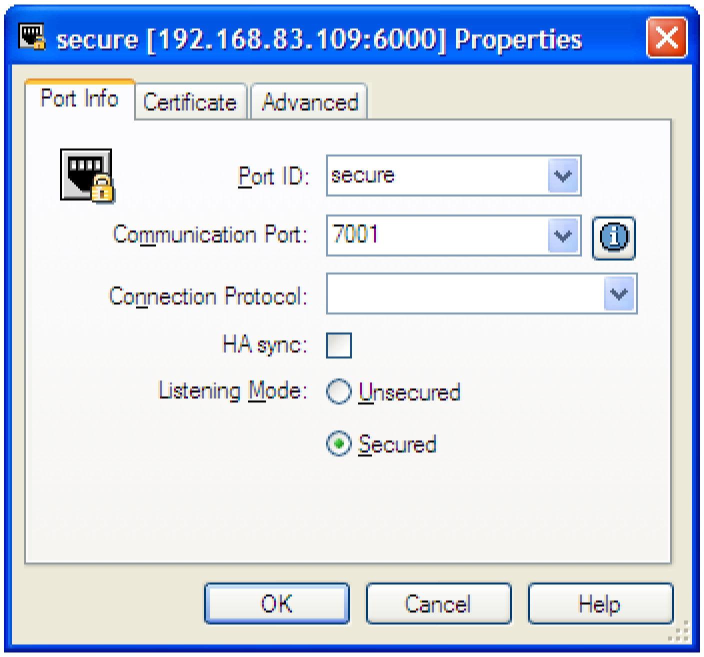
Certificate View
Host [✓ Application]

Certificate: "C:\certs\server-cert-3-cert.pem"
Description:""
Key: "C:\certs\server-cert-3-key.pem"

Backup Server: TLSServer Backup
Redundancy Type: Warm Standby
Reconnect Timeout: 10
Reconnect Attempts: 1

When adding or editing a port, TLS parameters are specified on the following tabs:

- *Port Info* — Turn on *Secured* listening mode for the port (the same as adding the *tls=1* string to transport parameters).
- *Certificate* — Show certificate information, open a certificate selection window, or delete the current certificate information.
- *Advanced* — Manually edit the *Transport Protocol Parameters* field. TLS parameters not reflected on the *Certificate* tab can be added here.


secure [192.168.83.109:6000] Properties

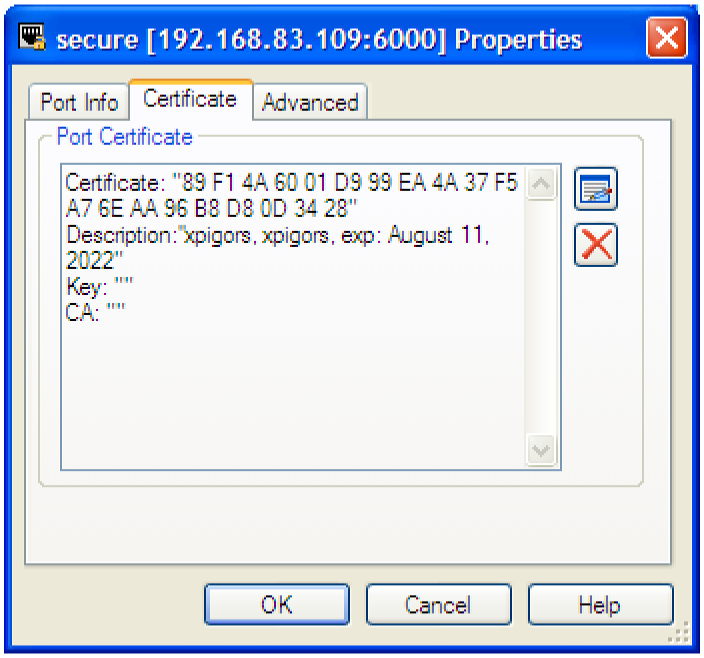
Port ID: secure

Communication Port: 7001

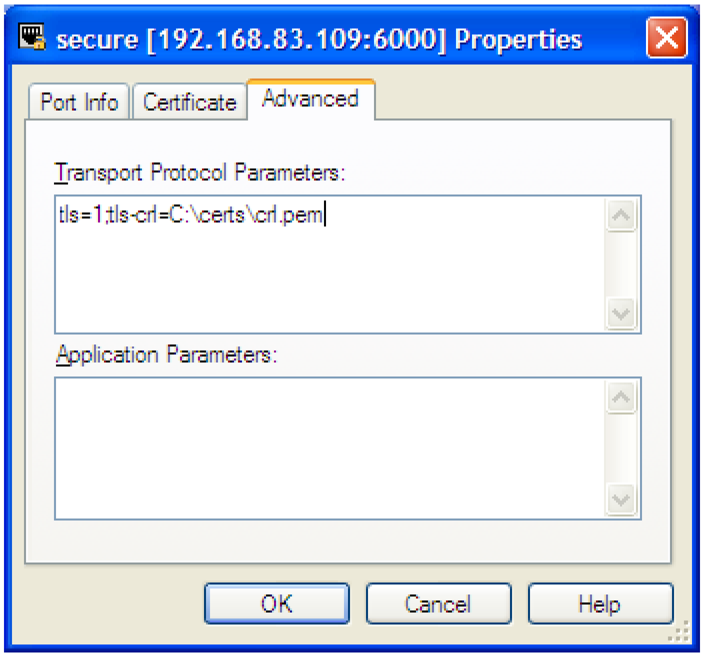
Connection Protocol:

HA sync: ☐

Listening Mode: ○ Unsecured [✓ ● Secured]


secure [192.168.83.109:6000] Properties

```
Certificate: "89 F1 4A 60 01 D9 99 EA 4A 37 F5
A7 6E AA 96 B8 D8 0D 34 28"
Description:"xpigors, xpigors, exp: August 11,
2022"
Key: ""
CA: ""
```



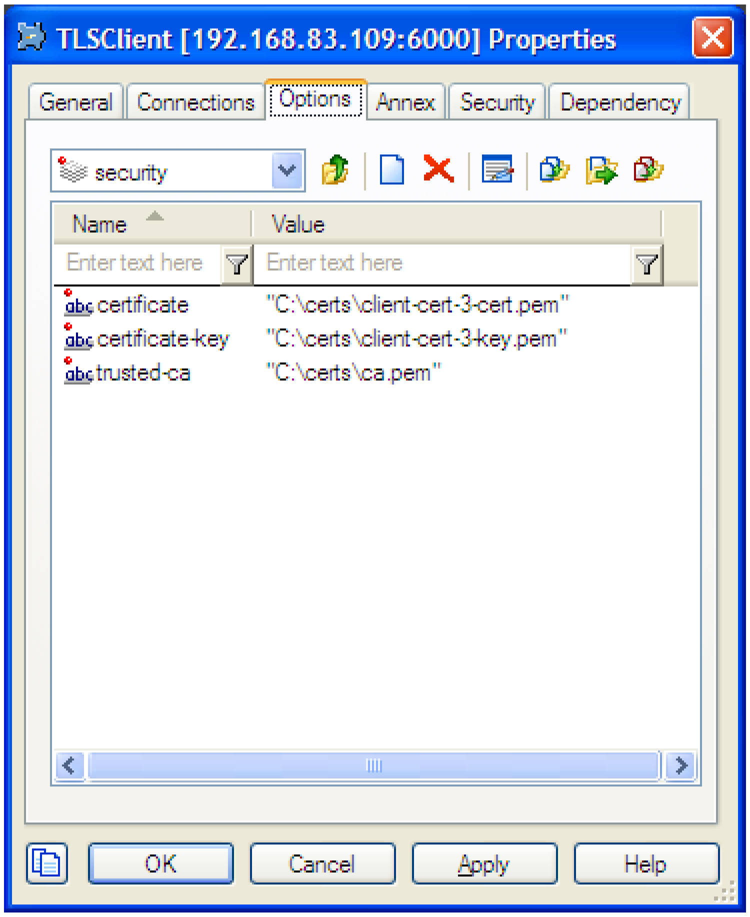
Transport Protocol Parameters:

```
tls=1;tls-crl=C:\certs\crl.pem
```

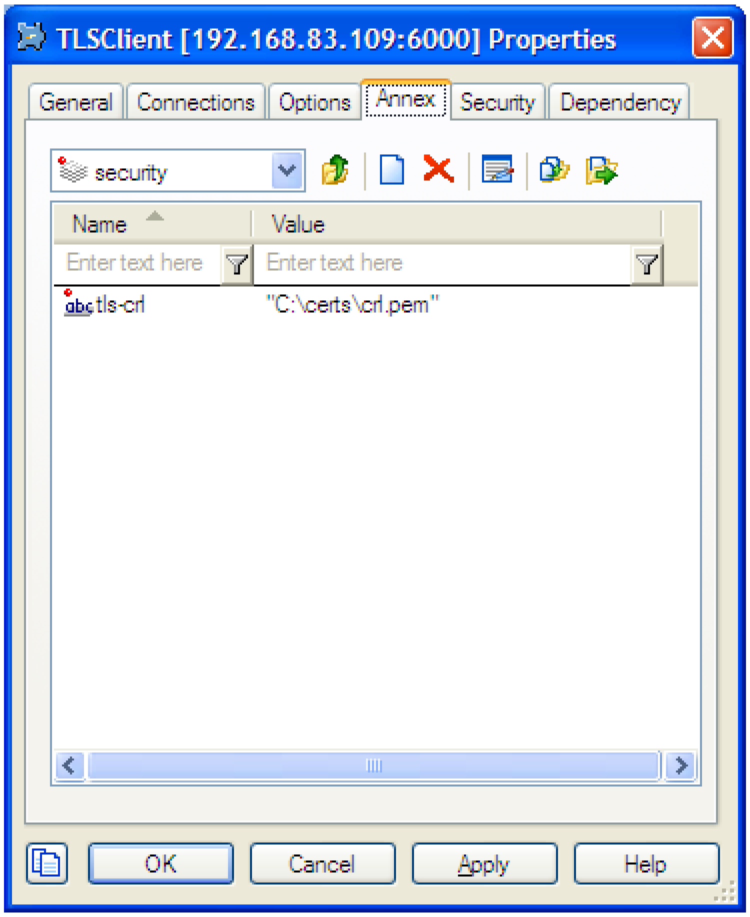
Application Parameters:

Client Application Object

For client Application objects, TLS-related fields are located under the *security* sections of both the *Options* and *Annex* tabs. There is no certificate selection window provided, but TLS parameters can be configured manually in either section.



Name	Value
certificate	"C:\certs\client-cert-3-cert.pem"
certificate-key	"C:\certs\client-cert-3-key.pem"
trusted-ca	"C:\certs\ca.pem"

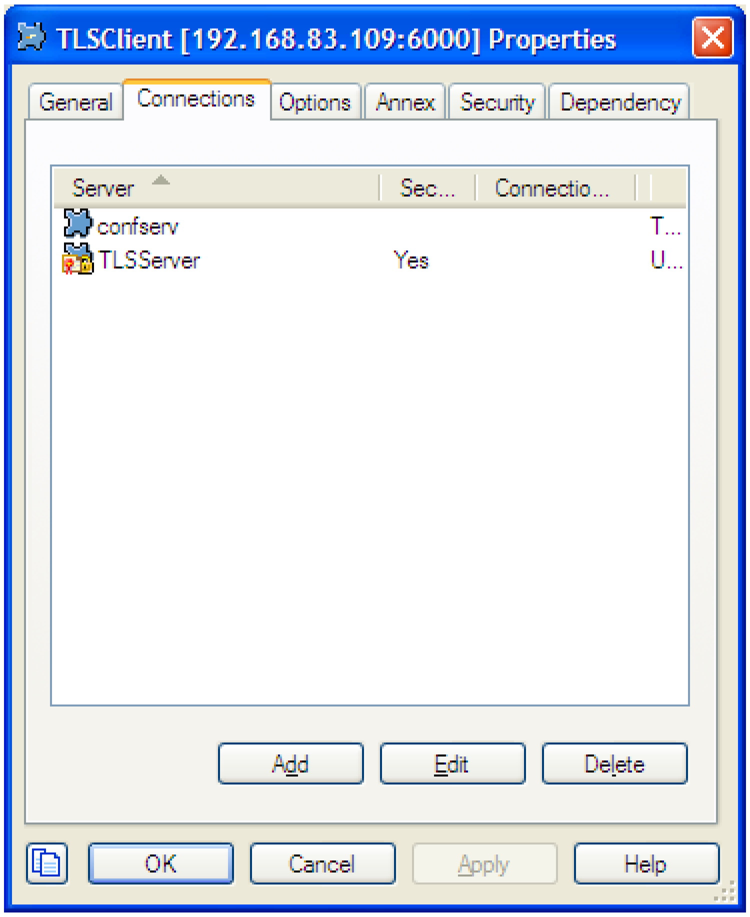


Name	Value
tls-crl	"C:\certs\crl.pem"

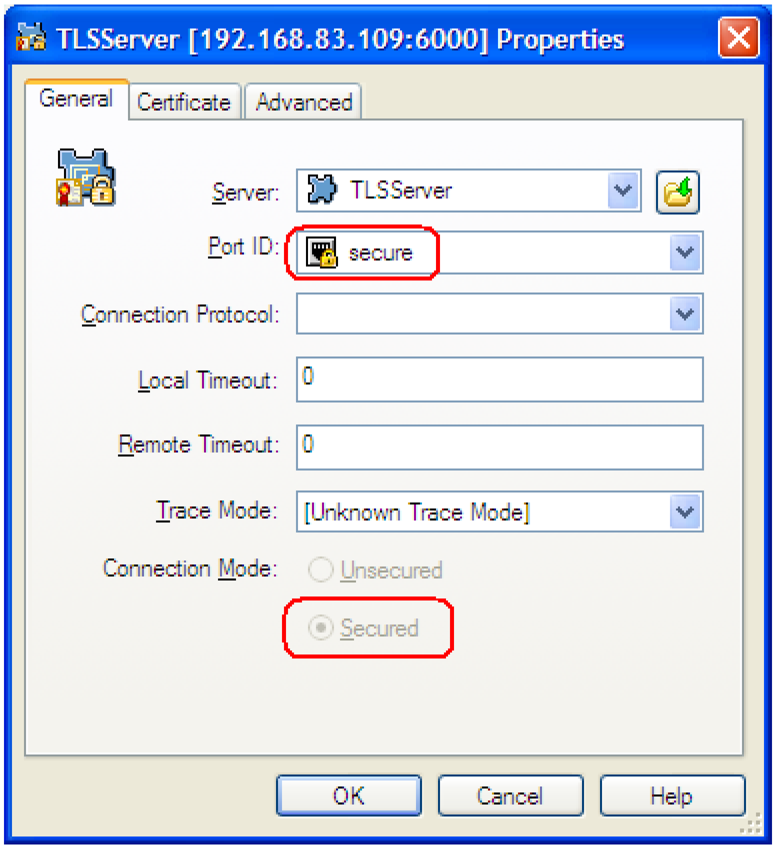
When processing a client Application object, Platform SDK looks at parameters from both sections. If any parameters are specified in both places, then the values from the *Options* tab take precedence.

Connection Object

The properties window for all Application objects includes a *Connection* tab where connections to servers can be added or edited. Each connection determines if TLS mode should be enabled based on port settings for the target server.

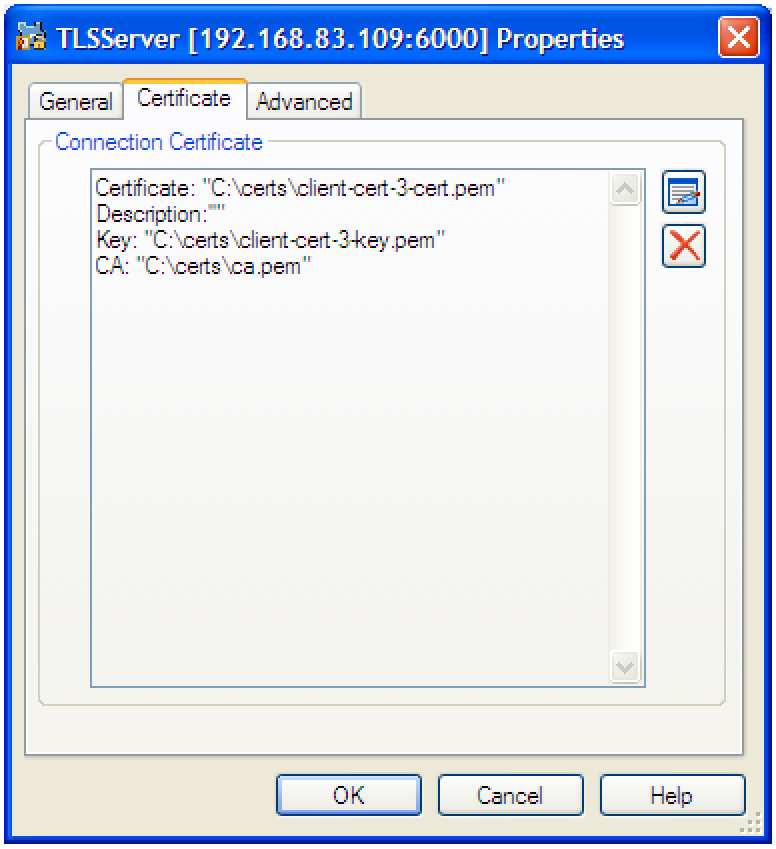


Server	Sec...	Connectio...	
confserv			T...
TLSServer	Yes		U...

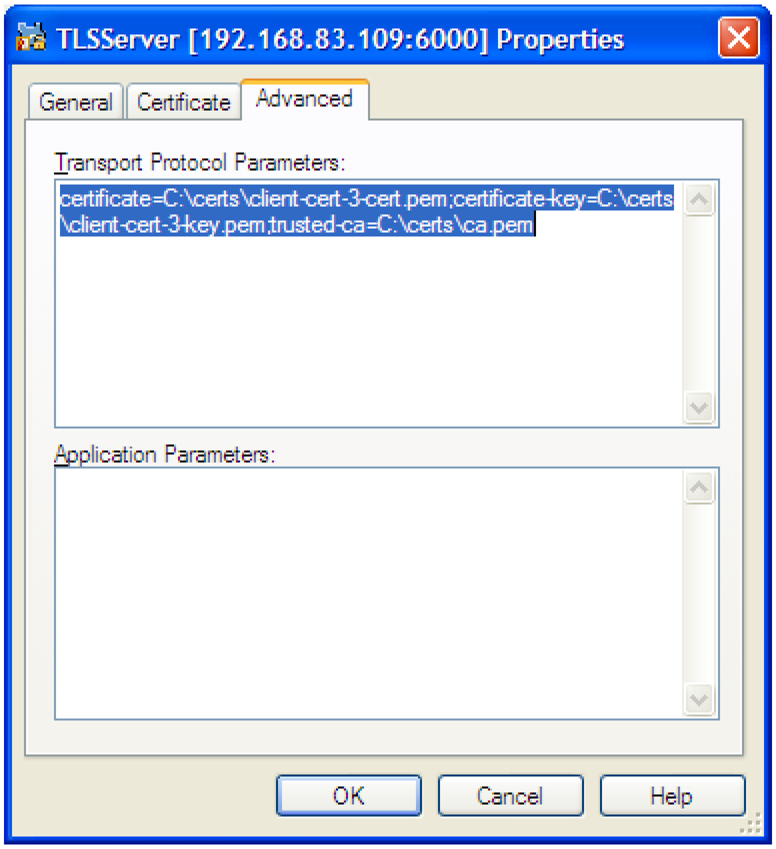


Server: TLSServer
Port ID: secure
Connection Protocol:
Local Timeout: 0
Remote Timeout: 0
Trace Mode: [Unknown Trace Mode]
Connection Mode: Unsecured [✓ Secured]

Similar to the *Port* properties window, the *Certificate* tab allows you to select from a list of certificates or manually edit certificate properties. You can also use the *Advanced* tab to edit TLS settings not included with the certificate. However, the *Transport Protocol Parameters* field behaves differently for this object — which may result in lost or incorrect settings in some cases. See the Notes and Issues section for details.



```
Certificate: "C:\certs\client-cert-3-cert.pem"
Description:""
Key: "C:\certs\client-cert-3-key.pem"
CA: "C:\certs\ca.pem"
```



Transport Protocol Parameters:

```
certificate=C:\certs\client-cert-3-cert.pem;certificate-key=C:\certs
\client-cert-3-key.pem;trusted-ca=C:\certs\ca.pem
```

Application Parameters:

List of TLS Parameters

The following table lists all TLS parameters supported by Platform SDK, with their valid value ranges and purpose:

Parameter Name	Acceptable Values	Purpose	Platform
tls	Boolean value. Possible values are "1"/"0", "yes"/"no", "on"/"off", "true"/"false". Example: • "tls=1"	Client: 1 - perform TLS handshake immediately after connecting to server. 0 – do not turn on TLS immediately but autodetect can still work.	Java, .NET
provider	"PEM", "MSCAPI", "PKCS11", "JKS" Not case-sensitive. Example:	Explicit selection of security provider to be used. For example, MSCAPI and PKCS11 providers can contain all other parameters in their internal database.	Java

Parameter Name	Acceptable Values	Purpose	Platform
	• "provider=MSCAPI"	This parameter allow configuration of TLS through security provider tools.	
certificate	**Java:** PEM provider: path to a X.509 certificate file in PEM format. Path can use both forward and backward slash characters. MSCAPI provider: thumbprint of a certificate – string with hexadecimal SHA-1 hash code of the certificate. Whitespace characters are allowed anywhere within the string. PKCS11 provider: this parameter is ignored. JKS provider: path to a java key store file. If 'provider' option isn't specified implicitly then the file must have 'jks' extension. Examples: • "certificate= C:\certs\client-cert-3-cert.pem" • "certificate=A4 7E A6 E4 7D 45 6A A6 2F 15 BE 89 FD 46 F0 EE 82 1A 58 B9" • "certificate= C:\certs\ mykeystore.jks" **.NET:** Thumbprint of a certificate – string with hexadecimal SHA-1 hash code of the certificate (Whitespace characters are allowed anywhere within the string).	Specifies location of X.509 certificate to be used by application. MSCAPI provider keeps certificates in internal database and can identify them by hash code; so called thumbprint. In Java, PKCS#11 provider does not allow selection of the certificate; it must be configured using provider tools. **Note:** When using autodetect (upgrade) TLS connection, this option MUST be specified in application configuration, otherwise Configuration Server would return empty TLS parameters even if other options are set.	Java, .NET
certificate-key	PEM provider: path to a PKCS#8 private key file without password protection in PEM format. Path can use both vforward and backward slash characters. • MSCAPI provider: this	Specifies location of PKCS#8 private key to be used in pair with the certificate by application. MSCAPI provider keeps private keys paired with certificates in internal database. In Java, PKCS#11 provider does not allow selection of the private	Java

Parameter Name	Acceptable Values	Purpose	Platform
	parameter is ignored; key is taken from the entry identified by "certificate" field. • PKCS11 provider: this parameter is ignored. • JKS provider: this parameter must not be used. Examples: • "certificate-key= C:\certs\client-cert-3-key.pem"	key; it must be configured using provider tools.	
trusted-ca	PEM provider: path to a X.509 certificate file in PEM format. Path can use both forward and backward slash characters. MSCAPI provider: thumbprint of a certificate – string with hexadecimal SHA-1 hash code of the certificate. Whitespace characters are allowed anywhere within the string. PKCS11 provider: this parameter is ignored. Examples: • "trusted-ca= C:\certs\ ca.pem" • "trusted-ca=A4 7E A6 E4 7D 45 6A A6 2F 15 BE 89 FD 46 F0 EE 82 1A 58 B9"	Specifies location of a X.509 certificate to be used by application to validate remote party certificates. The certificate is designated as Trusted Certification Authority certificate and application will only trust remote party certificates signed with the CA certificate. MSCAPI provider keeps CA certificates in internal database and can identify them by hash code; so called thumbprint. In Java, PKCS#11 provider does not allow selection of the CA certificate; it must be configured using provider tools.	Java
tls-mutual	Boolean value. Possible values are "1"/"0", "yes"/"no", "on"/"off", "true"/"false". Example: • "tls-mutual=1"	Has meaning only for server application. Client applications ignore this value. When turned on, server will require connecting clients to present their certificates and validate the certificates the same way as client applications do.	Java, .NET

Parameter Name	Acceptable Values	Purpose	Platform
tls-crl	**Java:** All providers: path to a Certificate Revocation List file in PEM format. Path can use both forward and backward slash characters. Example: • "tls-crl= C:\certs\crl.pem" **.NET:** Boolean value. Enable/disable to check of certificate's revocation. Possible values are "1"/"0", "yes"/"no", "on"/"off", "true"/"false". Example: • "tls-crl=1"	Applications will use CRL during certificate validation process to check if the (seemingly valid) certificate was revoked by CA. This option is useful to stop usage of leaked certificates by unauthorized parties.	Java, .NET
tls-target-name-check	"host" or none. Not case-sensitive. Example: • "tls-target-name-check=host"	When set to "host", enables matching of certificate’s Alternative Subject Name or Subject fields against expected host name. PSDK supports DNS names and IP addresses as expected host names.	Java, .NET
cipher-list	String consisting of space-separated cipher suit names. Information on cipher names can be found online. Example: • "cipher-list= TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA"	Used to calculate enabled cipher suites. Only ciphers present in both the cipher suites supported by security provider and the cipher-list parameter will be valid.	Java
fips140-enabled	Boolean value. Possible values are "1"/"0", "yes"/"no", "on"/"off", "true"/"false". Example: • "fips140-enabled=1"	PSDK Java: when set to true, effectively is the same as setting "provider=PKCS11" since only PKCS11 provider can support FIPS-140. If set to true while using other	Java

Parameter Name	Acceptable Values	Purpose	Platform
		provider type, PSDK will throw exception.	
sec-protocol	String value. Possible values are "SSLv23", "SSLv3", "TLSv1", "TLSv11", "TLSv12". Example: • "sec-protocol= TLSv1"	Starting with PSDK release 8.5.1, an application can specify the exact protocol to send and accept secure connection requests on one or more of its connections.	Java, .NET

tls-version	String value.
	
	This value is the algorithm name used to get an instance of Java `SSLContext`. It is used to create the Java `SSLEngine` used to handle a security protocol.
	
	Standard algorithm names supported by standard java security provider are available in the official Java documentation.
	
	This parameter must be used together with the `protocol-list` option, and was a legal and powerful way to choose a provider before the `sec-protocol` parameter was available.
	
	Examples:
	
	`tls-version=TLSv1.2 protocol-list= TLSv1.2`
	
	`tls-version=TLSv1.2 protocol-list= SSLv3 TLSv1.1 TLSv1.2`
	
	`tls-version=TLSv1.2 protocol-list= TLSv1`
		Together with the `protocol-list` parameter, this value helps to specify one or more security protocols
		
		that can be used (if the Java platform supports it).
		
		This option specifies a Java security protocol name that is used to select an appropriate Java security provider. The actual list of security protocols that can be used depends on:
		
		• Which Java security provider will be selected.
		
		• Which security protocols are supported by the security provider. Sometimes this can be customizable, such as by using the Windows Control Panel > Java > Advanced tab.
		
		• Which security protocols are enabled for use. This is specified in the `protocol-list` parameter.
		
		The default value depends on version of Java in use, and its configuration.
		
		• “TLSv1” for Java 6 and 7
		
		• “TLSv1.2” for Java 8
			Java

Parameter Name	Acceptable Values	Purpose	Platform

protocol-list	String
	
	Java:
	
	A space-separated list of security protocols that can be used. This value is passed to the `setEnabledProtocols` method of the Java SSLEngine, and is the standard way to choose a security provider that supports the requested protocol version from the system providers list.
	
	This parameter must be used together with the `tls-version` option, and was a legal and powerful way to choose a provider before the `sec-protocol` parameter was available.
	
	Examples:
	
	`tls-version=TLSv1.2 protocol-list= TLSv1.2`
	
	`tls-version=TLSv1.2 protocol-list= SSLv3 TLSv1.1 TLSv1.2`
	
	`tls-version=TLSv1.2 protocol-list= TLSv1`
	
	.NET:
	
	A comma-separated list of security protocols that can be used.
		Together with the `tls-version`, helps to specify one or more security protocols that can be used (if the Java platform supports it).
		
		Restricts use of supported security protocols by the selected security provider.
			Java, .NET

Warning

Currently there is no support for options that store JKS keystore, entry and truststore passwords. Thus configuration of the passwords has to be completed programmatically using the following `TLSConfiguration` methods:

- `setKeyStoreCallBackHandler`
- `setKeyStoreEntryCallBackHandler`
- `setTrustStoreCallback`

Notes and Issues

- Key/value pairs in *Transport Protocol Parameters* fields should be separated only with a single semicolon character. Adding space characters to improve readability can cause applications, including those based on Platform SDK, unable to parse these parameters correctly.
- *Transport Protocol Parameters* fields in Configuration Manager are limited to 256 characters in length. Be sure to keep your parameter list as short as possible. For example: certificate thumbprints for MSCAPI provider take 40 characters without spaces and 49 characters with them, and long paths to certificate files can easily eat up all available space.
- The Connection properties window behaves differently from the Port properties window, as described below. Be sure to double-check TLS settings for Connection objects.
  - It does not save content of the *Transport Protocol Parameters* field unless a certificate was selected using UI controls on the *Certificate* tab.
  - If certificate information is deleted from the *Certificate* tab, then all transport protocol parameters are also erased (including those entered manually).
  - In some cases it does not save additional TLS parameters that were entered manually.
- Configuration Server reads its own TLS parameters from Application or from Host object only during startup. If you use an Application or Host object as a source of TLS parameters for Configuration Server, be sure to restart the server after any changes to the parameters.

Using and Configuring Security Providers

Java

Tip

The contents of this page only apply to Java implementations.

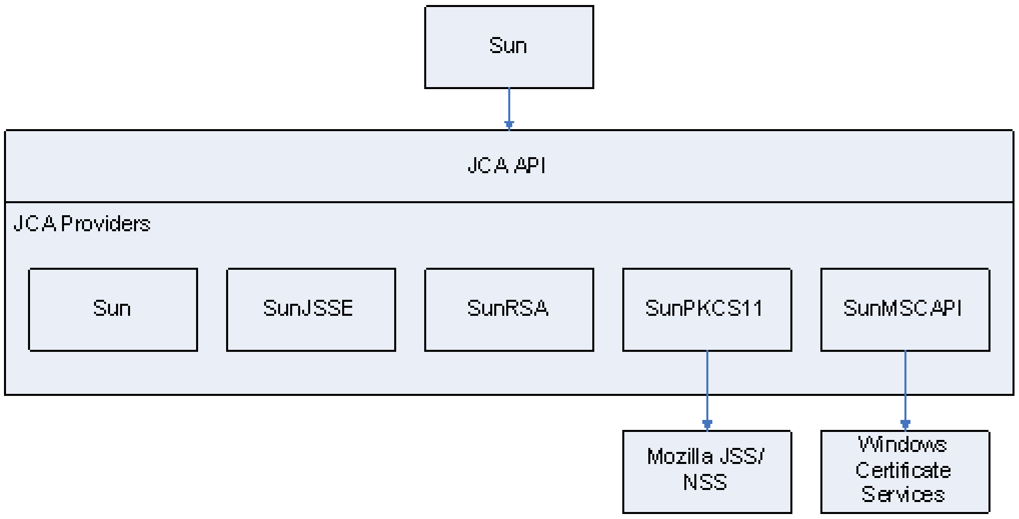
Introduction

This page deals with Security Providers — an umbrella term describing the full set of cryptographic algorithms, data formats, protocols, interfaces, and related tools for configuration and management when used together. The primary reasons for bundling together such diverse tools are: compatibility, support for specific standards, and implementation restrictions.

The security providers listed here were tested with the Platform SDK 8.1.1 implementation of TLS, and found to work reliably when used with the configuration described below.

Java Cryptography Architecture Notes

Java Cryptography Architecture (JCA) provides a general API, and a pluggable architecture for cryptography providers that supply the API implementation.



Some JCA providers (Sun, SunJSSE, SunRSA) come bundled with the Java platform and contain actual algorithm implementations, they are named PEM provider since they are used when working with certificates in PEM files. Some other (SunPKCS11, SunMSCAPI) serve as a façade for external providers. SunPKCS11 supports PKCS#11 standard for pluggable security providers, such as hardware cryptographic processors, smartcards or software tokens. Mozilla NSS/JSS is an example of pluggable software token implementation. SunMSCAPI provides access to Microsoft Cryptography API (MSCAPI), in particular, to Windows Certificate Services (WSC).

PEM Provider: OpenSSL

Note: Working with certificates and keys is also covered in the *Genesys 8.1 Security Deployment Guide*.

PEM stands for "Privacy Enhanced Mail", a 1993 IETF proposal for securing email using public-key cryptography. That proposal defined the PEM file format for certificates as one containing a Base64-encoded X.509 certificate in specific binary representation with additional metadata headers. Here, the term is used to refer to Java built-in security providers that are used in conjunction with certificates and private keys loaded from X.509 PEM files.

One of the most popular free tools for creating and manipulating PEM files is OpenSSL. Instructions for installing and configuring OpenSSL are provided below.

Installing OpenSSL

OpenSSL is available two ways:

- distributed as a source code tarball: http://www.openssl.org/source/
- as a binary distribution (specific links are subject to change): http://www.openssl.org/related/binaries.html

The installation process is very easy when using a binary installer; simply follow the prompts. The only additional step required is to add the *<OpenSSL-home>\bin* folder to your *Path* system variable so that OpenSSL can run from command line directly with the `openssl` command.

Configuring OpenSSL

The OpenSSL configuration file contains settings for OpenSSL itself, and also many field values for the certificates being generated including issuer and subject names, host names and URIs, and so on. You will need to customize your OpenSSL file with your own values before using the tool. An example of a customized configuration file is available here.

The OpenSSL database consists of a set of files and folders, similar to the sample database described in the table below. To start using OpenSSL, this structure should be created manually except for files marked as "Generated by OpenSSL". Other files can be left empty as long as they exist in the expected location.

OpenSSL database file/folder structure

File or Folder	Generated by OpenSSL?	Description
openssl-ca\		
openssl-ca\openssl.cfg		OpenSSL configuration file

File or Folder	Generated by OpenSSL?	Description
openssl-ca\.rnd	Yes	File filled with random data, used in key generation process.
openssl-ca\ca-password.txt		Stores the password for the CA private key. Reduces typing required, but is very insecure. Should only be used for testing and development.
openssl-ca\export-password.txt		Stores the password used to encrypt the private keys when exporting PKCS#12 files. Reduces typing required, but is very insecure. Should only be used for testing and development.
openssl-ca\ca\		CA root folder.
openssl-ca\ca\certs\		All generated certificates are copied here. Folder contents can be safely deleted.
openssl-ca\ca\crl\		Generated CRLs stored here. Folder contents can be safely deleted.
openssl-ca\ca\newcerts\		Certificates being generated are stored here. Folder contents can be safely deleted *once generation process is finished*.
openssl-ca\ca\private\		CA private files.
openssl-ca\ca\private\cakey.pem	Yes	CA private key. Must be kept secret.
openssl-ca\ca\crlnumber		Serial number of last exported CRL.
openssl-ca\ca\serial		Serial number of last signed certificate.
openssl-ca\ca\cacert.pem	Yes	CA certificate.
openssl-ca\ca\index.txt		Textual database of all certificates.

Short Command Line Reference

- This section assumes that the OpenSSL *bin* folder was added to the local PATH environment variable, and that *openssl-ca* is the current folder for all issued commands.
- Placeholders for parameters are shown in the following form: "<param-placeholder>".
- The frequently used parameter "<request-name>" should be a unique name that identifies the certificate files.

Task	Description	Command
Create a CA Certificate/Key	This is performed in three steps: 1. Create CA Private Key 2. Create CA Certificate 3. Export CA Certificate	1. `openssl genrsa -des3 -out ca\private\cakey.pem 1024 -passin file:ca-password.txt` 2. `openssl req -config openssl.cfg -new -x509 -days <days-ca-cert-is-valid> -key ca\private\cakey.pem -out ca\cacert.pem -passin file:ca-password.txt` 3. `openssl x509 -in ca\cacert.pem -outform PEM -out ca.pem`

Create a Leaf Certificate/Key Pair	This is performed in three steps:
	1. Create certificate request. Certificate fields and extensions are defined during this step, and the certificate's public and private keys are created in the process.
	2. Sign the request.
	3. Export the certificate.
		1. `openssl req -new -nodes -out requests\<request name>-req.pem -keyout requests\<request name>-key.pem -days 3650 -config openssl.cfg`
		2. `openssl ca -out requests\<request-name>-signed.pem -days 3650 -config openssl.cfg -passin file:ca-password.txt -infiles requests\<request-name>-req.pem`
		3. `openssl pkcs12 -export -in requests\<request-name>-signed.pem -inkey requests\<request-name>-key.pem -certfile ca\cacert.pem -name "<entry-name-in-p12-file>" -out <request-name>.p12 -passout file:export-password.txt`
		`openssl x509 -in requests\<request-name>-signed.pem -outform PEM -out <request-name>-cert.pem`
		`openssl pkcs8 -topk8 -nocrypt -in requests\<request-name>-key.pem -out <request-name>-key.pem`

Task	Description	Command
Revoke a Certificate		`openssl ca -revoke <certificate-pem-file> -config openssl.cfg -passin file:ca-password.txt`
Export the CRL		`openssl ca -gencrl -crldays <days-crl-is-valid> -out crl.pem -config openssl.cfg -passin file:ca-password.txt`

MSCAPI Provider: Windows Certificate Services

Note: Working with Windows Certificate Services (WCS) is also covered in *Genesys 8.1 Security Deployment Guide*.

MSCAPI stands for Microsoft CryptoAPI. This provider offers the following features:

- It is available only on Windows platform.
- It implies usage of WCS to store and retrieve certificates, private keys, and CA certificates.
- Every Windows account has its own WCS storage, including the System account.
- Depends heavily on OS configuration and system security policies.
- Has its own set of supported cipher suites, different from what is provided by Java.
- When used with Java, please use the latest available version of Java to run the application.
- Java does not support CRLs located in WCS. With Java MSCAPI, CRL should be specified as a file.
- Does not accept passwords from Java code programmatically via CallbackHandler. If private key is password-protected or prompt-protected, OS popup dialog will be shown to user.
- Certificates in WCS are configured using the Certificates snap-in for Microsoft Management Console (MMC).

Note: If the version of Java being used does not support MSCAPI, a "WINDOWS-MY KeyStore not available" exception appears in the application log. If you receive such exceptions, please consider switching to a newer version of Java.

Starting Certificates Snap-in

There are two methods for accessing the Certificates Snap-in:

- Enter "certmgr.msc" at the command line. (This only gives access to Certificates for the current user account.)
- Launch the MMC console and add the Certificates Snap-in for a specific account using the following steps:
  1. Enter "mmc" at the command line.
  2. Select *File > Add/Remove Snap-in...* from the main menu.

3. Select *Certificates* from the list of available snap-ins and click *Add*.
4. Select the account to manage certificates for (see Account Selection for important notes) and click *Finish*.
5. Click *OK*.

Account Selection

It is important to place certificates under the correct Windows account. Some applications are run as services under the Local Service or System account, while others are run under user accounts. The account chosen in MMC must be the same as the account used by the application that certificates are configured for, otherwise the application will not be able to access this WCS storage.

Note: Currently, most Genesys servers do not clearly report this error so WCS configuration must be checked every time there is a problem with the MSCAPI provider.

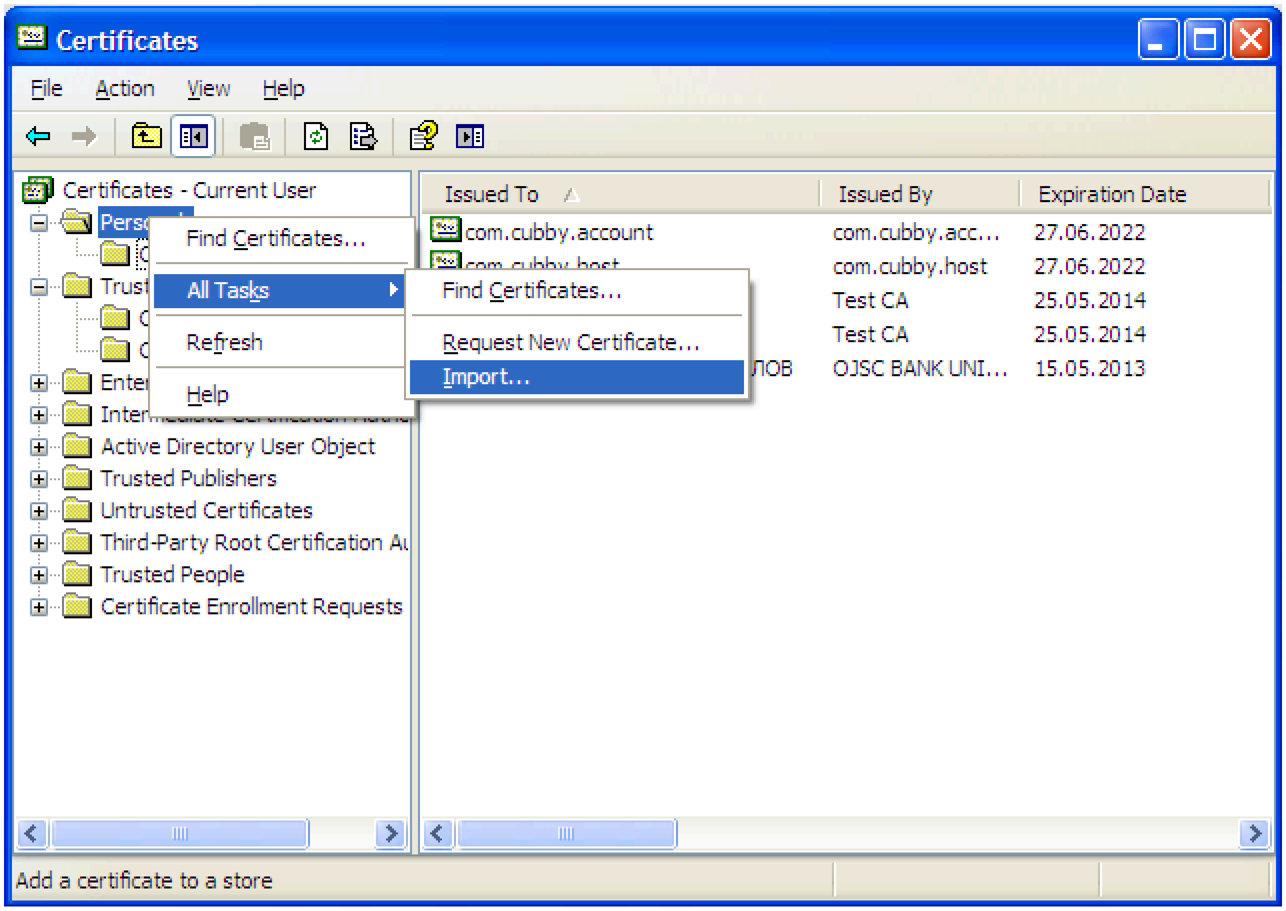
Note: Configuration Manager is also a regular application in this aspect and can access WCS only for the Local Computer (System) account on the local machine. It will not show certificates configured for different accounts or on remote machines. Please consult your system and/or security administrator for questions related to certificate configuration and usage.

Importing Certificates

There are many folders within WCS where certificates can be placed. Only two of them are used by Platform SDK:

- Personal/Certificates – Contains application certificates used by applications to identify themselves.
- Trusted Root Certification Authorities/Certificates – Contains CA certificates used to validate remote party certificates.

To import a certificate, right-click the appropriate folder and choose *All Tasks > Import...* from the context menu. Follow the steps presented by the Certificate Import Wizard, and once finished the imported certificate will appear in the certificates list.



Issued To	Issued By	Expiration Date
com.cubby.account	com.cubby.acc...	27.06.2022
com.cubby.host	com.cubby.host	27.06.2022
	Test CA	25.05.2014
	Test CA	25.05.2014
ЛОВ	OJSC BANK UNI...	15.05.2013

Add a certificate to a store

Although WCS can import X.509 PEM certificate files, these certificates cannot be used as application certificates because they do not contain a private key. It is not possible to attach a private key from a PKCS#7 PEM file to the imported certificate. To avoid this problem, import application certificates only from PKCS#12 files (*.*p12*) which contain a certificate and private key pair.

CA certificates do not have private keys attached, so it is safe to import CA certificates from X.509 PEM files.

It is possible to copy and paste certificates between folders and/or user accounts in the Management Console, but this approach is not recommended due to WCS errors which may result in the pasted certificate having an inaccessible private key. This error is not visible in Console, but applications would not be able to read the private key. A recommended and reliable workaround is to export the certificate to a file and then import from that file.

If you encounter the following error in the application log: “The credentials supplied to the package were not recognized”, the most likely cause is due to the private key being absent or inaccessible. In this case try deleting the certificate from WCS and re-importing it.

Importing CRL Files

CRL files can be imported to the following folder in WCS:

- Trusted Root Certification Authorities/Certificate Revocation List

The import procedure is the same as for importing certificate. CRL file types are automatically recognized by the import wizard.

Note: Although an MSCAPI provider may choose to use CRL while validating remote party certificates, this functionality is not guaranteed and/or supported by Platform SDK. Platform SDK implements its own CRL matching logic using CRL PEM files.

PKCS11 Provider: Mozilla NSS

PKCS11 stands for the PKCS#11 family of Public-Key Cryptography Standards (PKCS), published by RSA Laboratories. These standards define platform-independent API-to-cryptographic tokens, such as Hardware Security Modules (HSM) and smart cards, allowing you to connect to external certificate storage devices and/or cryptographic engines.

In Java, the PKCS#11 interface is a simple pass-through and all processing is done externally. When used together with a FIPS-certified security provider, such as Mozilla NSS, the whole provider chain is FIPS-compliant.

Platform SDK uses PKCS11 because it is the only way to achieve FIPS-140 compliance with Java.

Installing Mozilla NSS

Currently Platform SDK only supports FIPS when used with the Mozilla NSS security provider. (Java has FIPS certification only when working with a PKCS#11-compatible pluggable security provider, and the only provider with FIPS certification and Java support is Mozilla NSS.)

Note: In theory, BSafe can be used since it supports JCA interfaces. However, Platform SDK was not tested with RSA BSafe and such system would not be FIPS-certifiable as a while.

Generally, some security parameters and data must be configured on client host, requiring the involvement of a system/security administrator. At minimum, the client host must have a copy of the CA Certificate to be able to validate the Configuration Server certificate. The exact location of the CA certificate depends on the security provider being used. It can be present as a PEM file, Java Keystore file, a record in WCS, or as an entry in the Mozilla NSS database. Once the application is connected to Configuration Server, the Application Template Application Block can be used to extract connection parameters from Configuration Server and set up TLS.

Mozilla NSS is the most complex security provider to deploy and configure. In order to use NSS, the following steps must be completed:

1. Deploy Mozilla NSS.
2. Create Mozilla NSS database (a "soft token" in terms of NSS), and set it to FIPS mode.
3. Adjust the Java security configuration, or implement dynamic loading for the Mozilla NSS provider.
4. Import the CA certificate to the Mozilla NSS database.
5. Use the Platform SDK interface to select PKCS11 as a provider (with no specific configuration options required).

Configuring FIPS Mode in Mozilla NSS

To configure FIPS mode in Mozilla NSS, create a file named *nss-client.cfg* in Mozilla NSS deployment folder with the following values configured:

- name - Name of a software token.
- nssLibraryDirectory - Library directory, located in the Mozilla NSS deployment folder.
- nssSecmodDirectory - Folder where the Mozilla NSS database for the listed software token is located.
- nssModule - Indicates that FIPS mode should be used.

An example is provided below:

```
name = NSSfips
nssLibraryDirectory = C:/nss-3.12.4/lib
nssSecmodDirectory = C:/nss-3.12.4/client
nssModule = fips
```

More information about configuring FIPS mode can be found using external resources:

- https://davidvaleri.wordpress.com/2010/10/19/using-nss-for-fips-140-2-compliant-transport-security-in-cxf/ external sources

Configuring FIPS Mode in Java Runtime Environment (JRE)

To configure your Java runtime to use Mozilla NSS, the *java.security* file should be located in Java deployment folder and edited as shown below:

(Changes are shown in **bold red**, insertions are shown in **bold blue**)

```
#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=sun.security.ec.SunEC
#security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.net.ssl.internal.ssl.Provider SunPKCS11-NSSfips
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
security.provider.8=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.9=sun.security.smartcardio.SunPCSC
security.provider.10=sun.security.mscapi.SunMSCAPI
security.provider.11=sun.security.pkcs11.SunPKCS11 C:/nss-3.12.4/nss-client.cfg
```

After those updates are complete, the Java runtime instance works with FIPS mode, with only the PKCS#11/Mozilla NSS security provider enabled.

Short Command Line Reference

Please refer to the following reference for more information:

- https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Tools

Task	Command
Create CA Certificate	`certutil -S -k rsa -n "<CA-cert-name>" -s "CN=Test CA, OU=Miratech, O=Genesys, L=Kyiv, C=UA" -x -t "CTu,u,u" -m 600 -v 24 -d ./client -f "<keystore-password-file>"`
Import CA Certificate	`certutil -A -a -n “<CA-cert-name>” -t "CTu,u,u" -i <ca-cert-file> -d ./client -f "<keystore-password-file>"`
Create New Leaf Certificate	`certutil -S -k rsa -n "<cert-name>" -s "CN=Test CA, OU=Miratech, O=Genesys, L=Kyiv, C=UA" -x -t "u,u,u" -m 666 -v 24 -d ./client -f "<keystore-password-file>" -z "<noise-file>"`
Import Leaf Certificate	`pk12util -i <cert-file.p12> -n <cert-name> -d ./client -v -h "NSS FIPS 140-2 Certificate DB" -K <keystore-password>`
Create CRL	`crlutil -d ./client -f "<keystore-password-file>" -G -c "<crl-script-file>" -n "<CA-cert-name>" -l SHA512`
Modify CRL	`crlutil -d ./client -f "<keystore-password-file>" -M -c "<crl-script-file>" -n "<CA-cert-name>" -l SHA512 -B`
Show Certificate Information	`certutil -d ./client -f "<keystore-password-file>" -L -n "<cert-name>"`
Show CRL Information	`crlutil -d ./client -f "<keystore-password-file>" -L -n "<CA-cert-name>"`
List Certificates	`certutil -d ./client –L`
List CRLs	`crlutil -L -d ./client`

JKS Provider: Java Built-in

This provider is supported by the Platform SDK Commons library, but the Application Template Application Block does not support this provider due to compatibility guidelines with Genesys Framework Deployment.

This provider can only be used when TLS is configured programmatically by Platform SDK users.

Short Command Line Reference

Refer to the following references for more information:

- https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html
- https://docs.oracle.com/javase/8/docs/technotes/tools/windows/keytool.html

Task	Command
Creating and Importing - These commands allow you to generate a new Java Keytool keystore file,	

Task	Command
create a Certificate Signing Request (CSR), and import certificates. Any root or intermediate certificates will need to be imported before importing the primary certificate for your domain.	
Generate a Java keystore and key pair	`keytool -genkey -alias mydomain -keyalg RSA -keystore keystore.jks -keysize 2048`
Generate a certificate signing request (CSR) for an existing Java keystore	`keytool -certreq -alias mydomain -keystore keystore.jks -file mydomain.csr`
Import a root or intermediate CA certificate to an existing Java keystore	`keytool -import -trustcacerts -alias root -file Thawte.crt -keystore keystore.jks`
Import a signed primary certificate to an existing Java keystore	`keytool -import -trustcacerts -alias mydomain -file mydomain.crt -keystore keystore.jks`
Generate a keystore and self-signed certificate	`keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -storepass password -validity 360 -keysize 2048`
Java Keytool Commands for Checking - If you need to check the information within a certificate, or Java keystore, use these commands.	
Check a stand-alone certificate	`keytool -printcert -v -file mydomain.crt`
Check which certificates are in a Java keystore	`keytool -list -v -keystore keystore.jks`
Check a particular keystore entry using an alias	`keytool -list -v -keystore keystore.jks -alias mydomain`
Other Java Keytool Commands	
Delete a certificate from a Java Keytool keystore	`keytool -delete -alias mydomain -keystore keystore.jks`
Change a Java keystore password	`keytool -storepasswd -new new_storepass -keystore keystore.jks`
Export a certificate from a keystore	`keytool -export -alias mydomain -file mydomain.crt -keystore keystore.jks`
List Trusted CA Certs	`keytool -list -v -keystore $JAVA_HOME/jre/lib/security/cacerts`
Import New CA into Trusted Certs	`keytool -import -trustcacerts -file /path/to/ca/ca.pem -alias CA_ALIAS -keystore $JAVA_HOME/jre/lib/security/cacerts`

OpenSSL Configuration File

This page provides an example of a customized OpenSSL configuration file that has been edited to work with the Platform SDK implementation of TLS. For more details about OpenSSL and how it relates to the Platform SDK implementation of TLS, refer to the Using and Configuring Security Providers page.

Sample File

Customized file content is listed below.

- Changes are marked with **bold red**.
- Added lines are marked with **bold blue**.

```
#
# OpenSSL example configuration file.
# This is mostly being used for generation of certificate requests.
#

# This definition stops the following lines choking if HOME isn't
# defined.
HOME                        = .
RANDFILE               = $ENV::HOME/.rnd

# Extra OBJECT IDENTIFIER info:
#oid_file               = $ENV::HOME/.oid
oid_section              = new_oids

# To use this configuration file with the "-extfile" option of the
# "openssl x509" utility, name here the section containing the
# X.509v3 extensions to use:
# extensions                = 
# (Alternatively, use a configuration file that has only
# X.509v3 extensions in its main [= default] section.)

[ new_oids ]

# We can add new OIDs in here for use by 'ca', 'req' and 'ts'.
# Add a simple OID like this:
# testoid1=1.2.3.4
# Or use config file substitution like this:
# testoid2=${testoid1}.5.6

# Policies used by the TSA examples.
tsa_policy1 = 1.2.3.4.1
tsa_policy2 = 1.2.3.4.5.6
tsa_policy3 = 1.2.3.4.5.7

####################################################################
[ ca ]
default_ca        = CA_default               # The default ca section

####################################################################
```

```
[ CA_default ]

dir                = ./ca                        # Where everything is kept
certs                = $dir/certs                # Where the issued certs are kept
crl_dir                = $dir/crl                # Where the issued crl are kept
database        = $dir/index.txt        # database index file.
#unique_subject        = no                        # Set to 'no' to allow creation of
                                        # several ctificates with same subject.
new_certs_dir        = $dir/newcerts                # default place for new certs.

certificate        = $dir/cacert.pem         # The CA certificate
serial                = $dir/serial                # The current serial number
crlnumber        = $dir/crlnumber        # the current crl number
                                        # must be commented out to leave a V1 CRL
crl                = $dir/crl.pem                # The current CRL
private_key        = $dir/private/cakey.pem # The private key
RANDFILE        = $dir/private/.rand        # private random number file

x509_extensions        = usr_cert                # The extentions to add to the cert

# Comment out the following two lines for the "traditional"
# (and highly broken) format.
name_opt         = ca_default                # Subject Name options
cert_opt         = ca_default                # Certificate field options

# Extension copying option: use with caution.
# copy_extensions = copy

# Extensions to add to a CRL. Note: Netscape communicator chokes on V2 CRLs
# so this is commented out by default to leave a V1 CRL.
# crlnumber must also be commented out to leave a V1 CRL.
# crl_extensions        = crl_ext

default_days        = 365                        # how long to certify for
default_crl_days= 30                        # how long before next CRL
default_md        = sha256                # SHA-1 is deprecated, so use SHA-2 instead
preserve        = no                        # keep passed DN ordering

# A few difference way of specifying how similar the request should look
# For type CA, the listed attributes must be the same, and the optional
# and supplied fields are just that :-)
policy                = policy_anything

# For the CA policy
[ policy_match ]
countryName                = match
stateOrProvinceName        = match
organizationName        = match
organizationalUnitName        = optional
commonName                = supplied
emailAddress                = optional

# For the 'anything' policy
# At this point in time, you must list all acceptable 'object'
# types.
[ policy_anything ]
countryName                = optional
stateOrProvinceName        = optional
localityName                = optional
organizationName        = optional
organizationalUnitName        = optional
commonName                = supplied
emailAddress                = optional
```

```
####################################################################
[ req ]
default_bits                = 1024
default_keyfile          = privkey.pem
distinguished_name         = req_distinguished_name
attributes                 = req_attributes
x509_extensions         = v3_ca        # The extentions to add to the self signed cert

# Passwords for private keys if not present they will be prompted for
# input_password = secret
# output_password = secret

# This sets a mask for permitted string types. There are several options.
# default: PrintableString, T61String, BMPString.
# pkix         : PrintableString, BMPString (PKIX recommendation before 2004)
# utf8only: only UTF8Strings (PKIX recommendation after 2004).
# nombstr : PrintableString, T61String (no BMPStrings or UTF8Strings).
# MASK:XXXX a literal mask value.
# WARNING: ancient versions of Netscape crash on BMPStrings or UTF8Strings.
string_mask = utf8only

req_extensions = v3_req # The extensions to add to a certificate request

[ req_distinguished_name ]
countryName                        = Country Name (2 letter code)
countryName_default                = UA
countryName_min                        = 2
countryName_max                        = 2

stateOrProvinceName                = State or Province Name (full name)
stateOrProvinceName_default        = None

localityName                        = Locality Name (eg, city)
localityName_default         = Kyiv

0.organizationName                = Organization Name (eg, company)
0.organizationName_default        = Genesys

# we can do this but it is not needed normally :-)
#1.organizationName                = Second Organization Name (eg, company)
#1.organizationName_default        = World Wide Web Pty Ltd

organizationalUnitName                = Organizational Unit Name (eg, section)
organizationalUnitName_default        = Engineering

commonName                        = Common Name (that is, server FQDN or YOUR name)
commonName_default         = xpigors
commonName_max                        = 64

emailAddress                        = Email Address
emailAddress_max                = 64

# SET-ex3                         = SET extension number 3

[ req_attributes ]
challengePassword                = A challenge password
challengePassword_min                = 0
challengePassword_max                = 20

unstructuredName                = An optional company name

[ usr_cert ]
```

```
# These extensions are added when 'ca' signs a request.

# This goes against PKIX guidelines but some CAs do it and some software
# requires this to avoid interpreting an end user certificate as a CA.

basicConstraints=CA:FALSE

# Here are some examples of the usage of nsCertType. If it is omitted
# the certificate can be used for anything *except* object signing.

# This is OK for an SSL server.
# nsCertType                        = server

# For an object signing certificate this would be used.
# nsCertType = objsign

# For normal client use this is typical
# nsCertType = client, email

# and for everything including object signing:
# nsCertType = client, email, objsign

# This is typical in keyUsage for a client certificate.
# keyUsage = nonRepudiation, digitalSignature, keyEncipherment

# This will be displayed in Netscape's comment listbox.
nsComment                        = "OpenSSL Generated Certificate"

# PKIX recommendations harmless if included in all certificates.
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer

# This stuff is for subjectAltName and issuerAltname.
# Import the email address.
#subjectAltName=issue:copy
subjectAltName = @alt_names
# An alternative to produce certificates that aren't
# deprecated according to PKIX.
# subjectAltName=email:move

# Copy subject details
# issuerAltName=issuer:copy

#nsCaRevocationUrl = http://www.domain.dom/ca-crl.pem
#nsBaseUrl
#nsRevocationUrl
#nsRenewalUrl
#nsCaPolicyUrl
#nsSslServerName

# This is required for TSA certificates.
# extendedKeyUsage = critical,timeStamping

[ v3_req ]

# Extensions to add to a certificate request

basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
subjectAltName = @alt_names

[ alt_names ]
```

```
DNS.1 = hostname.emea.int.genesyslab.com
DNS.2 = hostname
IP.1 = 192.168.1.1
IP.2 = fe80::21d:7dff:fe0d:682c
IP.3 = fe80::ffff:ffff:fffd
IP.4 = fe80::5efe:192.168.1.1
URI.1 = http://hostname/
URI.2 = https://hostname/
email.1 = UserName1@genesyslab.com
email.2 = UserName2@genesyslab.com

[ v3_ca ]


# Extensions for a typical CA


# PKIX recommendation.

subjectKeyIdentifier=hash

authorityKeyIdentifier=keyid:always,issuer

# This is what PKIX recommends but some broken software chokes on critical
# extensions.
#basicConstraints = critical,CA:true
# So we do this instead.
basicConstraints = CA:true

# Key usage: this is typical for a CA certificate. However since it will
# prevent it being used as an test self-signed certificate it is best
# left out by default.
# keyUsage = cRLSign, keyCertSign

# Some might want this also
# nsCertType = sslCA, emailCA

# Include email address in subject alt name: another PKIX recommendation
# subjectAltName=email:copy
# Copy issuer details
# issuerAltName=issuer:copy

# DER hex encoding of an extension: beware experts only!
# obj=DER:02:03
# Where 'obj' is a standard or added object
# You can even override a supported extension:
# basicConstraints= critical, DER:30:03:01:01:FF

[ crl_ext ]

# CRL extensions.
# Only issuerAltName and authorityKeyIdentifier make any sense in a CRL.

# issuerAltName=issuer:copy
authorityKeyIdentifier=keyid:always

[ proxy_cert_ext ]
# These extensions should be added when creating a proxy certificate

# This goes against PKIX guidelines but some CAs do it and some software
# requires this to avoid interpreting an end user certificate as a CA.

basicConstraints=CA:FALSE
```

```
# Here are some examples of the usage of nsCertType. If it is omitted
# the certificate can be used for anything *except* object signing.

# This is OK for an SSL server.
# nsCertType                        = server

# For an object signing certificate this would be used.
# nsCertType = objsign

# For normal client use this is typical
# nsCertType = client, email

# and for everything including object signing:
# nsCertType = client, email, objsign

# This is typical in keyUsage for a client certificate.
# keyUsage = nonRepudiation, digitalSignature, keyEncipherment

# This will be displayed in Netscape's comment listbox.
nsComment                        = "OpenSSL Generated Certificate"

# PKIX recommendations harmless if included in all certificates.
subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid,issuer

# This stuff is for subjectAltName and issuerAltname.
# Import the email address.
# subjectAltName=email:copy
# An alternative to produce certificates that aren't
# deprecated according to PKIX.
# subjectAltName=email:move

# Copy subject details
# issuerAltName=issuer:copy

#nsCaRevocationUrl = http://www.domain.dom/ca-crl.pem
#nsBaseUrl
#nsRevocationUrl
#nsRenewalUrl
#nsCaPolicyUrl
#nsSslServerName

# This really needs to be in place for it to be a proxy certificate.
proxyCertInfo=critical,language:id-ppl-anyLanguage,pathlen:3,policy:foo

####################################################################
[ tsa ]

default_tsa = tsa_config1        # the default TSA section

[ tsa_config1 ]

# These are used by the TSA reply generation only.
dir                = ./demoCA                # TSA root directory
serial                = $dir/tsaserial        # The current serial number (mandatory)
crypto_device        = builtin                # OpenSSL engine to use for signing
signer_cert        = $dir/tsacert.pem         # The TSA signing certificate
                                        # (optional)
certs                = $dir/cacert.pem        # Certificate chain to include in reply
                                        # (optional)
signer_key        = $dir/private/tsakey.pem # The TSA private key (optional)
```

```
default_policy        = tsa_policy1                # Policy if request did not specify it
                                        # (optional)
other_policies        = tsa_policy2, tsa_policy3        # acceptable policies (optional)
digests                = md5, sha1                # Acceptable message digests (mandatory)
accuracy        = secs:1, millisecs:500, microsecs:100        # (optional)
clock_precision_digits  = 0        # number of digits after dot. (optional)
ordering                = yes        # Is ordering defined for timestamps?
                                # (optional, default: no)
tsa_name                = yes        # Must the TSA name be included in the reply?
                                # (optional, default: no)
ess_cert_id_chain        = no        # Must the ESS cert id chain be included?
                                # (optional, default: no)
```

Use Cases

Introduction

This page examines TLS functionality as a series of common use cases. Use cases are broken into two categories: server or application.

Examples and explanations are provided for some use cases, while others simply provide links to the related TLS documentation needed to understand the functionality.

Genesys Server Use Cases

Opening a TLS Port

Code snippets explaining how to open a basic TLS port are provided both with, and without using the Application Template Application Block:

- Opening a TLS port using the Platform SDK Commons Library
- Opening a TLS port using the Application Template Application Block

Opening a Mutual TLS Port (With Expiration, Revocation and CA Checks)

This use case is an advanced variation on opening a simple TLS port. As such, it already has a CA and expiration check, but needs additional parameters to turn on mutual mode and to enable a CRL check.

Mutual Mode

If TLS is configured programmatically, then the *mutualTLS* parameter should be set to *true* when creating an *SSLExtendedOptions* object:

```
SSLExtendedOptions sslOptions = new SSLExtendedOptions(true, (String) null);
```

If TLS is configured in Configuration Manager, then the *tls-mutual* parameter for the server port, application or host should be set to *1*. Please refer to the list of TLS parameters for details.

Revocation Check

If TLS is configured programmatically, then a valid path to the CRL file should be provided in the *crlFilePath* parameter when creating a trust manager:

```
X509TrustManager tm = TrustManagerHelper.createPEMTrustManager(
        "c:/cert/ca-cert.pem","c:/cert/crl.pem", null);
```

If TLS is configured in Configuration Manager, then the *tls-crl* parameter for the server port, application or host should contain the path to the CRL file located on server. Please refer to the list of TLS parameters for details.

Opening a FIPS-Compliant Port

FIPS mode is not a property of a port or application; it is defined mostly by the type of security provider in use and the OS/environment settings. For Java, the PKCS#11 security provider should be used to support FIPS; for .Net, FIPS is configured at the OS level (http://technet.microsoft.com/en-us/library/cc750357.aspx).

If TLS is configured programmatically, then a PKCS11 key/trust managers should be used:

```
X509TrustManager tm = TrustManagerHelper.createPKCS11TrustManager(
        new DummyPasswordCallbackHandler(), (String) null);
X509ExtendedKeyManager km = KeyManagerHelper.createPKCS11KeyManager(
        new DummyPasswordCallbackHandler());
```

If TLS is configured in Configuration Manager, then the *fips140-enabled* parameter for the server port, application or host should be set to "1". Please refer to the list of TLS parameters for details.

Note: This parameter is used to detect the security provider type to use. If this setting conflicts with other TLS parameters or points to a FIPS security provider that is not installed on host, then Platform SDK will generate an exception when attempting to accept or open a connection.

Genesys Application Use Cases

Opening a TLS Connection to a TLS Autodetect Server Port

TLS autodetect ports (also called upgrade mode ports) allow you to establish an unsecured connection to the server before specifying TLS settings. For details, please refer to Connecting to Upgrade Mode Ports in the quick start instructions.

Opening a TLS Connection to a Backend Server (With Expiration, Revocation and CA Checks)

Code snippets explaining how to open a basic TLS connection to a backend server are provided both with, and without using the Application Template Application Block:

- Configuring TLS for Client Connections using the Platform SDK Commons Library
- Configuring TLS for Client Connections using the Application Template Application Block

Opening a FIPS-Compliant Connection to a FIPS-Compliant Port

In this use case, the application does not need to provide any special behavior because the server will only handshake for FIPS-compliant ciphers. Details about setting up a FIPS-compliant port are described above.

Ensuring the Certificate is Checked with CA

If TLS is configured programmatically, then a valid CA certificate data should be provided when creating the trust manager:

```
X509TrustManager tm = TrustManagerHelper.createPEMTrustManager(
        "c:/cert/ca-cert.pem","c:/cert/crl.pem", null);
```

If TLS is configured in Configuration Manager, then the *trusted-ca* parameter for the port, connection, application or host should contain valid CA certificate data. Please refer to the list of TLS parameters for details.

Note: CA certificates are configured differently for each type of security provider. Please refer to the page on using and configuring security providers for detailed information.

Ensuring the Certificate Expiration is Checked

Certificate expiration is checked by default during the certificate validation process.

Note: If a server certificate is placed in a trusted certificates store on the client host, it will be automatically trusted without any validation. A trust certificates store should not include application certificates; instead, it should contain only CA certificates.

Handling a Certificate Revocation List

If TLS is configured programmatically, then a valid path to a CRL file should be provided in the *crlFilePath* parameter when creating trust manager:

```
X509TrustManager tm = TrustManagerHelper.createPEMTrustManager(
        "c:/cert/ca-cert.pem","c:/cert/crl.pem", null);
```

If TLS is configured in Configuration Manager, then the *tls-crl* parameter for the application connection, application or host should contain the path to the CRL file located on the application's host. Please refer to the list of TLS parameters for details.

Handling a User-Specified Cipher List

If TLS is configured programmatically, then the *enabledCipherSuites* constructor parameter should contain a list of allowed ciphers when the *SSLExtendedOptions* object is being created:

```
SSLExtendedOptions sslOptions = new SSLExtendedOptions(
        true, "TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA " +
        "TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA");
```

If TLS is configured in Configuration Manager, then the *cipher-list* parameter for the port, connection, application or host should be set to contain list of allowed ciphers. Please refer to the list of TLS parameters for details.

Using and Configuring TLS Protocol Versions

Java

The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating applications. Since there are various versions of TLS (1.0, 1.1, 1.2, and any future versions) and SSL (2.0 and 3.0), there needs to be a way to negotiate which specific protocol version to use. The TLS protocol provides a built-in mechanism for version negotiation so as not to bother other protocol components with the complexities of version selection.

Tip

By default, the following client-side TLS Protocol versions are enabled in Java:

- Java 6, Java 7 – TLSv1
- Java 8, Java 11 – TLSv1.2

Platform SDK for Java includes extended TLS support so that you can specify which version of TLS to use. There are several ways to accomplish this, as described below.

1) Using the `ConnectionConfiguration` class:

```
KeyValueConfiguration config = new KeyValueConfiguration(new KeyValueCollection());
config.setTLSEnabled(true);
config.setTLSVersion(TLSConfiguration.TLS_VERSION_1_1);
Endpoint ep = new Endpoint(host,  port, config);
ExternalServiceProtocol protocol = new ExternalServiceProtocol(ep);
```

2) Using the `TLSConfiguration` class:

```
String tlsVersion = TLSConfiguration.TLS_VERSION_1;
TLSConfiguration config = new TLSConfiguration();
config.setVersion(tlsVersion);
SSLContext sslContext = TLSConfigurationHelper.createSslContext(config);
SSLExtendedOptions sslOptions = TLSConfigurationHelper.createSslExtendedOptions(config);
ExternalServiceProtocol protocol = new ExternalServiceProtocol(new Endpoint("TLSClient",
host, port, connectionConfiguration, true, sslContext, sslOptions));
```

3) Using the `SSLContextHelper` and `TLSConfigurationHelper` classes:

```
KeyManager keyManager = new KeyManager() {  };
KeyManager[] keyManagers = {keyManager};
TrustManager trustManager = new TrustManager() { };
TrustManager[] trustManagers = {trustManager};
```

```
SecureRandom secureRandom = new SecureRandom();
SSLContextHelper sslContextHelper = new SSLContextHelper();
SSLContext sslContext = sslContextHelper.createSSLContext(keyManagers, trustManagers,
secureRandom, TLSConfiguration.TLS_VERSION_DEFAULT);
```

4) Using system property settings:

```
System.setProperty("com.genesyslab.platform.commons.connection.tlsDefaultVersion",
TLSConfiguration.TLS_VERSION_1_1);

PsdkCustomization.setOption(PsdkOption.PsdkTlsDefaultVersion,
TLSConfiguration.TLS_VERSION_1_2);
```

.NET

The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating applications. Since there are various versions of TLS (1.0, 1.1, 1.2, and any future versions) and SSL (2.0 and 3.0), there needs to be a way to negotiate which specific protocol version to use. The TLS protocol provides a built-in mechanism for version negotiation so as not to bother other protocol components with the complexities of version selection.

Tip

The list of available TLS protocol versions, with respect to .NET Framework versions, is provided below:

1. .NET Framework 3.5, 4.0: Ssl2, Ssl3, Tls (TLS 1.0).
2. .NET Framework 4.5.x, 4.6.x, 4.7.x: Ssl2, Ssl3, Tls (TLS 1.0), Tls11 (TLS 1.1), Tls12 (TLS 1.2).

For all .NET Framework versions, setting the TLS protocol version to `SslProtocols.Default` (which is the initial value) means that both SSL 3.0 and TLS 1.0 are acceptable for secure communications.

Platform SDK for .NET includes extended TLS support so that you can specify which version of TLS to use. There are several ways to accomplish this, as described below.

1) Using the `TlsSupport` class.

Server side:

```
var serverSslProtocols = SslProtocols.Tls;
var server = new ServerChannel(new Endpoint(host, port), new
ExternalServiceProtocolFactory());
server.TlsProperties.EnabledSslProtocols = serverSslProtocols;
```

Client side:

```
var clientSslProtocols = SslProtocols.Tls12;
var protocol = new ExternalServiceProtocol(new Endpoint(host, port));
protocol.TlsProperties.EnabledSslProtocols = clientSslProtocols;
```

2) Using the application configuration file:

```
<configuration>
  <appSettings>
    <add key="DefaultClientTLSVersion" value="TLS12 , SSL3"/>
    <add key="DefaultServerTLSVersion" value="tls11"/>
  </appSettings>
</configuration>
```

3) Using the `PsdkCustomization` class:

```
PsdkCustomization.TLSServerVersion.Value = SslProtocols.Tls12;
PsdkCustomization.TLSClientVersion.Value = SslProtocols.Tls11;
```

Lazy Parsing of Message Attributes

This page provides:

- an overview and list of requirements for the lazy parsing feature
- design details explaining how this feature works
- code examples showing how to implement lazy parsing in your applications

Introduction to Lazy Parsing

Lazy parsing allows users to specify which attributes should always be parsed immediately, and which attributes should be parsed only on demand.

Some complex attributes (such as the `ConfObject` attribute found in some Configuration Server protocol messages) are large and very complex. Unpacking these attributes can be time-consuming and, in cases when an application is not interested in that data, can affect program performance. This issue is resolved by using the "lazy parsing" feature included with the Platform SDK 8.1 release, which is described in this article.

When this feature is turned off, all message attributes are parsed immediately - which is normal behavior for previous version of the Platform SDK. When lazy parsing is enabled, any attributes that were tagged for lazy parsing are only parsed on demand. In this case, if the application does not explicitly check the value of an attribute tagged for lazy parsing then that attribute is never parsed at all.

Feature Overview

- Platform SDK includes configuration options to turn the lazy parsing functionality on or off for each individual protocol that supports this feature.
- Potentially time-consuming attributes that are candidates for lazy parsing are selected and marked by Platform SDK developers. Refer to your Platform SDK API Reference for details.
- To maintain backwards compatibility, there is no change in how user applications access attribute values.
- Default values:
  - In Platform SDK for Java, the lazy parsing feature is turned **on** by default.

    Note: The default value changed in release 8.1.4; in earlier releases of Platform SDK for Java, lazy parsing is turned off by default.
  - In Platform SDK for .NET, the lazy parsing feature is turned **off** by default.

Java

Important

This feature requires Configuration SDK protocol release 8.1 or later, and is currently only used with the `EventObjectsRead.ConfObject` property of the Configuration Platform SDK.

Design Details

This section describes the main classes and interfaces you will need to be familiar with to implement lazy parsing in your own application.

Enabling and Disabling the Lazy Parsing Feature

At any time, a running application can enable or disable lazy parsing for a specific protocol in just a few lines of code. This is done in three easy steps:

1. Create a new `KeyValueCollection` object.
2. Set the appropriate value for the `Connection.LAZY_PARSING_ENABLED_KEY` field. A value of `True` enables the feature, while `False` disables lazy parsing.
3. Use a `KeyValueConfiguration` object to apply that setting to the desired protocol(s).

Tip

Starting with release 8.1.4, the default value of the `Connection.LAZY_PARSING_ENABLED_KEY` field is always `True`, with the lazy parsing feature enabled.

Once lazy parsing mode is enabled for a protocol, the change is applied immediately. Every new message that is received takes the lazy parsing setting into account: parsing entire messages if the feature is disabled, or leaving some attributes unparsed until their values are requested if the feature is enabled.

To enable lazy parsing for the Configuration Server protocol, an application would use the following code:

```
[Java]

KeyValueCollection kv = new KeyValueCollection();
kv.addString(Connection.LAZY_PARSING_ENABLED_KEY, "true");
KeyValueConfiguration kvcfg = new KeyValueConfiguration(kv);
ConfServerProtocol cfgChannel = new ConfServerProtocol(endpoint);
cfgChannel.configure(kvcfg); //lazy parsing is immediately active after this line
```

To disable lazy parsing for the protocol only the second line of code is changed before re-applying the configuration, as shown below:

```
[Java]

kv.addString(Connection.LAZY_PARSING_ENABLED_KEY, "false");
```

.NET

Important

This feature requires Configuration SDK protocol release 8.1 or later, and is currently only used with the `EventObjectsRead.ConfObject` property of the Configuration Platform SDK.

Design Details

This section describes the main classes and interfaces you will need to be familiar with to implement lazy parsing in your own application.

Enabling and Disabling the Lazy Parsing Feature

At any time, a running application can enable or disable lazy parsing for a specific protocol in just a few lines of code. This is done in three easy steps:

1. Create a new `KeyValueCollection` object.
2. Set the appropriate value for the `CommonConnection.LazyParsingEnabledKey` field. A value of `True` enables the feature, while `False` disables lazy parsing.
3. Use a `KeyValueConfiguration` object to apply that setting to the desired protocol(s).

Tip

The default value of the `CommonConnection.LazyParsingEnabledKey` field is always `False`, with the lazy parsing feature disabled.

Once lazy parsing mode is enabled for a protocol, the change is applied immediately. Every new message that is received takes the lazy parsing setting into account: parsing entire messages if the feature is disabled, or leaving some attributes unparsed until their values are requested if the feature is enabled.

To enable lazy parsing for the Configuration Server protocol, an application would use the following code:

```
[C#]

KeyValueCollection kvc = new KeyValueCollection();
kvc[CommonConnection.LazyParsingEnabledKey] = "true";
KeyValueConfiguration kvcfg = new KeyValueConfiguration(kvc);
ConfServerProtocol cfgChannel = new ConfServerProtocol(endpoint);
cfgChannel.Configure(kvcfg); //lazy parsing is immediately active after this line
```

To disable lazy parsing for the protocol only the second line of code is changed before re-applying the configuration, as shown below:

```
[C#]

kvc[CommonConnection.LazyParsingEnabledKey] = "false";
```

Accessing Attribute Values

There is no difference in how applications access attribute values from returned messages. Whether the lazy parsing feature is enabled or disabled, whether the attribute being access supports lazy parsing or not, your code remains exactly the same.

However, you should consider differences in timing when accessing attribute values.

- When lazy parsing is disabled, the entire message is parsed immediately when it is received. Accessing attribute values is very fast, as the requested information is already prepared.
- When lazy parsing is enabled, the delay to parse the message upon arrival is smaller but accessing any attributes that support lazy parsing causes a slightly delay as that information must first be parsed. Note that accessing the same attribute a second time will not result in the attribute information being parsed a second time; Platform SDK saves parsed data.

Additional Notes

- XML Serialization — The `XmlMessageSerializer` class has been updated to support lazy parsing. If a message that contains unparsed attributes is serialized, then `XmlMessageSerializer` will trigger parsing before the serialization process begins.
- `ToString` method — Use of the `ToString` method does not trigger parsing of attributes that support lazy parsing. In this case, each unparsed attribute has its name printed along with a value of: "<value is not yet parsed>".

Log Filtering

Introduction

Debug Log Level in Platform SDK protocol may affect Application performance due to huge log information output.

The aim is to introduce the ability to dynamically configure the verbosity of Platform SDK message logging. This way, production applications will be able to provide appropriate traces for troubleshooting without hurting performance with overly verbose logs.

This feature should provide ability to set message filtering, for defining which messages should be logged and which should not.

Message filter can be executed only when Debug log level is enabled.

Create and assign message filter directly

Use `setLogMessageFilter()` to assign custom log filter implementation for the protocol objects:

```
protocol.setLogMessageFilter(new MessageFilter(){
    public boolean isMessageAccepted(Message message) {
        if(message.messageId()==123) {
            return true;
        }
        else {
              return false;
        }
    }
});
```

This filter allows to log messages with ID equals to 123.

It is possible to assign default filter implementation, see described below.

Use Application Template to setup filters

Application Template provides default filter implementations. This filter can read configuration and handle updates from Configuration Server.

Default filter implementation should be wired with a client protocol using `FilterConfigurationHelper.bind()` method.

User needs to provide application name where filter configuration was defined and Config Service (to read application):

```
import com.genesyslab.platform.applicationblocks.com.ConfService;
import com.genesyslab.platform.applicationblocks.com.objects.CfgApplication;
import com.genesyslab.platform.applicationblocks.com.queries.CfgApplicationQuery;
import com.genesyslab.platform.apptemplate.configuration.ClientConfigurationHelper;
import com.genesyslab.platform.apptemplate.configuration.GCOMApplicationConfiguration;
import com.genesyslab.platform.apptemplate.configuration.IGApplicationConfiguration.IGAppConnConfiguration;
import com.genesyslab.platform.configuration.protocol.types.CfgAppType;
import com.genesyslab.platform.commons.protocol.Endpoint;
import com.genesyslab.platform.reporting.protocol.StatServerProtocol;

import com.genesyslab.platform.apptemplate.filtering.FilterConfigurationHelper;

public class MyApp {
   public void init() {

        ...
        //read application settings and create protocol
        String appName = "my-app-name";
        CfgApplication cfgApplication = confService.retrieveObject(
                   CfgApplication.class, new CfgApplicationQuery(appName));
        GCOMApplicationConfiguration appConfiguration =
                   new GCOMApplicationConfiguration(cfgApplication);
        IGAppConnConfiguration connConfig = appConfiguration.getAppServer(CfgAppType.CFGStatServer);

        Endpoint endpoint= ClientConfigurationHelper.createEndpoint(
                        appConfiguration, connConfig,
                        connConfig.getTargetServerConfiguration());

         StatServerProtocol statProtocol = new StatServerProtocol(endpoint);
         statProtocol.setClientName(clientName);

         //assign message filters to the protocol
         FilterConfigurationHelper.bind(statProtocol, appConfiguration, confService);

         statProtocol.open();
    }
}
```

Important

For manually created Endpoints the server host and port must match the server host and port of the `IGAppConnConfiguration` object (corresponds to one of the "Connections" tab entries in provided application). Otherwise the `ConfigurationException` "No connection object was found in application for protocol endpoint..." will be thrown. However this will not happen if the `Endpoint` has been created using the `ClientConfigurationHelper.createEndpoint()` helper.

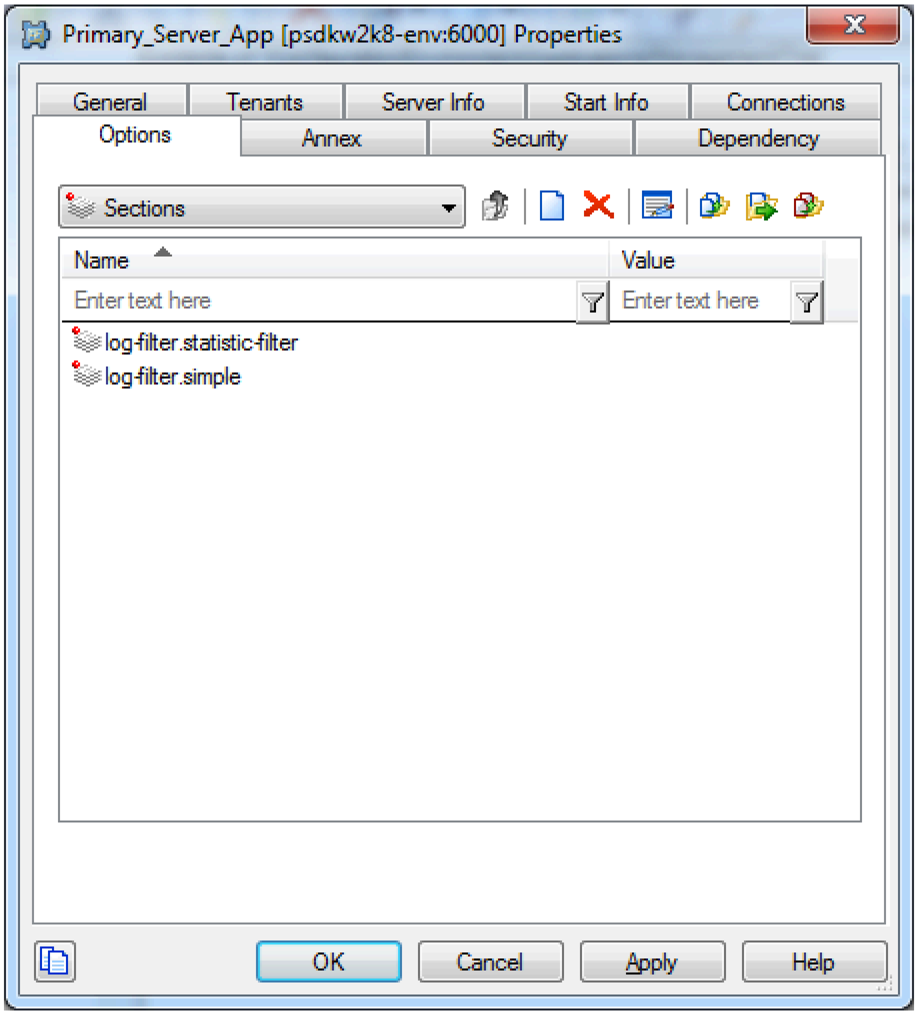
Helper method `FilterConfigurationHelper.bind()` reads application configuration, instantiates filter objects, assigns them to protocol, subscribe for Configuration Server notifications, registers handlers for protocol events and so on. When filters are not required anymore, release filtering infrastructure:

```
FilterConfigurationHelper.unbind(statProtocol,confService);
```

Use Configuration Manager to define filters, as described below.

Define filter in Configuration Manager

Filters are defined in the application "options" tab in configuration manager.



log-filter.statistic-filter
log-filter.simple

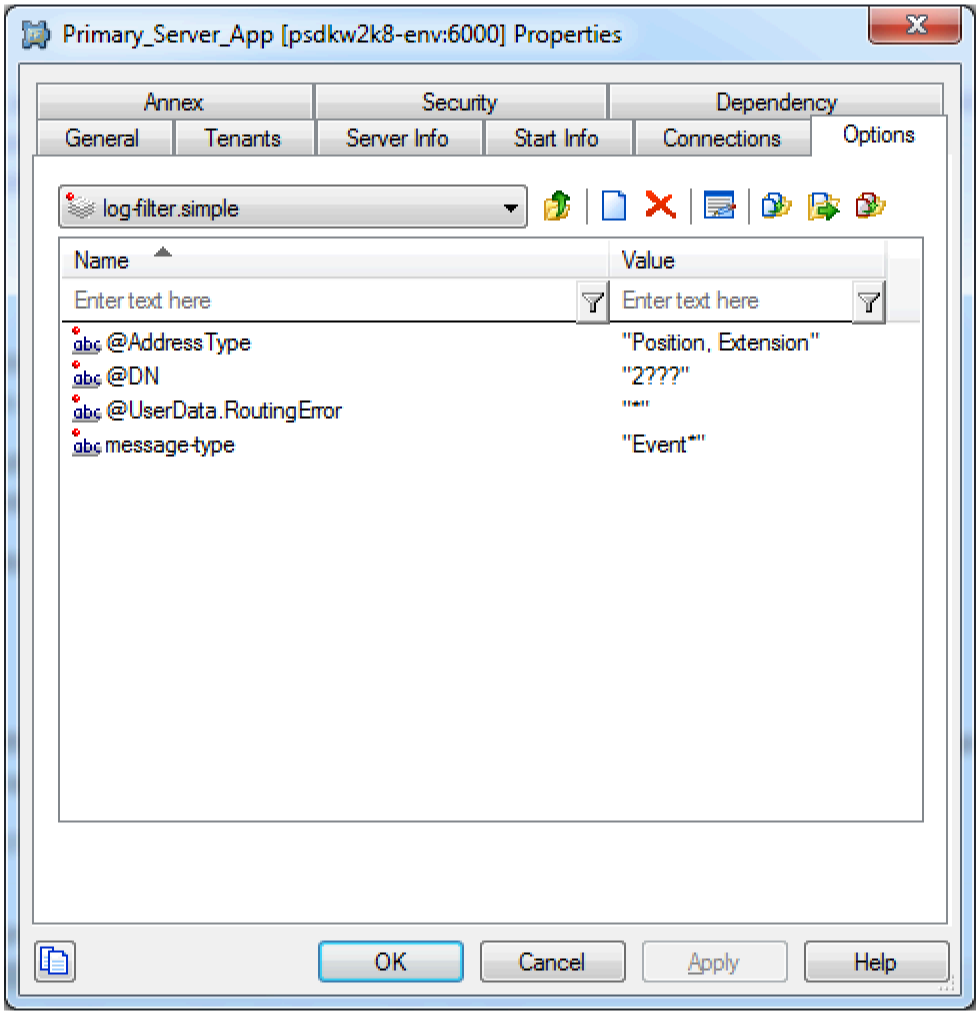
Filter name must be preceded with "log-filter." prefix.

For example: "`log-filter.my-filter`"

Filter names cannot start with "-", "!", or space symbols. Names such as "log-filter.-somefilter" are not allowed.

Filter options are specified under the "log-filter.name" section.

Example: Define filter for T-Server channel which will show only incoming events for DNs that are types of Positions or Extension and match "2???". Events should have user data with key "ROUTING_ERROR". Here is how configuration can be done:

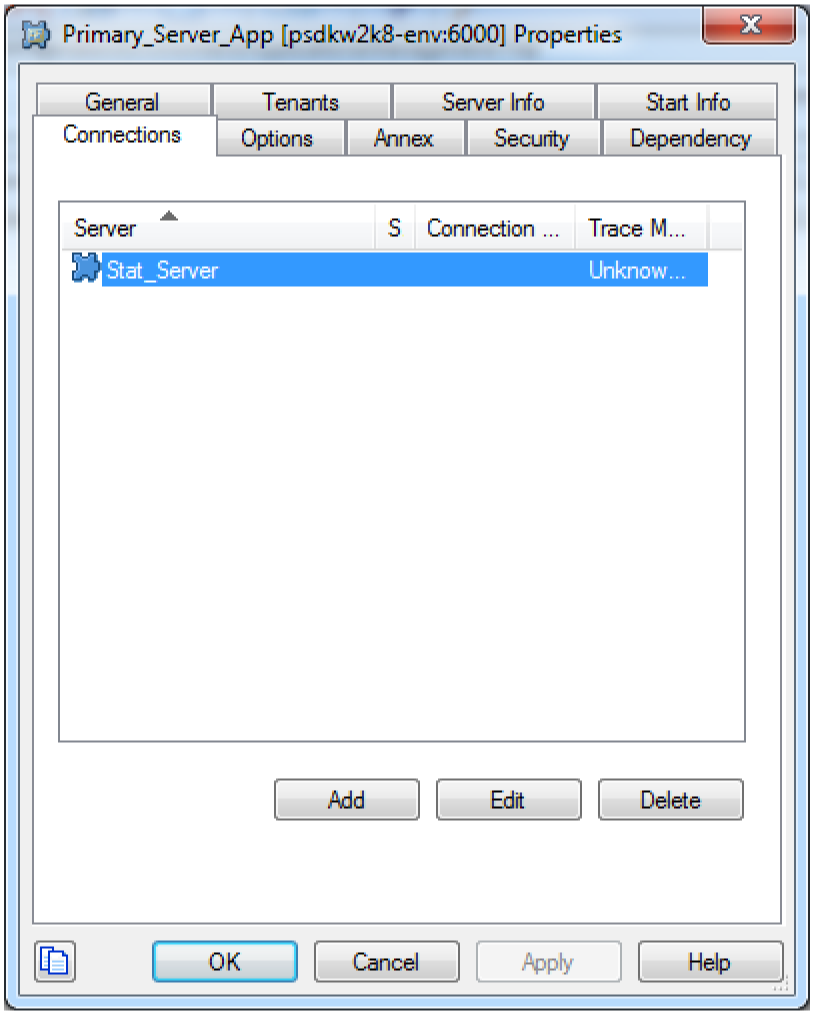


Name	Value
@AddressType	"Position, Extension"
@DN	"2???"
@UserData.RoutingError	"*"
message-type	"Event*"

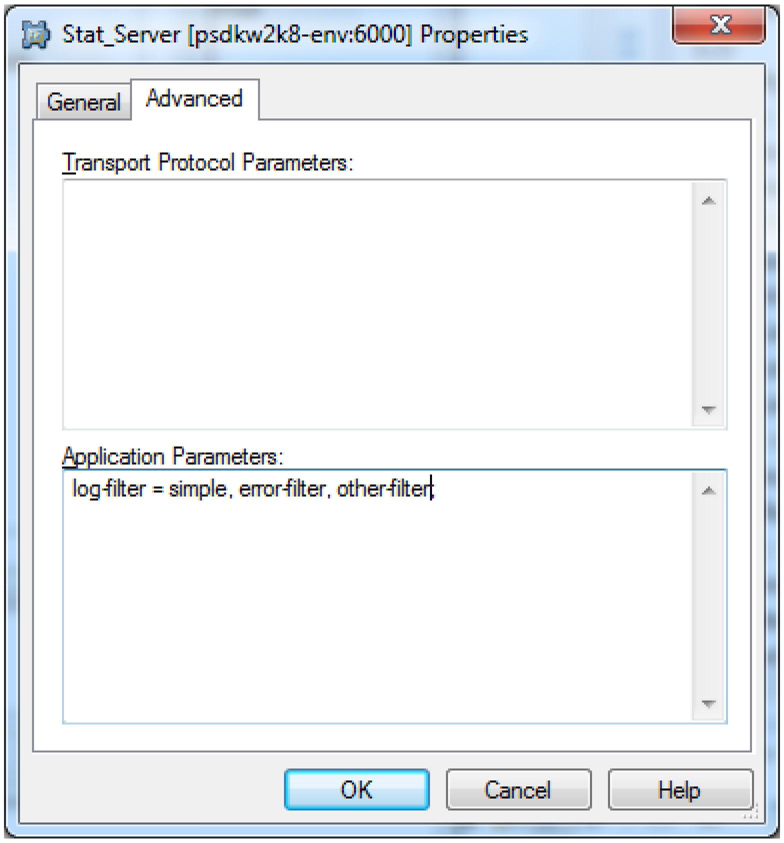
Filter options can represent one or more message attribute conditions.

```
log-filter.simple
    message-type = Event*
    @DN = 2???
    @AddressType = Position, Extension
    @UserData.RoutingError = *
```

After the filter is defined, assign it to one or more protocols on the application’s "Connection" tab.



Server	S	Connection ...	Trace M...
Stat_Server			Unknow...


Stat_Server [psdkw2k8-env:6000] Properties
General
Advanced
Transport Protocol Parameters:
Application Parameters:
OK
Cancel
Help

log-filter = simple, error-filter, other-filter

In Application Parameters it is possible to specify one or more filters for a protocol:

```
log-filter = simple, error-filter, other-filter
```

See Filter Chain for details.

Filter Syntax

It is possible to specify one or more conditions to filter messages by their names or (and) attribute values.

List of elementary conditions evaluate with "AND" operation.

Message Type Conditions

Evaluates message name or message id.

```
message-type = constant value
```

Example:

Option	Description
`message-type = EventInfo`	Filter accepting only EventInfo message
`message-type = 125,126,127`	Filter accepting messages with id 125,126 and 127.
`message-type = Event*`	Filter accepting all Events

Attribute conditions

Evaluates message attribute value.

```
@attribute-name = constant value
```

Example:

Option	Description
`@ReferenceId = 50`	Matches ReferenceId attribute value with 50.
`@DN = 2???`	Matches DN attribute value with 4-character string starting from "2". Attribute values like "2999" or "2ef7" will be accepted. Value "299999" will not be accepted.
`@UserData.CustomerID = 87624FAC`	Matches "CustomerID" key of the complex "UserData" attribute with "87624FAC" value.
`@StatisticObject.ObjectId = place*`	Matches statistic object which name start from "place"

Attribute names

Attribute name is specified after the "@" symbol. Attribute names should match getter name of the corresponding message class. To find out attribute name, see API reference guide for the corresponding message.

For example:

```
message.getStatisticObject().getObjectType().equals(...);
```

equals to filter condition

```
"@StatisticObject.ObjectType = ... "
```

To access sub element of the complex attribute (`KeyValueCollection, CompoundValue`), specify the name of inner element. Names should be delimited with "." symbol:

```
@attribute-name.element-1.element-n.
```

Supported attribute types

Currently supported message attribute types:

- string,
- int,

- enum,
- KeyValueCollection,
- CompoundValue (complex attributes like StatisticObject.)

If attribute has complex type, it is possible to specify matching condition only for one of its inner elements of a simple type: *string*, *integer* or *enum*. For example, here is how to specify condition for "TenantName" element of the complex "StatisticObject" attribute:

```
@StatisticObject.TenantName = 101
```

Constant values

Condition can have one or more constant values, delimited with ",". If one of the specified constants matching attribute value (or message name), condition will return true.

Constant values supports wildcards:

Symbol	Description
`*`	Any sequence of symbols. For example, "Event*" matching any message name, starting from Event
?	Any symbol at specified position. For example, 555?5 will match 555A5 or 55505 strings.
\	Escape symbol. For example, 555\?5 will match only 555?5 string.

Constant examples:

```
@AgentPlace =  place1000
```

```
@AgentPlace = place*
```

```
@AgentPlace = place100??
```

```
@AgentPlace = place110, place120, place2??
```

Inverted conditions

It is possible to invert attribute condition or message condition result by specifying "#not" prefix:

```
#not message-type = RequestAgentLogin
```

or

```
#not @DN = 10
```

Empty filters

Filter with empty options will return negative evaluation result. It can be used to deny all messages for a protocol object.

Stateful filters

Sometimes user doesn't know what attribute value should be specified in a filter condition. For example, in Statistic protocol, the `ReferenceId` attribute (which uniquely identifies `EventInfo` message with statistic data), initializing at a runtime, during statistic opening. To find out `ReferenceId` value, user needs to search corresponding `RequestOpenStatistic` in logs.

Use case: trace `EventInfo` messages with any statistic for "place100".

Default log filter implementation allows to save attribute value from one message and re-use it to trace other messages.

Configuration sample:

```
log-filter.stat-filter
    message-type = RequestOpenStatistic
    @StatisticObject.ObjectId = place100
    trace-on-attribute = ReferenceId
    trace-until-message-type = EventStatisticClosed
```

When filter meets condition ("RequestOpenStatistic" message with "place100" statistic object), the ReferenceId is added to the list of saved values.

When filter receives message ("EventInfo" in statistics protocol) with `ReferenceId` matching to any of the saved values, it allows to log the messages. More then one statistics for "place100" could be opened, so it is possible to store several `ReferenceId` values.

When filter receives "EventStatisticClosed" with `ReferenceId` matching to any of the previously saved values, this value is removed from the list. When values list is empty, no messages could be logged.

Note 1: Filters processing messages only when debug log level is enabled. In order to save `ReferenceId` value, debug log level should be enabled before `RequestOpenStatistic` is sent to server.

Note 2: Saved values are cleared upon protocol close.

Note 3: Number of saved values is limited to 1024 to prevent high memory consumption upon incorrect filter conditions. It can be changed with system property "com.genesyslab.platform.filtering.valuelist.capacity". Changing to greater value is not recommended.

Filter Chain

List of filters on the "Connection tab" represent a filter chain. By default, filter chain evaluates filter results as "OR" expression. If one of the filter accept message message will be logged and other filters will not be evaluated. Filters can be of two types: "accept" and "deny" filters

Use cases

Filter type	Use case
accept (default)	User exactly knows criteria by which messages should be logged. For example, log all Events and Requests with "DN" attribute "2000".
deny	User see a lot of unneeded messages in log with common data. User can specify "deny" filter to truncate those messages.

Filter chain behavior

Filter type	Message evaluation result	Filter chain behavior
accept (default)	TRUE\FALSE	If TRUE - allow log, do not execute other filters. Otherwise, execute next filter. Deny log if last filter returned FALSE
deny	TRUE\FALSE	If TRUE - deny log, do not execute other filters. Otherwise, execute next filter. Allow log if last filter returned FALSE

Syntax

Filter name, prefixed with "-" means "deny" filter. Names without prefix mean "accept" filter.

Example:

`"log-filter = -filter"`

`"log-filter = f1, f2, -f3"`.

Filter result negation

Optionally, it is possible to invert message evaluation result for a filter with "!" symbol.

Example:

`"log-filter = !filter"`

`"log-filter = f1, !f2, f3"`

```
"log-filter = f1, -f2, -!f3".
```

Special filters

While delivering message from TCP connection to the client's receiver (or in opposite direction), Platform SDK can trace message on the different points of its way:

```
2014-07-31 15:07:38,168 [New I/O worker #1] DEBUG otocolMessagePackagerImpl  - New message #2
....
2014-07-31 15:07:38,168 [New I/O worker #1] DEBUG ns.protocol.DuplexChannel  - Complete
message handling: 2
```

It is possible to disable such log entries with special filter "skip-trace-msg". This filter can be specified as a stand-alone filter, or can be used together with other filters in a filter chain:

Example:

```
log-filter = skip-trace-msg
log-filter = filter-1, filter-2, filter-n, skip-trace-msg
```

Hide or Tag Sensitive Data in Logs

The Genesys Security Deployment Guide describes common options to filter out or tag sensitive data in logs (in `KeyValueCollection` entries).

- The `default-filter-type` option in the `[log-filter]` section defines the treatment for all Key-Value pairs.
- The <`key-name`> options in the `[log-filter-data]` section define the treatment for specific keys in the log, overriding the default treatment specified by `default-filter-type`.

Corresponding configurations can also be applied for the Platform SDK `KeyValuePrinter`:

```
KeyValuePrinter.setDefaultPrinter(new KeyValuePrinter(globalPrinterOpts,
individualKeyMapping));
```

where `globalPrinterOpts` and `individualKeyMapping` are `KeyValueCollection` objects with filter names and filter options.

Using Default Filters

Most `KeyValueCollection` objects (`CfgApplication` configuration options) can be read from Configuration Server and applied to the `KeyValuePrinter` directly:

```
CfgApplication application = ...;

KeyValueCollection options = application.getOptions();
KeyValueCollection globalPrinterOpts= options.getList("log-filter");
KeyValueCollection individualKeyMapping = options.getList("log-filter-data");
KeyValuePrinter.setDefaultPrinter(new KeyValuePrinter(globalPrinterOpts,
individualKeyMapping));
```

Prior to 8.5.102.00, standard tag filters configuration could not be applied as-is and required additional parsing.

The table below demonstrates filter samples from the Genesys Security Deployment Guide and corresponding `KeyValuePrinter` settings.

Masking Partial Values	
Configuration options in Administrator	**Corresponding KeyValueCollection content**

[log-filter]
default-filter-type=hide-first,3
	[Java]
	`KeyValueCollection globalPrinterOpts = new KeyValueCollection();`
	`globalPrinterOpts.addString("default-filter-type", "hide-first,3");`
	`KeyValuePrinter.setDefaultPrinter(new KeyValuePrinter(globalPrinterOpts , null));`

Masking Partial Values	

	[.NET]
	
	`var globalPrinterOpts = new KeyValueCollection();`
	`globalPrinterOpts.Add("default-filter-type", "hide-first,3");`
	`KeyValuePrinter.DefaultPrinter = new KeyValuePrinter(globalPrinterOpts , null);`

KVList:
 	
`    'DNIS' [str] = "***0"`
`    'PASSWORD' [str] = "***111111"`
`    'RECORD_ID' [str] = "***3427"`	

Using Default Tags	
Configuration options in Administrator	**Corresponding KeyValueCollection content**

[log-filter]
 	
default-filter-type=tag()
	[Java]
	
	`KeyValueCollection globalPrinterOpts = new KeyValueCollection();`
	`globalPrinterOpts.addString("default-filter-type", "tag()");`
	`KeyValuePrinter.setDefaultPrinter(new KeyValuePrinter(globalPrinterOpts , null));`
	
	[Java, Prior to 8.5.102.00]
	
	`KeyValueCollection globalPrinterOpts = new KeyValueCollection();`
	`globalPrinterOpts.addString("default-filter-type", "custom-filter");`
	`globalPrinterOpts.addString("custom-filter-type", "PrefixPostfixFilter");`
	`KeyValueCollection filterOpts = new KeyValueCollection();`
	`filterOpts.addString("key-prefix-string", "");`
	`filterOpts.addString("key-postfix-string", "");`
	`filterOpts.addString("value-prefix-string", "<#");`
	`filterOpts.addString("value-postfix-string", "#>");`
	`globalPrinterOpts.addList("custom-filter-options", filterOpts);`
	`KeyValuePrinter.setDefaultPrinter(new KeyValuePrinter(globalPrinterOpts, null));`
	
	[.NET]
	
	`var globalPrinterOpts = new KeyValueCollection();`
	`globalPrinterOpts.Add("default-filter-type", "tag()");`
	`KeyValuePrinter.DefaultPrinter = new KeyValuePrinter(globalPrinterOpts , null);`

Masking Partial Values	

	[.NET Prior to 8.5.102.00]
	
	`var globalPrinterOpts = new`
	`KeyValueCollection();`
	`globalPrinterOpts.Add("default-filter-`
	`type", "custom-filter");`
	`globalPrinterOpts.Add("custom-filter-type",`
	
	`"Genesyslab.Platform.Commons.Collections.Filters.PrefixPostfi`
	`var filterOpts = new KeyValueCollection();`
	`filterOpts.Add("key-prefix-string", "");`
	`filterOpts.Add("key-postfix-string", "");`
	`filterOpts.Add("value-prefix-string", "<#");`
	`filterOpts.Add("value-postfix-string",`
	`"#>");`
	`globalPrinterOpts.Add("custom-filter-`
	`options", filterOpts);`
	`KeyValuePrinter.DefaultPrinter = new`
	`KeyValuePrinter(globalPrinterOpts, null);`

KVList:
 	
`'DNIS' [str] = <#"8410"#>`
`'PASSWORD' [str] = <#"111111111"#>`
`'RECORD_ID' [str] = <#"8313427"#>`	

Using User-defined Tags for All Attributes	
Configuration options in Administrator	**Corresponding KeyValueCollection content**

[log-filter]
 	
default-filter-type=tag(<**,**>)
	[Java]
	
	`KeyValueCollection globalPrinterOpts = new`
	`KeyValueCollection();`
	`globalPrinterOpts.addString("default-filter-`
	`type", "tag(<**,**>)");`
	`KeyValuePrinter.setDefaultPrinter(new`
	`KeyValuePrinter(globalPrinterOpts , null));`
	
	[Java, Prior to 8.5.102.00]
	
	`KeyValueCollection globalPrinterOpts = new`
	`KeyValueCollection();`
	`globalPrinterOpts.addString("default-filter-`
	`type", "custom-filter");`
	`globalPrinterOpts.addString("custom-filter-`
	`type", "PrefixPostfixFilter");`
	`KeyValueCollection filterOpts = new`
	`KeyValueCollection();`
	`filterOpts.addString("key-prefix-string",`
	`"");`
	`filterOpts.addString("key-postfix-string",`
	`"");`
	`filterOpts.addString("value-prefix-string",`
	`"<**");`
	`filterOpts.addString("value-postfix-`
	`string", "**>");`
	`globalPrinterOpts.addList("custom-filter-`

Masking Partial Values	

	`options", filterOpts);`
	`KeyValuePrinter.setDefaultPrinter(new`
	`KeyValuePrinter(globalPrinterOpts, null));`
	
	[.NET]
	
	`var globalPrinterOpts = new`
	`KeyValueCollection();`
	`globalPrinterOpts.Add("default-filter-`
	`type", "tag(<**,**>)");`
	`KeyValuePrinter.DefaultPrinter = new`
	`KeyValuePrinter(globalPrinterOpts , null);`
	
	[.NET Prior to 8.5.102.00]
	
	`var globalPrinterOpts = new`
	`KeyValueCollection();`
	`globalPrinterOpts.Add("default-filter-`
	`type", "custom-filter");`
	`globalPrinterOpts.Add("custom-filter-type",`
	
	`"Genesyslab.Platform.Commons.Collections.Filters.PrefixPostfi`
	`var filterOpts = new KeyValueCollection();`
	`filterOpts.Add("key-prefix-string", "");`
	`filterOpts.Add("key-postfix-string", "");`
	`filterOpts.Add("value-prefix-string",`
	`"<**");`
	`filterOpts.Add("value-postfix-string",`
	`"**>");`
	`globalPrinterOpts.Add("custom-filter-`
	`options", filterOpts);`
	`KeyValuePrinter.DefaultPrinter = new`
	`KeyValuePrinter(globalPrinterOpts, null);`

KVList:
 	
`'DNIS' [str] = <**"8410"**>`
`'PASSWORD' [str] = <**"111111111"**>`
`'RECORD_ID' [str] = <**"8313427"**>`	

Masking Individual Values in Selected KV Pairs	
Configuration options in Administrator	**Corresponding KeyValueCollection content**

[log-filter-data]
 	
PASSWORD=hide
	[Java]
	
	`KeyValueCollection individualKeyMapping =`
	`new KeyValueCollection();`
	`individualKeyMapping.addString("PASSWORD",`
	`"hide");`
	`KeyValuePrinter.setDefaultPrinter(new`
	`KeyValuePrinter(null,`
	`individualKeyMapping));`
	
	[.Net]
	
	`var individualKeyMapping = new`
	`KeyValueCollection();`
	`individualKeyMapping.Add("PASSWORD",`

Masking Partial Values	

	`"hide");`
	`KeyValuePrinter.DefaultPrinter = new`
	`KeyValuePrinter(null, individualKeyMapping);`

KVList:
 	
`'DNIS' [str] = "8410"`
`'PASSWORD' [output suppressed]`
`'RECORD_ID' [str] = "8313427"`	

Masking Partial Values in Selected KV Pairs	
Configuration options in Administrator	**Corresponding KeyValueCollection content**

[log-filter-data]
 	
PASSWORD=unhide-last,5
	[Java]
	
	`KeyValueCollection individualKeyMapping =`
	`new KeyValueCollection();`
	`individualKeyMapping.addString("PASSWORD",`
	`"unhide-last,5");`
	`KeyValuePrinter.setDefaultPrinter(new`
	`KeyValuePrinter(null,`
	`individualKeyMapping));`
	
	[.NET]
	
	`var individualKeyMapping = new`
	`KeyValueCollection();`
	`individualKeyMapping.Add("PASSWORD",`
	`"unhide-last,5");`
	`KeyValuePrinter.DefaultPrinter = new`
	`KeyValuePrinter(null, individualKeyMapping);`

KVList:
 	
`'DNIS' [str] = "8410"`
`'PASSWORD' [str] = "****11111"`
`'RECORD_ID' [str] = "8313427"`	

Tagging Specific KV Pairs with Default Tags	
Configuration options in Administrator	**Corresponding KeyValueCollection content**

[log-filter-data]
 	
PASSWORD=tag()
	[Java]
	
	`KeyValueCollection individualKeyMapping =`
	`new KeyValueCollection();`
	`individualKeyMapping.addString("PASSWORD",`
	`"tag()");`
	`KeyValuePrinter.setDefaultPrinter(new`
	`KeyValuePrinter(null,`
	`individualKeyMapping));`
	
	[Java, Prior to 8.5.102.00]
	
	`KeyValueCollection customFilter = new`
	`KeyValueCollection();`

Masking Partial Values

```
customFilter.addString("custom-filter-
type", "PrefixPostfixFilter");
KeyValueCollection individualKeyMapping =
new KeyValueCollection();
individualKeyMapping.addList("PASSWORD",
customFilter);
KeyValueCollection filterOpts = new
KeyValueCollection();
filterOpts.addString("key-prefix-string",
"");
filterOpts.addString("key-postfix-string",
"");
filterOpts.addString("value-prefix-string",
"<#");
filterOpts.addString("value-postfix-
string", "#>");
customFilter.addList("custom-filter-
options", filterOpts);
KeyValuePrinter.setDefaultPrinter(new
KeyValuePrinter(null,
individualKeyMapping));
```

[.NET]

```
var individualKeyMapping = new
KeyValueCollection();
individualKeyMapping.Add("PASSWORD",
"tag()");
KeyValuePrinter.DefaultPrinter = new
KeyValuePrinter(null, individualKeyMapping);
```

[.NET, Prior to 8.5.102.00]

```
var customFilter = new KeyValueCollection();
customFilter.Add("custom-filter-type",

"Genesyslab.Platform.Commons.Collections.Filters.PrefixPostfi
var individualKeyMapping = new
KeyValueCollection();
individualKeyMapping.Add("PASSWORD",
customFilter);
var filterOpts = new KeyValueCollection();
filterOpts.Add("key-prefix-string", "");
filterOpts.Add("key-postfix-string", "");
filterOpts.Add("value-prefix-string", "<#");
filterOpts.Add("value-postfix-string",
"#>");
customFilter.Add("custom-filter-options",
filterOpts);
KeyValuePrinter.DefaultPrinter = new
KeyValuePrinter(null, individualKeyMapping);
```

KVList:

```
    'DNIS' [str] = "8410"
    'PASSWORD' [str] = <#"111111111"#>
    'RECORD_ID' [str] = "8313427"
```

Tagging Individual KV Pairs with Different Tags

Masking Partial Values	
Configuration options in Administrator	**Corresponding KeyValueCollection content**

[log-filter-data]
 	
PASSWORD=tag() RECORD_ID=tag(<**,**>)
	[Java]
	
	`KeyValueCollection individualKeyMapping = new KeyValueCollection();`
	`individualKeyMapping.addString("PASSWORD", "tag(<!--,-->)");`
	`individualKeyMapping.addString("RECORD_ID", "tag(<**,**>)");`
	`KeyValuePrinter.setDefaultPrinter(new KeyValuePrinter(null, individualKeyMapping));`
	
	[Java prior to 8.5.102.00]
	
	`KeyValueCollection opts1 = new KeyValueCollection();`
	`opts1.addString("key-prefix-string", "");`
	`opts1.addString("key-postfix-string", "");`
	`opts1.addString("value-prefix-string", "<!--");`
	`opts1.addString("value-postfix-string", "-->");`
	
	`KeyValueCollection opts2 = new KeyValueCollection();`
	`opts2.addString("key-prefix-string", "");`
	`opts2.addString("key-postfix-string", "");`
	`opts2.addString("value-prefix-string", "<**");`
	`opts2.addString("value-postfix-string", "**>");`
	
	`KeyValueCollection filter1 = new KeyValueCollection();`
	`filter1.addString("custom-filter-type", "PrefixPostfixFilter");`
	`filter1.addList("custom-filter-options", opts1);`
	`KeyValueCollection filter2 = new KeyValueCollection();`
	`filter2.addString("custom-filter-type", "PrefixPostfixFilter");`
	`filter2.addList("custom-filter-options", opts2);`
	
	`KeyValueCollection individualKeyMapping = new KeyValueCollection();`
	`individualKeyMapping.addList("PASSWORD", filter1);`
	`individualKeyMapping.addList("RECORD_ID", filter2);`
	`KeyValuePrinter.setDefaultPrinter(new KeyValuePrinter(null, individualKeyMapping));`
	
	[.NET]
	
	`var individualKeyMapping = new`

Masking Partial Values

```
KeyValueCollection();
individualKeyMapping.Add("PASSWORD",
"tag(<!--,-->)");
individualKeyMapping.Add("RECORD_ID",
"tag(<**,**>)");
KeyValuePrinter.DefaultPrinter = new
KeyValuePrinter(null, individualKeyMapping);
```

[.NET prior to 8.5.102.00]

```
var opts1 = new KeyValueCollection();
opts1.Add("key-prefix-string", "");
opts1.Add("key-postfix-string", "");
opts1.Add("value-prefix-string", "<!--");
opts1.Add("value-postfix-string", "-->");

var opts2 = new KeyValueCollection();
opts2.Add("key-prefix-string", "");
opts2.Add("key-postfix-string", "");
opts2.Add("value-prefix-string", "<**");
opts2.Add("value-postfix-string", "**>");

var filter1 = new KeyValueCollection();
filter1.Add("custom-filter-type",
"Genesyslab.Platform.Commons.Collections.Filters.PrefixPostfi
filter1.Add("custom-filter-options", opts1);
var filter2 = new KeyValueCollection();
filter2.Add("custom-filter-type",
"Genesyslab.Platform.Commons.Collections.Filters.PrefixPostfi
filter2.Add("custom-filter-options", opts2);

var individualKeyMapping = new
KeyValueCollection();
individualKeyMapping.Add("PASSWORD",
filter1);
individualKeyMapping.Add("RECORD_ID",
filter2);
KeyValuePrinter.DefaultPrinter = new
KeyValuePrinter(null, individualKeyMapping);
```

KVList:

```
    'DNIS' [str] = "8410"
    'PASSWORD' [str] = <!--"111111111"-->
    'RECORD_ID' [str] = <**"8313427"**>
```

Note that the `KeyValuePrinter` class has predefined String constants. For example, `KeyValuePrinter.DEF_FILTER_OPTION` is equivalent to `default-filter-type`. See the `KeyValuePrinter` documentation in the Platform SDK API Reference guide for details.

Implement Custom Filter

It is possible to write your own filter implementation. To do that, extend the `KeyValueAbstractOutputFilter` class and register it in `KeyValuePrinter` using the `custom-`

`filter-type` option.

A sample filter implementation is provided below:

```
public class SimpleHideFilter extends KeyValueAbstractOutputFilter {
   private KeyValueCollection opts;
   public void configure(final KeyValueCollection options) {
       this.opts = options;
   }
   @Override
   protected String doAppendPairValue(final StringBuffer buf,
           final String key, final Object value,
           final KeyValuePrinterContext context) {
       if (opts != null && "true".equals(opts.getString("enabled"))) {
           buf.append("*** Hidden by simple filter ***");
       } else {
           super.doAppendPairValue(buf, key, value, context);
       }
       return null;
   }
}
```

And here is some code showing how to register your filter:

```
KeyValueCollection filterOpts = new KeyValueCollection();
filterOpts.addString("enabled", "true");

KeyValueCollection customFilterDef = new KeyValueCollection();
customFilterDef.addString(KeyValuePrinter.CUSTOM_FILTER_TYPE,
SimpleHideFilter.class.getName());
customFilterDef.addList(KeyValuePrinter.CUSTOM_FILTER_OPTIONS, filterOpts);

KeyValueCollection individualKeyMapping = new KeyValueCollection();
individualKeyMapping.addList("PASSWORD", customFilterDef);
KeyValuePrinter.setDefaultPrinter(new KeyValuePrinter(null, individualKeyMapping));
```

The resulting log might look like this:

```
'EventPartyInfo' (109) attributes:
    AttributeCallType [int] = 4 [Consult]
    AttributeConnID [long] = 008b012ece62c8be
    AttributeUserData [bstr] = KVList:
       'DNIS' [int] = 8410
       'PASSWORD' [str] = *** Hidden by simple filter ***
       'RECORD_ID' [int] = 8313427
    AttributeThisDN [str] = "8899"
```

Profiling and Performance Services

Java

Using JMX Agent

Java Management Extensions (JMX) provide built-in profiling and management options, including an API for monitoring Java applications. This API provides access to information such as:

- Number of classes loaded and threads running
- Virtual machine uptime, system properties, and JVM input arguments
- Thread state, thread contention statistics, and stack trace of live threads
- Memory consumption
- Garbage collection statistics
- Low memory detection
- On-demand deadlock detection
- Operating system information

To enable monitoring for a Java application, first launch the JVM with the default JMX agent turned on using the `com.sun.management.jmxremote.port=<portNum>` system property.

```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=portNum
```

Once these system properties are set, you can use jconsole to monitor the intended Java application. First use the appropriate system tools to determine the process ID for your running application. For example, on Windows systems you can use Task Manager to find the process ID of the java or javaw process. Once the process ID is known you can start jconsole from the jdk/bin directory:

```
jconsole <processID>
```

Heap memory is the runtime data area from which the JVM allocates memory for all class instances and arrays. The heap may be of a fixed or variable size. The garbage collector is an automatic memory management system that reclaims heap memory for objects.

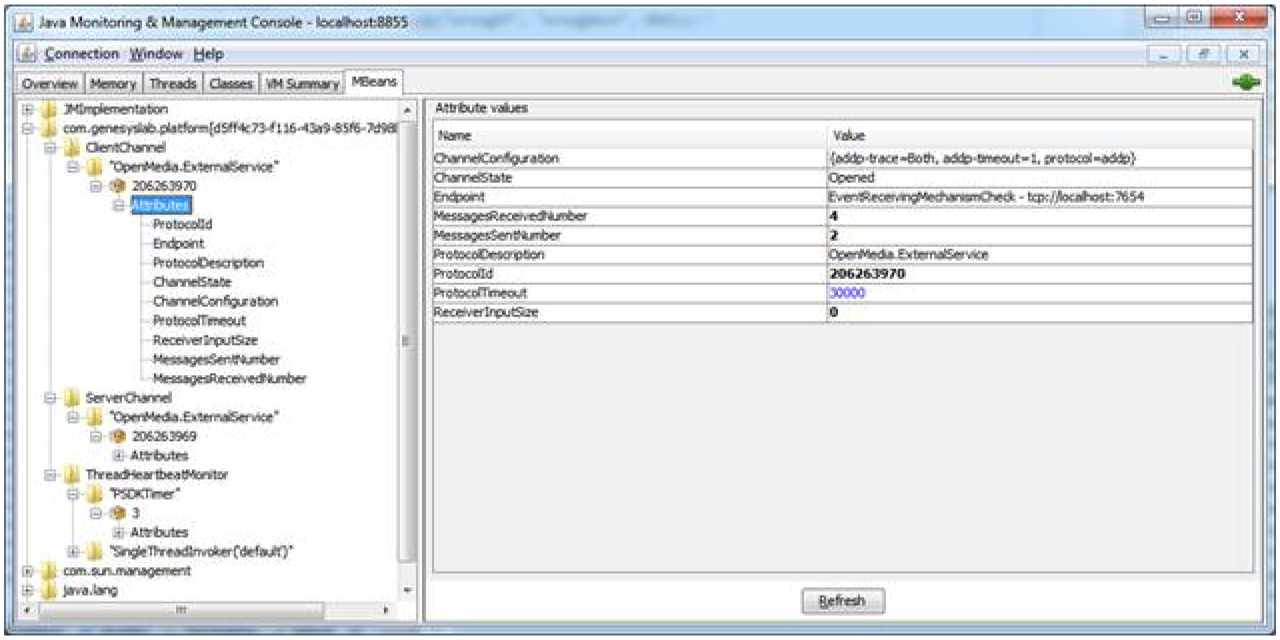
Non-heap memory includes a method area shared among all threads and memory required for the internal processing or optimization for the JVM. It stores per-class structures such as a runtime constant pool, field and method data, and the code for methods and constructors. The method area is logically part of the heap but, depending on implementation, a JVM may not garbage collect or compact it. Like the heap, the method area may be of fixed or variable size. The memory for the method area does not need to be contiguous

Platform SDK MBeans

There are three types of Platform SDK MBeans available:

- ClientChannel Monitor
- ServerChannel Monitor
- ThreadHeartbeatMonitor

Descriptions of each MBean and a sample JConsole screenshot showing what they might look like are provided below.


Java Monitoring & Management Console - localhost:8855
Connection Window Help
Overview Memory Threads Classes VM Summary MBeans
Refresh

```
JMImplementation
com.genesyslab.platform[d5ff4c73-f116-43a9-85f6-7d98
  ClientChannel
    "OpenMedia.ExternalService"
      206263970
        Attributes
          ProtocolId
          Endpoint
          ProtocolDescription
          ChannelState
          ChannelConfiguration
          ProtocolTimeout
          ReceiverInputSize
          MessagesSentNumber
          MessagesReceivedNumber
  ServerChannel
    "OpenMedia.ExternalService"
      206263969
        Attributes
  ThreadHeartbeatMonitor
    "PSDKTimer"
      3
        Attributes
    "SingleThreadInvoker('default')"
com.sun.management
java.lang
```

Attribute values

Name	Value
ChannelConfiguration	{addp-trace=Both, addp-timeout=1, protocol=addp}
ChannelState	Opened
Endpoint	EventReceivingMechanismCheck - tcp://localhost:7654
MessagesReceivedNumber	4
MessagesSentNumber	2
ProtocolDescription	OpenMedia.ExternalService
ProtocolId	206263970
ProtocolTimeout	30000
ReceiverInputSize	0

ClientChannel Monitor

The ClientChannel MBean object represents a single instance of an opened client protocol connection.

It exposes several properties of the client connection for monitoring/debugging purposes, such as:

- protocol type
- protocol ID (that is, the unique ID of the protocol instance)
- a string representation of the client connection endpoint
- connection configuration
- counters for the number of sent and received messages

The `ReceiverInputSize` property reflects the number of asynchronous incoming messages which were received by the protocol connection, but were not retrieved by the application. If this property has a large value during runtime then it may indicate that there was an overload, hang-up, or memory leak in the application.

The only property that you can modify (using JMX) in the ClientChannel MBean is `ProtocolTimeout`.

ServerChannel Monitor

The ServerChannel MBean object represents a single instance of the Platform SDK ServerChannel.

This MBean exposes similar information to the ClientChannel Monitor, but also includes counters for the number of accepted and active client collections.

ThreadHeartbeatMonitor

ThreadHeartbeatMonitor is designed to support the Genesys Management Framework Hang-up Detection feature.

Platform SDK provides this class for server-type applications to allow integration with the Hang-up Detection feature. Developers may create instances of this class in their applications to act as a heartbeat for functional threads, and then add "ticking" calls in their code. An opened LCA connection with proper configuration will then allow Genesys Management Framework to collect information about the heartbeats. If the heartbeat counter stops and specific configuration options are enabled, then Solution Control Server may request that LCA restart the application as hanged up.

Platform SDK exposes the following internal threads: *PSDK Timer*, and workers of *SingleTreadInvoker* instances. Other internal Platform SDK threads (including connection layer and Netty executors) do not use the heartbeat functionality.

.NET

Thread Monitoring

Thread Monitoring Functionality in Platform SDK for .NET

All threads (both user thread and internal Platform SDK threads) for any user application built with Genesys Platform SDK can be monitored, and run-time information for each thread can be gathered at any time. This information gathering and accessibility are supported by standard .NET framework technology related to `PerformanceCounters`. See http://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter.aspx for details.

PerformanceCounter Name Constants

String constants (names) which are used for managing `PerformanceCounters` are as follows:

```
public const string CategoryName = "Genesyslab PSDK .NET";
public const string HeartbeatCounterName = "Thread Heartbeat";
public const string StateCounterName = "Thread State";
public const string ProcessIdCounterName = "ProcessId";
public const string OsThreadIdCounterName = "OsThreadId";
```

These constants are defined in the `ThreadMonitoring` class. In addition to these custom Platform SDK performance counters, users can also use standard counters. For example, both "% Processor Time" and "% User Time" are defined in the *Thread* category.

Tip

To enable monitoring and performance profiling feature the application configuration file has to contain the following section:

```
<configuration>
  <appSettings>
    <add key="ProfilingEnabled" value="true"/>
  </appSettings>
</configuration>
```

Other Diagnostic and Monitoring Tools in Platform SDK

Messages Sent/Received

Platform SDK automatically collects information about the number of messages which were sent and received for each individual channel, along with a total for all channels. String constants (names) which are used for managing `PerformanceCounters` are as follows:

```
public const string CategoryName = "Genesyslab PSDK .NET Messages";
public const string MessagesSentCounterName = "Messages Sent/sec";
public const string MessagesReceivedCounterName = "Messages Received/sec";
```

And the name of the instance which collects total information is:

```
public const string TotalInstanceName = "_Total";
```

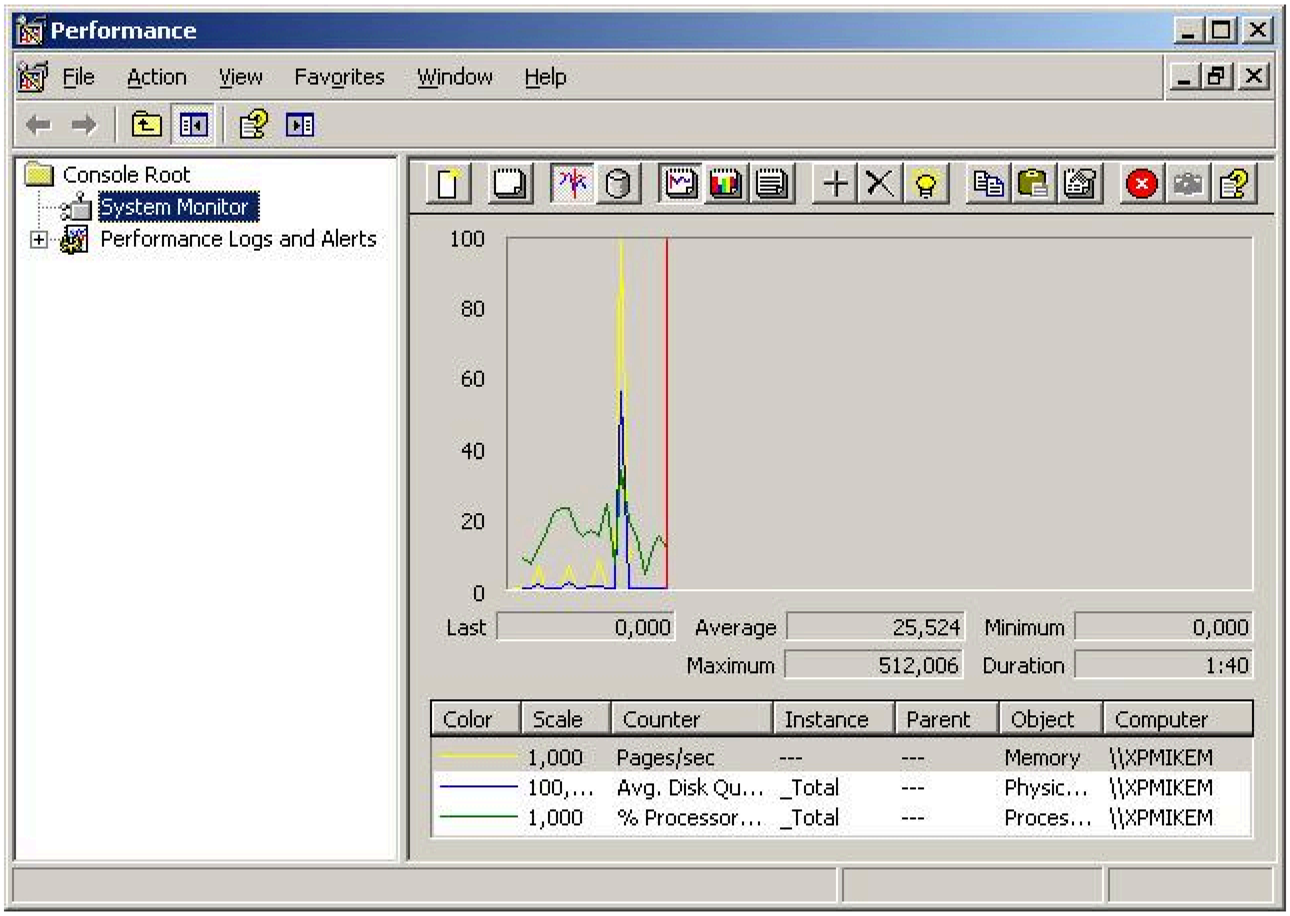
They all are defined in `MessagesMonitoring` class in the `Genesyslab.Platform.Commons.Protocols.Diagnostics` namespace.

Viewing Genesys Platform SDK Diagnostic Data Using 'perfmon.exe'

Performance Monitor Tool

The Performance Monitor tool ('perfmon.exe' included in Windows OS starting with NT 3.0) can be used to view and monitor all of the Genesys Platform SDK diagnostic values described above.

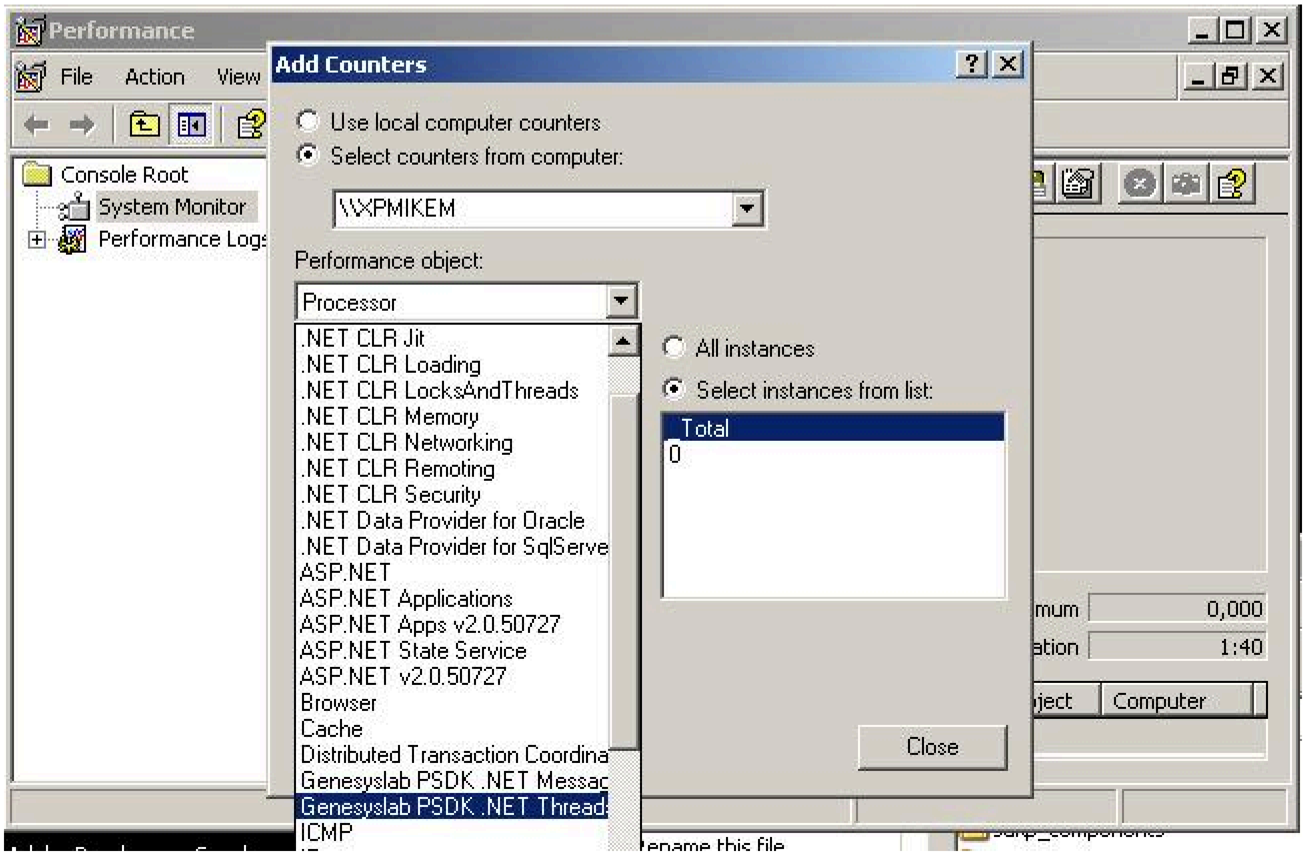
To start 'perfmon.exe', press *Start* > *Run...*, type "perfmon" into the opened window, and than press Ok. The following application will be opened:



Last 0,000 Average 25,524 Minimum 0,000
Maximum 512,006 Duration 1:40

Color	Scale	Counter	Instance	Parent	Object	Computer
🟨	1,000	Pages/sec	---	---	Memory	\\XPMIKEM
🟦	100,...	Avg. Disk Qu...	_Total	---	Physic...	\\XPMIKEM
🟩	1,000	% Processor...	_Total	---	Proces...	\\XPMIKEM

Then you can add Genesyslab PSDK counters. Press 'Add Counters...' (or press Ctrl+I), choose the host where you want to see counters, and then select 'Genesyslab PSDK .NET' performance object in the appropriate combo-box:



Use local computer counters [✓ Select counters from computer:] \\XPMIKEM

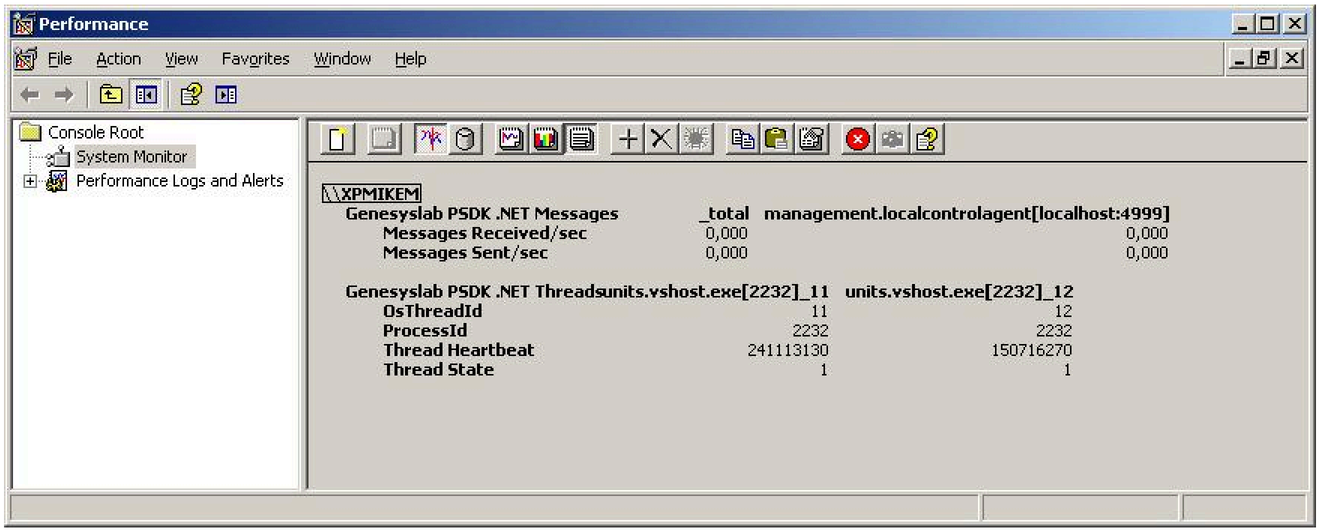
Performance object: Processor

```
.NET CLR Jit
.NET CLR Loading
.NET CLR LocksAndThreads
.NET CLR Memory
.NET CLR Networking
.NET CLR Remoting
.NET CLR Security
.NET Data Provider for Oracle
.NET Data Provider for SqlServe
ASP.NET
ASP.NET Applications
ASP.NET Apps v2.0.50727
ASP.NET State Service
ASP.NET v2.0.50727
Browser
Cache
Distributed Transaction Coordina
Genesyslab PSDK .NET Messac
Genesyslab PSDK .NET Thread:
ICMP
```

All instances [✓ Select instances from list:]

```
_Total
0
```

After that you can select the exact performance counters and which instances they are related to. Now 'perfmon' will show the selected values, for example in 'Report' view as following:



```
\\XPMIKEM
  Genesyslab PSDK .NET Messages            _total   management.localcontrolagent[localhost:4999]
       Messages Received/sec               0,000                                         0,000
       Messages Sent/sec                   0,000                                         0,000

  Genesyslab PSDK .NET Threadsunits.vshost.exe[2232]_11  units.vshost.exe[2232]_12
       OsThreadId                                     11                         12
       ProcessId                                    2232                       2232
       Thread Heartbeat                        241113130                  150716270
       Thread State                                    1                          1
```

IPv6 Resolution

Java

Overview

Platform SDK provides two connection configuration options that control IPv4/IPv6 address resolution:

Option Name	Java Constant	Values	Description
enable-ipv6	Connection.ENABLE_IPV6_KEY	`0` (default) `1`	This option enables and disables IPv6 support. When set to `0`, IPv6 support is disabled, even if IPv6 is supported by the platform.
ip-version	Connection.IP_VERSION_KEY	`4,6` (default) `6,4`	Defines the order in which connection attempts will be made to IPv6 and IPv4 addresses. Option values do not contain spaces. This option has no effect if the option `enable-ipv6` is set to `0`. **Note:** This option only applies to clients. **Note:** In Java you can use the predefined value constants: `Connection.IP_VERSION_4_6` or `Connection.IP_VERSION_6_4`

To enable the use of a Netty connection with OIO transport, use one of the following methods:

- start your Java application with the follow JVM option: `-Dcom.genesyslab.platform.commons.connection.impl.netty.transport=OIO`
- include the following code at the beginning of your application, before any Platform SDK classes are used:

```
    System.setProperty(NettyConnectionFactory.TRANSPORT_TYPE_PARAMETER, "OIO");
```

For additional information about working with IPv6, refer to the Networking IPv6 User Guide (https://docs.oracle.com/javase/8/docs/technotes/guides/net/ipv6_guide/index.html).

Code Samples

[+] Genesys Server needs to open the IPv6-only port for listening

```
PropertyConfiguration cfg = new PropertyConfiguration();
cfg.setIPv6Enabled(true); // cfg.setBoolean(Connection.ENABLE_IPV6_KEY, true) OR
cfg.setOption("enable-ipv6", "1")

Endpoint endpoint = new Endpoint("testServer", "::", 1234, cfg);
ServerChannel server = new ServerChannel(endpoint, new SomeProtocolFactory());
server.open();
```

[+] Genesys Server needs to open IPv4-only port for listening

```
PropertyConfiguration cfg = new PropertyConfiguration();
cfg.setIPv6Enabled(false); // cfg.setBoolean(Connection.ENABLE_IPV6_KEY, false) OR
cfg.setOption("enable-ipv6", "0")

Endpoint endpoint = new Endpoint("testServer", "0.0.0.0", 1234, cfg);
ServerChannel server = new ServerChannel(endpoint, new SomeProtocolFactory());
server.open();
```

[+] Genesys Server needs to open IPv4/IPv6 dual stack port for listening

```
PropertyConfiguration cfg = new PropertyConfiguration();
cfg.setIPv6Enabled(true); // cfg.setBoolean(Connection.ENABLE_IPV6_KEY, true) OR
cfg.setOption("enable-ipv6", "1")

Endpoint endpoint = new WildcardEndpoint("testServer", 1234, cfg);
ServerChannel server = new ServerChannel(endpoint, new SomeProtocolFactory());
server.open();
```

[+] Genesys application needs to open connection to IPv6 network interface of backend server

```
PropertyConfiguration cfg = new PropertyConfiguration();
cfg.setIPv6Enabled(true); // cfg.setBoolean(Connection.ENABLE_IPV6_KEY, true) OR
cfg.setOption("enable-ipv6", "1")
cfg.setIPVersion(Connection.IP_VERSION_6_4); // cfg.setOption(Connection.IP_VERSION_KEY,
Connection.IP_VERSION_6_4) OR cfg.setOption("ip-version", "6,4")

Endpoint endpoint = new Endpoint("testServer", 1234, cfg);
SomeProtocol protocol = new SomeProtocol(endpoint);
protocol.open();
```

[+] Genesys application needs to open connection to IPv4 network interface of backend server

```
PropertyConfiguration cfg = new PropertyConfiguration();
```

```
cfg.setIPv6Enabled(false); // cfg.setBoolean(Connection.ENABLE_IPV6_KEY, false) OR
cfg.setOption("enable-ipv6", "0")

Endpoint endpoint = new Endpoint("testServer", 1234, cfg);
SomeProtocol protocol = new SomeProtocol(endpoint);
protocol.open();
```

[+] Genesys application needs to open connection to IPv6 or IPv4 network interface of backend server, with explicit order of preference (try IPv4 then IPv6)

```
PropertyConfiguration cfg = new PropertyConfiguration();
cfg.setIPv6Enabled(true); // cfg.setBoolean(Connection.ENABLE_IPV6_KEY, true) OR
cfg.setOption("enable-ipv6", "1")
cfg.setIPVersion(Connection.IP_VERSION_4_6); // cfg.setOption(Connection.IP_VERSION_KEY,
Connection.IP_VERSION_4_6) OR cfg.setOption("ip-version", "4,6")

Endpoint endpoint = new Endpoint("testServer", 1234, cfg);
SomeProtocol protocol = new SomeProtocol(endpoint);
protocol.open();
```

Using the Application Template Application Block

Refer to the Using the Application Template Application Block article for details about defining the IPv6 options in Configuration Manager or loading connection configuration details.

.NET

Overview

Platform SDK provides two connection configuration options that control IPv4/IPv6 address resolution:

Option Name	C# Constant	Values	Description
enable-ipv6	CommonConnection.EnableIPv6Key	`0` (default) `1`	This option enables and disables IPv6 support. When set to `0`, IPv6 support is disabled, even if IPv6 is supported by the platform.
ip-version	CommonConnection.IpVersionKey	`4,6` (default) `6,4`	Defines the order in which connection attempts will be made to IPv6 and IPv4 addresses. Option values do not contain

Option Name	C# Constant	Values	Description
			spaces. This option has no effect if the option `enable-ipv6` is set to `0`. **Note:** This option only applies to clients.

Code Samples

[+] Genesys Server needs to open IPv6-only port for listening

```
PropertyConfiguration cfg = new PropertyConfiguration();
cfg.IPv6Enabled = true; // cfg.SetBoolean(CommonConnection.EnableIPv6Key, true) OR
cfg.SetOption("enable-ipv6", "1")

Endpoint endpoint = new Endpoint("testServer", "::", 1234, cfg);
ServerChannel server = new ServerChannel(endpoint, new SomeProtocolFactory());
server.Open();
```

[+] Genesys Server needs to open IPv4-only port for listening

```
PropertyConfiguration cfg = new PropertyConfiguration();
cfg.IPv6Enabled = false; // cfg.SetBoolean(CommonConnection.EnableIPv6Key, false) OR
cfg.SetOption("enable-ipv6", "0")

Endpoint endpoint = new Endpoint("testServer", "::", 1234, cfg);
ServerChannel server = new ServerChannel(endpoint, new SomeProtocolFactory());
server.Open();
```

[+] Genesys Server needs to open IPv4/IPv6 dual stack port for listening

```
PropertyConfiguration cfg = new PropertyConfiguration();
cfg.IPv6Enabled = true; // cfg.SetBoolean(CommonConnection.EnableIPv6Key, true) OR
cfg.SetOption("enable-ipv6", "1")

Endpoint endpoint = new WildcardEndpoint("testServer", 1234, cfg);
ServerChannel server = new ServerChannel(endpoint, new SomeProtocolFactory());
server.Open();
```

[+] Genesys application needs to open connection to IPv6 network interface of backend server

```
PropertyConfiguration cfg = new PropertyConfiguration();
cfg.IPv6Enabled = true; // cfg.SetBoolean(CommonConnection.EnableIPv6Key, true) OR
cfg.SetOption("enable-ipv6", "1")
cfg.IPVersion = "6,4"; // cfg.SetOption(CommonConnection.IpVersionKey, "6,4") OR
cfg.SetOption("ip-version", "6,4")

Endpoint endpoint = new Endpoint("testServer", 1234, cfg);
SomeProtocol protocol = new SomeProtocol(endpoint);
protocol.Open();
```

[+] Genesys application needs to open connection to IPv4 network interface of backend server

```
PropertyConfiguration cfg = new PropertyConfiguration();
cfg.IPv6Enabled = false; // cfg.SetBoolean(CommonConnection.EnableIPv6Key, false) OR
cfg.SetOption("enable-ipv6", "0")

Endpoint endpoint = new Endpoint("testServer", 1234, cfg);
SomeProtocol protocol = new SomeProtocol(endpoint);
protocol.Open();
```

[+] Genesys application needs to open connection to IPv6 or IPv4 network interface of backend server, with explicit order of preference (try IPv4 then IPv6)

```
PropertyConfiguration cfg = new PropertyConfiguration();
cfg.IPv6Enabled = true; // cfg.SetBoolean(CommonConnection.EnableIPv6Key, true) OR
cfg.SetOption("enable-ipv6", "1")
cfg.IPVersion = "4,6"; // cfg.SetOption(CommonConnection.IpVersionKey, "4,6") OR
cfg.SetOption("ip-version", "4,6")

Endpoint endpoint = new Endpoint("testServer", 1234, cfg);
SomeProtocol protocol = new SomeProtocol(endpoint);
protocol.Open();
```

Managing Protocol Configuration

Even after a protocol object has been created, you can still manage and update the configuration for that protocol. This article gives an overview of how to manage protocol configuration, including code samples and a list of properties that can be changed.

Managed Configuration

Starting with Platform SDK release 8.5, each protocol tracks configuration changes and applies them in any state. Some properties (such as running timer) have a deferred effect, while others are applied immediately.

The following code samples show Genesys recommendations for changing a protocol's configuration in any state. You do not need to directly set the new configuration to an Endpoint.

[+] Java Code Sample

```
// Example 1
ConnectionConfiguration cfg = protocol.getEndpoint().getConfiguration();
if (cfg instanceof ClientConnectionOptions) {
  ClientConnectionOptions options =
(ClientConnectionOptions)protocol.getEndpoint().getConfiguration();
  options.setUseAddp(true);
  options.setAddpClientTimeout(5000);
  options.setAddpServerTimeout(5000);
  options.setAddpTraceMode(AddpTraceMode.Local);
}

// Example 2
ConnectionConfiguration cfg = protocol.getEndpoint().getConfiguration();
cfg.setOption(Interceptor.PROTOCOL_NAME_KEY, AddpInterceptor.NAME);
cfg.setOption(AddpInterceptor.TIMEOUT_KEY, "5");
cfg.setOption(AddpInterceptor.REMOTE_TIMEOUT_KEY, "5");
cfg.setOption(AddpInterceptor.TRACE_KEY, "1");
```

[+] .NET Code Sample

```
// Example 1
var configuration = protocol.Endpoint.GetConfiguration() as IClientConnectionOptions;
if (configuration != null)
{
  configuration.AddpClientTimeout = 15;
  configuration.AddpServerTimeout = 20;
  configuration.AddpTraceMode = AddpTraceMode.Both;
  configuration.UseAddp = true;
}

// Example 2
var configuration = protocol.Endpoint.GetConfiguration();
if (configuration != null)
{
  configuration.SetOption(AddpInterceptor.TimeoutKey,"15");
  configuration.SetOption(AddpInterceptor.RemoteTimeoutKey, "20");
  configuration.SetOption(AddpInterceptor.TraceKey, AddpTraceMode.Both.ToString("F"));
  configuration.SetOption(CommonConnection.ProtocolNameKey,AddpInterceptor.Name);
}

// Example 3
var configuration = protocol.Endpoint.GetConfiguration();
if (configuration != null)
{
  configuration.SetOption("addp-timeout","15");
  configuration.SetOption("addp-remote-timeout", "20");
  configuration.SetOption("addp-trace", "both");
  configuration.SetOption("protocol","addp");
}
```

Managed Properties

The following tables list properties that can be changed at any time.

Properties that relate to all protocols:

Property Name	Property Type	Mnemonic Constant in Java	Mnemonic Constant in .NET
protocol	string (protocol name)	Interceptor.PROTOCOL_NAME_KEY	CommonConnection.ProtocolNameKey
addp-timeout	float (in seconds)	AddpInterceptor.TIMEOUT_KEY	AddpInterceptor.TimeoutKey
addp-remote-timeout	float (in seconds)	AddpInterceptor.REMOTE_TIMEOUT_KEY	AddpInterceptor.RemoteTimeoutKey
addp-trace	int	AddpInterceptor.TRACE_KEY	AddpInterceptor.TraceKey
string-attribute-encoding	string	Connection.STR_ATTR_ENCODING_NAME_KEY	ConnectionBase.StringAttributeEncodin

Properties that are supported by the Voice protocol:

Property Name	Property Type	Mnemonic Constant in Java	Mnemonic Constant in .NET
tspAppName	string	TServerProtocol.APP_NAME_KEY	TServerProtocol.ApplicationNameKey
tspPassword	string	TServerProtocol.PASS_KEY	TServerProtocol.PassKey
tspSwitchoverTimeout	long	TServerProtocol.SWITCHOVER_TIMEOUT_KEY	TServerProtocol.SwitchoverTimeoutKey
tspBackupReconnectInterval	long	TServerProtocol.BACKUP_RECONNECT_INTERVAL_KEY	TServerProtocol.BackupReconnectInterv

Properties that are supported by the WebMedia protocol:

Property Name	Property Type	Mnemonic Constant in Java	Mnemonic Constant in .NET
replace-illegal-unicode-chars	boolean	WebmediaChannel.OPTION_NAME_REPLACE_ILLEGAL_UNICODE_CH	WebmediaChannel.OptionNameReplace
illegal-unicode-chars-replacement	string	WebmediaChannel.OPTION_NAME_ILLEGAL_UNICODE_CHARS_REPL	WebmediaChannel.OptionNameIllegal

Managing Configuration Prior to Release 8.5

For releases prior to 8.5, the configuration of an existing protocol object can still be changed. However, any configuration changes made will only take effect if the protocol object is in a "Closed" state; otherwise the changes are applied the next time that protocol is opened.

The following code examples show how Genesys recommends managing protocol configuration:

[+] Java Code Sample

```
// Example 1
ConnectionConfiguration cfg = protocol.getEndpoint().getConfiguration();
if (cfg instanceof ClientConnectionOptions) {
  ClientConnectionOptions options = (ClientConnectionOptions)cfg;
  options.setUseAddp(true);
  options.setAddpClientTimeout(5000);
  options.setAddpServerTimeout(5000);
  options.setAddpTraceMode(AddpTraceMode.Local);
  protocol.configure(cfg); // method configure is deprecated
}

// Example 2
ConnectionConfiguration cfg = protocol.getEndpoint().getConfiguration();
cfg.setOption(Interceptor.PROTOCOL_NAME_KEY, AddpInterceptor.NAME);
cfg.setOption(AddpInterceptor.TIMEOUT_KEY, "5");
cfg.setOption(AddpInterceptor.REMOTE_TIMEOUT_KEY, "5");
cfg.setOption(AddpInterceptor.TRACE_KEY, "1");
protocol.configure(cfg); // method configure is deprecated

// Example 3
ConnectionConfiguration cfg = protocol.getEndpoint().getConfiguration();
cfg.setOption("protocol", "addp");
cfg.setOption("addp-timeout", "5");
cfg.setOption("addp-remote-timeout", "5");
cfg.setOption("addp-trace", "1");
protocol.configure(cfg); // method configure is deprecated
```

[+] .NET Code Sample

```
// Example 1
var configuration = protocol.Endpoint.GetConfiguration() as IClientConnectionOptions;
if (configuration != null)
{
  configuration.UseAddp = true;
  configuration.AddpClientTimeout = 15;
  configuration.AddpServerTimeout = 20;
  configuration.AddpTraceMode = AddpTraceMode.Both;
  protocol.Configure(configuration as IConnectionConfiguration); // method Configure is
obsolete
}

// Example 2
var configuration = protocol.Endpoint.GetConfiguration();
if (configuration != null)
{
  configuration.SetOption(CommonConnection.ProtocolNameKey,AddpInterceptor.Name);;
  configuration.SetOption(AddpInterceptor.TimeoutKey,"15");
  configuration.SetOption(AddpInterceptor.RemoteTimeoutKey, "20");
  configuration.SetOption(AddpInterceptor.TraceKey, AddpTraceMode.Both.ToString("F"));
  protocol.Configure(configuration); // method Configure is obsolete
}

// Example 3
var configuration = protocol.Endpoint.GetConfiguration();
if (configuration != null)
{
  configuration.SetOption("protocol","addp");;
  configuration.SetOption("addp-timeout","15");
  configuration.SetOption("addp-remote-timeout", "20");
  configuration.SetOption("addp-trace", "both");
  protocol.Configure(configuration); // method Configure is obsolete
}
```

Friendly Reaction to Unsupported Messages

Java

Overview

This feature allows Platform SDK to deliver protocol messages which are unknown for the current protocol version (that is, messages with an unsupported message ID). This allows a user application to receive and react to "abstract" messages from the server which have no corresponding protocol Event class.

Note: Prior to release 8.5.0 of Platform SDK, there was no way to receive or react to unsupported messages.

Unsupported messages can be received as an asynchronous or unsolicited event by calling `MessageHandler.onMessage()`. The received message only has one protocol attribute declared: a protocol-specific Reference ID. It is also possible to use this attribute to receive unsupported messages with a synchronous request as shown below:

```
Message response = protocol.request(req);
```

Providing a friendly reaction to unsupported messages is optional. The feature is enabled by default, but may be disabled for all protocol types or for particular protocols in case of backward compatibility issues.

Tip

This feature is implemented for protocols based on Genesys-proprietary protocols messages. This includes most of the Platform SDK protocols, excluding the following XML-based ones: Chat/Callback/Email, and ESP-based UCS/EspEmail protocols).

Tip

This feature is designed to receive "unknown messages" from a server, not for sending these unsupported messages to a server.

The Protocol Unknown Message

A new Platform SDK internal "protocol unknown message" class was introduced in Platform SDK release 8.5.0 to represent unsupported messages.

Genesys recommends that users do not rely on this specific class, its specific attributes or properties when handling the message. In this case, the following basic Message attributes are valuable:

- `MessageId`
- `ProtocolId`
- `Endpoint`
- `ProtocolDescription`

This base API will help provide "forward compatibility", when user application gets newer Platform SDK version which is extended with a particular message support. To ensure backwards compatibility in the future, this message does not support any protocol attributes except for `ReferenceId`.

Previously, the scenario for adding support of new protocol messages to Platform SDK required the following steps:

1. collecting technical details of new protocol message(s)
2. making a feature request to Genesys for the Platform SDK to be extended
3. waiting for development and testing to be completed
4. getting the extended version of Platform SDK and using it to update your application

This feature allows your applications to support unknown events without waiting for this process to be completed.

In this case, the attribute subscription functionality may be helpful. Your application should subscribe for needed attributes on specific message(s) using `AttributeSubscriptionList`. For example:

```
// Initialize protocol:
<AnyServer>Protocol protocol = new <AnyServer>Protocol();
protocol.set...

// Initialize attribute subscription:
final int unsupportedMessageId = 123;
final String attrId = "9";
AttributeSubscriptionList subscriptionList = new AttributeSubscriptionList();
subscriptionList.addAttribute(unsupportedMessageId, attrId);
subscriptionList.applyToContext(protocol.connectionContext());

// Initialize MessageHandler:
protocol.setMessageHandler(new MessageHandler() {
        public void onMessage(final Message message) {
            if (message.messageId() == unsupportedMessageId) {
                Object valAttr9 = message.getMessageAttribute(attrId);
                // do something with raw value of the attribute
            }
            // ...
        }
});
```

```
// Open protocol connection:
protocol.open();

// Send some request to the server to initiate responding:
protocol.send(rq);
```

This code is designed to let your application behave in the same way after Platform SDK is updated to include message support for the added messages.

Backward Compatibility

Automatically enabling this feature may cause a change in behavior for some Platform SDK protocols.

Genesys servers usually maintain backwards compatibility, so receiving unsupported messages is not expected in normal situations. However, this feature may have an effect in scenarios when your application tries to open a protocol client connection to a server of the wrong type. So there may be some cases where you need to disable this feature to keep your applications working as expected. This section describes how to disable the feature for such cases.

In Platform SDK for Java, there is a new `PsdkCustomization` utility class that contains an option to enable or disable this feature. Option values for this class can be configured in three ways:

- a specific configuration file
- JVM system properties
- explicit calls to the `PsdkCustomization` API

The flag to enable this feature is "branchable" for particular protocols. This means that it is possible to disable the feature for specific protocol types, while keep the default value of enabled for all others. Or it is possible to disable the feature for all protocols, and then enable it only for specific types.

The following example shows how to disable the feature by default and then enables it for StatServerProtocol objects using two of the available methods:

Using PsdkCustomization API

```
 PsdkCustomization.setOption(PsdkOption.DisableUnknownProtocolMessageDelivery, "true");
 PsdkCustomization.setOption(PsdkOption.DisableUnknownProtocolMessageDelivery,
"Reporting.StatServer", "false");
```

Using JVM System Properties

```
-Dcom.genesyslab.platform.disable-unknown-incoming-messages=true
-Dcom.genesyslab.platform.Reporting.StatServer.disable-unknown-incoming-messages=false
```

.NET

Overview

This feature allows Platform SDK to deliver protocol messages which are unknown for the current protocol version (that is, messages with an unsupported message ID). This allows a user application to receive and react to "abstract" messages from the server which have no corresponding protocol Event class.

Note: Prior to release 8.5.0 of Platform SDK, there was no way to receive or react to unsupported messages.

Unsupported messages can be received as an asynchronous or unsolicited event using the `Received` event. The received message only has one protocol attribute declared: a protocol-specific Reference ID. It is also possible to use this attribute to receive unsupported messages with a synchronous request as shown below:

```
IMessage response = protocol.Request(req);
```

Providing a friendly reaction to unsupported messages is optional. The feature is enabled by default, but may be disabled for all protocol types or for particular protocols in case of backward compatibility issues.

Tip

This feature is implemented for protocols based on Genesys-proprietary protocols messages. This includes most of the Platform SDK protocols, excluding the following XML-based ones: Chat/Callback/Email, and ESP-based UCS/EspEmail protocols).

Tip

This feature is designed to receive "unknown messages" from a server, not for sending these unsupported messages to a server.

The Protocol Unknown Message

A new Platform SDK internal "protocol unknown message" class was introduced in Platform SDK release 8.5.0 to represent unsupported messages.

Genesys recommends that users do not rely on this specific class, its specific attributes or properties when handling the message. In this case, the following basic Message attributes are valuable:

- `MessageId`
- `ProtocolId`
- `Endpoint`

- `ProtocolDescription`

This base API will help provide "forward compatibility", when user application gets newer Platform SDK version which is extended with a particular message support. To ensure backwards compatibility in the future, this message does not support any protocol attributes except for `ReferenceId`.

Previously, the scenario for adding support of new protocol messages to Platform SDK required the following steps:

1. collecting technical details of new protocol message(s)
2. making a feature request to Genesys for the Platform SDK to be extended
3. waiting for development and testing to be completed
4. getting the extended version of Platform SDK and using it to update your application

This feature allows your applications to support unknown events without waiting for this process to be completed.

In this case, it may be helpful to add a message handler to identify unknown messages. For example:

```
protocol.Received += (sender, e) => // assign message handler
{
  var args = e as MessageEventArgs;
  if ((args != null) && (args.Message != null))
  {
    switch (args.Message.Id){
      case 2854:{ // unknown message is handled by identifier
        // TODO: process message with id 2854
        break;
      }
      default:{
        // TODO: do something with others incoming messages
        break;
      }
    }
  }
};
```

This code is designed to let your application behave in the same way after Platform SDK is updated to include message support for the added messages.

Backward Compatibility

Automatically enabling this feature may cause a change in behavior for some Platform SDK protocols.

Genesys servers usually maintain backwards compatibility, so receiving unsupported messages is not expected in normal situations. However, this feature may have an effect in scenarios when your application tries to open a protocol client connection to a server of the wrong type. So there may be some cases where you need to disable this feature to keep your applications working as expected. This section describes how to disable the feature for such cases.

Backwards compatibility can be preserved in .NET by using an application configuration file. The example below shows how to disable this feature for Configuration Protocol:

```
<configuration>
  <appSettings>
    <add key="ConfServerProtocolUnknownMessageEnabled" value="false"/>
  </appSettings>
</configuration>
```

Each protocol has an individual key which can be added into the application configuration file with a value of "false" to restore behavior to the way it was in earlier versions of Platform SDK. The key name can be formed using the following rules:

```
<key  name> ::= <protocol name><suffix>
<protocol name> ::= "ConfServer" | "MessageServer" | "TServer" | ... | etc.
(equal to IProtocolDescriptionSupport.ProtocolDescription.ProtocolName)
<suffix> := "ProtocolUnknownMessageEnabled"
```

To disable this feature for all protocols used in your application, update the configuration file and assign a value of false to the `ProtocolUnknownMessageEnabled` key, as shown here:

```
<configuration>
  <appSettings>
    <add key="ProtocolUnknownMessageEnabled" value="false"/>
  </appSettings>
</configuration>
```

Creating Custom Protocols

Java

Overview

The External Service Protocol (ESP) was developed to simplify creation of custom protocols. It contains a minimal set of messages for exchanging information between a client and server. All messages contain a reference field to correlate the response with the request. The payload of messages is contained in key-value structures which are used as message properties. Because key-value collections can be used recursively, the total number of properties depends on the message. The custom protocol implements binary transport and obeys common rules for Genesys protocols.

Set of Messages

Message	Description
`Class Request3rdServer`	The `Request3rdServer` class holds requests for your custom server. This class extends the `Message` class and adds three additional fields: • `ReferenceId` - This integer type (32-bit) field is used to correlate this request with related events received as a server response. • `Request` - This `KeyValueCollection` type field is designed to contain a request for the server. Some ESP-based protocols such as `UniversalContactServer` protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols can use this field as desired. • `UserData` - This `KeyValueCollection` type fields is designed to have additional information related to the request. Most known protocols leave this field as is; custom protocols can use this field as desired.
`Class Event3rdServerResponse`	The `Event3rdServerResponse` class is used to send a response to clients. This class extends the `Message` class and adds three additional fields:

Message	Description
	• `ReferenceId` field - This integer type (32-bit) field is used to correlate this event with the related client request. • `Request` field - This `KeyValueCollection` type field is designed to contain the server response to a client request. Some ESP-based protocols such as `UniversalContactServer` protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols can use this field as desired. • `UserData` field - This `KeyValueCollection` type fields is designed to have additional information related to the request. Most known protocols leave this field as is; custom protocols can use this field as desired.
`Class Event3rdServerFault`	The `Event3rdServerFault` class is sent to clients if the request cannot be processed for some reason. This class extends the `Message` class and adds two additional fields: • `ReferenceId` field - This integer type (32-bit) field is used to correlate this server response with the related client request. • `Request` field - This `KeyValueCollection` type field is designed to contain a reason why the error occurred. Some ESP-based protocols such as `UniversalContactServer` protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols can use this field as desired.

Using ESP on the Client Side

Creating a Custom Protocol

To create the simplest ESP-based protocol, all you need to do is create a class inherited from the `ExternalServiceProtocol` class. However, this protocol only provides a way to send data. Your custom protocol still has to handle incoming and outgoing messages.

For example:

```
public class MessageProcessor {
    private static final String PROTOCOL_DESCRIPTION = "CustomESPProtocol";
    /**
     * Processes message which is has to be sent.
     * @param msg message to be sent.
     * @param clientSide flag indicates that the message is processing on a client side
     * @return message instance if message was processed successfully otherwise null.
     */
    static Message processSendingMessage(String msg, boolean clientSide)
    {
      Message outMsg = clientSide ? Request3rdServer.create() :
Event3rdServerResponse.create();
      KeyValueCollection request = new KeyValueCollection();
      request.addString("Protocol", PROTOCOL_DESCRIPTION);
      request.addString("Request", msg);

      if (outMsg instanceof Request3rdServer) {
          ((Request3rdServer)outMsg).setRequest(request);
      }
      else if (outMsg instanceof Event3rdServerResponse) {
          ((Event3rdServerResponse)outMsg).setRequest(request);
      }
      return outMsg;
    }

    /**
     * Handles incoming message.
     * @param message Incoming message.
     * @return Message instance if message was processed successfully otherwise null
     */
    static String processEvent(Message message)
    {
        KeyValueCollection request = null;
        if (message instanceof Event3rdServerResponse) {
            request = ((Event3rdServerResponse)message).getRequest();
        }
        else if (message instanceof Request3rdServer) {
            request = ((Request3rdServer)message).getRequest();
        }
        if (request == null) {
            return null;
        }
        String requestString = request.getString("Request");
        if (requestString == null) {
            return null;
        }
        String protocolDescr = request.getString("Protocol");
        return PROTOCOL_DESCRIPTION.equals(protocolDescr) ? requestString : null;
    }
}

public class EspExtension extends ExternalServiceProtocol {

    private static ILogger log = Log.getLogger(EspExtension.class);

    public EspExtension(Endpoint endpoint) {
        super(endpoint);
    }

    public String request(String message) throws ProtocolException
    {
      Message newMessage = MessageProcessor.processSendingMessage(message, true);
      if (newMessage != null) {
          return MessageProcessor.processEvent(request(newMessage));
      }
      if (log.isDebug()) {
          log.debugFormat("Cannot send message: '{0}'", message);
      }
      return null;
    }
}
```

Using ESP on the Server Side

Class ExternalServiceProtocolListener

The `ExternalServiceProtocolListener` class provides server-side functionality based on the Platform SDK `ServerChannel` class, and implements External Service Protocol. The simplest server side logic is shown in the following example:

```
public class EspServer {

    private static final ILogger log = Log.getLogger(EspServer.class);

    private final ExternalServiceProtocolListener listener;

    public EspServer(Endpoint settings)
    {
        listener = new ExternalServiceProtocolListener(settings);

        listener.setClientRequestHandler(new ClientRequestHandler() {

            @Override
            public void processRequest(RequestContext context) {
                try {
                    EspServer.this.processRequest(context);
                } catch (ProtocolException e) {
                    if (log.isError()) {
                        log.error("Message processing error:\n" +
context.getRequestMessage(), e);
                    }
                }
            }
        });
    }


    public ExternalServiceProtocolListener getServer() {
        return listener;
    }

    private void processRequest(RequestContext context) throws ProtocolException {
        //TODO: Return to client reversed source request

        Message requestMessage = context.getRequestMessage();
        if (requestMessage == null) {
            return;
        }
        if (log.isDebug()) {
            log.debugFormat("Request: {0}", requestMessage);
        }
        String msg = MessageProcessor.processEvent(requestMessage);
        if (msg != null)
        {
          String reversedMsg = new StringBuilder(msg).reverse().toString();
          Message outMsg = MessageProcessor.processSendingMessage( reversedMsg, false);
          if (outMsg instanceof Referenceable
                  && requestMessage instanceof Referenceable) {

((Referenceable)outMsg).updateReference(((Referenceable)requestMessage).retreiveReference());
          }
          if (log.isDebug()) {
              log.debugFormat("Request: {0}", requestMessage);
          }
          context.respond(outMsg); // or context.getClientChannel().send(outMsg);
        }
    }
}
```

Testing Your Protocol

A simple test example is shown below:

```
public class TestEsp {

    final String REQUEST = "Hello world!!!";

    @Test
    public void testMirrorSerializedMessage() throws ChannelNotClosedException,
ProtocolException, InterruptedException
    {
      String response = null;
      ExternalServiceProtocolListener server = new EspServer(new
WildcardEndpoint(0)).getServer();
      server.open();
      InetSocketAddress ep = server.getLocalEndPoint();
      if (ep != null) {
          EspExtension client = new EspExtension(new Endpoint("localhost", ep.getPort()));
          client.open();
        response = client.request(REQUEST);
        client.close();
      }
      server.close();

      System.out.println("Request: \n" + REQUEST);
      System.out.println("Response: \n" + response);
      Assert.assertNotNull(response);

      String expected = new StringBuilder(REQUEST).reverse().toString();
      Assert.assertEquals(response, expected);
  }
}
```

.NET

Overview

The External Service Protocol (ESP) was developed to simplify creation of custom protocols. It contains a minimal set of messages for exchanging information between a client and server. All messages contain a reference field to correlate the response with the request. The payload of messages is contained in key-value structures which are used as message properties. Because key-value collections can be used recursively, the total number of properties depends on the message. The custom protocol implements binary transport and obeys common rules for Genesys protocols.

Set of Messages

Message	Description
`Class Request3rdServer`	The `Request3rdServer` class serves for request the server. This class contains the base set of properties inherent to the `IMessage` interface and three additional fields. • `ReferenceId` field - This integer type (32-bit) field is used to correlate this request with related events received as a server response. • `Request` field - This `KeyValueCollection` type field is designed to contain a request to server. Some ESP-based protocols such as `UniversalContactServer` protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols can use this field as desired. • `UserData` field - This `KeyValueCollection` type field is designed to have additional information related to the request. Most known protocols leave it as is; custom protocols can use this field as desired.
`Class Event3rdServerResponse`	The `Event3rdServerResponse` class is used to send a response to clients. This class contains the base set of properties inherent to the `IMessage` interface and three additional fields: • `ReferenceId` field - This integer type (32-bit) field is used to correlate this event with the related client request. • `Request` field - This `KeyValueCollection` type field is designed to contain a server response to a client request. Some ESP-based protocols such as `UniversalContactServer` protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols

Message	Description
	can use this field as desired. • `UserData` field - This `KeyValueCollection` type field is designed to have additional information of request. Most known protocols leave it as is; custom protocols can use this field as desired.
`Class Event3rdServerFault`	The `Event3rdServerFault` class is sent to clients if the request cannot be processed for some reasons. This class contains the base set of properties inherent to the `IMessage` interface and two additional fields: • `ReferenceId` field - This integer type (32-bit) field is used to correlate this server response with the related client request. • `Request` field - This `KeyValueCollection` type field is designed to contain a reason why the error occurred. Some ESP-based protocols such as `UniversalContactServer` protocol can parse and bind content of this structure with some classes to have more convenient representation of the data. Custom protocols can use this field as desired.

Using ESP on the Client Side

Creating a Custom Protocol

To create the simplest ESP-based protocol, all you need to do is create a class inherited from the `ExternalServiceProtocol` class. However, this protocol only provides a way to send data. Your custom protocol still has to handle incoming and outgoing messages.

For example:

```
  internal static class MessageProcessor
  {
    private const string ProtocolDescriptionStr = "CustomESPProtocol";

    /// <summary>
    /// Processes message which is has to be sent.
    /// </summary>
    /// <param name="msg">message to be sent</param>
    /// <param name="clientSide">flag indicates that message is processing on client
side</param>
    /// <returns>IMessage instance if message was processed successfuly otherwise
null</returns>
    internal static IMessage ProcessSendingMessage(string msg, bool clientSide)
    {
      IMessage outMsg = clientSide ? Request3rdServer.Create() :
```

```
Event3rdServerResponse.Create() as IMessage;
      var request = new KeyValueCollection {{"Protocol", ProtocolDescriptionStr}, {"Request",
msg}};
      var req = outMsg as Request3rdServer;
      if (req != null) req.Request = request;
      var evt = outMsg as Event3rdServerResponse;
      if (evt != null) evt.Request = request;
      return outMsg;
    }
    /// <summary>
    /// Handles incoming message
    /// </summary>
    /// <param name="message">Incoming message</param>
    /// <returns>IMessage instance if message was processed successfuly otherwise
null</returns>
    internal static string ProcessEvent(IMessage message)
    {
      var response = message as Event3rdServerResponse;
      var req = message as Request3rdServer;
      KeyValueCollection request = null;
      if ((response != null) && (response.Request != null))
      {
        request = response.Request;
      }
      else
      {
        if ((req != null) && (req.Request != null))
          request = req.Request;
      }
      if (request == null) return null;
      var requestStr = request["Request"] as String;
      if (String.IsNullOrEmpty(requestStr)) return null;
      var protocolDescr = request["Protocol"] as string;
      return (ProtocolDescriptionStr.Equals(protocolDescr))?requestStr:null;
    }
  }

  public class EspExtension : ExternalServiceProtocol
  {

    public EspExtension(Endpoint endPoint) : base(endPoint) { }
    public string Request(string message)
    {
      var newMessage = MessageProcessor.ProcessSendingMessage(message, true);
      if (newMessage != null)
      {
        return MessageProcessor.ProcessEvent(Request(newMessage));
      }
      if ((Logger != null) && (Logger.IsDebugEnabled))
        Logger.DebugFormat("Cannot send message: '{0}'", message);
      return null;
    }
  }
```

Using ESP on the Server Side

Class ExternalServiceProtocolListener

The `ExternalServiceProtocolListener` class provides server-side functionality based on the Platform SDK `ServerChannel` class, and implements External Service Protocol. The simplest server side logic is shown in the following example:

```
  public class EspServer
  {
    private readonly ExternalServiceProtocolListener _listener;
    public EspServer(Endpoint settings)
    {
      _listener = new ExternalServiceProtocolListener(settings);
      _listener.Received += ListenerOnReceived;
    }
    public ExternalServiceProtocolListener Server { get { return _listener; } }

    private void ListenerOnReceived(object sender, EventArgs eventArgs)
    {
      //TODO: Return to client source request
      var duplexChannel = sender as DuplexChannel;
      var args = eventArgs as MessageEventArgs;
      if ((duplexChannel == null) || (args == null)) return;
      Console.WriteLine(args.Message);
      var msg = MessageProcessor.ProcessEvent(args.Message);
      if (msg != null)
      {
        var outMsg = MessageProcessor.ProcessSendingMessage(new
string(msg.Reverse().ToArray()), false);
        var source = args.Message as IReferenceable;
        var dest = outMsg as IReferenceable;
        if ((source!=null) && (dest!=null))
         dest.UpdateReference(source.RetrieveReference());
        Console.WriteLine(outMsg);
        duplexChannel.Send(outMsg);
      }
    }
  }
```

Testing Your Protocol

A simple test example is shown below:

```
  [TestClass]
  public class TestEsp
  {
    [TestMethod]
    public void TestMirrorSerializedMessage()
    {
      const string request = "Hello world!!!";
      string response = null;
      var server = new EspServer(new WildcardEndpoint(0)).Server;
      server.Open();
      var ep = server.LocalEndPoint as IPEndPoint;
      if (ep != null)
      {
        var client = new EspExtension(new Endpoint("localhost", (server.LocalEndPoint as
IPEndPoint).Port));
        client.Open();
        response = client.Request(request);
        client.Close();
      }
      server.Close();

      Console.WriteLine("Request: \n{0}", request);
      Console.WriteLine("Response: \n{0}", response);
      Assert.IsNotNull(response);
      Assert.IsTrue(request.Equals(new string(response.Reverse().ToArray())));    }
  }
```

JSON Support

Java

Starting with release 8.5.201.04, Platform SDK for Java has been extended with functionality for serialization and deserialization of protocol messages to JSON string representation. (For older 8.5.201.x releases, refer to the Legacy Content section at the bottom of this article.)

Design

The serializer supports two different types of JSON representations for Platform SDK protocol messages: with `MessageName` attribute, and without it.

Adding the `MessageName` attribute helps users to deserialize a protocol message when its type is not known from context. Deserialization without the inner `MessageName` attribute can be used to support existing solutions, or to avoid duplicating some request context data in order to optimize network traffic, CPU, and memory usage.

> **Important**
> Each serializer instance should work with one protocol type. If you need to process different types (such as UCS and StatServer) then you must create one serializer per protocol.

Examples

Using the Platform SDK JSON Message Serializer

> **Tip**
> This serializer is designed to work with Platform SDK messages only. Serialization of other custom classes is not supported.

Example 1: Message Deserialization (with messageName in JSON)

```
PsdkJsonSerializer ser = PsdkJsonSerializer.createContactServerSerializer();
```

```
String json = "{ \"messageName\":\"RequestRefresh\", "
       + "\"query\":\"test-Query-4\", \"file\":\"test-File-3\",\"indexName\":\"Contact\",
\"persistents\":\"test-Persistents-2\"}";

Message message = ser.deserialize(json);
```

Example 2: Message Deserialization (Without messageName in JSON)

```
PsdkJsonSerializer ser = PsdkJsonSerializer.createContactServerSerializer();

String json = "{ \"query\":\"test-Query-4\", \"file\":\"test-
File-3\",\"indexName\":\"Contact\", \"persistents\":\"test-Persistents-2\"}";

Message message = ser.deserialize(json, "RequestRefresh");
// RequestRefresh message = ser.deserialize(json, RequestRefresh.class);
```

Example 3: Message Serialization (With messageName in JSON)

```
PsdkJsonSerializer ser = PsdkJsonSerializer.createContactServerSerializer();

RequestRefresh request = RequestRefresh.create();
request.setQuery("test-Query-4");
request.setFile("test-File-3");
request.setIndexName(IndexNameType.Contact);
request.setPersistents("test-Persistents-2");

String json = ser.serialize(request);
```

Using the Jackson Framework

Example 1: Message Deserialization (with messageName in JSON)

```
ObjectMapper m = new ObjectMapper();
m.registerModule(new ContactServerModule(true));

String json = "{ \"messageName\":\"RequestRefresh\", "
       + "\"query\":\"test-Query-4\", \"file\":\"test-File-3\",\"indexName\":\"Contact\",
\"persistents\":\"test-Persistents-2\"}";

Message message = (Message)m.readValue(json, ContactServerMessage.class);
```

Example 2: Message Deserialization (Without messageName in JSON)

```
ObjectMapper m = new ObjectMapper();
PSDKCommonModule mod = new ContactServerModule();
m.registerModule(mod);

String json = "{ \"query\":\"test-Query-4\", \"file\":\"test-
File-3\",\"indexName\":\"Contact\", \"persistents\":\"test-Persistents-2\"}";

Message message = m.readValue(json, mod.getMessageClass("RequestRefresh"));
//RequestRefresh message = m.readValue(json, RequestRefresh.class);
```

Example 3: Message Serialization (Without messageName in JSON)

```
ObjectMapper m = new ObjectMapper();
m.registerModule(new ContactServerModule());

RequestRefresh request = RequestRefresh.create();
```

```
request.setQuery("test-Query-4");
request.setFile("test-File-3");
request.setIndexName(IndexNameType.Contact);
request.setPersistents("test-Persistents-2");

String json = m.writeValueAsString(request);

// We expect to get JSON like the following:
//  "{ "\"query\":\"test-Query-4\", \"file\":\"test-File-3\",\"indexName\":\"Contact\",
\"persistents\":\"test-Persistents-2\"}";
```

Example 4: Message Serialization (With messageName in JSON)

```
ObjectMapper m = new ObjectMapper();
m.registerModule(new ContactServerModule(true));

RequestRefresh request = RequestRefresh.create();
request.setQuery("test-Query-4");
request.setFile("test-File-3");
request.setIndexName(IndexNameType.Contact);
request.setPersistents("test-Persistents-2");

String json = m.writeValueAsString(request);

// We expect to get JSON like the following:
//  "{ \"messageName\":\"RequestRefresh\", "
//       + "\"query\":\"test-Query-4\", \"file\":\"test-File-3\",\"indexName\":\"Contact\",
\"persistents\":\"test-Persistents-2\"}";
```

Legacy Content

Prior to release 8.5.201.04, Platform SDK for Java did not offer native support for JSON - only XML serialization was supported. This section describes how provide JSON support within your Platform SDK for Java applications for legacy applications that use JSON format for data.

[+] Display Legacy Content

Overview

In most Genesys projects, the Jackson framework is used for serialization/deserialization of plain old Java objects (POJO) in/from JSON. Not all Platform SDK messages are POJO that can be transformed in/from JSON automatically, however.

So to provide full JSON support Platform SDK for Java, a Jackson module is included with your distribution. This module is a separate JAR library; no dependency to the Jackson library has been added directly within Platform SDK.

Using the Module

This section contains code snippets that illustrate how to use the new module.

Example 1: Using the ConfServerModule

```
// create Jackson's JSONizator
```

```
ObjectMapper mapper = new ObjectMapper();
// register our new Platform SDK module
mapper.registerModule( new ConfServerModule() );
// create Platform SDK message
RequestCreateObject request= RequestCreateObject.create();
// serialize message
String json = mapper.writeValueAsString(msg);
// deserialize message
RequestCreateObject msg = objectMapper.readValue(json, RequestCreateObject.class);
```

Example 2: Using a specified set of metadata from Configuration Server

This approach is useful when you are connecting to a more recent version of Configuration Server then Platform SDK supports.

Configuration Server messages that are deserialized using the metadata included with your release of Platform SDK can be sent to older releases of Configuration Server, but in this scenario the unknown (new) attributes for Configuration Server will be ignored while sending the message.

```
ConfServerProtocol c= new ...;
c.open();
CfgMetadata metadata =
((ConfServerContext)c.connectionContext().serverContext()).getMetadata();
// create Jackson's JSONizator
ObjectMapper mapper = new ObjectMapper();
// register our new Platform SDK module
mapper.registerModule( new ConfServerModule(metadata) );
// create Platform SDK message
RequestCreateObject request= RequestCreateObject.create();
// serialize message
String json = mapper.writeValueAsString(msg);
// deserialize message
RequestCreateObject msg = objectMapper.readValue(json, RequestCreateObject.class);
```

Example 3: Configuring Jackson ObjectMapper to support multiple Platform SDK protocols

```
// create Jackson's JSONizator
ObjectMapper mapper = new ObjectMapper();
// register our new Platform SDK modules
mapper.registerModules( new ConfServerModule(), new ContactServerModule() );
mapper.registerModule( new StatServerModule() );
```

Notable Jackson Features

Jackson has many configurable features. This section describes a few features that are particularly noteworthy for Platform SDK developers.

Property Naming

Use this feature carefully, because if you change property naming for serialization then anyone who wants to deserialize content must use the same configuration or else the deserialization will fail. An example of setting a custom property naming strategy is included below:

```
ObjectMapper mapper = new ObjectMapper();
mapper.setPropertyNamingStrategy(PropertyNamingStrategy.LOWER_CASE);
```

Handling Unknown Properties

By default, Jackson fails if a unknown property occurs during deserialization. It is possible to change this behavior so that unknown properties can be ignored, as shown below:

```
ObjectMapper mapper = new ObjectMapper();
mapper.disable(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);
mapper.enable(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES);
```

Relation to Platform SDK ObjectMapper

All Jackson modules included with Platform SDK make changes to the `ObjectMapper` configuration while registering, as shown here:

```
mapper.setSerializationInclusion(Include.NON_NULL);
mapper.configure(SerializationFeature.FAIL_ON_EMPTY_BEANS, false);
mapper.setDateFormat(new ISO8601DateFormatWithMilliseconds());
```

If you want to customer how the `ObjectMapper` will be configured, be sure to apply your changes *after* the Jackson module has registered.

JSON Format Examples

Several examples of the resulting JSON format are provided here for illustrative purposes.

JSON Representation of KVList

```
[ {
  "key" : "int1",
  "type" : "int",
  "value" : 1
}, {
  "key" : "string1",
  "type" : "str",
  "value" : "string-value1"
}, {
  "key" : "utf-string1",
  "type" : "utf",
  "value" : "utf-string-value1"
}, {
  "key" : "binary1",
  "type" : "byte[]",
  "value" : "AQIECP8="
}, {
  "key" : "kv1",
  "type" : "kvlist",
  "value" : [ {
                "key" :"int2",
                "type" : "int",
                "value" : 2
              }, {
                "key" : "string2",
                "type" : "str",
                "value" : "string-value2"
              }, {
                "key" : "utf-string2",
                "type" : "utf",
                "value" : "utf-string-value2"
              }, {
                "key" : "binary2",
                "type" : "byte[]",
                "value" : "AwQFBr8="
               } ]
```

Notes:

- KeyValuePair property "type" can be skipped for the following types: `int`, `str`, `kvlist` (KeyValueCollection). Platform SDK will deserialize similar data by determining the type heuristically.

JSON Representation of RequestInsertInteraction

```
{
  "interactionAttributes" : {
    "id" : "some-id",
    "entityTypeId" : "Chat",
    "lang" : "en"
  },
  "interactionContent" : {
    "mimeType" : "text/plain",
    "text" : "some text"
  },
  "entityAttributes" : {
    "$type" : ".ChatEntityAttributes",
    "establishedDate" : "2015-07-24",
    "nullAttributes" : [ "ReleasedDate" ]
  }
}
```

Notes:

- $type supports the following values:
  - ".ChatEntityAttributes"
  - ".EmailInEntityAttributes"
  - ".EmailOutEntityAttributes"
  - ".CoBrowseEntityAttributes"
  - ".PhoneCallEntityAttributes"

JSON Representation of RequestContactListGet

```
{
  "contactCount" : true,
  "tenantId" : 15,
  "attributeList" : [ "attr-1", "attr-5", "attr-a" ],
  "sortCriteria" : [ {
    "$type" : ".SimpleSearchCriteria",
    "attrName" : "attr-1",
    "sortIndex" : 0,
    "sortOperator" : "Ascending"
  }, {
    "$type" : ".SimpleSearchCriteria",
    "attrName" : "attr-a",
    "sortIndex" : 1,
    "sortOperator" : "Descending"
  } ],
  "searchCriteria" : [ {
    "$type" : ".ComplexSearchCriteria",
    "prefix" : "And",
    "criterias" : [ {
      "$type" : ".SimpleSearchCriteria",
      "attrName" : "a",
      "operator" : "Greater",
      "attrValue" : "5"
    } ]
  }, {
    "$type" : ".ComplexSearchCriteria",
    "prefix" : "And",
    "criterias" : [ {
      "$type" : ".SimpleSearchCriteria",
      "attrName" : "b",
      "operator" : "Lesser",
      "attrValue" : "2"
    } ]
  }, {
    "$type" : ".ComplexSearchCriteria",
    "prefix" : "And",
    "criterias" : [ {
      "$type" : ".SimpleSearchCriteria",
      "attrName" : "c",
      "operator" : "Equal",
      "attrValue" : "11"
    } ]
  }, {
    "$type" : ".ComplexSearchCriteria",
    "prefix" : "And",
    "criterias" : [ {
      "prefix" : "Or",
      "criterias" : [ {
        "$type" : ".SimpleSearchCriteria",
        "attrName" : "e",
        "operator" : "Lesser",
        "attrValue" : "1"
      } ]
    }, {
      "$type" : ".ComplexSearchCriteria",
      "prefix" : "Or",
      "criterias" : [ {
        "$type" : ".SimpleSearchCriteria",
        "attrName" : "k",
        "operator" : "GreaterOrEqual",
        "attrValue" : "0"
      } ]
    } ]
  } ]
}
```

Notes:

- $type supports the following values:
  - ".SimpleSearchCriteria"
  - ".ComplexSearchCriteria"

Frequently Asked Questions

Q: The KV lists can contain pointers to outer KVlists, creating circular dependencies. Will such structures be serialized/deserialized?
A: No. Platform SDK does not support sending or receiving such structures.

Q: In what format are binary values serialized? Base64 or something else?
A: Base64

Q: Why is the "type" attribute is optional?
A: The attribute is only optional for the following types: integer, string and key-value collection (it can be skipped when you type JSON requests manually). The Platform SDK Jackson module will always serialize the type attribute to JSON, but can deserialize it without the optional attribute "type" attribute.

Q: Is GEnum serialization/deserialization supported?
A: Yes. As you can see the example above where the GEnum `EntityTypes` is serialized as simple string value "chat".

Q: Does the module perform direct serialization/deserialization or does it use intermediate XML?
A: It performs direct serialization from Platform SDK messages to JSON (and vice versa) using Jackson objectmapper with registered PSDKModule.

.NET

Starting with release 8.5.201.04, Platform SDK for .NET has been extended with functionality for serialization and deserialization of protocol messages to JSON string representation.

Design

The serializer supports two different types of JSON representations for Platform SDK protocol messages: with `MessageName` attribute, and without it.

Adding the `MessageName` attribute helps users to deserialize a protocol message when its type is not known from context. Deserialization without the inner `MessageName` attribute can be used to support existing solutions, or to avoid duplicating some request context data in order to optimize network traffic, CPU, and memory usage.

Important

Each serializer instance should work with one protocol type. If you need to process different types (such as UCS and StatServer) then you must create one serializer per protocol.

Examples

Example 1: Bi-directional JSON Serialization

```
public void TestRequestGetContacts()
{
  var req = RequestGetContacts.Create();
  req.ReferenceId = 1;
  req.SearchCriteria = new SearchCriteriaCollection();
  req.SortCriteria = new SortCriteriaCollection();
  var json = PsdkJsonSerializer.Serialize(req);
  Console.WriteLine(json);
  var newRq = PsdkJsonSerializer.Deserialize(req.GetType(), json);
  var json2 = PsdkJsonSerializer.Serialize(newRq);
  Console.WriteLine(json2);
  Assert.IsTrue(json.Equals(json2));
}
```

Example 2: Using Platform SDK JSON Serializer for UCS Protocol

```
public void TestRequestGetVersion()
{
  var request = RequestGetVersion.Create();
  request.ReferenceId = 1;
  Console.WriteLine(request);
  var serializer = JsonSerializationFactory.GetSerializer<UniversalContactServerProtocol>();
  var json = serializer.Serialize(request);
  Console.WriteLine(json);
  var deserializedMessage = serializer.Deserialize(json);
  Console.WriteLine(deserializedMessage);
  Assert.IsTrue(request.Equals(deserializedMessage));
}
```

Example 3: Sample JSON Object

Message:

```
'RequestRefresh' ('60')
message attributes:
Query          = test-Query-4
File           = test-File-3
IndexName      = Contact
Persistents    = test-Persistents-2
```

JSON Representation:

```
{"messageName":"RequestRefresh","query":"test-Query-4","file":"test-
File-3","indexName":"Contact","persistents":"test-Persistents-2"}
```

Java and .NET Compatibility Note

The internal implementation of messages from the Configuration Server protocol differs for .NET and Java platforms. Data objects in .NET messages are based on the XDocument class, while Java message use data classes for mapping of data objects. This means that the result of serialization for the same messages from Java and .NET platforms will be different for the Configuration Server protocol (although bi-directional serialization is supported for both platforms).

All other Platform SDK protocols support cross-platform serialization.

Working with Custom Servers

Java

The `ServerChannel` class was designed to give you the ability to develop custom servers using Platform SDK. A `ServerChannel` instance can accept incoming connections, receive and handle incoming messages, and send responses to the clients.

Creating a ServerChannel Instance

Before creating a `ServerChannel` instance, your application should define an instance of some class which implements the `ProtocolFactory` interface. You can use any of the existing Platfrom SDK message factories, although most of these classes cannot follow the logic of existing servers because the messages used in the handshake procedure are hidden.

The most flexible protocol for any extension is `ExternalServiceProtocol`, because it does not require any handshake by default. The following example will use this protocol to create an instance of `ServerChannel`:

```
final ServerChannel server = new ServerChannel(new WildcardEndpoint(11111),
        new ExternalServiceProtocolFactory());
```

Defining Handlers to Process Incoming (Closed) Connections

`ServerChannel` generates two events to manage client connections. When a new client tries to connect, `ServerChannel` raises the `onClientChannelOpened` event, and when a client disconnects `ServerChannel` raises an `onClientChannelClosed` event. Your code can then process these events, as shown in the example below:

```
server.addChannelListener(new ServerChannelListener() {
        public void onClientChannelOpened(OutputChannel channel) { /* … */ }
        public void onClientChannelClosed(ChannelClosedEvent event) { /* … */ }
        public void onChannelOpened(EventObject event) { /* … */ }
        public void onChannelError(ChannelErrorEvent event) { /* … */ }
        public void onChannelClosed(ChannelClosedEvent event) { /* … */ }
});
```

Starting the Server

```
server.open();
```

Processing Incoming Messages

`ServerChannel` supports multiple ways to receive and process incoming messages:

- receiveRequest Method
- External Receiver
- Message Handler

More details about each approach are explored below.

Using the receiveRequest Method

Using this method allows your to define exactly when messages are read. However, you should remember that the internal queue which contains incoming messages is not unlimited. The maximum capacity of this queue will be equal to 4k elements, and once that capacity is filled each new incoming message will cause the oldest one to be lost.

The most popular way of using the `receiveRequest` method is inside a dedicated thread, as shown here:

```
new Thread() {
        @Override
        public void run() {
                while (running) {
                        RequestContext request = server.receiveRequest();
                        if (request != null) {
                                Message requestMessage = request.getRequestMessage();
                                Message respondMessage;
                                // TODO generate respondMessage
                                if (respondMessage != null) {
                                        request.respond(respondMessage);
                                }
                        }
                        Thread.yield();
                }
        }
}.start();
```

Using an External Receiver

To use an external receiver, your should create a class which implements the `RequestReceiverSupport` interface, and then use the `ServerChannel.setReceiver(RequestReceiverSupport receiver)` method to assign this receiver to `ServerChannel`.

The simplest implementation to process incoming messages is shown below:

```
RequestReceiverSupport receiver = new RequestReceiverSupport() {

        public void onChannelOpened(EventObject event) { /* ... */}
        public void onChannelError(ChannelErrorEvent event) { /* ... */}
        public void onChannelClosed(ChannelClosedEvent event) { /* ... */}
        public void setInputSize(int inputSize) { /* ... */}
        public void releaseReceivers() { /* ... */}
        public int getInputSize() { return 0; }
        public void clearInput() { /* ... */}
        public RequestContext receiveRequest(long timeout) { return null; }
        public RequestContext receiveRequest() { return null; }

        public void processRequest(RequestContext request) {

                Message requestMessage = request.getRequestMessage();
                Message respondMessage;
                 // TODO generate respondMessage
                 if (respondMessage != null) {
                         request.respond(respondMessage);
                 }
        }
};

server.setReceiver(receiver);
```

Using Message Handler

Starting with release 8.5.1, Platform SDK has included a new mechanism to handle incoming messages. `ServerChannel` was extended with a new method `setClientRequestHandler`, that can be used as shown in the following example:

```
server.setClientRequestHandler(new ClientRequestHandler() {

        @Override
        public void processRequest(RequestContext context) {
                Message requestMessage = request.getRequestMessage();
                Message respondMessage;
                 // TODO generate respondMessage
                 if (respondMessage != null) {
                         request.respond(respondMessage);
                 }
        }
});
```

Closing ServerChannel

Closing the server channel causes all active incoming connections to be closed also. To close server channel use the `ServerChannel.close()` method.

```
server.close();
```

.NET

The `ServerChannel` class was designed to give you the ability to develop custom servers using Platform SDK. A `ServerChannel` instance can accept incoming connections, receive and handle incoming messages, and send responses to the clients.

Creating a ServerChannel Instance

Before creating a `ServerChannel` instance, your application should define an instance of some class which implements the `IMessageFactory` interface. You can use any of the existing Platfrom SDK message factories, although most of these classes cannot follow the logic of existing servers because the messages used in the handshake procedure are hidden.

The most flexible protocol for any extension is `ExternalServiceProtocol`, because it does not require any handshake by default. The following example will use this protocol to create an instance of `ServerChannel`:

```
const int portNumber = 22222;
var server = new ServerChannel(new WildcardEndpoint(portNumber),
        new ExternalServiceProtocolFactory());
```

Defining Handlers to Process Incoming (Closed) Connections

`ServerChannel` generates two events to manage client connections. When a new client tries to connect, `ServerChannel` raises the `ClientChannelOpened` event, and when a client disconnects `ServerChannel` raises an `ClientChannelClosed` event. Your code can then process these events, as shown in the example below:

```
server.ClientChannelOpened += (sender, args) =>
{
        var arg = args as NewChannelEventArgs;
        if (arg != null)
        {
                var incomingChannel = arg.Channel;
                // TODO: do something with incoming channel
        }
};
server.ClientChannelClosed += (sender, args) =>
{
        var closedChannel = sender as DuplexChannel;
        var arg = args as ClosedEventArgs;
        var cause = (arg != null)?arg.Cause:null;
        // TODO: process closed channel with known arguments and reason of closing
};
```

Starting the Server

```
server.Open();
```

Processing Incoming Messages

`ServerChannel` supports multiple ways to receive and process incoming messages:

- ReceiveRequest Method

- External Receiver
- Message Handler

More details about each approach are explored below.

Using the ReceiveRequest Method

Using this method allows your to define exactly when messages are read. However, you should remember that the internal queue which contains incoming messages is not unlimited. If messages are kept in the queue for a long time (5 seconds by default, although this value can be changed by setting the PsdkCustomization.ReceiveQueueTimeLimit property) without being read, then the maximum capacity of this queue will be equal to 4k elements and each new incoming message will lead to lose the eldest one.

The most popular way of using the `ReceiveRequest` method is inside a dedicated thread, as shown here:

```
var processMessagesThreadActiveFlag = new ManualResetEvent(false);
var processMessagesThread = new Thread(() =>
{
        while (!processMessagesThreadActiveFlag.WaitOne(100))
        {
                var request = server.ReceiveRequest(TimeSpan.FromMilliseconds(0));
                if (request == null) continue; // nothing to do
                var message = request.RequestMessage;
                IMessage respond = null;
                // TODO: respond = result of process request
                if (respond != null)
                request.Respond(respond);
        }
});
processMessagesThread.Start();

// TODO:

processMessagesThreadActiveFlag.Set();
processMessagesThread.Join();
```

Using an External Receiver

To use an external receiver, your should create a class which implements the `IRequestReceiverSupport` interface, and then use the `ServerChannel.SetReceiver(IRequestReceiverSupport receiver)` method to assign this receiver to `ServerChannel`.

One implementation to process incoming messages is shown below:

```
class ServerRequestReceiver : IRequestReceiverSupport
{
        public void ClearInput(){}
        public int InputSize { get; set; }
        public void ReleaseReceivers(){}
        public IRequestContext ReceiveRequest(){ return null; }
        public IRequestContext ReceiveRequest(TimeSpan timeout){ return null; }
        public void ProcessRequest(IRequestContext request)
        {
                if (request == null) return; // nothing to do
                var message = request.RequestMessage;
                IMessage respond = null;
                // TODO: respond = result of process request
                if (respond != null)
                        request.Respond(respond);
        }
}
server.SetReceiver(new ServerRequestReceiver());
```

Using Message Handler

Starting with release 8.5.1, Platform SDK has included a new mechanism to handle incoming messages. `ServerChannel` was extended with a new event called `Received`, that can be used as shown in the following example:

```
server.Received += (sender, args) =>
{
        var channel = sender as DuplexChannel;
        var arg = args as MessageEventArgs;
        if (arg == null) return;
        var incomingMessage = arg.Message;
        IMessage outgoingMessage = null;
        // TODO: outgoingMessage = result of processing incomingMessage
        if ((outgoingMessage != null) && (channel != null))
          channel.Send(outgoingMessage);
};
```

Closing ServerChannel

Closing the server channel causes all active incoming connections to be closed also. To close server channel use the `ServerChannel.Close()` method.

```
server.Close();
```

Bidirectional Messaging

The primary function of Platform SDK is to provide client protocols for communication with Genesys servers, where your applications would send requests for information and receive related events.

However, the introduction of bidirectional messaging provides an opportunity for client applications to send their own "events" and receive "requests" from the server. Reversing the direction of messages in this way allows your application to implement server side logic - which allows you to implement new servers or emulating existing servers.

Java

Existing Server Side Support

Releases of Platform SDK prior to 8.5.3 functionality allowed some server-side functionality to be implemented, but server-side code had to implement all custom preprocessing of incoming and outgoing messages. This could be inconvenient for server development, especially for complex protocols which act as containers for inner protocols.

To make this easier, the bidirectional messaging feature takes responsibility for preprocessing incoming and outgoing messages. For complex protocols, this increases transport protocol transparency and allows you to work with concrete protocols.

Bidirectional Messaging Support

Starting with Platform SDK 8.5.302, customized server channels are available for the following protocols:

Protocol	Client Handler Class Name
ConfServer	com.genesyslab.platform.configuration.protocol.ConfServerProtoc
Basic Chat	com.genesyslab.platform.webmedia.protocol.ChatProtocolListener
Flex Chat	com.genesyslab.platform.webmedia.protocol.ChatProtocolListener
Email	com.genesyslab.platform.webmedia.protocol.EmailProtocolListene
UCS	com.genesyslab.platform.contacts.protocol.UniversalContactServe
ESP Email	com.genesyslab.platform.webmedia.protocol.EspEmailProtocolList
BasicChat + FlexChat	com.genesyslab.platform.webmedia.protocol.ChatProtocolListener
Callback	com.genesyslab.platform.webmedia.protocol.CallbackProtocolListe
ESP	com.genesyslab.platform.openmedia.protocol.ExternalServiceProto

Protocol	Client Handler Class Name
InteractionServer	com.genesyslab.platform.openmedia.protocol.InteractionServerPro

Important

Platform SDK provides class com.genesyslab.platform.webmedia.protocol.ChatProtocolListener that can recognize the dialect of BasiChat and FlexChat protocols.

Using Bidirectional Messaging

The following example shows how to create and use a custom server channel for UCS.

UniversalContactServerProtocolListener Example

```
ManagedConfiguration cfg = new ManagedConfiguration(new PropertyConfiguration());
cfg.setBoolean(UniversalContactServerProtocol.USE_UTF_FOR_RESPONSES, false); // do not change
string encoding
cfg.setBoolean(TKVCodec.UTF_STRING_KEY, false);
UniversalContactServerProtocolListener listener =
  new UniversalContactServerProtocolListener(new WildcardEndpoint(0, cfg)); // creates server
channel. Port is unknown before opened.
UniversalContactServerProtocol client = new UniversalContactServerProtocol(); // creates
client channel. Endpoint is unknown before server opens.
final AtomicReference messageReference = new AtomicReference();
try{
    listener.setClientRequestHandler(new ClientRequestHandler() {
        @Override
        public void processRequest(RequestContext context){
            try {
                Message msg = context.getRequestMessage();
                context.respond(msg); // return message to sender
                messageReference.set(msg); // save link to the received message
            }catch (Exception e){}
        }
    });
    listener.open();
    int port = listener.getLocalEndPoint().getPort();
    client.setEndpoint(new Endpoint("localhost",port, cfg));
    client.open();

    EventSearch request = EventSearch.create(); // create request
    DocumentList documentList = new DocumentList(); // fill request data
    DocumentData documentData = new DocumentData();
    KeyValueCollection data = new KeyValueCollection();
    data.addObject("E-mail","email@email.com");
    documentData.setDocumentIndex(0);
    documentData.setFields(data);
    documentList.add(documentData);
    request.setDocuments(documentList);
    client.send(request);
    Message received = client.receive(10000);
    /*
```

```
       Correct conditions:
         1. received!=null
         2. request.equals(received)
         3. received.equals(messageReference.get())
         4. received!=request
    */
}finally {
    client.close();
    if (listener.getState()== ChannelState.Opened)
        listener.close();
}
```

Handshake issues

Platform SDK offers no server-side handshake procedure. Even if most protocols have a simple "unconditional" registration, your application is responsible for validating the clients itself.

.NET

Existing Server Side Support

Releases of Platform SDK prior to 8.5.3 provide specific server channel classes for ESP and Custom Routing Protocol (`ExternalServiceProtocolListener` and `UrsCustomProtocolListener` respectively). There is also an unspecified base implementation of server channel with the `ServerChannel` class.

However, these classes have some restrictions, such as being unable to switch transport layers for incoming connections, that make them unusable with Web Media Protocols.

Bidirectional Messaging Support

Starting with Platform SDK release 8.5.201, an additional server channel class was available: `ServerChannel<T>` (where T extends the `ClientChannelHandler` abstract class).

Platform SDK also includes extensions of `ClientChannelHandler` for all supported protocols. These extensions have server side messaging logic, and are responsible for substitution of XML transport instead of binary transport for Web Media Protocols.

Platform SDK .NET 8.5.3 provides extensions for the following protocols:

Protocol	Client Handler Class Name
ConfServer	Genesyslab.Platform.Configuration.Protocols.ConfServerProtocol.Cl
Basic Chat	Genesyslab.Platform.WebMedia.Protocols.BasicChatProtocol.ClientH
Flex Chat	Genesyslab.Platform.WebMedia.Protocols.FlexChatProtocol.ClientH
Email	Genesyslab.Platform.WebMedia.Protocols.EmailProtocol.ClientHand

Protocol	Client Handler Class Name
Callback	Genesyslab.Platform.WebMedia.Protocols.CallbackProtocol.ClientHa
ESP	Genesyslab.Platform.OpenMedia.Protocols.ExternalServiceProtocol
InteractionServer	Genesyslab.Platform.OpenMedia.Protocols.InteractionServerProtoc
UCS	Genesyslab.Platform.Contacts.Protocols.UniversalContactServerPro
ESP Email	Genesyslab.Platform.WebMedia.Protocols.EspEmail.EspEmailProtoc
BasicChat + FlexChat	Genesyslab.Platform.WebMedia.Protocols.ChatServerClientHandler
Stat Server	Genesyslab.Platform.Reporting.Protocols.StatServerProtocol.Client
TServer	Genesyslab.Platform.Voice.Protocols.TServerProtocol.ClientHandler

Important

Platform SDK provides the `Genesyslab.Platform.WebMedia.Protocols.ChatServerClientHandler` class that can recognize the dialect of `BasicChat` and `FlexChat` protocols. Using this class together with `ServerChannel<T>` allows the following ChatServer logic.

Using Bidirectional Messaging

The following example shows how to create and use a custom server channel for Basic Chat Protocol.

ServerChannel<T> Example

```
var cfg = new ManagedConnectionConfiguration(null) { WrapUtfString = true, }; // allows to
use utf values in KvLists
server = new ServerChannel<BasicChatProtocol.ClientHandler>(new WildcardEndpoint(0,cfg)); //
creates server channel. Port is unknown before opened.
server.Received += (sender, args) =>
{
  var msgArg = args as MessageEventArgs;
  if (msgArg == null) return; // wrong arguments (in general it's impossible with
ServerChannel<>)
  var channel = sender as DuplexChannel;
  if (channel == null) return; // wrong client (in general it's impossible with
ServerChannel<>)
  var incomingMessage = msgArg.Message; // gets incoming message
  IMessage outgoingMessage= null;
  // TODO: handle incoming message...
  if (outgoingMessage!=null) channel.Send(outgoingMessage); // sends response to client
};
server.Open(); // opens server
client = new BasicChatProtocol(new Endpoint("localhost", (server.LocalEndPoint as
IPEndPoint).Port, cfg)); // create client
// client has to be created only after server will be opened for case of unknown port
client.AutoRegister = false; // skip handshake
client.Open(); // opens client

var msg = RequestMessage.Create("12345",Visibility.All, MessageText.Create(null));
client.Send(msg); // send message
```

```
var response = client.Receive(TimeSpan.FromSeconds(5)); // receive response
// TODO: handle response...
```

Handshake Issues

Platform SDK offers no server-side handshake procedure. Even if most protocols have a simple "unconditional" registration, your application is responsible for validating the clients itself.

Hide Message Attributes in Logs

This article describes how to hide message attributes on logs generated by your Platform SDK applications.

The new `ToStringHelper` class described here was introduced in release 8.5.300.02 of Platform SDK.

How to Configure Hidden Attributes

There are two ways to configure hidden attributes within log files generated by Platform SDK, depending on what type of protocols you are working with.

Working With ESP Protocols

The UCS and EspEmail protocols use `Request3rdServer` and `Event3rdServerResponse` messages from the underlying ESP protocol to transport their messages. Thus, `Request3rdServer` and `Event3rdServerResponse` are printed to logs and `KeyValuePrinter` must be configured to hide any sensitive data in logs.

For example:

```
KeyValueCollection defaultCfg = new KeyValueCollection();
KeyValueCollection specificCfg = new KeyValueCollection();
specificCfg.addString("StructuredText","hide");
KeyValuePrinter defaultKVPrinter = new KeyValuePrinter(defaultCfg, specificCfg);
KeyValuePrinter.setDefaultPrinter(defaultKVPrinter);
```

This is the standard Genesys `[log-filter-data]` implementation described by the Hide or Tag Sensitive Data in Logs article.

In cases where UCS protocol messages are printed out directly (for example, by calling `interactionContent.toString()` in your application code) it is possible to hide sensitive data by using the new `ToStringHelper` class:

```
/// Declare hidden attributes for Protocol message or structure
ToStringHelper.hideAttribute("ContactServer",  "InteractionContent", "StructuredText");
```

Working With Non-ESP Protocols

To hide specific keys from message attributes of KVLists type in the logs, configure `KeyValuePrinter`.

To hide protocol messages attributes in logs, use the new `ToStringHelper` class. For example:

```
ToStringHelper.hideAttribute("FlexChat", "EventInfo", "Text");
```

Using the Application Template Helper to Read Configuration

from Config Manager

The Application Template will process the `CfgApplication` configuration, received from Config Server, and pass that information to `ToStringHelper.setHiddenAttributes();`.

> **Important**
>
> There is no standard format for defining log hidden attributes in Config Manger; only a format for `KVLists` structure exists).

The example below shows how to specify hidden attributes for log files that can be parsed by the Platform SDK AppTemplate mini-helper. However, you can specify settings in any other suitable format and parse them on your own.

```
[log-hidden-attributes]
<ProtocolName>.<Message Name | Complex Attribute Name> = <Attributes List>
```

For example:

```
[log-hidden-attributes]
ContactServer.InteractionContent = "Text, StructuredText"
ContactServer.SomeMessage = "Attr1, Attr2, Attr-n"
TServer.SomeEvent = "Attr-x"
```

Usage Sample

```
//Read application from ConfServer and set new log hidden attributes configuration:
String sectionName = "log-hidden-attributes"; //name of config section in the app options
CfgApplication app = confService.retrieveObject(CfgApplication.class, new
CfgApplicationQuery("AppName"));
MessagePrinterHelper.setHiddenAttributes(app.getOptions().getList(sectionName));
...

//Handle updates from Config Server:
if (cfgEvent.getCfgObject() instanceof CfgDeltaApplication) {
    CfgDeltaApplication delta = (CfgDeltaApplication) cfgEvent.getCfgObject();

    //update application configuration state
    app.update(delta);
    ...

    //set new configuration, if "log-hidden-attributes" section has been added\changed\removed
    if(MessagePrinterHelper.isConfigurationChanged(delta, sectionName)) {
        MessagePrinterHelper.setHiddenAttributes(app.getOptions().getList(sectionName));
    }
...
}
```

Resources Releasing in an Application Container

To improve performance, Platform SDK only releases internal resources (such as threads) after a slight delay so that they can be reused in the near future if beneficial.

This delay in releasing resources can lead to warnings about memory leaks from application containers like Tomcat, because Tomcat checks if all application threads are stopped immediately without taking into account that Platform SDK intentionally holds resources for a short time.

Starting with release 8.5.300.02, Platform SDK for Java includes a `PSDKRuntime` class that allows your applications to stop gracefully. When used in your application, this class does the following things:

- wait until all Platform SDK resources are released
- reduce time required to release PSDK resources

Design Notes

The `PSDKRuntime` class provides two methods:

- awaitTermination with timeout
- awaitTermination without timeout

PSDKRuntime.java

```
public final class PSDKRuntime {

  public static void awaitTermination() throws InterruptedException { ... } // It is similar
to call of awaitTermination(Long.MAX_VALUE, TimeUnit.MILLISECONDS)
  public static void awaitTermination(long timeout, TimeUnit timeUnit) throws
InterruptedException, TimeoutException { ... }

}
```

Important
These two methods wait until all Platform SDK resources are released, and reduce the release time for Platform SDK resources.

To initiate Platform SDK resources releasing:

- all Platform SDK channels have to be closed;

- if Platform SDK invokers were requested using `InvokerFactory.namedInvoker(String)` or `InvokerFactory.namedInvoker(String, int)`, then those invokers must be released (as many times as they were requested) using `InvokerFactory.releaseInvoker(String);`
- if `SingleThreadInvoker` instances were created by user then these invokers have to be released using `SingleThreadInvoker.release()` ;
- if you scheduled timer actions by using `Scheduler.schedule(long, long, TimerAction)` then these timer actions have to be canceled using `TimerActionTicket.cancel()`.

Code Sample

The following sample provides an example of how you can correctly finalize a Platform SDK-based application in J2EE containers.

TestServlet.java

```
@WebServlet("/TestServlet")
public class TestServlet extends HttpServlet {

    ConfServerProtocol protocol;
    AsyncInvoker myInvoker;
    SingleThreadInvoker myInvoker2;
    TimerActionTicket ticket;

    @Override
    public void init(ServletConfig config) throws ServletException {
        super.init(config);

        String name = "Case0001731406Test";
        String host = "host";
        String clientName = "clientName";
        String userName = "userName";
        String password = "password";
        int port = 2020;

        // creates PSDK channel
        protocol = new ConfServerProtocol(new Endpoint(name, host, port));
        protocol.setClientName(clientName);
        protocol.setUserName(userName);
        protocol.setUserPassword(password);
        protocol.setTimeout(Long.MAX_VALUE);

        // request PSDK named invoker
        myInvoker = InvokerFactory.namedInvoker("myInvoker");
        protocol.setInvoker(myInvoker);

        // request it 2nd time (it increases its reference counter)
        InvokerFactory.namedInvoker("myInvoker");

        // request PSDK invoker
        myInvoker2 = new SingleThreadInvoker();

        // open PSDK channel
        try {
            protocol.open();
        } catch (Exception e) { /*...*/ }

        // creates PSDK timer action
        TimerAction action = new TimerAction() {
            public void onTimer() {   /* ... */  }
        };

        // schedule the periodic timer action
        ticket = TimerFactory.getTimer().schedule(500, 1000, action);
    }

    public void destroy() {

        // stop periodic timer action scheduled before in PSDK timer
        if (ticket != null) {
            ticket.cancel();
            ticket = null;
        }

        // close all opened PSDK channels
        if (protocol != null) {
            try {
                protocol.close();
            } catch (Exception e) { /*...*/ }
            protocol.setInvoker(null);
            protocol = null;
        }

        // release 1st invoker
        if (myInvoker != null) {
            myInvoker = null;
            InvokerFactory.releaseInvoker("myInvoker");
            InvokerFactory.releaseInvoker("myInvoker"); // named invoker have to be released
as many times as it was requested before
        }

        // release 2nd invoker
        if (myInvoker2 != null) {
            myInvoker2.release();
            myInvoker2 = null;
        }

        // wait until all PSDK resources are stopped
        try {
            PSDKRuntime.awaitTermination(20000, TimeUnit.MILLISECONDS);
        } catch (InterruptedException e) { /*...*/ }
    }


    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException { /* ... */ }
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException { /* ... */ }

    // ...
}
```

Transport Layer Substitution

Java

The Transport Layer Substitution feature, introduced with Platform SDK release 8.5.300.02, allows you to isolate the transport layer and provide alternative ways of transporting messages. It is up to your code to exchange messages and manage connections. Goals of this feature are:

- Simulation of different server behavior in test scenarios.
- Replacement of binary protocol with any other; that is, allowing alternative message delivery sub-systems such as AMQP, MQTT, or STOMP.
- Creation of proxies, hubs etc.

Design Notes

The injected transport layer has to implement the following interface:

ExternalTransport API

```
package com.genesyslab.platform.commons.protocol;

/** Describes API of external transport. */
public interface ExternalTransport {
    /**
     * Connects to a specified destination asynchronously.
     * This method is called during the protocol is opening.
     * Method has to notify {@link ExternalTransportListener#onConnected()}
     * otherwise it must notify {@link ExternalTransportListener#onDisconnected(Throwable)}
     * @param endpoint Endpoint which describes destination address and contains
configuration.
     */
    void connect(Endpoint endpoint);

    /**
     * Disconnects from destination.
     * Method has to notify {@link ExternalTransportListener#onDisconnected(Throwable)}
     */
    void disconnect();

    /**
     * Sends message to the destination.
     * @param message which has to be sent.
     */
    void sendMessage(Message message);

    /**
     * Set external transport listener.
     * @param listener that will handle the transport events.
     */
    void setTransportListener(ExternalTransportListener listener);
}
```

ExternalTransportListener API

```
package com.genesyslab.platform.commons.protocol;

/**
 * API for external transport events handling.
 * <p>
 *     All events have to be notified within some transport notification thread.
 *
 *     Any events can't be notified by transport inside of calls of
 *     {@link ExternalTransport#connect(Endpoint)},
 *     {@link ExternalTransport#disconnect()} or
 *     {@link ExternalTransport#sendMessage(Message)}
 *     or {@link RecursiveCallException} will be thrown.
 * </p>
 */
public interface ExternalTransportListener {

    /**
     * The method is called (using channel's invoker)
     * as soon as the external transport connected to the destination.
     * @throws RecursiveCallException when it is notified inside calls of the methods
     * {@link ExternalTransport#connect(Endpoint)},
     * {@link ExternalTransport#disconnect()},
     * {@link ExternalTransport#sendMessage(Message)}
     * or {@link ExternalTransport#setTransportListener(ExternalTransportListener)}
     */
    void onConnected();

    /**
     * The method is called (using channel's invoker)
     * as soon as the external transport disconnected from the destination.
     * @param cause of disconnection. It is null if the disconnection happened due to call
     * of {@link ExternalTransport#disconnect()}.
     * @throws RecursiveCallException when it is notified inside calls of the methods
     * {@link ExternalTransport#connect(Endpoint)},
     * {@link ExternalTransport#disconnect()},
     * {@link ExternalTransport#sendMessage(Message)}
     * or {@link ExternalTransport#setTransportListener(ExternalTransportListener)}
     */
    void onDisconnected(Throwable cause);

    /**
     * The method is called (using channel's invoker)
     * for each message received by the external transport.
     * @param message that is received by the external transport.
     * @throws RecursiveCallException when it is notified inside calls of the methods
     * {@link ExternalTransport#connect(Endpoint)},
     * {@link ExternalTransport#disconnect()},
     * {@link ExternalTransport#sendMessage(Message)}
     * or {@link ExternalTransport#setTransportListener(ExternalTransportListener)}
     */
    void onMessageReceived(Message message);
}


/**
 * Used to control caller thread and prevent some recursive calls.
 */
public class RecursiveCallException extends RuntimeException {
    public RecursiveCallException(String message) {
        super(message);
    }
}
```

To substitute the transport layer used for a protocol instance in your application, you must complete the following steps:

1. Set either the system property `com.genesyslab.common.protocol.transport.factory` or `PsdkCustomization.PsdkOption.TransportFactoryImpl` to be equal to a fully qualified `ExternalTransportFactory` implementation class name. This helps to control which transport layer will be used for a specified pair (protocol description and endpoint) of arguments.
2. Use the `DuplexChannel.setExternalTransport(ExternalTransport transport)` method. This helps to substitute the transport layer for a specified instance of a channel. An external transport layer (non-null) that is set using `DuplexChannel.setExternalTransport` has more priority then the previous method of substitution.

ExternalTransportFactory API

```
package com.genesyslab.platform.commons.protocol;

/**
 * Factory of external transport for a specified protocol description and an endpoint.
 */
public interface ExternalTransportFactory {

    /**
     * Gets external transport for a specified protocol description and an endpoint.
     * @param protocolDescription specifies a protocol.
     * @param endpoint specifies host, port and configuration.
     * @return external transport instance
     * or null if PSDK have to use the default implementation.
     */
    ExternalTransport getTransport(ProtocolDescription protocolDescription, Endpoint
endpoint);
}
```

Notes:

- The `Connect` operation uses an active Endpoint of the channel, which has its own configuration. Using the settings for encoding, ADDP, TLS and other values is the responsibility of the injected transport layer.
- Sending `Connected`, `Disconnected`, and `ReceivedMessage` events is the responsibility of the injected transport layer. All transport layer implementations notify a listener, but if the notifications are performed directly inside calls to the connect, disconnect, send, or `setTransportListener` methods then `RecursiveCallException` is thrown.

.NET

The Transport Layer Substitution feature, introduced with Platform SDK release 8.5.300.02, allows you to isolate the transport layer and provide alternative ways of transporting messages. It is up to your code to exchange messages and manage connections. Goals of this feature are:

- Simulation of different server behavior in test scenarios.

- Replacement of binary protocol with any other; that is, allowing alternative message delivery sub-systems such as AMQP, MQTT, or STOMP.
- Creation of proxies, hubs etc.

Design Notes

The injected transport layer has to implement the following interface:

API of External Transport

```
/// <summary>
/// Describes API of external transport
/// </summary>
public interface IExternalTransport
{
  /// <summary>
  /// Returns state of connection to the destination.
  /// </summary>
  ConnectionState State { get; }
  /// <summary>
  /// Connects to destination.
  /// This method is called during the protocol is opening.
  /// Method has to raise event <see cref="Connected"/> if connection was successful,
  /// otherwise it must raise event <see cref="Disconnected"/>.
  /// <param name="endpoint">Endpoint which describes destination address
  /// and contains configuration.</param>
  /// </summary>
  void Connect(Endpoint endpoint);

  /// <summary>
  /// Disconnects from destination.
  /// Method has to raise event <see cref="Disconnected"/> when connection disconnected.
  /// </summary>
  void Disconnect();

  /// <summary>
  /// Fired, when connection to the destination becomes opened.
  /// </summary>
  event EventHandler Connected;

  /// <summary>
  /// Fired, when connection to the destination becomes closed.
  /// </summary>
  event EventHandler<ConnectionEventArgs> Disconnected;

  /// <summary>
  /// Sends message to the destination
  /// </summary>
  /// <param name="message">Message which has to be sent</param>
  void SendMessage(IMessage message);

  /// <summary>
  /// This event hast to be fired when received IMessage is ready for user.
  /// </summary>
  event EventHandler<MessageEventArgs> ReceivedMessage;
}
```

The injection of a new transport layer is made by using the public method of DuplexChannel class. Its signature is:

```
public void SetExternalTransport(IExternalTransport transport)
```

Notes:

- The `Connect` operation uses an active Endpoint of the channel, which has its configuration. Using settings of encoding, ADDP, TLS etc. becomes a responsibility of the injected transport layer.
- Sending `Connected`, `Disconnected`, and `ReceivedMessage` events is the responsibility of the injected transport layer.

Abstract Implementation

Platform SDK may provide base abstract implementation in order to simplify the 3rd-party code which uses an external transport layer. It may have the following signature:

Abstract Implementation of Basic Functionality

```
/// <summary>
/// Abstract implementation of basic functionality of the
/// <see cref="IExternalTransport"/> interface.
/// </summary>
public abstract class ExternalTransportBase:AbstractLogEnabled, IExternalTransport
{
  /// <summary>
  /// Returns state of connection to the destination.
  /// </summary>
  public ConnectionState State { get; protected set; }

  /// <summary>
  /// Sends message to the destination
  /// </summary>
  /// <param name="message">Message which has to be sent</param>
  public abstract void SendMessage(IMessage message);

  /// <summary>
  /// Connects to the destination. No need to raise any events.
  /// After successful connection property <see cref="State"/> has to be set to <see
cref="ConnectionState.Opened"/> state.
  /// </summary>
  /// <param name="endpoint">Endpoint which describes destination address
  /// and contains configuration.</param>
  /// <exception cref="Exception">If connection is unsuccessful.</exception>
  public abstract void DoConnect(Endpoint endpoint);


  /// <summary>
  /// Disconnects from the destination.
  /// </summary>
  public abstract void DoDisconnect();

  /// <summary>
  /// Fired, when connection to the destination becomes opened.
  /// </summary>
  public event EventHandler Connected;

  /// <summary>
  /// Fired, when connection to the destination becomes closed.
  /// </summary>
  public event EventHandler<ConnectionEventArgs> Disconnected;

  /// <summary>
  /// This event hast to be fired when received IMessage is ready for user.
  /// </summary>
  public event EventHandler<MessageEventArgs> ReceivedMessage;

  /// <summary>
  /// Simulates receiving message.
  /// </summary>
  /// <param name="message">Message which is received.</param>
  public void OnMessageReceived(IMessage message) {…}

  void IExternalTransport.Disconnect() {…}

  void IExternalTransport.Connect(Endpoint endpoint) {…}

  /// <summary>
  /// Raise <see cref="Disconnected"/> event.
  /// </summary>
  /// <param name="args"><see cref="ConnectionEventArgs"/> parameter.</param>
  protected void FireDisconnect(ConnectionEventArgs args) {…}

  /// <summary>
  /// Raise <see cref="Disconnected"/> event.
  /// </summary>
  /// <param name="args"><see cref="ConnectionEventArgs"/> parameter.</param>
  protected void FireConnect(EventArgs args) {…}
}
```

Usage of this implementation allows end-user simplify own implementation.

Code Sample

Example of External Transport Implementation

```
internal class TheNewTransport : IExternalTransport
{
  private readonly IMessageFactory _factory = new ConfServerProtocolFactory();
  public ConnectionState State { get; private set; }
  public void Connect(Endpoint endpoint)
  {
    State = ConnectionState.Opening;
    Console.WriteLine("Connecting to: {0}", endpoint.ToString());
    ThreadPool.QueueUserWorkItem(state =>
    {
      Thread.Sleep(100); //
      // TODO: do something to connect to destination
      var handler = Connected;
      State = ConnectionState.Opened;
      if (handler != null) handler(this, null);
    });
  }
  public void Disconnect()
  {
    State = ConnectionState.Closing;
    Console.WriteLine("Dicsonnecting");
    ThreadPool.QueueUserWorkItem(state =>
    {
      Thread.Sleep(100);
      // TODO: do something to disconnect from destination
      var handler = Disconnected;
      State = ConnectionState.Closed;
      if (handler != null) handler(this, null);
    });
  }
  public event EventHandler Connected;
  public event EventHandler<ConnectionEventArgs> Disconnected;
  public void SendMessage(IMessage message)
  {
    Console.WriteLine("Sending message: {0}", message);
    if (message.Id == 49) // request ‘RequestProtocolVersion’
    {
      var msg = _factory.CreateMessage(50);
      // response ‘EventProtocolVersion’
      msg["ReferenceId"] = message["ReferenceId"];
      msg["OldProtocolVersion"] = message["ProtocolVersion"];
      ReceiveMessage(msg);
    }
    if (message.Id == 3) // request ‘RequestRegisterClient’
    {
      var msg = _factory.CreateMessage(19);
      // response ‘EventClientRegistered’
      msg["ReferenceId"] = message["ReferenceId"];
      msg["OldProtocolVersion"] = message["ProtocolVersion"];
      msg["ProtocolVersion"] = message["ProtocolVersion"];
      ReceiveMessage(msg);
    }
  }
  private void ReceiveMessage(IMessage message)
  {
    var handler = ReceivedMessage;
    if (handler != null)
    {
      var args = new MessageEventArgs(message);
      handler(this, args);
    }
  }
  public event EventHandler<MessageEventArgs> ReceivedMessage;
}
```

Using Injected Transport (Test Method Snippet)

```
var client = new ConfServerProtocol(new Endpoint("localhost", 56789))
client.Timeout = TimeSpan.FromSeconds(3);
client.SetExternalTransport(new TheNewTransport());
client.Open();
Assert.IsTrue(client.State == ChannelState.Opened);
client.Close();
Assert.IsTrue(client.State == ChannelState.Closed);
```

Example Using Base Implementation

```
internal class CustomExternalTransport : ExternalTransportBase
{
  public override void SendMessage(IMessage message)
  {
    Console.WriteLine("Message received: {0}",message); // log to console
    IMessage returnMessage = null;
    // TODO: process handshake and other logic
    if (returnMessage!=null)
      OnMessageReceived(returnMessage); // returns message to sender
  }
  public override void DoConnect(Endpoint endpoint)
  {
    // TODO: connect (or do nothing if there is no need to connect anywhere)
    State = ConnectionState.Opened;
  }
  public override void DoDisconnect()
  {
    // TODO: disconnect
  }
}

[TestMethod]
public void TestExternalTransport()
{
  var protocol = new ConfServerProtocol(new Endpoint("localhost",12345));
  protocol.SetExternalTransport(new CustomExternalTransport());
  protocol.Open();
  Assert.IsTrue(protocol.State == ChannelState.Opened);
  var request = RequestReadLocale.Create(123);
  var response = protocol.Request(request);
  Assert.AreSame(request,response);
  protocol.Close();
  Assert.IsTrue(protocol.State==ChannelState.Closed);
}
```

Dynamically Linked Factory

An external transport layer may be linked to an existing application dynamically by using the factory interface implementation contained in the external assembly. This interface has the following description:

IExternalTransportFactory description

```
/// <summary>
/// Describes API of external transport factory.
/// It may be dynamically linked and used by ClientChannel before opening.
/// </summary>
public interface IExternalTransportFactory
{
  /// <summary>
  /// Creates instance of external transport for given endpoint.
  /// </summary>
  /// <param name="protocolDescription">Description of protocol</param>
  /// <param name="endpoint">Endpoint which is used as key to create external
transport</param>
  /// <returns>Instance of external transport, or null if it is not needed</returns>
  IExternalTransport GetTransport(ProtocolDescription protocolDescription, Endpoint endpoint);
}
```

The public class which implements this interface must have a public constructor with no parameters. Otherwise it won't load correctly.

This factory provides allows you to create different implementations - not only for different protocols, but also even for different instances of the same protocol - by using the `protocolDescription` and `endpoint` parameters.

The location of your assembly and class name should be defined in the app.config file before your application starts:

App.Config Configuration File

```
<?xml version="1.0"?>
<configuration>
  <appSettings>
    <add key="ExternalTransport.AssemblyFileName" value="[path]\..."/>
    <add key="ExternalTransport.ClassName" value="..."/>
  </appSettings>
</configuration>
```

Server-Specific Overviews

Telephony (T-Server)

Java

You can use the Voice Platform SDK to write Java or .NET applications that monitor and handle voice interactions from a traditional or IP-based telephony device. These applications can range from the simple to the advanced. This document shows how to implement the basic functions you will need to write a simple voice application. It is organized to show the kind of structure you will probably use to write your own applications.

Setting Up a TServerProtocol Object

The first thing you need to do to use the Voice Platform SDK is instantiate a `TServerProtocol` object. To do that, you must supply information about the T-Server you want to connect with. This example provides the server's name, host, and port information:

```
[Java]

TServerProtocol tServerProtocol =
        new TServerProtocol(
                new Endpoint(
                        tServerName, host, port));
```

After instantiating the `TServerProtocol` object, you need to open the connection to the T-Server:

```
[Java]

tServerProtocol.open();
```

Registering an Address

Now you need to register a DN for your agent to use. To do this, you must send a `RequestRegisterAddress` request to the server.

Here is how to create this request:

```
[Java]

RequestRegisterAddress requestRegisterAddress =
        RequestRegisterAddress.create(
                thisDn,
                RegisterMode.ModeShare,
                ControlMode.RegisterDefault,
                AddressType.DN);
```

The `thisDn` argument refers to the DN you want to associate with your agent, while `RegisterMode.ModeShare` tells the T-Server to share information about the DN with other applications. The next argument asks to use the switch's default value for deciding whether to let the switch know that you have registered this DN. And finally, you are specifying that the object you are registering is a DN.

After you create the request, you will need to send it to the T-Server:

```
[Java]

Message response =
        tServerProtocol.request(requestRegisterAddress);
```

Remember that the `request()` method is synchronous. If you use this method, your application will block until you hear back from the server. When you get the response, you can execute code to handle the response. In this case, you probably don't need to do anything if the request is successful:

```
[Java]

switch(response.messageId())
{
        case EventRegistered.ID:
        case EventUnregistered.ID:
                break;
        .
        .
        .
}
```

Logging in an Agent

Once you have registered a DN to your agent, you can log him or her in. To do this, you need to create a `RequestAgentLogin` request:

```
[Java]

RequestAgentLogin requestAgentLogin =
        RequestAgentLogin.create(
                thisDn,
                AgentWorkMode.AutoIn);
```

After you create the request, you will need to indicate the queue the agent will be using, and you may need to supply the agent's user name and password. Once you have done this, you can send the request to the server:

```
[Java]

requestAgentLogin.setThisQueue(thisQueue);
// Your switch may not need a user name and password:
requestAgentLogin.setAgentID(userName);
requestAgentLogin.setPassword(password);
Message response = tServerProtocol.request(requestAgentLogin);
```

If your request is successful, the server will respond with an `EventAgentLogin` event. At that point, you may need to update the state of your user interface to indicate that the agent can no longer log in, but that, for example, he or she can now log out.

Answering a Call

Now that your agent is logged in, he or she can handle calls. Let's start by answering a call.

When a call comes in, your application will receive an `EventRinging` message. When you get this message, you will probably want to enable an answer button. Here is how to do that:

```
[Java]

switch(response.messageId())
{
        .
        .
        .
        case EventRinging.ID:
                EventRinging eventRinging = (EventRinging) response;
                connId = eventRinging.getConnID();
                if (eventRinging.getThisDN() == thisDn)
                {
                        AnswerButton.enabled = true;
                }
                break;
        .
        .
        .
}
```

It is important to note that an `EventRinging` event will also be triggered when you are sending an outbound call. So this particular snippet is only enabling the answer button if the call is ringing on `thisDN`. As you can also see, when you receive an `EventRinging` you will want to store the `ConnID` of the call associated with it.

After the agent clicks the answer button, you need to send a request to answer the call, using your DN and the `ConnID` of the call:

```
[Java]

RequestAnswerCall requestAnswerCall =
        RequestAnswerCall.create(
        thisDn,
        connId);
Message response = tServerProtocol.request(requestAnswerCall);
```

If the request is successful, you will receive an `EventEstablished`.

Releasing a Call

When your agent is finished with the call, he or she will need to release it:

```
[Java]

RequestReleaseCall requestReleaseCall =
        RequestReleaseCall.create(
                thisDn,
                connId);
```

```
Message response = tServerProtocol.request(requestReleaseCall);
```

If the request is successful, you will receive an `EventReleased`.

Making a Call

Here is how to make a call:

```
[Java]

RequestMakeCall requestMakeCall =
        RequestMakeCall.create(
                thisDn,
                thatDn,
                MakeCallType.DirectAgent);
Message response = tServerProtocol.request(requestMakeCall);
```

If the request is successful, you will receive an `EventDialing` message, an `EventRinging` message, and then, when your party responds, an `EventEstablished` message.

Setting up a Conference Call

After you make or answer a call, you can add another party to the call. Here is how to perform an ordinary two-step conference call.

To start off, you need to initiate a conference call, supplying your own DN, the connection ID of the existing call, and the DN of the party you want to add to the call:

```
[Java]

RequestInitiateConference requestInitiateConference =
        RequestInitiateConference.create(
                thisDn,
                connId,
                otherDn);
Message response = tServerProtocol.request(requestInitiateConference);
```

Tip

In a real telephony application, the events you would receive in response to the kinds of conferencing requests shown here could also be generated by other requests. For example, you might receive an `EventDialing` or an `EventEstablished` in response to a `RequestMakeCall` or `RequestInitiateTransfer`. Because of this, a real-world application will need to keep track of the requests that initiate these events in order to interpret them correctly.

If the initiate request is successful, you will receive an `EventDialing` message and an `EventHeld` message. When your party picks up the call, you will also receive an `EventEstablished` message.

Now you need to complete the conference call.

When you received the `EventDialing` message from the `RequestInitiateConference`, you were given a new connection ID associated with the party you want to establish the conference call with. You will need that connection ID, in addition to your own DN and the original connection ID, in order to complete the conference call:

```
[Java]

RequestCompleteConference requestCompleteConference =
        RequestCompleteConference.create(
                thisDn,
                connId,
                secondConnId);
response = tServerProtocol.request(requestCompleteConference);
```

If the completion request is successful, you will receive `EventReleased`, `EventRetrieved`, `EventPartyAdded`, and `EventAttachedDataChanged` messages.

Transferring a Call

After you make or answer a call, you may also want to transfer that call. Here is how to perform an ordinary two-step transfer.

To start off, you need to initiate a transfer, supplying your own DN, the connection ID of the existing call, and the DN of the party you want to transfer the call to.

```
[Java]

RequestInitiateTransfer requestInitiateTransfer =
        RequestInitiateTransfer.create(
                thisDn,
                connId,
                otherDn);
Message response = tServerProtocol.request(requestInitiateTransfer);
```

Tip

In a real telephony application, the events you would receive in response to the kinds of transfer requests shown here could also be generated by other requests. For example, you might receive an `EventDialing` or an `EventEstablished` in response to a `RequestMakeCall` or `RequestInitiateConference`. Because of this, a real-world application will need to keep track of the requests that initiate these events in order to interpret them correctly.

If the initiate request is successful, you will receive an `EventDialing` message and an `EventHeld` message. When the party you want to transfer to picks up the call, you will also receive an `EventEstablished` message.

Now you need to complete the transfer.

When you received the `EventDialing` message from the `RequestInitiateTransfer`, you were given a new connection ID associated with the party you want to transfer the call to. You will need that connection ID, in addition to your own DN and the original connection ID, in order to complete the transfer:

```
[Java]

RequestCompleteTransfer requestCompleteTransfer =
        RequestCompleteTransfer.create(
                thisDn,
                connId,
                secondConnId);
response = tServerProtocol.request(requestCompleteTransfer);
```

If the completion request is successful, you will receive two `EventReleased` messages and you will no longer be a party to the call.

Closing the Connection

Finally, when you are finished communicating with the T-Server, you should close the connection to minimize resource utilization:

```
[Java]

tServerProtocol.close();
```

.NET

You can use the Voice Platform SDK to write Java or .NET applications that monitor and handle voice interactions from a traditional or IP-based telephony device. These applications can range from the simple to the advanced. This document shows how to implement the basic functions you will need to write a simple voice application. It is organized to show the kind of structure you will probably use to write your own applications.

Setting Up a TServerProtocol Object

The first thing you need to do to use the Voice Platform SDK is instantiate a `TServerProtocol` object. To do that, you must supply information about the T-Server you want to connect with. This example uses the URI of the T-Server, but you can also use name, host, and port information:

```
[C#]

TServerProtocol tServerProtocol =
        new TServerProtocol(
                new Endpoint(
                        tServerUri));
```

After instantiating the `TServerProtocol` object, you need to open the connection to the T-Server:

```
[C#]

tServerProtocol.Open();
```

Registering an Address

Now you need to register a DN for your agent to use. To do this, you must send a `RequestRegisterAddress` request to the server.

Here is how to create this request:

```
[C#]

RequestRegisterAddress requestRegisterAddress =
        RequestRegisterAddress.Create(
        thisDn,
        RegisterMode.ModeShare,
        ControlMode.RegisterDefault,
        AddressType.DN);
```

The `thisDn` argument refers to the DN you want to associate with your agent, while `RegisterMode.ModeShare` tells the T-Server to share information about the DN with other applications. The next argument asks to use the switch's default value for deciding whether to let the switch know that you have registered this DN. And finally, you are specifying that the object you are registering is a DN.

After you create the request, you will need to send it to the T-Server:

```
[C#]

IMessage response = tServerProtocol.Request(requestRegisterAddress);
```

Remember that the `Request()` method is synchronous. If you use this method, your application will block until you hear back from the server. When you get the response, you can execute code to handle the response. In this case, you probably don't need to do anything if the request is successful:

```
[C#]

switch(response.Id )
{
        case EventRegistered.MessageId:
        case EventUnregistered.MessageId:
                break;
        .
        .
        .
}
```

Logging in an Agent

Once you have registered a DN to your agent, you can log him or her in. To do this, you need to create a `RequestAgentLogin` request:

```
[C#]

RequestAgentLogin requestAgentLogin =
        RequestAgentLogin.Create(
        thisDn,
        AgentWorkMode.AutoIn);
```

After you create the request, you will need to indicate the queue the agent will be using, and you may need to supply the agent's user name and password. Once you have done this, you can send the request to the server:

```
[C#]

requestAgentLogin.ThisQueue = thisQueue;
// Your switch may not need a user name and password:
requestAgentLogin.AgentID = userName;
requestAgentLogin.Password = password;
IMessage response = tServerProtocol.Request(requestAgentLogin);
```

If your request is successful, the server will respond with an `EventAgentLogin` event. At that point, you may need to update the state of your user interface to indicate that the agent can no longer log in, but that, for example, he or she can now log out.

Answering a Call

Now that your agent is logged in, he or she can handle calls. Let's start by answering a call.

When a call comes in, your application will receive an `EventRinging` message. When you get this message, you will probably want to enable an answer button. Here is how to do that:

```
[C#]

switch(response.Id)
{
        .
        .
        .
        case EventRinging.MessageId:
                EventRinging eventRinging = (EventRinging) response;
                connId = eventRinging.ConnID;
                if (eventRinging.ThisDN == thisDn)
                {
                        AnswerButton.Enabled = true;
                }
                break;
        .
        .
        .
}
```

It is important to note that an `EventRinging` event will also be triggered when you are sending an outbound call. So this particular snippet is only enabling the answer button if the call is ringing on `thisDN`. As you can also see, when you receive an `EventRinging` you will want to store the `ConnID` of the call associated with it.

After the agent clicks the answer button, you need to send a request to answer the call, using your

DN and the `ConnID` of the call:

```
[C#]

RequestAnswerCall requestAnswerCall =
        RequestAnswerCall.Create(
        thisDn,
        connId);
IMessage response = tServerProtocol.Request(requestAnswerCall);
```

If the request is successful, you will receive an `EventEstablished`.

Releasing a Call

When your agent is finished with the call, he or she will need to release it:

```
[C#]

RequestReleaseCall requestReleaseCall =
        RequestReleaseCall.Create(
        thisDn,
        connId);
IMessage response = tServerProtocol.Request(requestReleaseCall);
```

If the request is successful, you will receive an `EventReleased`.

Making a Call

Here is how to make a call:

```
[C#]

RequestMakeCall requestMakeCall =
        RequestMakeCall.Create(
        thisDn,
        thatDn,
        MakeCallType.DirectAgent);
IMessage response = tServerProtocol.Request(requestMakeCall);
```

If the request is successful, you will receive an `EventDialing` message, an `EventRinging` message, and then, when your party responds, an `EventEstablished` message.

Setting up a Conference Call

After you make or answer a call, you can add another party to the call. Here is how to perform an ordinary two-step conference call.

To start off, you need to initiate a conference call, supplying your own DN, the connection ID of the existing call, and the DN of the party you want to add to the call:

```
[C#]

RequestInitiateConference requestInitiateConference =
        RequestInitiateConference.Create(
        thisDn,
        connId,
        otherDn);
IMessage response = tServerProtocol.Request(requestInitiateConference);
```

Tip

In a real telephony application, the events you would receive in response to the kinds of conferencing requests shown here could also be generated by other requests. For example, you might receive an `EventDialing` or an `EventEstablished` in response to a `RequestMakeCall` or `RequestInitiateTransfer`. Because of this, a real-world application will need to keep track of the requests that initiate these events in order to interpret them correctly.

If the initiate request is successful, you will receive an `EventDialing` message and an `EventHeld` message. When your party picks up the call, you will also receive an `EventEstablished` message.

Now you need to complete the conference call.

When you received the `EventDialing` message from the `RequestInitiateConference`, you were given a new connection ID associated with the party you want to establish the conference call with. You will need that connection ID, in addition to your own DN and the original connection ID, in order to complete the conference call:

```
[C#]

RequestCompleteConference requestCompleteConference =
        RequestCompleteConference.Create(
        thisDn,
        connId,
        secondConnId);
response = tServerProtocol.Request(requestCompleteConference);
```

If the completion request is successful, you will receive `EventReleased, EventRetrieved, EventPartyAdded`, and `EventAttachedDataChanged` messages.

Transferring a Call

After you make or answer a call, you may also want to transfer that call. Here is how to perform an ordinary two-step transfer.

To start off, you need to initiate a transfer, supplying your own DN, the connection ID of the existing call, and the DN of the party you want to transfer the call to.

```
[C#]

RequestInitiateTransfer requestInitiateTransfer =
        RequestInitiateTransfer.Create(
        thisDn,
        connId,
        otherDn);
IMessage response = tServerProtocol.Request(requestInitiateTransfer);
```

Tip

In a real telephony application, the events you would receive in response to the kinds of transfer requests shown here could also be generated by other requests. For example, you might receive an `EventDialing` or an `EventEstablished` in response to a `RequestMakeCall` or `RequestInitiateConference`. Because of this, a real-world application will need to keep track of the requests that initiate these events in order to interpret them correctly.

If the initiate request is successful, you will receive an `EventDialing` message and an `EventHeld` message. When the party you want to transfer to picks up the call, you will also receive an `EventEstablished` message.

Now you need to complete the transfer.

When you received the `EventDialing` message from the `RequestInitiateTransfer`, you were given a new connection ID associated with the party you want to transfer the call to. You will need that connection ID, in addition to your own DN and the original connection ID, in order to complete the transfer:

```
[C#]

RequestCompleteTransfer requestCompleteTransfer =
        RequestCompleteTransfer.Create(
        thisDn,
        connId,
        secondConnId);
response = tServerProtocol.Request(requestCompleteTransfer);
```

If the completion request is successful, you will receive two `EventReleased` messages and you will no longer be a party to the call.

Closing the Connection

Finally, when you are finished communicating with the T-Server, you should close the connection to minimize resource utilization:

```
[C#]

tServerProtocol.Close();
```

Introduction to TLib Functions and Data

This page provides a flat list of TLib functions, datatypes, and unstructured data that you should be familiar with during development.

For detailed descriptions and additional information, please see the TLib Reference Guide.

List of TLib Functions

The following table provides a convenient list of TLib Functions that are available.

TAgentLogin	TEventGetStringAttr	TNetworkMerge	TSendEvent
TAgentLogout	TFreeEvent	TNetworkPrivateService	TSendEventEx
TAgentSetIdleReason	TGetAccessNumber	TNetworkReconnect	TSendUserEvent
TAgentSetNotReady	TGetMessageTypeName	TNetworkSingleStepTransfer	TSetCallAttributes
TAgentSetReady	TGetReferenceID	TNetworkTransfer	TSetDNDOff
TAlternateCall	TGetRouteTypeNames	TOpenServer	TSetDNDOn
TAnswerCall	TGetTreatmentTypeNames	TOpenServerEx	TSetInputMask
TApplyTreatment	TGetXCaps	TOpenServerX	TSetMessageWaitingOff
TAttachUserData	TGiveMusicTreatment	TOpenVoiceFile	TSetMessageWaitingOn
TCallCancelForward	TGiveRingBackTreatment	TPlayVoice	TSetMuteOff
TCallSetForward	TGiveSilenceTreatment	TPrivateService	TSetMuteOn
TCancelMonitoring	THoldCall	TQueryAddress	TSetParamHA
TCancelReqGetAccessNumber	TInitiateConference	TQueryCall	TSetRefIDLimit
TClearCall	TInitiateTransfer	TQueryLocation	TSetReferenceID
TCloseServer	TLibSetCompatibMode	TQueryServer	TSetSocketChangeCallback
TCloseVoiceFile	TListenDisconnect	TQuerySwitch	TSingleStepConference
TCollectDigits	TListenReconnect	TReconnectCall	TSingleStepTransfer
TCompleteConference	TLoginMailBox	TRedirectCall	TSockInfoStructure
TCompleteTransfer	TLogoutMailBox	TRegisterAddress	TSyncIsSet
TCopyEvent	TMakeCall	TReleaseCall	TSyncSetSelectMask
TDeleteAllUserData	TMakePredictiveCall	TReserveAgent	TUnregisterAddress
TDeleteFromConference	TMergeCalls	TRetrieveCall	TUpdateUserData
TDeleteUserData	TMonitorNextCall	TRouteCall	TXCapsSupported
TDispatch	TMuteTransfer	TScanServer	connid_to_decimal

TEventGetConnID	TNetworkAlternate	TScanServerEx	connid_to_str
TEventGetIntAttr	TNetworkConsult	TSendDTMF	decimal_to_connid
			str_to_connid

List of TLib Datatypes

The following table provides a convenient list of TLib Datatypes that are available.

Important
The names of most of these datatypes start with a "T", but the Voice Platform SDK uses names that do not contain an initial "T". For example, the `TRegisterMode` datatype mentioned in this section is known to the Voice Platform SDK as `RegisterMode`.

AddressStatusInfoType	TCallState	TKVResult	TRegisterMode
AssociationInfoType	TCallType	TKVType	TReliability
MsgWaitingInfoType	TClearFlag	TLocationInfoType	TRemoteParty
TAddressInfoStatus	TConnectionID	TMakeCallType	TRouteType
TAddressInfoType	TControlMode	TMediaType	TScanServerMode
TAddressType	TDNRole	TMergeType	TServer
TAgentID	TDirectoryNumber	TMessageType	TServerRole
TAgentPassword	TEvent	TMonitorNextCallType	TSetOpType
TAgentType	TEventMask	TNetworkCallState	TSwitchInfoType
TAgentWorkMode	TFile	TNetworkDestState	TTime
TAttribute	TForwardMode	TNetworkPartyRole	TTimeStamp
TCallHistoryInfo	TInterruptFlag	TOpenMode	TTreatmentType
TCallID	TKVList	TPartyState	TXCaps
TCallInfoType	TKVPair	TPrivateMsgType	TXRouteType

List of TLib Unstructured Data

The following table provides a convenient list of TLib Unstructured Data functions that are available.

These functions deal exclusively with transaction-related user data on the client side and allow you to work with all three categories of unstructured data: *User Data*, *Extensions*, and *Reasons*. None of these functions generate any requests to T-Server. The result of the function execution is confirmed by the value that the function returns.

TKVListAddBinary	TKVListCreate	TKVListGetListValue	TKVListNextPair
TKVListAddInt	TKVListDeleteAll	TKVListGetStringValue	TKVListPrint
TKVListAddList	TKVListDeletePair	TKVListGetUnicodeValue	TKVListStringValue
TKVListAddString	TKVListDup	TKVListInitScanLoop	TKVListType
TKVListAddUnicode	TKVListFree	TKVListIntValue	TKVListUnicodeValue
TKVListBinaryLength	TKVListGetBinaryValue	TKVListKey	TVKListGetPair
TKVListBinaryValue	TKVListGetIntValue	TKVListListValue	

Configuration

You can use the Configuration Platform SDK to write Java or .NET applications that access and update information from the Genesys Configuration Layer. These applications can range from the simple to the advanced.

This article shows how to implement the basic functions you will need to write a simple Configuration Layer application.

Once you have reviewed the information in this document, you should familiarize yourself with Configuration Layer Objects. Since the Configuration Platform SDK uses these objects for nearly everything it does, you will need to understand them before you start using this SDK.

Tip

The Platform SDK includes the Configuration Object Model (COM) Application Block, which is a high-performance component you can use to query on, and to create, update, and delete, Configuration Layer objects. Genesys recommends that you use this application block for most of the work you do with Configuration Layer objects.

When you are ready to write more complicated applications, take a look at the classes and methods described in the Platform SDK API Reference.

Java

Setting Up a ConfServerProtocol Object

The first thing you need to do to use the Configuration Platform SDK is instantiate a `ConfServerProtocol` object. To do that, you must supply information about the Configuration Server you want to connect with. This example uses the URI of the Configuration Server, but you can also use the server's name, host, and port information:

```
ConfServerProtocol confServerProtocol =
        new ConfServerProtocol(
                new Endpoint(
                        confServerUri));
```

Configuration Server needs some additional information in order to create a successful connection. This information includes the type of client you wish to create, your client's name, and your user name and password:

```
confServerProtocol.setClientApplicationType(CfgAppType.CFGSCE.asInteger());
confServerProtocol.setClientName("default");
confServerProtocol.setUserName(userName);
confServerProtocol.setUserPassword(password);
```

After instantiating the `ConfServerProtocol` object, you need to open the connection to the Configuration Server:

```
confServerProtocol.open();
```

Creating a Query

Now that you have opened a connection, you can create a query and send it to Configuration Server. Starting with release 8.1.4, there are two types of queries supported:

- Filter-based Queries (using `RequestReadObjects`)
- XPath-based Queries (using `RequestReadObjects2`)

If the request is successful, you will receive an `EventObjectsRead` message with the matching data.

Tip

When you send a RequestReadObjects message, Configuration Server may send more than one `EventObjectsRead` messages in response, depending on whether there is too much data to be handled by a single `EventObjectsRead`. Once you have received all of the `EventObjectsRead` messages, Configuration Server will also send an `EventObjectsSent`, which confirms that it has completed your request. For more information, refer to the article on event handling.

Examples of both query types are shown below, showing how you could retrieve information about a particular agent.

Filter-based Queries

For this type of query, you will need to supply the agent's user name using a *filter key*. The filter key tells Configuration Server to narrow your query to a specific agent, rather than retrieving information about all of the persons in your contact center:

```
KeyValueCollection filterKey = new KeyValueCollection();
filterKey.addObject("user_name", userName);
```

You can find the names of the filter keys for Person objects by looking in the *Filter Keys* section of the CfgPerson entry.

Tip

A similar reference page is available for each Configuration Layer object.

Now you are ready to create the request. For filter-based queries, this is done using `RequestReadObjects`.

As you may know, Configuration Server considers agents to be objects of type CfgPerson. So you will need to create a request for information about a `Person` who has the user name you specified in the filter key:

```
CfgObjectType objectType = CfgObjectType.CFGPerson;
int intPerson = objectType.asInteger();
RequestReadObjects requestFilterQuery =
        RequestReadObjects.create(
                intPerson,
                filterKey);
```

Important

While the Configuration Layer supports the full character set in defining object names, using certain characters can cause problems in the behavior of some Genesys applications. Avoid using spaces, dashes, periods, or special characters in object names. Consider using underscores where you might normally use spaces or dashes.

After you have created your request, you can send it to Configuration Server, as shown here:

```
confServerProtocol.send(requestFilterQuery);
```

XPath-based Queries

Submitting XPath-based queries is similar to filter-based queries, but does not require any filters or additional objects - instead an XPath search expression is passed to `RequestReadObjects2` as a string.

As in the example above, Configuration Server considers agents to be objects of type CfgPerson. So you will need to create a request for information about a `Person` who has the user name you are looking for:

```
CfgObjectType objectType = CfgObjectType.CFGPerson;
int intPerson = objectType.asInteger();
RequestReadObjects2 requestXpathQuery =
        RequestReadObjects2.create(
                intPerson,
                "CfgPerson[@firstName='John']");
```

Important

While the Configuration Layer supports the full character set in defining object names, using certain characters can cause problems in the behavior of some Genesys applications. Avoid using spaces, dashes, periods, or special characters in object names. Consider using underscores where you might normally use spaces or dashes.

After you have created your request, you can send it to Configuration Server, as shown here:

```
confServerProtocol.send(requestXpathQuery);
```

Interpreting the Response

The 8.5.0 release of Platform SDK introduces new structures for handling configuration object data, instead of the heavyweight DOM trees used in previous Platform SDK releases.

Information that you request is returned by invoking the `getObjects` method of the `EventObjectsRead` message. This method returns a `ConfObject` collection that may contain one or more configuration objects depending on the request query filter.

The `ConfObject` structure is represented as a set of object properties linked with the actual object type metadata description (that is, the schema definition for the configuration server objects).

Using the `toString()` method to dump a sample application object might result in the following:

```
ConfObject(CfgApplication) {
        "DBID" = 631
        "name" = "SampleServerApp-SV-B"
        "type" = 107
        "version" = "8"
        "isServer" = 2
        "serverInfo" = ConfStructure(CfgServerInfo) {
                "hostDBID" = 123
                "port" = "7007"
                "backupServerDBID" = 0
                "timeout" = 10
                "attempts" = 1
        }
        "state" = 1
        "appPrototypeDBID" = 177
        "workDirectory" = "C:\GCTI\SampleServerApp-2\"
        "commandLine" = "SampleServerApp.cmd"
        "autoRestart" = 1
        "startupTimeout" = 90
        "shutdownTimeout" = 90
        "redundancyType" = 2
        "isPrimary" = 1
        "startupType" = 1
        "portInfos" = ConfStructureCollection[1 item(s)] = {
                [0]: ConfStructure(CfgPortInfo) {
                        "id" = "default"
                        "port" = "7007"
                        "longField1" = 0
                        "longField2" = 0
                        "longField3" = 0
                        "longField4" = 0
                }
        }
        "componentType" = 0
}
```

Actual property values can be get or set using this property schema name, as shown below:

```
Integer objDbid = (Integer) confObject.getPropertyValue("DBID");
String objName = (String) confObject.getPropertyValue("name");

confObject.setPropertyValue("name", objNewName);
```

Configuration protocol metadata descriptions for a configuration object or its properties may be received using the following code:

```
CfgDescriptionObject classInfo = confObject.getClassInfo();
CfgDescriptionAttribute attrInfo = confObject.getPropertyInfo("workDirectory");
```

This API is designed as low-level protocol structures with the ability to support different protocol/ schema versions, including forward compatibility features. That is, you can connect to some "future" version of Configuration Server, load its schema information with new objects types and/or with new objects properties, and handle its data in the same way.

Updating an Object

The Configuration Server protocol to update a configuration object uses a special kind of data structure - a "delta object". Delta objects contain object identification (DBID) and new values for changed properties.

Delta objects may be created and filled directly, or with the help of a delta utility (see the `ConfDeltaUtility` class, included in the configuration protocol library).

```
ConfObjectDelta delta0 = new ConfObjectDelta(metadata, CfgObjectType.CFGApplication);
ConfObject inDelta = (ConfObject) delta0.getOrCreatePropertyValue("deltaApplication");
inDelta.setPropertyValue("DBID", objCreated.getObjectDbid());
inDelta.setPropertyValue("name", "ConfObject-new-name");

ConfIntegerCollection delTenants = (ConfIntegerCollection)
        delta0.getOrCreatePropertyValue("deletedTenantDBIDs");
delTenants.add(105);

RequestUpdateObject reqUpdate = RequestUpdateObject.create();
reqUpdate.setObjectDelta(delta0);

Message resp = protocol.request(reqUpdate);
if (resp == null) {
    // timeout
} else if (resp instanceof EventObjectUpdated) {
    // the object has been updated
} else if (resp instanceof EventError) {
    // fail((EventError) resp);
} else {
    // unexpected server response
}
```

Creating a New Object

The new Platform SDK Configuration Protocol data structures allow users to create objects on Configuration Server without using the COM application, as shown below:

```
ConfObject obj0 = new ConfObject(metadata, CfgObjectType.CFGApplication);
obj0.setPropertyValue("name", "ConfObject-App-create-test");
obj0.setPropertyValue("type", CfgAppType.CFGGenericServer.asInteger());
obj0.setPropertyValue("version", "8.5.000.00");

obj0.setPropertyValue("workDirectory", ".");
obj0.setPropertyValue("commandLine", ".");

ConfIntegerCollection tenants = (ConfIntegerCollection)
        obj0.getOrCreatePropertyValue("tenantDBIDs");
tenants.add(101);
tenants.add(105);

ConfStructureCollection connInfos = (ConfStructureCollection)
        obj0.getOrCreatePropertyValue("appServerDBIDs");
ConfStructure connInfo = connInfos.createStructure();
connInfo.setPropertyValue("id", "conn1");
connInfo.setPropertyValue("appServerDBID", getSomeAppDbid(CfgAppType.CFGStatServer));
connInfo.setPropertyValue("connProtocol", "addp");
connInfo.setPropertyValue("timoutLocal",  5);
connInfo.setPropertyValue("timoutRemote", 7);
connInfo.setPropertyValue("mode", 3);
connInfo.setPropertyValue("transportParams", "strings-attr-encoding=utf-8");
connInfo.setPropertyValue("longField1", 0);
connInfos.add(connInfo);

RequestCreateObject reqCreate = RequestCreateObject.create();
reqCreate.setObject(obj0);

Message resp = protocol.request(reqCreate);
if (resp == null) {
    // timeout
} else if (resp instanceof EventObjectCreated) {
    ConfObject objCreated = ((EventObjectCreated) resp).getObject();
    // here we have created object from the server side with assigned DBID
} else if (resp instanceof EventError) {
    // fail((EventError) resp);
} else {
    // error: unexpected server response
}
```

Closing the Connection

Finally, when you are finished communicating with the Configuration Server, you should close the connection, in order to minimize resource utilization:

```
confServerProtocol.close();
```

Working with Delta Objects

When using the Configuration Platform SDK to change attribute values of a configuration object, it is important to understand how "delta structures" work.

A delta structure contains values for each attribute in the configuration object. When a change is requested, a delta object is created that contains values for each attribute. Delta values are initialized to either zero (for integer values) or a null string - defaults that indicate no change should be made for that attribute. To change attributes of a configuration object, you first set the delta value for that attribute and then send the request to Configuration Server to be processed. Only attribute values that are changing should be specified in the delta structure for that object.

Any attributes with a delta value set to zero are left unchanged, so there are two special cases to remember when updating integer values in a configuration object:

- leaving the integer as 0 (zero) means that attribute does not change;
- setting a delta value to the current value of the configuration object attribute will change that attribute value to zero.

For example, if an Agent skill level is currently set to 5, then the following table illustrates the effect of various delta structure values:

Initial Attribute Value	**Delta Structure Value**	**Updated Attribute Value**	**Comment**
5	3	3	Setting the delta structure value to a non-zero integer will change the attribute to that value.
5	0	5	Leaving the delta structure value as zero will leave the attribute unchanged.
5	5	0	Setting the delta structure value to the current attribute value will change the attribute to zero.

Requests sent by SOAP clients and formed in an XML format do not use delta structures, because these types of request do not require all attributes to be present. The COM application block (which is shipped with the Platform SDKs) provides a "non-delta" interface and uses delta objects internally to help users update objects, as shown in the following code snippet:

```
//retrieve an agent that has a single skill, with skill level set to 5
CfgPersonQuery query = new CfgPersonQuery();
query.setUserName("userName");
CfgPerson person = confService.retrieveObject(CfgPerson.class, query);

//Setting the skill level to 5 again will NOT result in a change in skill level  (ie: it will
remain 5).
((List<CfgSkillLevel>)person.getAgentInfo().getSkillLevels()).get(0).setLevel(5);
person.save();
```

```
//Setting the skill level to 0 will actually change the current skill level value.
((List<CfgSkillLevel>)person.getAgentInfo().getSkillLevels()).get(0).setLevel(0);
person.save();
```

To aid you in working with delta objects, Configuration SDK provides the `ConfDeltaUtility` helper class, which contains two public methods:

- `public ConfObjectDelta createDelta(ConfObject actualObj, ConfObject changedObj);`
- `public void applyDelta(ConfObject theObject, ConfObjectDelta delta);`

An instance of this helper class can be created using the actual configuration metadata from an open Configuration Server protocol connection:

```
ConfDeltaUtility deltaUtil = new
ConfDeltaUtility(csProtocol.getServerContext().getMetadata());
```

`ConfObject` objects are cloneable, so it is possible to use the delta utility in the following manner:

```
ConfObject someObject = ...; // get some object from config server
ConfObject objClone = (ConfObject) someObject.clone(); // create a copy

objClone.setPropertyValue("name", "ConfObject-new-name"); // change some property(-ies)
objClone.set...(...);

ConfObjectDelta delta = deltaUtil.createDelta(someObject, objClone);
// use 'delta' to update the object on config server with RequestUpdateObject
```

.NET

Setting Up a ConfServerProtocol Object

The first thing you need to do to use the Configuration Platform SDK is instantiate a `ConfServerProtocol` object. To do that, you must supply information about the Configuration Server you want to connect with. This example uses the URI of the Configuration Server, but you can also use the server's name, host, and port information:

```
ConfServerProtocol confServerProtocol =
        new ConfServerProtocol(
                new Endpoint(
                        confServerUri));
```

Configuration Server needs some additional information in order to create a successful connection. This information includes the type of client you wish to create, your client's name, and your user name and password:

```
confServerProtocol.ClientApplicationType = (int) CfgAppType.CFGSCE;
confServerProtocol.ClientName = clientName;
confServerProtocol.UserName = userName;
confServerProtocol.UserPassword = password;
```

After instantiating the `ConfServerProtocol` object, you need to open the connection to the Configuration Server:

```
confServerProtocol.Open();
```

Creating a Query

Now that you have opened a connection, you can create a query and send it to Configuration Server. Starting with release 8.1.4, there are two types of queries supported:

- Filter-based Queries (using `RequestReadObjects`)
- XPath-based Queries (using `RequestReadObjects2`)

If the request is successful, you will receive an `EventObjectsRead` message with the matching data.

Tip

When you send a RequestReadObjects message, Configuration Server may send more than one `EventObjectsRead` messages in response, depending on whether there is too much data to be handled by a single `EventObjectsRead`. Once you have received all of the `EventObjectsRead` messages, Configuration Server will also send an `EventObjectsSent`, which confirms that it has completed your request. For more information, refer to the article on event handling.

Examples of both query types are shown below, showing how you could retrieve information about a particular agent.

Filter-based Queries

For this type of query, you will need to supply the agent's user name using a *filter key*. The filter key tells Configuration Server to narrow your query to a specific agent, rather than retrieving information about all of the persons in your contact center:

```
KeyValueCollection filterKey = new KeyValueCollection();
filterKey.Add("user_name", userName);
```

You can find the names of the filter keys for Person objects by looking in the *Filter Keys* section of the CfgPerson entry.

Tip

A similar reference page is available for each Configuration Layer object.

Now you are ready to create the request. For filter-based queries, this is done using

`RequestReadObjects`.

As you may know, Configuration Server considers agents to be objects of type CfgPerson. So you will need to create a request for information about a `Person` who has the user name you specified in the filter key:

```
RequestReadObjects requestFilterQuery =
        RequestReadObjects.Create(
                (int) CfgObjectType.CFGPerson,
                filterKey);
```

Important

While the Configuration Layer supports the full character set in defining object names, using certain characters can cause problems in the behavior of some Genesys applications. Avoid using spaces, dashes, periods, or special characters in object names. Consider using underscores where you might normally use spaces or dashes.

After you have created your request, you can send it to Configuration Server, as shown here:

```
confServerProtocol.Send(requestFilterQuery);
```

XPath-based Queries

Submitting XPath-based queries is similar to filter-based queries, but does not require any filters or additional objects - instead an XPath search expression is passed to `RequestReadObjects2` as a string.

As in the example above, Configuration Server considers agents to be objects of type CfgPerson. So you will need to create a request for information about a `Person` who has the user name you are looking for:

```
RequestReadObjects2 requestXpathQuery =
        RequestReadObjects2.Create(
                (int) CfgObjectType.CFGPerson,
                "CfgPerson[@firstName='John']");
```

Important

While the Configuration Layer supports the full character set in defining object names, using certain characters can cause problems in the behavior of some Genesys applications. Avoid using spaces, dashes, periods, or special characters in object names. Consider using underscores where you might normally use spaces or dashes.

After you have created your request, you can send it to Configuration Server, as shown here:

```
confServerProtocol.Send(requestXpathQuery);
```

Interpreting the Response

The information you asked for is returned in the `ConfObject` property of the `EventObjectsRead` message.

Here is a sample of how you might print the XML document:

```
EventObjectsRead objectsRead = theMessage;

StringBuilder xmlAsText = new StringBuilder();
XmlWriterSettings xmlSettings = new XmlWriterSettings();
xmlSettings.Indent = true;

using (XmlWriter xmlWriter =
        XmlWriter.Create(xmlAsText, xmlSettings))
{
        XDocument resultDocument = objectsRead.ConfObject;
        resultDocument.WriteTo(xmlWriter);
}

Console.WriteLine("This is the response:\n"
                + xmlAsText.ToString() + "\n\n");
```

And this is what the XML document might look like:

```
 <ConfData>
   <CfgPerson>
     <DBID value="105"/>
     <tenantDBID value="101"/>
     <lastName value="agent1"/>
     <firstName value="Agent"/>
     <employeeID value="agent1"/>
     <userName value="agent1"/>
     <password value="204904E461002B28511D5880E1C36A0F"/>
     <isAgent value="2"/>
     <CfgAgentInfo>
       <placeDBID value="102"/>
       <skillLevels>
         <CfgSkillLevel>
           <skillDBID value="101"/>
           <level value="9"/>
         </CfgSkillLevel>
       </skillLevels>
       <agentLogins>
         <CfgAgentLoginInfo>
           <agentLoginDBID value="103"/>
           <wrapupTime value="0"/>
         </CfgAgentLoginInfo>
       </agentLogins>
       <capacityRuleDBID value="127"/>
     </CfgAgentInfo>
     <isAdmin value="1"/>
     <state value="1"/>
     <userProperties>
       <list_pair key="desktop-redial">
         <str_pair key="phone-number0" value="5551212"/>
         <str_pair key="phone-number1" value=""/>
         <str_pair key="phone-number2" value=""/>
         <str_pair key="phone-number3" value=""/>
         <str_pair key="phone-number4" value=""/>
```

```
        <str_pair key="phone-number5" value=""/>
        <str_pair key="phone-number6" value=""/>
        <str_pair key="phone-number7" value=""/>
        <str_pair key="phone-number8" value=""/>
        <str_pair key="phone-number9" value=""/>
      </list_pair>
      <list_pair key="multimedia">
        <str_pair key="last-media-logged"
            value="voice,email"/>
      </list_pair>
    </userProperties>
    <emailAddress value="agent1@techpubs3"/>
  </CfgPerson>
</ConfData>
```

This XML document contains information about a Person. To interpret the information contained in the document, look at the Parameters section of the CfgPerson entry in the list of Configuration Objects.

If you compare the elements in this XML document to the CfgPerson entry, you can see that some of them contain information that is explained in detail in another entry. For example, the `CfgAgentInfo` element contains information that is described in the CfgAgentInfo entry. Similarly, the `CfgAgentLoginInfo` element contains information described in the CfgAgentLoginInfo entry.

Updating an Object

You can update a Configuration Layer object by passing in an XML document (of type `XDocument`) containing the appropriate information about that object:

```
RequestUpdateObject requestUpdateObject =
        RequestUpdateObject.Create(
                (int) CfgObjectType.CFGPerson,
                xDocument);
```

Creating a New Object

You can also create a new Configuration Layer object by sending an XML Document (of type `XDocument`) to Configuration Server, as shown here:

```
RequestCreateObject requestCreateObject =
        RequestCreateObject.Create(
                (int) CfgObjectType.CFGPerson,
                xDocument);
```

Closing the Connection

Finally, when you are finished communicating with the Configuration Server, you should close the connection, in order to minimize resource utilization:

```
confServerProtocol.Close();
```

Working with Delta Objects

When using the Configuration Platform SDK to change attribute values of a configuration object, it is important to understand how "delta structures" work.

A delta structure contains values for each attribute in the configuration object. When a change is requested, a delta object is created that contains values for each attribute. Delta values are initialized to either zero (for integer values) or a null string - defaults that indicate no change should be made for that attribute. To change attributes of a configuration object, you first set the delta value for that attribute and then send the request to Configuration Server to be processed. Only attribute values that are changing should be specified in the delta structure for that object.

Any attributes with a delta value set to zero are left unchanged, so there are two special cases to remember when updating integer values in a configuration object:

- leaving the integer as 0 (zero) means that attribute does not change;
- setting a delta value to the current value of the configuration object attribute will change that attribute value to zero.

For example, if an Agent skill level is currently set to 5, then the following table illustrates the effect of various delta structure values:

Initial Attribute Value	Delta Structure Value	Updated Attribute Value	Comment
5	3	3	Setting the delta structure value to a non-zero integer will change the attribute to that value.
5	0	5	Leaving the delta structure value as zero will leave the attribute unchanged.
5	5	0	Setting the delta structure value to the current attribute value will change the attribute to zero.

Note that requests sent by SOAP clients and formed in an XML format do not use delta structures, because these types of request do not require all attributes to be present. The COM application block (which is shipped with the Platform SDKs) provides a "non-delta" interface and uses delta objects internally to help users update objects, as shown in the following code snippet:

```
//retrieve a particular agent whose last name is "Jones"
CfgPersonQuery query = new CfgPersonQuery();
query.UserName = "userName";
query.LastName = "Jones";
CfgPerson person = myConfService.RetrieveObject<CfgPerson>(query);

//Setting the last name to the same value will NOT result in a change
person.LastName = "Jones";
person.Save();
```

```
//Setting the last name to a different value will change the actual value
person.LastName = "Smith";
person.Save();
```

Connecting Using UTF-8 Character Encoding

Java

Scenarios

Genesys Configuration Server 8.1.2 added the ability to be configured to support multiple languages at a same time using UTF-8 encoding. Once Configuration Server is installed, configured and started in multilingual (UTF-8) mode it cannot be switched to regular mode. If Configuration Server is installed and started in normal mode, then it cannot be switched to multilingual (UTF-8) mode later.

One known issue is that the UTF-enabled protocol breaks backward compatibility, so users must add their own code for connection reconfiguration. The following samples describe connection scenarios with Platform SDK:

Scenario 1

Configuration Server is release 8.1.2 or later and is NOT configured as multilingual (without UTF-8 transport), or is an earlier version without support for the UTF-8 feature.

In this scenario, Platform SDK connections can be created in the usual way.

Scenario 2

Configuration Server is release 8.1.2 or later and configured as multilingual (with UTF-8 transport), with:

A) Platform SDK release 8.1.3 in use.

Reconfiguration for encoding is automatically handled by Platform SDK as described in the section below – no user action is required.

B) Platform SDK release 8.1.1 or 8.1.2 in use.

Platform SDK provides information that Configuration Server is UTF-8, so, the connection can be reopened using new connection configuration with following user code.

```
PropertyConfiguration config = new PropertyConfiguration();
config.setUseAddp(true);
config.setAddpClientTimeout(11);
config.setAddpServerTimeout(21);

ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint(name, host, port, config));
```

```
protocol.setClientName(clientName);
protocol.setClientApplicationType(clientType.ordinal());
protocol.setUserName(username);
protocol.setUserPassword(password);

protocol.open();

Integer cfgServerEncoding = protocol.getServerContext().getServerEncoding();
if (cfgServerEncoding != null && cfgServerEncoding.intValue() == 1) {
   protocol.close();
   config.setStringsEncoding("UTF-8");
   protocol.setEndpoint(new Endpoint(name, host, port, config));
   protocol.open();
 }
```

It may be more comfortable to move the flag value evaluation to a separated method where a temporary `ConfServerProtocol` instance may be created - especially in the case of `ChannelListeners` usage, messages handlers, etc.

Important

This is not the best solution for wide usage. The `ServerEncoding` value evaluation method may fail if non-ASCII symbols are found inside the username or password, which may lead to a handshake procedure error such as "invalid username/password". This issue may be resolved with an additional test connection retry with UTF-8 enabled, but this workaround is not a best practice solution.

C) Platform SDK release 8.0.1 through 8.1.1 in use:

Platform SDK does NOT indicate whether Configuration Server is using UTF-8 mode or not, so user application should take care to evaluate this information (or have it defined by the design or configuration of the application).

In this case we have no `protocol.getServerContext().getServerEncoding()`, but we are able to configure the connection for Unicode usage.

It may be recommended to add one more property to the application configuration/parameters (along with the existing Configuration Server host and port) such as a boolean "isCSUTF8" value.

```
PropertyConfiguration config = new PropertyConfiguration();
if (isCSUTF8) {
   config.setOption(Connection.STR_ATTR_ENCODING_NAME_KEY, "UTF-8");
}

ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint(name, host, port, config));
protocol.setClientName(clientName);
protocol.setClientApplicationType(clientType.ordinal());
protocol.setUserName(username);
protocol.setUserPassword(password);

protocol.open();
```

D) Platform SDK release of 8.0.0 or earlier in use.

No support if provided for string encoding of connection configuration options. The only way is to use this feature is to upgrade your release of Platform SDK.

Automatic UTF-8 Character Encoding Set Up on Handshake

Starting in Platform SDK Release 8.1.3 (which incorporates Configuration Protocol Release 3.79), support for UTF-8 encoding can be automatically detected.

The process for this features is described here:

1. The first handshake message, `EventProtocolVersion`, now includes the extra `ServerEncoding` attribute. If this attribute is 1 then Platform SDK updates string encoding for that connection to the server as UTF-8.
2. The next message from the client requests authentication from the server. These messages (`RequestRegisterClient` or `RequestRegisterClient2`) have been expanded with the `ClientEncoding` attribute, which must have the same value as the `ServerEncoding` attribute received previously.
3. After the handshake is complete, string encoding for this channel may be different from the string encoding specified in the original configuration parameters. You can access the current value through `Endpoint.GetConfiguration()` of the `ConfServerProtocol` instance.

.NET

Scenarios

Genesys Configuration Server 8.1.2 added the ability to be configured to support multiple languages at a same time using UTF-8 encoding. Once Configuration Server is installed, configured and started in multilingual (UTF-8) mode it cannot be switched to regular mode. If Configuration Server is installed and started in normal mode, then it cannot be switched to multilingual (UTF-8) mode later.

One known issue is that the UTF-enabled protocol breaks backward compatibility, so users must add their own code for connection reconfiguration. The following samples describe connection scenarios with Platform SDK:

Scenario 1

Configuration Server is release 8.1.2 or later and is NOT configured as multilingual (without UTF-8 transport), or is an earlier version without support for the UTF-8 feature.

In this scenario, Platform SDK connections can be created in the usual way.

Scenario 2

Configuration Server is release 8.1.2 or later and configured as multilingual (with UTF-8 transport), with:

A) Platform SDK release 8.1.3 in use.

Reconfiguration for encoding is automatically handled by Platform SDK as described in the section below – no user action is required.

B) Platform SDK release 8.1.1 or 8.1.2 in use.

Platform SDK provides information that Configuration Server is UTF-8, so, the connection can be reopened using new connection configuration with following user code.

```
PropertyConfiguration config = new PropertyConfiguration();
config.UseAddp = true;
config.AddpClientTimeout = 11;
config.AddpServerTimeout = 21;

ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint(_name, _host, _port,
config));
protocol.ClientName = _clientName;
protocol.ClientApplicationType = _clientType;
protocol.UserName = _userName;
protocol.UserPassword = _password;

protocol.Open();

int? cfgServerEncoding = protocol.Context.ServerEncoding;
if (cfgServerEncoding != null && cfgServerEncoding.Value == 1)
{
  protocol.Close();
  config.StringsEncoding = "UTF-8";
  protocol.Endpoint = new Endpoint(_name, _host, _port, config);
  protocol.Open();
}
```

It may be more comfortable to move the flag value evaluation to a separated method where a temporary `ConfServerProtocol` instance may be created - especially in the case of `ChannelListeners` usage, messages handlers, etc.

Important

This is not the best solution for wide usage. The `ServerEncoding` value evaluation method may fail if non-ASCII symbols are found inside the username or password, which may lead to a handshake procedure error such as "invalid username/password". This issue may be resolved with an additional test connection retry with UTF-8 enabled, but this workaround is not a best practice solution.

C) Platform SDK release 8.0.1 through 8.1.1 in use:

Platform SDK does NOT indicate whether Configuration Server is using UTF-8 mode or not, so user application should take care to evaluate this information (or have it defined by the design or configuration of the application).

In this case we have no `protocol.Context.ServerEncoding` property, but we are able to configure the connection for Unicode usage.

It may be recommended to add one more property to the application configuration/parameters (along with the existing Configuration Server host and port) such as a boolean "isCSUTF8" value.

```
PropertyConfiguration config = new PropertyConfiguration();
if (_isCsutf8)
{
  config.SetOption(CommonConnection.StringAttributeEncodingKey, "UTF-8");
}

ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint(_name, _host, _port,
config));
protocol.ClientName = _clientName;
protocol.ClientApplicationType = _clientType;
protocol.UserName = _userName;
protocol.UserPassword = _password;

protocol.Open();
```

D) Platform SDK release of 8.0.0 or earlier in use.

No support if provided for string encoding of connection configuration options. The only way is to use this feature is to upgrade your release of Platform SDK.

Automatic UTF-8 Character Encoding Set Up on Handshake

Starting in Platform SDK Release 8.1.3 (which incorporates Configuration Protocol Release 3.79), support for UTF-8 encoding can be automatically detected.

The process for this features is described here:

1. The first handshake message, `EventProtocolVersion`, now includes the extra `ServerEncoding` attribute. If this attribute is 1 then Platform SDK updates string encoding for that connection to the server as UTF-8.
2. The next message from the client requests authentication from the server. These messages (`RequestRegisterClient` or `RequestRegisterClient2`) have been expanded with the `ClientEncoding` attribute, which must have the same value as the `ServerEncoding` attribute received previously.
3. After the handshake is complete, string encoding for this channel may be different from the string encoding specified in the original configuration parameters. You can access the current value through `Endpoint.GetConfiguration()` of the `ConfServerProtocol` instance.

Change Password On Next Login

This page shows examples of how a Configuration Server connection can be opened, using either a standard login or with a forced password change on login.

Java

Scenario: Standard Login

The following code shows a standard login process, without this feature enabled:

```
ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint("cfgsrv", csHost, csPort));
protocol.setClientName(clientAppName);
protocol.setClientApplicationType(clientAppType.ordinal());
protocol.setUserName(userName);
protocol.setUserPassword(userPasswd);
protocol.open();
```

Scenario: Change Password on Next Login

When the user has enabled the Change Password on Next Login feature, `protocol.open()` throws `ChangePasswordException`. This exception should be caught and handled to force the password update.

So the resulting code may look like:

```
ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint("cfgsrv", csHost, csPort));
protocol.setClientName(clientAppName);
protocol.setClientApplicationType(clientAppType.ordinal());
protocol.setUserName(userName);
protocol.setUserPassword(userPasswd);
try {
  protocol.open();
} catch (ChangePasswordException e) {
  String newPasswd = ...; // obtain new user password
  protocol.useChangePasswordRegistration(newPasswd);
  protocol.open();
}
```

After a successful open procedure, the new password value will be accepted and `protocol.getUserPassword()` will be set to the newPasswd value that was specified during login.

.NET

Scenario: Standard Login

The following code shows a standard login process, without this feature enabled:

```
ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint("cfgsrv", csHost, csPort));
protocol.ClientName = clientAppName;
protocol.ClientApplicationType = clientAppType;
protocol.UserName = userName;
protocol.UserPassword = userPassword;
protocol.Open();
```

Scenario: Change Password on Next Login

When the user has enabled the Change Password on Next Login feature, `protocol.open()` throws `ChangePasswordException`. This exception should be caught and handled to force the password update.

So the resulting code may look like:

```
ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint("cfgsrv", csHost, csPort));
protocol.ClientName = clientAppName;
protocol.ClientApplicationType = clientAppType;
protocol.UserName = userName;
protocol.UserPassword = userPassword;
try
{
  protocol.Open();
}catch(ChangePasswordException)
{
  string newPassword= ...; // obtain new user password
  protocol.UseChangePasswordRegistration(newPassword);
  protocol.Open();
}
```

After a successful open procedure, the new password value will be accepted and `protocol.UserPassword` will be set to the newPasswd value that was specified during login.

Getting the Last Login Info

Java

Tip

The appropriate Configuration Server version is required to use this feature, and so is the correct security configuration. For details, refer to Chapter 11 (Last Logged In Display) in Genesys 8.0 Security Deployment Guide.

Configuration Server provides last login information during the user authentication (handshake) procedure, and the Platform SDK Configuration Protocol provides it "as-is" in the form of a `KeyValueCollection`:

- `ConfServerProtocol.getServerContext().getLastLoginInfo()`

An example of the resulting `KeyValueCollection` could look like:

```
KVList:
   'LAST_LOGIN_PERSON' [int] = 100
   'LAST_LOGIN_TIME' [int] = 1259161588
```

Tip

This information is only available while the connection is opened.

Note that "last login" is configured on Configuration Server through the `confserv.cfg` file:

```
[confserv]
...
last-login = true
last-login-synchronization = true
```

Platform SDK obtains the information using the `EventClientRegister` message:

```
2012-08-21 10:05:49,306 [New I/O client worker #4-4] DEBUG ns.protocol.DuplexChannel null -
Handling message:  'EventClientRegistered' (19) attributes:
        IATRCFG_SESSIONNUMBER [int] = 22
        IATRCFG_CFGSERVERDBID [int] = 99
        SATRCFG_PROTOCOL [str] = "CfgProtocol 5.1.3.54"
        IATRCFG_EXTERNALAUTH [int] = 0
        SATRCFG_PARAMETERS [KvListString] = KVList:
'LAST_LOGIN_PERSON' [int] = 1227
'LAST_LOGIN_TIME' [int] = 1345532749
'LAST_LOGIN_APPLICATION' [str] = "PSDK_CFGSCI"
```

```
        IATRCFG_BACKUPCFGSERVERDBID [int] = 0
        IATRCFG_UNSOLEVENTNUM [int] = 73770
        IATRCFG_CRYPTPASSW [int] = 1
        SATRCFG_SCHEMAVERSION [str] = "8.1.100.05"
        IATRCFG_REQUESTID [int] = 6
        SATRCFG_PROTOCOLEX [str] = "CfgProtocol 5.1.3.77"
```

There are two methods available in Platform SDK for retrieving last login details:

- `protocol.getServerContext().getLastLoginInfo()`
- `protocol.getServerContext().getCfgLastLogin()` (deprecated, not recommended for use)

If these methods return null, then you need to check whether Configuration Server gave the required info by looking in the debug logs for either Platform SDK or Configuration Server.

.NET

Tip

The appropriate Configuration Server version is required to use this feature, and so is the correct security configuration. For details, refer to Chapter 11 (Last Logged In Display) in Genesys 8.0 Security Deployment Guide.

Configuration Server provides last login information during the user authentication (handshake) procedure, and the Platform SDK Configuration Protocol provides it "as-is" in the form of a `KeyValueCollection`:

- `protocol.Context.LastLoginInfo`

An example of values from the `KeyValueCollection` could look like:

```
KVList:
   'LAST_LOGIN_PERSON' [int] = 100
   'LAST_LOGIN_TIME' [int] = 1259161588
```

Tip

This information is only available while the connection is opened.

Note that "last login" is configured on Configuration Server through the `confserv.cfg` file:

```
[confserv]
...
last-login = true
last-login-synchronization = true
```

Platform SDK obtains the information using the `EventClientRegister` message:

```
2012-08-21 10:05:49,306 [New I/O client worker #4-4] DEBUG ns.protocol.DuplexChannel null -
Handling message:  'EventClientRegistered' (19) attributes:
        IATRCFG_SESSIONNUMBER [int] = 22
        IATRCFG_CFGSERVERDBID [int] = 99
        SATRCFG_PROTOCOL [str] = "CfgProtocol 5.1.3.54"
        IATRCFG_EXTERNALAUTH [int] = 0
        SATRCFG_PARAMETERS [KvListString] = KVList:
'LAST_LOGIN_PERSON' [int] = 1227
'LAST_LOGIN_TIME' [int] = 1345532749
'LAST_LOGIN_APPLICATION' [str] = "PSDK_CFGSCI"
        IATRCFG_BACKUPCFGSERVERDBID [int] = 0
        IATRCFG_UNSOLEVENTNUM [int] = 73770
        IATRCFG_CRYPTPASSW [int] = 1
        SATRCFG_SCHEMAVERSION [str] = "8.1.100.05"
        IATRCFG_REQUESTID [int] = 6
        SATRCFG_PROTOCOLEX [str] = "CfgProtocol 5.1.3.77"
```

There are two properties available in Platform SDK for retrieving last login details, shown below with related code snippets:

- `protocol.Context.LastLoginInfo`

```
ConfServerProtocol protocol = new ConfServerProtocol(new Endpoint(_name, _host, _port));
protocol.ClientName = _clientName;
protocol.ClientApplicationType = _clientType;
protocol.UserName = _userName;
protocol.UserPassword = _password;
protocol.Open();
if (protocol.Context.LastLoginInfo!=null)
{
  object  lastLoginPerson = protocol.Context.LastLoginInfo["LAST_LOGIN_PERSON"];
  object lastLoginTime = protocol.Context.LastLoginInfo["LAST_LOGIN_TIME"];
  // TODO ... use obtained data ...
}
```

- `protocol.Context.CfgLastLogin` (deprecated, not recommended for use)

```
 protocol.ClientName = _clientName;
 protocol.ClientApplicationType = _clientType;
 protocol.UserName = _userName;
 protocol.UserPassword = _password;
 protocol.Open();
 if (protocol.Context.CfgLastLogin != null)
 {
   // TODO parse XDocument to obtain data
 }
```

If these methods return null, then you need to check whether Configuration Server gave the required info by looking in the debug logs for either Platform SDK or Configuration Server.

Using the Configuration Object Model Application Block

Important

This application block is a reusable production-quality component. It has been designed using industry best practices and provided with source code so it can be used "as is," extended, or tailored if you need to. Please see the License Agreement for details.

The Configuration Object Model Application Block provides developers with a consistent and intuitive object model for working with Configuration Server objects.

Java

Architecture and Design

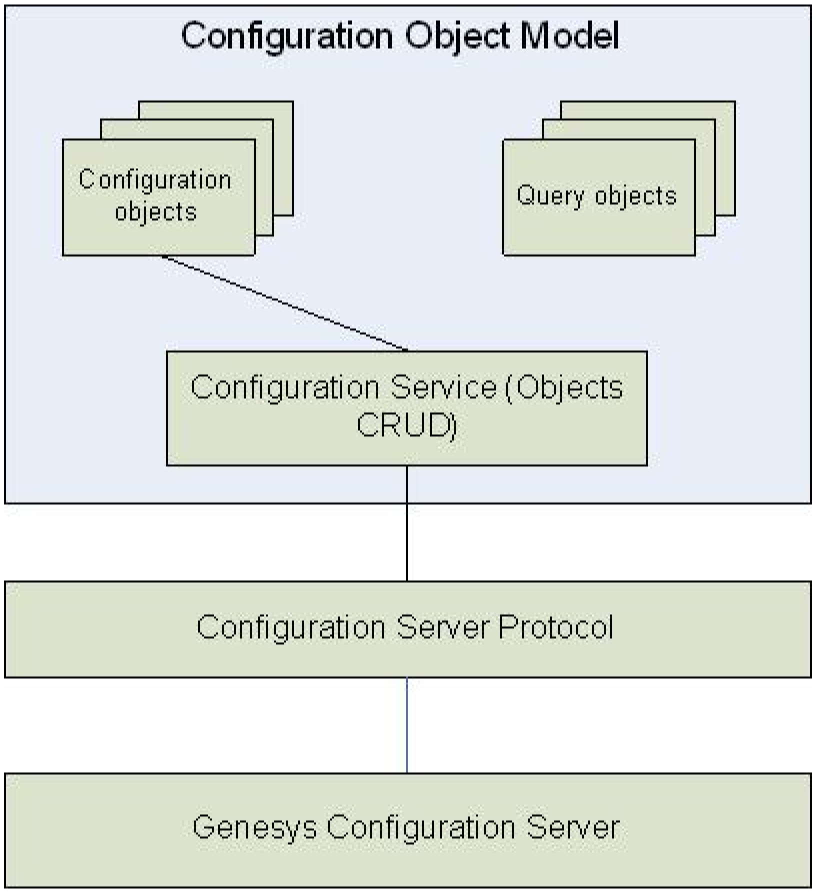
The Configuration Platform SDK allows you to work with objects in the Genesys Configuration Layer by using the interface provided by Configuration Server. Unfortunately, this interface can be difficult to work with. For example, in order to update or create Configuration Layer objects, you have to use special “delta” objects that are distinct from the objects used to retrieve information about Configuration Layer objects.

The Configuration Object Model Application Block provides a consistent and intuitive object model that hides many of the complexities involved in working with Configuration Layer objects. This object model is implemented by way of an event subscription/delivery model, which hides key-value details of the current protocol, and is integrated with the rest of the object model.

The architecture of the Configuration Object Model Application Block consists of three functional components:

- Configuration Objects
- Configuration Service
- Query Objects
- Cache Objects

These components are shown below.


Configuration Object Model
Configuration objects
Query objects
Configuration Service (Objects CRUD)
Configuration Server Protocol
Genesys Configuration Server

Configuration Objects

Classes and Structures

The Configuration Object Model Application Block supports two types of configuration objects:

- Classes, which can be retrieved directly from Configuration Server using queries.
- Structures, which only exist as properties of classes, and cannot be retrieved directly from Configuration Server.

Classes and structures are different in many ways, but in order to determine whether a given object is a class or a structure, all you need to do is check to see whether the object has a “DBID” property. Classes have this property, while structures do not.

Classes and structures are also different in the following ways:

- Each structure is a property of another class or structure, and therefore must have a “parent” class.
- Classes can be changed and saved to the Configuration Server and structures can only be saved through their “parent” classes.
- Clients can subscribe to events on changes in a class, but not in a structure. To retrieve events on

changes in a structure, clients have to subscribe to changes in its parent class.

Property Types

Both classes and structures have properties. Each property has its own getter and setter methods, and each property is an instance of one of the following types:

- *Simple* — A property that is represented by a value type. Configuration Server supports two types of simple properties - `string` and `integer`. For example, the `CfgPerson` object has `FirstName` and `LastName` properties, both of the `string` type.
- *KV-list* — Tree-like properties that are represented by the `KeyValueCollection` class in the Configuration Object Model. Examples of this property include `userProperties` and `CfgPerson`.
- *Structure* — A complex property that includes one or more properties. In the Configuration Object Model, structures are represented by instances of classes that are similar to configuration objects, but cannot be created directly. For example, in the `CfgPerson` class, its `AgentInfo` property contains *simple*, *kv-list* and other property types.
- *List of structures* — A property that represents more than one structure. In Configuration Object Model, lists of structures are represented by a generic type `IList<structure_type>`, so that the collection is typed, and clients can easily iterate through the collection.
- *Links to a single object* — In Configuration Server, these properties are stored as DBIDs of external objects. The Configuration Object Model automatically resolves these DBIDs into the real objects, which can be manipulated in the same way as the objects directly retrieved from Configuration Server. Links are initialized at the time of the initial request to one of its properties.

> **Tip**
> For each link, there are two ways to set the new value of a link. There is a setter method of the property, which uses an object reference to set a new value of a link. There is also a `Set...DBID` method, which uses an integer DBID value.

- *Links to multiple objects* — A property that contains more than one link. In the Configuration Object Model, lists of structures are represented by a generic type `IList<class_type>`, so that the collection is typed, and clients can easily iterate through the collection.

Creating Instances

One way to create an instance of an object in the Configuration Object Model is to invoke a `Retrieve...` method of a `ConfService` class. This set of methods returns instances of objects that already exist in Configuration Server.

To create a new object in Configuration Server, a client must create a new instance of a COM or "*detached*" object. The detached object does not correspond to any objects in Configuration Server until it is saved. The detached object is created using the regular Object-Oriented language object instantiation. For example, a new detached `CfgPerson` object is created using the following construction:

```
[Java]

CfgPerson person = new CfgPerson(confService);
```

An object instance can also be created by using links to external objects. The Component Object Model creates a new object instance whenever the link is called, or any of the properties of a linked object are called. For example, you can write:

```
[Java]

// person has already been retrieved from Configuration Server.
CfgTenant tenant = person.getTenant(); // this is a link to an external object. It is
initialized internally right now
CfgAddress address = tenant.getAddress();
```

Common Methods

Each configuration class contains the following methods:

- Generic `GetProperty(string propertyName)` — Retrieves the property value by its name.
- Generic `SetProperty(string propertyName)` — Sets the new value of the property by its name.
- `Save()` — Commits all changes previously made to the object to Configuration Server. If the object was created detached from Configuration Server and has never been saved, a new object is created in Configuration Server using the `RequestCreateObject` method. If the object has been saved or has been retrieved from Configuration Server, a delta-object, which contains all changes to the object, is formed and sent to Configuration Server by means of the `RequestUpdateObject` method.
- `Delete()` — Deletes the object from the Configuration Server Database.
- `Refresh()` — Retrieves the latest version of the object and refreshes the value of all its properties.

Tip

In this release, all configuration objects are “static,” which means that if the object changes in the Configuration Server, the instance of a class is not automatically changed in the Configuration Object Model. Clients must subscribe to the corresponding event and manually refresh the COM object in order for these changes to take effect.

Configuration Service

Tip

The `IConfService` interface was added to COM in release 8.0. All applications should now use this interface to work with the configuration service instead of the old `ConfService` class. This change is an example of how all COM types in the interface are now referred to by interface; for instance, if a method previously returned `CfgObject` it now returns `ICfgObject`. This is not compatible with existing code, but upgrading should not be difficult as the new interfaces support the same methods as the implementing types.

The Configuration Service (`IConfService`) interface provides services such as retrieval of objects and subscription to events from Configuration Server. Each connection to a Configuration Server (represented by a `ConfServerProtocol` class of Platform SDK) requires its own instance of the `IConfService` interface.

The protocol class should be created and initialized in the client code prior to `IConfService` initialization.

The `ConfServiceFactory` class is used to create the `IConfService`. This class uses the following syntax:

```
[Java]

IConfService service = ConfServiceFactory.CreateConfService(protocol);
```

Retrieving Objects

Objects can be retrieved from Configuration Service by using one of the following methods:

- `RetrieveObject` — Accepts a query that returns one object. If multiple objects are returned, an exception is thrown.
- `RetrieveMultipleObjects` — Accepts a query that returns one or more objects. A collection of objects is returned.

Each of the `Retrieve...` methods can be either specific (by using generic criteria entries, an object of a specified type is returned) or general (a general object is returned).

Handling Events

The following methods must be called before receiving events from Configuration Server:

1. Register - The application must register its callback by calling the `Register` method from the Configuration Service. This method supplies the client’s filter, which enables the client to receive only requested events.
2. Subscribe - The application must subscribe to events from Configuration Server by calling the `Subscribe` method from the Configuration Service. This method provides a notification query object as a parameter.

   The `NotificationQuery` object determines whether the object (or set of objects) to which the client wants to subscribe has changed. The `NotificationQuery` object contains such parameters as `object type, object DBID` and `tenant DBID`.

   After calling the `Subscribe` method, Configuration Server starts sending events to the client. These events are objects, which contain information such as:

   - which object (ID and type) is affected
   - the type of event sent to the client
   - any additional information

   There are three types of events that the client might receive:

   - `ObjectCreated` — A new object has been added to Configuration Server.
   - `ObjectChanged` — Some of the object properties have been modified in Configuration Server.
   - `ObjectDeleted` — The object has been removed from Configuration Server.

Releasing a Configuration Service

Whenever a `ConfService` instance is no longer needed, the `ReleaseConfService` method can be used to remove it from the internal list.

```
[Java]

ConfServiceFactory.ReleaseConfService(service);
```

Query Objects

A query object is an instance of a class that contains information required for a successful query to a Configuration Server. This information includes an object type and its attributes (such as *name* and *tenant*), which are used in the search process.

The inheritance structure of configuration server queries is designed to allow for future expansion. The `CfgQuery` object is the base class for all query objects. Other classes extend `CfgQuery` to provide more specific functionality for different types of queries - for example, all filter-based queries use the `CfgFilterBasedQuery` class. This allows room for future query types (such as XPath) to implemented in this Application Block.

A list of currently available query types is provided below:

- CfgFilterBasedQuery — Contains mapped attribute name-value pairs, as well as the object type.

A special query class is supplied for each configuration object type, in order to facilitate the process of making queries to Configuration Server. For each searchable attribute, the query class has a property that can be set. All of these classes inherit attributes from the `CfgQuery` object, and can be supplied as parameters to the Retrieve... methods which are used to perform searches in Configuration Server.

Cache Objects

The cache functionality is intended to enhance the Configuration Object Model by allowing configuration objects to be stored locally, thereby minimizing requests to configuration server, as well as enhancing ease of use by providing automatic synchronization between locally stored objects and their server-side counterparts.

The cache functionality was designed with the following principles in mind:

- The cache functionality is designed to be extendable with custom implementations of provided interfaces and not via inheritance.
- The cache component is not designed to replicate the Configuration Server query engine or other Configuration Server functionality on the client side.
- Caching must be an optional feature. Work with Configuration Server should not be affected if caching is not used.

Use Cases

Analysis of use cases provides insight into the requirements for applications likely to require configuration cache functionality. The use cases described in the following table were selected for analysis in order to highlight different functional requirements. There are several possible actors which are referenced in the use cases. The actors are as follows:

- Application - Any application which uses the Configuration Object Model application block
- User - Human (or software) user who may perform actions upon objects in the configuration which are separate from the Application

Use Case	Description	Actor	Steps
PLACE OBJECT INTO CACHE	Place a configuration object into the configuration cache (note the object must have been saved — ie must have a DBID in order to exist in the cache).	Application	1. Application adds object to the cache
PLACE OBJECT INTO CACHE ON SAVE	Place a newly created configuration object into the configuration cache when it is saved.	Application	1. Application creates object 2. Application saves object 3. Configuration Object Model Application Block adds object to the cache
PLACE OBJECT INTO CACHE ON RETRIEVE	Allow for automatic insertion of configuration objects into the cache upon retrieval from configuration server.	Application	1. Application retrieves configuration object 2. Configuration Object Model Application Block retrieves the configuration object from the server 3. Configuration Object Model Application Block places the configuration object into the cache 4. Configuration Object Model Application Block returns the object to the application
OBJECT REMOVED IN CONFIGURATION SERVER	When configuration objects are deleted in the configuration server, the cache can delete the local representation of the object as well.	User	1. User deletes object in the Configuration Server 2. Cache removes

Use Case	Description	Actor	Steps
			corresponding local object upon receiving delete notification 3. Cache sends notification of object deletion to Application
SYNCHRONIZE OBJECT PROPERTIES WITH CONFIGURATION SERVER	When an object stored in the cache is updated in the Configuration Server the object must be updated locally as well.	User	1. User updates a configuration object 2. Cache receives notification about object update 3. Cache updates the object based on the received delta 4. Cache fires event informing any subscribers of object change
FIND OBJECT IN CACHE	The cache must support the ability to find a specific configuration object in the cache using object DBID and type as the criteria for the search.	Application	1. Application retrieves object from cache. 2. If object is in the cache, the cache returns the object. Otherwise the application is notified that the requested object is not in the cache.
ACCESS CACHED OBJECTS	The cache must provide its full object collection to the application.	Application	1. Application requests a complete list of objects from the cache. 2. The cache returns a collection of all cached objects.
RETRIEVE LINKED OBJECT FROM CACHE	If caching is turned on, object links which the Configuration Object Model currently resolves through lazy	Application	1. Application accesses a property which requires link resolution

Use Case	Description	Actor	Steps
	initialization (i.e. if a property linking to another object is accessed, we retrieve the referred-to object from configuration server) must be resolvable through cache access.		2. Configuration Object Model Application Block retrieves the linked object from configuration server and stores it in the cache before returning to the application 3. Application again accesses the property and this time the Configuration Object Model Application Block retrieves the object from the cache
PROVIDE CACHE TRANSPARENCY ON RETRIEVE	A cache search should be performed on attempt to retrieve an object from Configuration Server. If the requested object is found in the cache then the Configuration Object Model should return the cached object rather than accessing Configuration Server.		1. Application creates query to retrieve configuration object 2. Application executes query using the Configuration Object Model 3. Configuration Object Model Application Block searches the cache • If object present, return the object • If object not present, query configuration server for the object
CACHE SERIALIZATION	The cache should support serialization.	Application	1. Application provides a stream to the cache 2. The cache serializes itself into the stream in an XML format 3. Application restarts 4. Application provides

Use Case	Description	Actor	Steps
			the cache a stream of cache data in the same XML format as in step 2 5. Cache restores itself 6. Cache subscribes for updates on the restored objects

Implementation Overview

Two new interfaces for cache management have been added to the Configuration Object Model: the `IConfCache` interface and a default cache implementation (`DefaultConfCache`). Note that the `ConfCache` also implements the Subscriber interface from `MessageBroker` so that the user can subscribe to notifications from Configuration Server, as discussed in *Notification And Delta Handling*.

The `IConfCache` interface provides methods for basic functionality such as adding, updating, retrieving, and removing objects in the cache. It also includes a `Policy` property that defines cache behavior and affects method implementation. (For more details about policies, see *Cache Policy*).

The `DefaultConfCache` component provides a default implementation of the `IConfCache` interface. It serializes and deserializes cache objects using the XML format described in the *XML Format* section, below.

To enable and configure caching functionality, and to specify `ConfService` policy, there are three `CreateConfService` methods available from `ConfServiceFactory`. The original `CreateConfService` method (not shown here) creates a `ConfService` instance that uses the default policy and does not use caching.

```
[Java]

public static IConfService createConfService(Protocol protocol, boolean enableCaching)
```

This method creates an instance of a Configuration Service based on the specified protocol. If caching is enabled, the default caching policy will be used. If `enableCaching` is set to true, caching functionality will be turned on. If caching is disabled, all policy flags related to caching will be false.

```
[Java]

public static IConfService createConfService(Protocol protocol,
        IConfServicePolicy confServicePolicy, IConfCache cache)
```

This method creates a configuration service with the specified policy information. The created service will have caching enabled if a cache object (implementing the `IConfCache` interface) is passed as a parameter.

```
[Java]

public static IConfService createConfService(Protocol protocol,
        IConfServicePolicy confServicePolicy, IConfCachePolicy confCachePolicy)
```

This method creates a configuration service with the specified policy information. The created service will have caching enabled by default with the cache using the specified cache policy.

XML Format

The "Cache" node will be the root of the configuration cache XML, while "ConfData" is a child of the "Cache" node. The ConfData node contains a collection of XML representations for each configuration object in the cache. The XML format of each object is identical to that which is returned by the `ToXml` method supported by each the Configuration Object Model configuration object.

The "CacheConfiguration" element is a child of the "Cache" node. There can only be one instance of this node and it contains all cache configuration parameters, as follows:

- CONFIGURATIONSERVER NODE — There can be 1..n instances of this element. Each one will represent a configuration server for which the cache is applicable (a cache can be applicable to multiple configuration servers if they are working with the same database as in the case of a primary and backup configuration server pair). Each ConfigurationServer element will have a `URI` attribute specifying the unique URI identifying the Configuration Server, as well as a `Name` attribute specifying the name associated with the endpoint.

The example provided below shows a cache that is applicable for the configuration server at "server:2020" with some policy details specified. There are two objects in the cache for this example: a `CfgDN` and a `CfgService` object.

```
[XML]

<Cache>
  <CacheConfiguration>
    <ConfigurationServer name="serverName" uri="tcp://server:2020"/>
  </CacheConfiguration>
  <ConfData>
    <CfgDN>
      <DBID value="267" />
      <switchDBID value="111" />
      <tenantDBID value="1" />
      <type value="3" />
      <number value="1111" />
      <loginFlag value="1" />
      <registerAll value="2" />
      <groupDBID value="0" />
      <trunks value="0" />
      <routeType value="1" />
      <state value="1" />
      <name value="DNAlias" />
      <useOverride value="2" />
      <switchSpecificType value="1" />
      <siteDBID value="0" />
      <contractDBID value="0" />
      <accessNumbers />
      <userProperties />
    </CfgDN>

    <CfgService>
      <DBID value="102" />
      <name value="Solution1" />
      <type value="2" />
      <state value="1" />
      <solutionType value="1" />
      <components>
        <CfgSolutionComponent>
```

```
          <startupPriority value="3" />
          <isOptional value="2" />
          <appDBID value="153" />
        </CfgSolutionComponent>
      </components>
      <SCSDBID value="102" />
      <assignedTenantDBID value="101" />
      <version value="7.6.000.00" />
      <startupType value="2" />
      <userProperties />
      <componentDefinitions />
      <resources />
    </CfgService>
  </ConfData>
</Cache>
```

Cache Policy

The configuration cache can be assigned a policy represented by a `Policy` interface. A default implementation of the interface will be provided in the `DefaultConfCachePolicy` class.

The `IConfCache` interface interprets the policy as follows:

1. `CacheOnCreate` — When an object is created in the configuration server, the policy will be checked with the created object as the parameter. If the method returns true, the object will be added to the cache, if it is false, the object will not be added. Default implementation will always return false.
2. `RemoveOnDelete` — When an object is deleted in the configuration server, the policy will be checked with the deleted object as the parameter. If the method returns true, the object will be deleted in the cache, if it is false, the notification will be ignored. Default implementation will always return true.
3. `TrackUpdates` — When an object is updated in the configuration server, the policy will be checked with the current version of the object as the parameter. If the method returns true, the object will be updated with the received delta, if it is false, the notification will be ignored. Default implementation will always return true.
4. `ReturnCopies` — Determines whether the cache should return copies of objects when they are retrieved from the cache, or the original, cached versions. False by default.

IConfServicePolicy Interface

The `IConfServicePolicy` interface can be used to define the policy settings for the `ConfService`. Two default implementations are available:

1. `DefaultConfServicePolicy` contains the settings for a non-caching configuration service. That is, all of the cache-related policy flags will always return false.
2. `CachingConfServicePolicy` defines the default behavior for a configuration service with caching enabled. (Note that when referring to the "default" value below, we will be referring to this implementation.)

The policy interface settings are interpreted as follows:

- AttemptLinkResolutionThroughCache — Whenever a link resolution attempt is made, this policy will be checked for the type of object the link refers to. If this method returns true, the link resolution attempt will first be made through the cache. If the method returns false, or if the object has not been found in the cache, the server will queried. Default value is always true.

- `CacheOnRetrieve` — This method will be called for each object retrieved from the configuration. If the return value is "true" the object will be added to the cache. Default value is always true.
- `CacheOnSave` — This method will be called for each object that is being saved. If the return value is true, the object will be added to the cache. If the object is already in the cache, it will not be overwritten. Default value is always true.
- `ValidateBeforeSave` — This is a property from the `ConfService` which will be moved to the policy interface and is not related to caching. It is used to indicate whether property values are checked for valid values against the schema before a save attempt is made. Default value is true.
- `QueryCacheOnRetrieve` — This method will be called every time a retrieve operation is performed using a query. The `ConfService` will first check the cache for the existence of the requested configuration object. If the object exists, it will be returned and no configuration server request will be made. If there are no values returned, the `ConfService` will query the configuration server (see *Query Engine*). Default value is always false.
- `QueryCacheOnRetrieveMultiple` — This method will be called every time a retrieve multiple operation is performed. The `ConfService` will first execute the query against cache. If the returned object count is greater than 0 the found object collection will be returned and no configuration server request will be made. If there are no values returned, the `ConfService` will query the configuration server (see *Query Engine*). Default value is always false.

Note that the `RetrieveMultiple` operation is NOT implemented in the default query engine, so providing a policy where this method returns true will require a new query engine implementation.

Cache Extendability

Consistent with the design principles outlined above, the configuration cache is extendable via custom implementations of provided interfaces. The two areas of the cache which can be extended are the cache storage and the cache query engine.

Cache Storage

The storage interface defines the method by which objects are stored in the cache. When an instance of an implementing object is provided to the cache, the cache will store all cached objects in the storage component.

The default storage implementation stores cached objects using the object type and DBID as keys. Note that this means that objects in the cache are assumed to be from one configuration database. The default implementation is also thread safe using a reader/writer lock which allows for multiple concurrent readers and one writer. The storage methods are as follows:

- Add — Adds a new object to the storage. If object already exists in the storage, the default implementation thrown an exception.
- Update — Overwrites an existing object in the storage. If the object is not found in the storage, the default implementation creates a new version of the object.
- Remove — Removes an object from the storage.
- Retrieve — Retrieves an enumerable list of all objects in the storage (filtered by type), and possibly influenced by an optional helper parameter. Note that the helper parameter is not meant to provide querying logic — that should be done in the query engine. Because the query engine is to some degree dependent on the storage implementation, the helper parameter allows for some flexibility in the way stored objects are enumerated for the query engine. The default implementation can take a

`CfgObjectType` as a helper parameter.

- Clear — Removes all objects in the storage.

Query Engine

The query engine provides the ability to define the method by which objects are located in the cache.

Depending on the `IConfService` policy, `Retrieve` requests as well as link resolution can first be attempted through the cache. If the requested object is found in the cache, then that cached object is returned instead of sending a request to Configuration Server. If the object is not present in the cache, a request to Configuration Server is made.

A user-definable query engine module exists inside the cache to achieve this functionality. A query engine must implement the `IConfCacheQueryEngine` interface, which provides methods to retrieve objects (either individually, or as a list) and to test a query and determine if it can be executed.

If enabled by the policy, `IConfService` will attempt a query to its cache using the cache's query engine interface. If a result is returned, the `IConfService` will not query the Configuration Server. By following this contract, the Configuration Object Model user is then able to create a custom implementation of the `IConfCacheQueryEngine` with any extended search capabilities which may be missing from the simple default implementation.

Two implementations of the `IConfCacheQueryEngine` interface are provided in the Configuration Object Model, as described below:

- DEFAULTCONFCACHEQUERYENGINE CLASS - The `DefaultConfCacheQueryEngine` class is a default implementation of the `IConfCacheQueryEngine` interface.
- COMPOSITECONFCACHEQUERYENGINE CLASS - This class is a more advanced implementation of the query engine which allows child query engine modules to be registered in order to interpret different types of queries. It does not have a default query engine implementation, only the mechanism for working with multiple child query engines.

Notification and Delta Handling

The default configuration cache will implement the `Subscriber<ConfEvent>` interface which will allow the cache to be subscribed to receive configuration events. When a cache instance is associated with a Configuration Service, it will automatically be subscribed for configuration events from that service (note that if a custom cache implementation also implements this interface it will be subscribed for events as well). The way the cache is updated based on these notifications is determined by the cache policy.

In addition, a new filter class will be added in order to allow the subscriber to filter the cache events. The `ConfCacheFilter` will implement the MessageBroker's `Predicate` interface, allowing for the filter to be passed during registration for events via `SubscriptionService`. The `ConfCacheFilter`'s properties will specify the parameters by which the events will be filtered. Initially, the supported parameters will be object type, object DBID, and update type, allowing the user to filter events by one or a combination of these parameters assuming an AND relationship between the parameters specified.

Using the Application Block

Installing the Configuration Object Model Application Block

Before you install the Configuration Object Model Application Block, it is important to review the software requirements established in the Genesys Supported Operating Environment Reference Manual.

Building the Configuration Object Model Application Block

> **Tip**
>
> Starting with release 8.5.0, the common interfaces for COM Application Block and Message Broker code have been moved to an individual `commonsappblock.jar` file.

To build the Configuration Object Model Application Block:

1. Open the `<Platform SDK Folder>\applicationblocks\com` folder.
2. Run either `build.bat` or `build.sh`, depending on your platform.

This will create the `commonsappblock.jar` file, located within the `<Platform SDK Folder>\applicationblocks\com\dist\lib` directory.

Using the QuickStart Application

The easiest way to start using the Configuration Object Model Application Block is to use the bundled QuickStart application. This application ships in the same folder as the application block.

Configuring the QuickStart Application

In order to use the QuickStart application, you will need to change some lines of code in the `quickstart.properties` file, located in the `<Platform SDK Folder>\applicationblocks\com\quickstart` directory. Change the following lines to point to your Configuration Server, and then save the updated file:

```
ConfServerUri = tcp://:

ConfServerUser =
ConfServerPassword =

ConfServerClientName = default
ConfServerClientType = CFGSCE
```

Building the QuickStart Application

1. Open the `<Platform SDK Folder>\applicationblocks\com\quickstart` folder.
2. Run either `build.bat` or `build.sh`, depending on your platform.

Running the QuickStart Application

1. Open a Command Prompt or Terminal window.
2. Navigate to the `<Platform SDK Folder>\applicationblocks\com\quickstart` directory.
3. Run either `quickstart.bat` or `quickstart.sh`, depending on your platform.

How to Properly Initialize the ConfService Instance

To work with Configuration Server, the `ConfService` instance needs `ConfServerProtocol`.

Platform SDK protocol connections allow users to manage connections, setup custom asynchronus `MessageHandler` objects, substitute message receivers, and subscribe for protocol messages and channel events. So, to maintain Platform SDK flexibility, the Configuration Object Model Application Block does not manage a `ConfServerProtocol` connection inside of the `ConfService` - this must be done by the user. Instead users may create a simple instance and initialize it with `WarmStandbyService`.

It is important to note that asycnhronous protocol events may be configured for delivery to a single destination, with only one `MessageHandler` or `MessageReceiver` for one protocol instance. Starting from Platform SDK release 8.1.1, `ConfService` may be initialized without use of legacy Message Broker Application Block. Starting from version 8.5, this is the only way to create `ConfService`.

If your application needs to receive asynchronous protocol messages from Configuration Server on the protocol instance where `ConfService` is initialized, that can be done using `ConfService.setUserMessageHandler(messageHandler)`.

Protocol Initialization

A `ConfServerProtocol` instance is required for the creation of `ConfService`. It should be initialized with an Endpoint and handshake properties, but without setting either `confServerProtocol.setMessageHandler()` or `confServerProtocol.setReceiver()`.

```
// Initialize ConfService:
PropertyConfiguration       config;
ConfServerProtocol          confServerProtocol;
IConfService                confService;

config = new PropertyConfiguration();
config.setUseAddp(true);
config.setAddpClientTimeout(15);

confServerProtocol = new ConfServerProtocol(new Endpoint("ConfigServer", csHost, csPort,
config));
confServerProtocol.setUserName(userName);
confServerProtocol.setUserPassword(password);
confServerProtocol.setClientName(clientName);
confServerProtocol.setClientApplicationType(clientType.ordinal());
```

Important

Do *not* open the protocol before ConfService is created. ConfService sets its own internal MessageHandler, and this operation can only be done on a closed channel.

ConfService Initialization

```
confService = ConfServiceFactory.createConfService(confServerProtocol);
confServerProtocol.open();
```

ConfService Shutdown

```
confServerProtocol.close();
ConfServiceFactory.releaseConfService(confService);
confService = null;
```

Application Components Usage Notes

Older releases of ProtocolManagementService do not support using ConfService without the Message Broker service - an exception raised when users try to create the ConfService object on a protocol instance initialized by the Protocol Manager Application Block. To migrate away from Protocol Manager Application Block usage, we recommend creating and configuring ConfServerProtocol without Protocol Manager Application Block usage, as shown above.

MessageHandler is not compatible with the deprecated MessageReceiver; it is only possible to use one of these components on a protocol instance. Specific to Platform SDK for Java is the limitation that one protocol instance may have only one instance of MessageHandler. So, if an application uses a custom MessageHandler on a protocol used for ConfService, then only one handler will be able to receive asynchronous protocol events.

If application overwrites the ConfService object after creation, then that service will be unable to receive Configuration Server notifications or to perform multiple objects reading operations - a timeout exception will occur. If there is a need to get those protocol messages separately from ConfService logic, it is possible to initialize custom MessageHandler with confService.setUserMessageHandler(messageHandler).

Notes for Previous Releases of Platform SDK

'''[+] Platform SDK 8.1.0 Specific Notes'''

Platform SDK 8.1.0 included some improvements to the Message Broker Application Block.

There was a new EventReceivingBrokerService class that implements the receiver interface, which can be used as an external receiver for Platform SDK protocols. When this class is in use, protocol messages will be handled a little bit faster (compared to the older Message Broker service) with no redundant intermediate queue, and there is no additional thread sleeping/waiting.

```
EventReceivingBrokerService broker = new EventReceivingBrokerService();
broker.setInvoker(new SingleThreadInvoker("COMBrokerService-" + cfgsrvEndpointName));
```

```
ConfServerProtocol protocol = new ConfServerProtocol(endpoint);
protocol.setReceiver(broker);
protocol.setUserName(...);
protocol.set...();
protocol.open();

IConfService confService = ConfServiceFactory.createConfService(protocol, broker);
```

To shutdown the Configuration Object Model Application Block, you can use the following code:

```
protocol.close();
ConfServiceFactory.releaseConfService(confService);
```

'''[+] Platform SDK 8.0, 7.6 Specific Notes'''

In earlier releases of Platform SDK, the initialization logic could look like this:

```
ConfServerProtocol protocol = new ConfServerProtocol(endpoint);
protocol.setUserName(...);
protocol.set...();
protocol.open();

EventBrokerService broker = BrokerServiceFactory.CreateEventBroker(protocol);
IConfService confService = ConfServiceFactory.createConfService(protocol, broker);
```

If the protocol has an external receiver initialized (for example, with Protocol Manager usage), then the `EventBrokerService` should be initialized on that receiver instead of the protocol itself:

```
EventBrokerService broker =
BrokerServiceFactory.CreateEventBroker(protocolManager.getReceiver());
```

To shutdown the Configuration Object Model Application Block, you can use the following code:

```
protocol.close();
broker.dispose();
ConfServiceFactory.releaseConfService(confService);
```

> **Important**
> Legacy `EventBrokerService` objects need to be disposed on shutdown because they include an internal reading thread which should be stopped.

Editing Capacity Rules

The Configuration Object Model Application Block includes the `CapacityRuleHelper` class (introduced in release 8.1.4) which allows you to edit and update Capacity Rules. This helper class presents an XML representation for `CfgScript` objects of type `CfgScriptType.CFGCapacityRule`, which can be updated and saved to edit existing Capacity Rules.

An example of how to edit capacity rules is provided below.

```
IConfService service = (IConfService)ConfServiceFactory.createConfService(protocol);
service.getProtocol().open();
CfgScriptQuery query = new CfgScriptQuery(service);
CfgScript script = (CfgScript)service.retrieveObject(query);
```

An instance of the `CapacityRuleHelper` class can now be created with static method `create`. This method validates the input script object and can throw an exception: `ConfigException` if the script has an invalid format, or `IllegalArgumentException` if the script is null or the script type is not valid. Once the instance is created, `getXMLPresentation()` allows you access to Capacity Rules.

```
CapacityRuleHelper helper = CapacityRuleHelper.create(script);
Document doc = helper.getXMLPresentation();
// edit xml document here
```

The `setXMLPresentation()` method allows you to save changes to the XML into the `CapacityRuleHelper` class instance. Once changes have been made to the XML document, apply your changes using the following code:

```
helper.setXMLPresentation(doc);
helper.getCfgScript().save();
service.getProtocol().close();
ConfServiceFactory.releaseConfService(service);
```

.NET

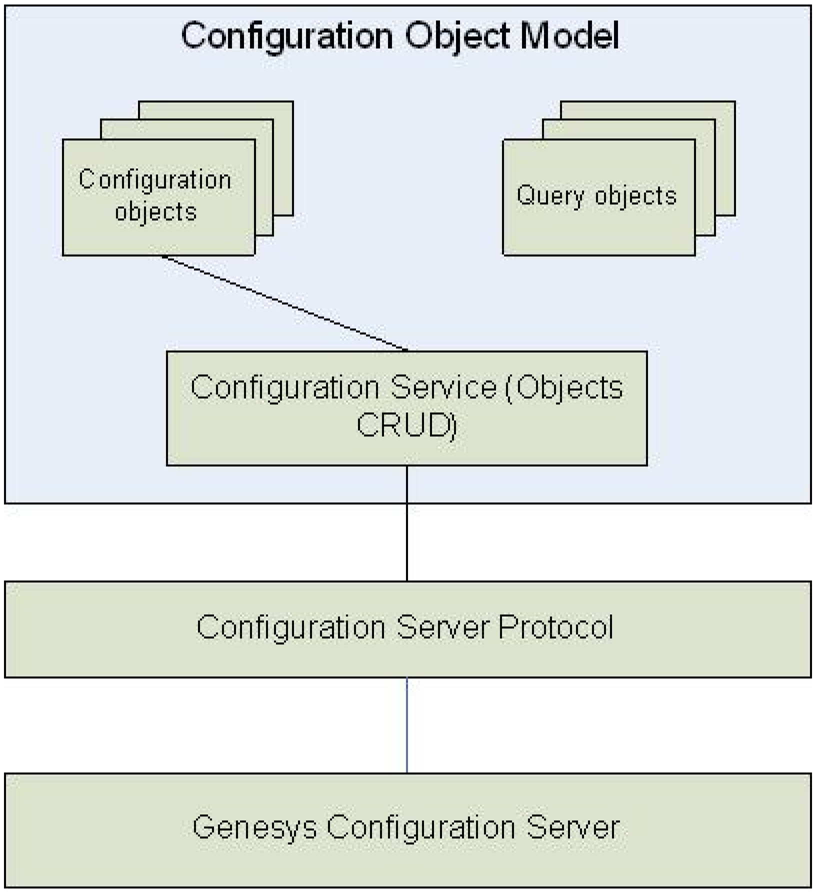
Architecture and Design

The Configuration Object Model Application Block provides a consistent and intuitive object model for working with Configuration Server objects, as well as a straightforward object model for queries with different filters. This Application Block hides the complexities of object creation and changing by means of "delta" objects. It also creates an event subscription/delivery model, which hides key-value details of the current protocol, and is integrated with the rest of the object model.

The architecture of the Configuration Object Model Application Block consists of three functional components:

- Configuration Objects
- Configuration Service
- Query Objects
- Cache Objects

These components are shown in the figure below.


Configuration Object Model
Configuration objects
Query objects
Configuration Service (Objects CRUD)
Configuration Server Protocol
Genesys Configuration Server

Configuration Objects

Classes and Structures

There are two types of configuration objects are supported by the Configuration Object Model Application Block:

- Classes, which can be retrieved directly from Configuration Server using queries.
- Structures, which only exist as properties of classes, and cannot be retrieved directly from Configuration Server.

The main differences between classes and structures are as follows:

1. Each structure is a property of another class or structure, and therefore must have a "parent" class.
2. Classes can be changed and saved to the Configuration Server and structures can only be saved through their "parent" classes.
3. Clients can subscribe to events on changes in a class, but not in a structure. To retrieve events on changes in a structure, clients have to subscribe to changes in a parent class.

Property Types

Both classes and structures have properties. For each property, the object has getter and setter methods which retrieve the value of the property and set a new value correspondingly. However, some properties are read-only and therefore will only have a getter method. For each object, its properties can be one of the following types:

- *Simple* — A property that is represented by a value type. Configuration Server supports two types of simple properties - `string` and `integer`. For example, the `CfgPerson` object has `FirstName` and `LastName` properties, both of the string type.
- *KV-list* — Tree-like properties that are represented by the `KeyValueCollection` class in the Configuration Object Model. Examples of this property include `userProperties` of `CfgPerson`.
- *Structure* — A complex property that includes one or more properties. In the Configuration Object Model, structures are represented by instances of classes that are similar to configuration objects, but cannot be created directly. For example, in the `CfgPerson` class, its `AgentInfo` property contains simple, kv-list and other property types.
- *List of structures* — A property that represents more than one structure. In Configuration Object Model, lists of structures are represented by a generic type `IList<structure_type>`, so that the collection is typed, and clients can easily iterate through the collection.
- *Links to a single object* — In Configuration Server, these properties are stored as DBIDs of external objects. The Configuration Object Model automatically resolves these DBIDs into the real objects, which can be manipulated in the same way as the objects directly retrieved from Configuration Server. Links are initialized at the time of the initial request to one of its properties.

> Tip
>
> For each link, there are two ways to set the new value of a link. There is a setter method of the property, which uses an object reference to set a new value of a link. There is also a Set DBID method, which uses an integer DBID value.

- *Links to multiple objects* — A property that contains more than one link. In the Configuration Object Model, lists of structures are represented by a generic type `IList<class_type>`, so that the collection is typed, and clients can easily iterate through the collection.

Creating Instances

One way to create an instance of an object in the Configuration Object Model is to invoke one of the Retrieve methods of the `ConfService` class. This set of methods returns instances of objects that already exist in Configuration Server.

To create a new object in Configuration Server, a client must create a new instance of a COM or "detached" object. The detached object does not correspond to any objects in Configuration Server until it is saved. The detached object is created using the regular object-oriented language object instantiation. For example, a new detached `CfgPerson` object is created using the following construction:

```
[C#]

CfgPerson person = new CfgPerson(confService);
```

An object instance can also be created by using links to external objects. The Component Object Model creates a new object instance whenever the link is called, or any of the properties of a linked object are called. For example, you can write:

```
[C#]

// Person has already been retrieved from Configuration Server.
CfgTenant tenant = person.Tenant;
// This is a link to an external object. It is initialized internally right now...
CfgAddress address = tenant.Address;
```

Common Methods

Each configuration class contains the following methods:

- `Generic GetProperty(string propertyName)` — Retrieves the property value by its name.
- `Generic SetProperty(string propertyName)` — Sets the new value of the property by its name.
- `Save()` — Commits all changes previously made to the object to Configuration Server. If the object was created detached from Configuration Server and has never been saved, a new object is created in Configuration Server using the `RequestCreateObject` method. If the object has been saved or has been retrieved from Configuration Server, a delta-object, which contains all changes to the object, is formed and sent to Configuration Server by means of the `RequestUpdateObject` method.
- `Delete()` — Deletes the object from the Configuration Server Database.
- `Refresh()` — Retrieves the latest version of the object and refreshes the value of all its properties.

Tip

In this release, all configuration objects are "static," which means that if the object changes in the Configuration Server, the instance of a class is not automatically changed in the Configuration Object Model. Clients must subscribe to the corresponding event and manually refresh the COM object in order for these changes to take effect.

Configuration Service

Important

The `IConfService` interface was added to COM in release 8.0. All applications should now use this interface to work with the configuration service instead of the old `ConfService` class. This change is an example of how all COM types in the interface are now referred to by interface; for instance, if a method previously returned `CfgObject` it now returns `ICfgObject`. This is not compatible with existing code, but upgrading should not be difficult as the new interfaces support the same methods as the implementing types.

The Configuration Service (`IConfService`) interface provides services such as retrieval of objects and subscription to events from Configuration Server. Each connection to a Configuration Server (represented by a `ConfServerProtocol` class of Platform SDK) requires its own instance of the `IConfService` interface.

The protocol class should be created and initialized in the client code prior to `IConfService` initialization.

The `ConfServiceFactory` class is used to create the `IConfService`. This class uses the following syntax:

```
[C#]

IConfService service = ConfServiceFactory.CreateConfService(protocol);
```

Retrieving Objects

Objects can be retrieved from Configuration Service by using one of the following methods:

- `RetrieveObject` — Accepts a query that returns one object. If multiple objects are returned, an exception is thrown.
- `RetrieveMultipleObjects` — Accepts a query that returns one or more objects. A collection of objects is returned.

Each of the `Retrieve` methods can be can be strongly typed (with use of generics, an object of a specified type is returned) or general (a general object is returned).

Handling Events

The following methods must be called before receiving events from Configuration Server:

1. Register

The application must register its callback by calling the `Register` method from the Configuration Service. This method supplies the client's filter, which enables the client to receive only requested events.

2. Subscribe

The application must subscribe to events from Configuration Server by calling the `Subscribe` method from the Configuration Service. This method provides a notification query object as a parameter.

The `NotificationQuery` object determines whether the object (or set of objects) to which the client wants to subscribe has changed. The `NotificationQuery` object contains such parameters as object type, object DBID and tenant DBID.

After calling the `Subscribe` method, Configuration Server starts sending events to the client. These events are objects, which contain information such as:

- which object (ID and type) is affected
- the type of event sent to the client
- any additional information

There are three types of events that the client might receive:

- `ObjectCreated` — A new object has been added to Configuration Server.
- `ObjectChanged` — Some of the object properties have been modified in Configuration Server.
- `ObjectDeleted` — The object has been removed from Configuration Server.

Logging Messages

Configuration Object Model Application Block supports logging through the standard Platform SDK logging interfaces. The `IConfService` interface inherits the `EnableLogging` method that provides the ability to log messages through the provided `ILogger` interface.

Releasing a Configuration Service

Whenever a `ConfService` instance is no longer needed, the `ReleaseConfService` method can be used to remove it from the internal list.

```
[C#]

ConfServiceFactory.ReleaseConfService(service);
```

Query Objects

A query object is an instance of a class that contains information required for a successful query to a Configuration Server. This information includes an object type and its attributes (such as `name` and `tenant`), which are used in the search process.

The inheritance structure of configuration server queries is designed to allow for future expansion. The `CfgQuery` object is the base class for all query objects. Other classes extend `CfgQuery` to provide more specific functionality for different types of queries - for example, all filter-based queries use the `CfgFilterBasedQuery` class. This allows room for future query types (such as XPath) to be implemented in this Application Block.

A list of currently available query types is provided below:

- `CfgFilterBasedQuery` — Contains mapped attribute name-value pairs, as well as the object type.

A special query class is supplied for each configuration object type, in order to facilitate the process of making queries to Configuration Server. For each searchable attribute, the query class has a property that can be set. All of these classes inherit attributes from the `CfgQuery` object, and can be supplied as parameters to the `Retrieve` methods which are used to perform searches in Configuration Server.

Cache Objects

The cache functionality is intended to enhance the Configuration Object Model by allowing configuration objects to be stored locally, thereby minimizing requests to configuration server, as well as enhancing ease of use by providing automatic synchronization between locally stored objects and their server-side counterparts.

The cache functionality was designed with the following principles in mind:

- The cache functionality is designed to be extendable with custom implementations of provided interfaces and not via inheritance.
- The cache component is not designed to replicate the Configuration Server query engine or other Configuration Server functionality on the client side.
- Caching must be an optional feature. Work with Configuration Server should not be affected if caching is not used.

Use Cases

Analysis of use cases provides insight into the requirements for applications likely to require configuration cache functionality. The use cases described in the following table were selected for analysis in order to highlight different functional requirements. There are several possible actors which are referenced in the use cases. The actors are as follows:

- Application - Any application which uses the Configuration Object Model application block
- User - Human (or software) user who may perform actions upon objects in the configuration which are separate from the Application

Use Case	Description	Actor	Steps
PLACE OBJECT INTO CACHE	Place a configuration object into the configuration cache (note the object must have been saved — ie must have a DBID in order to exist in the cache).	Application	1. Application adds object to the cache
PLACE OBJECT INTO CACHE ON SAVE	Place a newly created configuration object into the configuration cache when it is saved.	Application	1. Application creates object 2. Application saves object 3. Configuration Object Model Application Block adds object to the cache
PLACE OBJECT INTO CACHE ON RETRIEVE	Allow for automatic insertion of configuration objects into the cache upon retrieval from configuration server.	Application	1. Application retrieves configuration object 2. Configuration Object Model Application Block retrieves the configuration object from the server 3. Configuration Object Model Application Block places the configuration object

Use Case	Description	Actor	Steps
			into the cache 4. Configuration Object Model Application Block returns the object to the application
OBJECT REMOVED IN CONFIGURATION SERVER	When configuration objects are deleted in the configuration server, the cache can delete the local representation of the object as well.	User	1. User deletes object in the Configuration Server 2. Cache removes corresponding local object upon receiving delete notification 3. Cache sends notification of object deletion to Application
SYNCHRONIZE OBJECT PROPERTIES WITH CONFIGURATION SERVER	When an object stored in the cache is updated in the Configuration Server the object must be updated locally as well.	User	1. User updates a configuration object 2. Cache receives notification about object update 3. Cache updates the object based on the received delta 4. Cache fires event informing any subscribers of object change
FIND OBJECT IN CACHE	The cache must support the ability to find a specific configuration object in the cache using object DBID and type as the criteria for the search.	Application	1. Application retrieves object from cache. 2. If object is in the cache, the cache returns the object. Otherwise the application is notified that the requested object is not in the cache.

Use Case	Description	Actor	Steps
ACCESS CACHED OBJECTS	The cache must provide its full object collection to the application.	Application	1. Application requests a complete list of objects from the cache. 2. The cache returns a collection of all cached objects.
RETRIEVE LINKED OBJECT FROM CACHE	If caching is turned on, object links which the Configuration Object Model currently resolves through lazy initialization (i.e. if a property linking to another object is accessed, we retrieve the referred-to object from configuration server) must be resolvable through cache access.	Application	1. Application accesses a property which requires link resolution 2. Configuration Object Model Application Block retrieves the linked object from configuration server and stores it in the cache before returning to the application 3. Application again accesses the property and this time the Configuration Object Model Application Block retrieves the object from the cache
PROVIDE CACHE TRANSPARENCY ON RETRIEVE	A cache search should be performed on attempt to retrieve an object from Configuration Server. If the requested object is found in the cache then the Configuration Object Model should return the cached object rather than accessing Configuration Server.		1. Application creates query to retrieve configuration object 2. Application executes query using the Configuration Object Model 3. Configuration Object Model Application Block searches the cache • If object present, return the object • If object not present, query configuration

Use Case	Description	Actor	Steps
			server for the object
CACHE SERIALIZATION	The cache should support serialization.	Application	1. Application provides a stream to the cache 2. The cache serializes itself into the stream in an XML format 3. Application restarts 4. Application provides the cache a stream of cache data in the same XML format as in step 2 5. Cache restores itself 6. Cache subscribes for updates on the restored objects

Implementation Overview

Two new interfaces for cache management have been added to the Configuration Object Model: the `IConfCache` interface and a default cache implementation (`DefaultConfCache`). Note that the `ConfCache` also implements the `ISubscriber` interface from `MessageBroker`. The cache implements `ISubscriber` in order to allow the user to subscribe to notifications from Configuration Server, as discussed in *Notification And Delta Handling*.

The `IConfCache` interface provides methods for basic functionality such as adding, updating, retrieving, and removing objects in the cache. It also includes a `Policy` property that defines cache behavior and affects method implementation. (For more details about policies, see *Cache Policy*).

The `DefaultConfCache` component provides a default implementation of the `IConfCache` interface. It serializes and deserializes cache objects using the XML format described in the *XML Format* section, below.

To enable and configure caching functionality, and to specify `ConfService` policy, there are three `CreateConfService` methods available from `ConfServiceFactory`. The original `CreateConfService` method (not shown here) creates a `ConfService` instance that uses the default policy and does not use caching.

```
[C#]

public static IConfService CreateConfService(IProtocol protocol, bool enableCaching)
```

This method creates an instance of a Configuration Service based on the specified protocol. If caching is enabled, the default caching policy will be used. If `enableCaching` is set to true, caching functionality will be turned on. If caching is disabled, all policy flags related to caching will be false.

```
[C#]

public static IConfService CreateConfService(IProtocol protocol,
        IConfServicePolicy confServicePolicy, IConfCache cache)
```

This method creates a configuration service with the specified policy information. The created service will have caching enabled if a cache object (implementing the `IConfCache` interface) is passed as a parameter.

```
[C#]

public static IConfService CreateConfService(IProtocol protocol,
        IConfServicePolicy confServicePolicy, IConfCachePolicy confCachePolicy)
```

This method creates a configuration service with the specified policy information. The created service will have caching enabled by default with the cache using the specified cache policy.

XML Format

The "Cache" node will be the root of the configuration cache XML, while "ConfData" is a child of the "Cache" node. The ConfData node contains a collection of XML representations for each configuration object in the cache. The XML format of each object is identical to that which is returned by the `ToXml` method supported by each the Configuration Object Model configuration object.

The "CacheConfiguration" element is a child of the "Cache" node. There can only be one instance of this node and it contains all cache configuration parameters, as follows:

- CONFIGURATIONSERVER NODE – There can be 1..n instances of this element. Each one will represent a configuration server for which the cache is applicable (a cache can be applicable to multiple configuration servers if they are working with the same database as in the case of a primary and backup configuration server pair). Each `ConfigurationServer` element will have a `URI` attribute specifying the unique URI identifying the Configuration Server, as well as a `Name` attribute specifying the name associated with the endpoint.

The example provided below shows a cache that is applicable for the configuration server at "server:2020" with some policy details specified. There are two objects in the cache for this example: a `CfgDN` and a `CfgService` object.

```
[XML]

<Cache>
        <CacheConfiguration>
                <ConfigurationServer name="serverName" uri="tcp://server:2020"/>
        </CacheConfiguration>

        <ConfData>
                <CfgDN>
                        <DBID value="267" />
                        <switchDBID value="111" />
                        <tenantDBID value="1" />
                        <type value="3" />
                        <number value="1111" />
                        <loginFlag value="1" />
                        <registerAll value="2" />
                        <groupDBID value="0" />
                        <trunks value="0" />
                        <routeType value="1" />
                        <state value="1" />
                        <name value="DNAlias" />
                        <useOverride value="2" />
                        <switchSpecificType value="1" />
                        <siteDBID value="0" />
                        <contractDBID value="0" />
                        <accessNumbers />
                        <userProperties />
                </CfgDN>

                <CfgService>
                        <DBID value="102" />
                        <name value="Solution1" />
                        <type value="2" />
                        <state value="1" />
                        <solutionType value="1" />
                        <components>
                                <CfgSolutionComponent>
                                        <startupPriority value="3" />
                                        <isOptional value="2" />
                                        <appDBID value="153" />
                                </CfgSolutionComponent>
                        </components>
                        <SCSDBID value="102" />
                        <assignedTenantDBID value="101" />
                        <version value="7.6.000.00" />
                        <startupType value="2" />
                        <userProperties />
                        <componentDefinitions />
                        <resources />
                </CfgService>
        </ConfData>

</Cache>
```

Cache Policy

The configuration cache can be assigned a policy represented by a Policy interface. A default implementation of the interface will be provided in the `DefaultConfCachePolicy` class.

The `IConfCache` interface will interpret the policy as follows:

1. `CacheOnCreate` – When an object is created in the configuration server, the policy will be checked with the created object as the parameter. If the method returns true, the object will be added to the cache, if it is false, the object will not be added. Default implementation will always return false.
2. `RemoveOnDelete` – When an object is deleted in the configuration server, the policy will be checked with the deleted object as the parameter. If the method returns true, the object will be deleted in the cache, if it is false, the notification will be ignored. Default implementation will always return true.
3. `TrackUpdates` – When an object is updated in the configuration server, the policy will be checked with the current version of the object as the parameter. If the method returns true, the object will be updated with the received delta, if it is false, the notification will be ignored. Default implementation will always return true.
4. `ReturnCopies` – Determines whether the cache should return copies of objects when they are retrieved from the cache, or the original, cached versions. False by default.

IConfServicePolicy Interface

The `IConfServicePolicy` interface can be used to define the policy settings for the `ConfService`. Two default implementations are available:

1. `DefaultConfServicePolicy` contains the settings for a non-caching configuration service. That is, all of the cache-related policy flags will always return false.
2. `CachingConfServicePolicy` defines the default behavior for a configuration service with caching enabled. (Note that when referring to the "default" value below, we will be referring to this implementation.)

The policy interface settings are interpreted as follows:

- `AttemptLinkResolutionThroughCache` – Whenever a link resolution attempt is made, this policy will be checked for the type of object the link refers to. If this method returns true, the link resolution attempt will first be made through the cache. If the method returns false, or if the object has not been found in the cache, the server will queried. Default value is always true.
- `CacheOnRetrieve` – This method will be called for each object retrieved from the configuration. If the return value is "true" the object will be added to the cache. Default value is always true.
- `CacheOnSave` – This method will be called for each object that is being saved. If the return value is true, the object will be added to the cache. If the object is already in the cache, it will not be overwritten. Default value is always true.
- `ValidateBeforeSave` – This is a property from the `ConfService` which will be moved to the policy interface and is not related to caching. It is used to indicate whether property values are checked for valid values against the schema before a save attempt is made. Default value is true.
- `QueryCacheOnRetrieve` – This method will be called every time a retrieve operation is performed using a query. The `ConfService` will first check the cache for the existence of the requested configuration object. If the object exists, it will be returned and no configuration server request will be made. If there are no values returned, the `ConfService` will query the configuration server (see *Query Engine*). Default value is always false.
- `QueryCacheOnRetrieveMultiple` – This method will be called every time a retrieve multiple operation is performed. The ConfService will first execute the query against cache. If the returned object count is greater than 0 the found object collection will be returned and no configuration server request will be made. If there are no values returned, the `ConfService` will query the configuration server (see *Query Engine*). Default value is always false.

Note that the `RetrieveMultiple` operation is NOT implemented in the default query engine, so providing a policy where this method returns true will require a new query engine implementation.

Cache Extendability

Consistent with the design principles outlined above, the configuration cache is extendable via custom implementations of provided interfaces. The two areas of the cache which can be extended are the cache storage and the cache query engine.

Cache Storage

The storage interface defines the method by which objects are stored in the cache. When an instance of an implementing object is provided to the cache, the cache will store all cached objects in the storage component.

The default storage implementation stores cached objects using the object type and DBID as keys. Note that this means that objects in the cache are assumed to be from one configuration database. The default implementation is also thread safe using a reader/writer lock which allows for multiple concurrent readers and one writer. The storage methods are as follows:

- Add – Adds a new object to the storage. If object already exists in the storage, the default implementation thrown an exception.
- Update – Overwrites an existing object in the storage. If the object is not found in the storage, the default implementation creates a new version of the object.
- Remove – Removes an object from the storage.
- Retrieve – Retrieves an enumerable list of all objects in the storage (filtered by type), and possibly influenced by an optional helper parameter. Note that the helper parameter is not meant to provide querying logic – that should be done in the query engine. Because the query engine is to some degree dependent on the storage implementation, the helper parameter allows for some flexibility in the way stored objects are enumerated for the query engine. The default implementation can take a `CfgObjectType` as a helper parameter.
- Clear–Removes all objects in the storage.

Query Engine

The query engine provides the ability to define the method by which objects are located in the cache.

Depending on the `IConfService` policy, `Retrieve` requests as well as link resolution can first be attempted through the cache. If the requested object is found in the cache, then that cached object is returned instead of sending a request to Configuration Server. If the object is not present in the cache, a request to Configuration Server is made.

A user-definable query engine module exists inside the cache to achieve this functionality. A query engine must implement the `IConfCacheQueryEngine` interface, which provides methods to retrieve objects (either individually, or as a list) and to test a query and determine if it can be executed.

If enabled by the policy, `IConfService` will attempt a query to its cache using the cache's query engine interface. If a result is returned, the `IConfService` will not query the Configuration Server. By following this contract, the Configuration Object Model user is then able to create a custom implementation of the `IConfCacheQueryEngine` with any extended search capabilities which may be missing from the simple default implementation.

Two implementations of the `IConfCacheQueryEngine` interface are provided in the Configuration Object Model, as described below:

- DEFAULTCONFCACHEQUERYENGINE CLASS - The `DefaultConfCacheQueryEngine` class is a default implementation of the `IConfCacheQueryEngine` interface.
- COMPOSITECONFCACHEQUERYENGINE CLASS - This class is a more advanced implementation of the query engine which allows child query engine modules to be registered in order to interpret different types of queries. It does not have a default query engine implementation, only the mechanism for working with multiple child query engines.

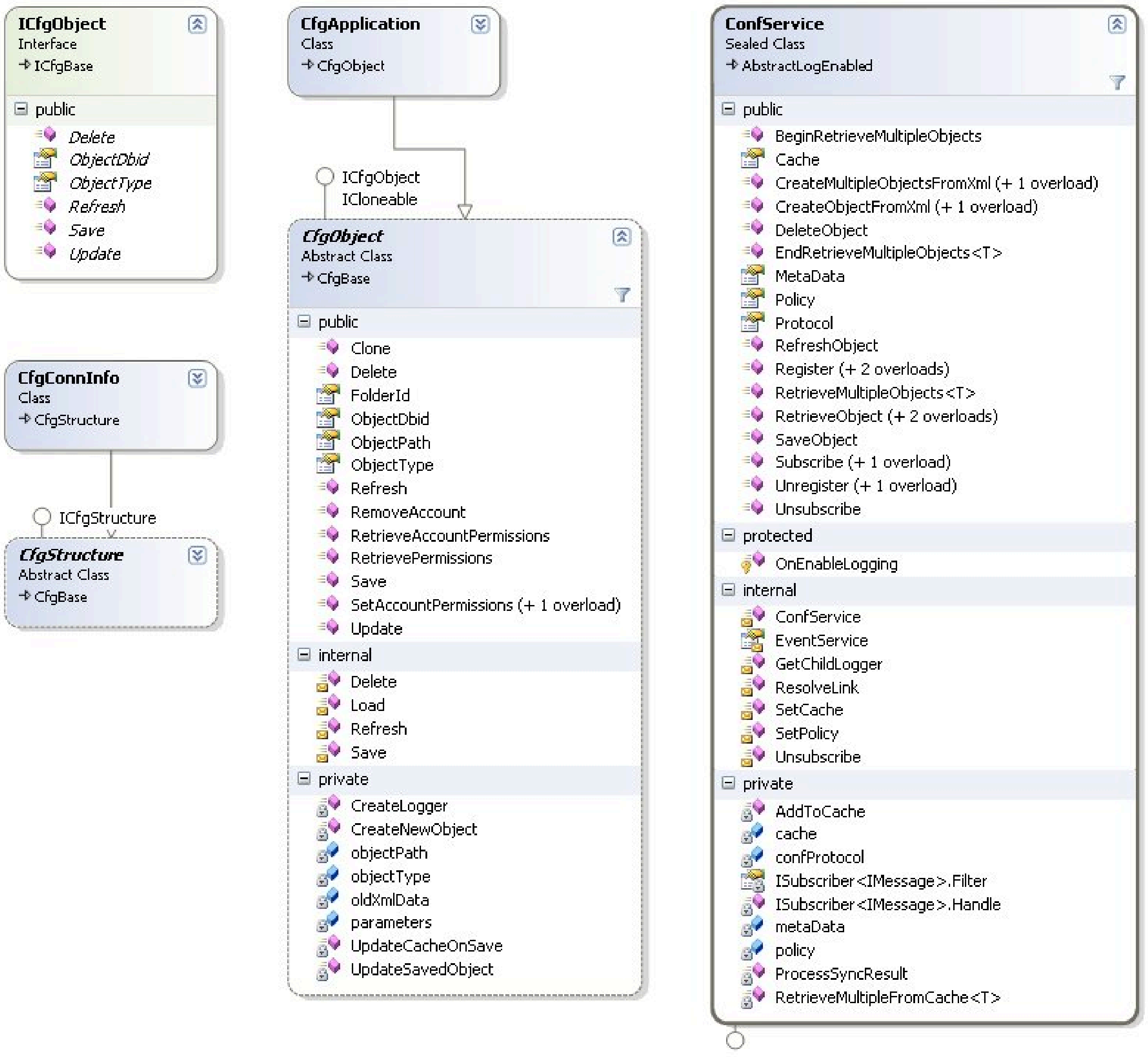
Notification and Delta Handling

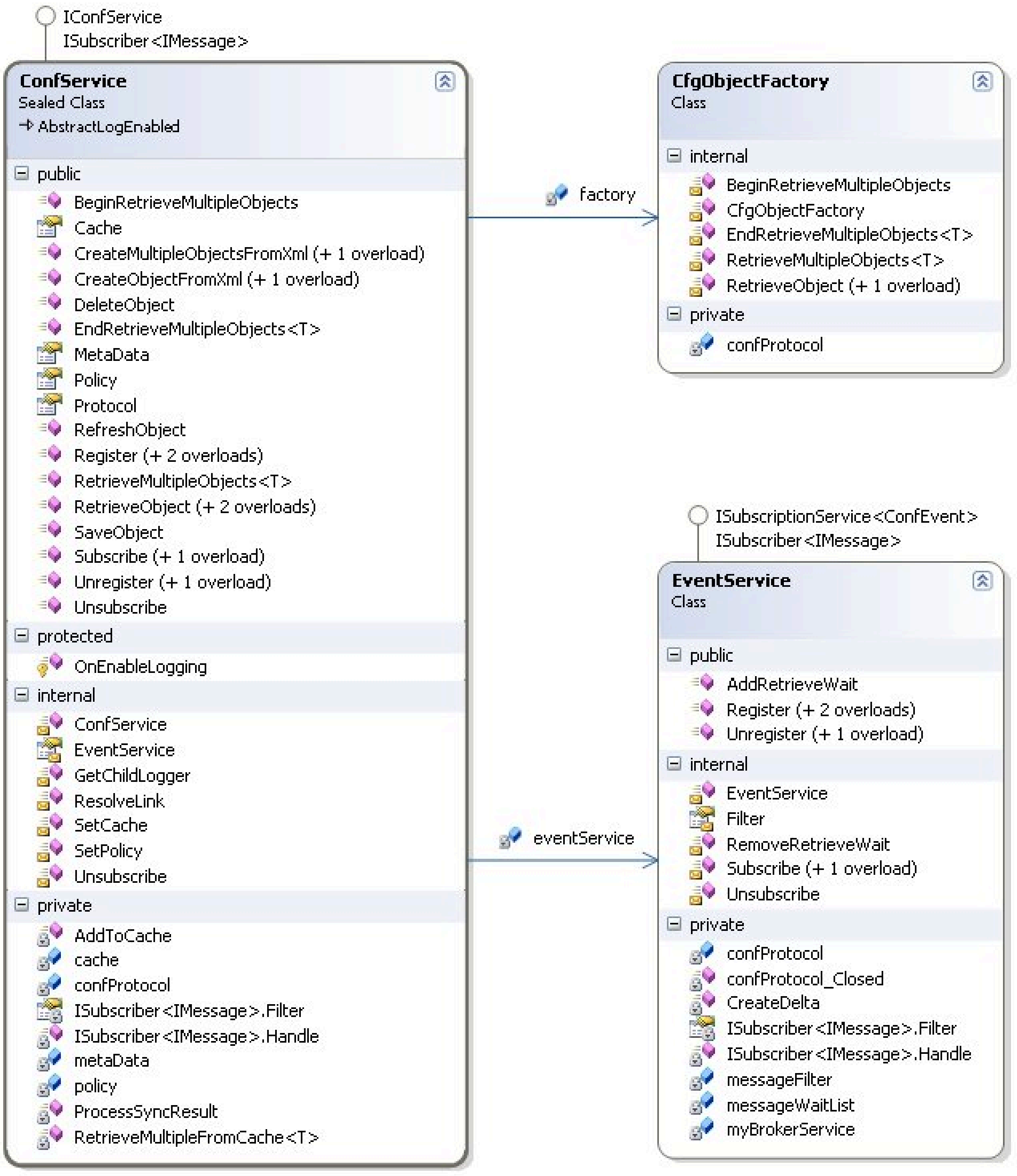
The default configuration cache will implement the `ISubscriber<ConfEvent>` interface which will allow the cache to be subscribed to receive configuration events. When a cache instance is associated with a Configuration Service, it will automatically be subscribed for configuration events from that service (note that if a custom cache implementation also implements this interface it will be subscribed for events as well). The way the cache is updated based on these notifications is determined by the cache policy.

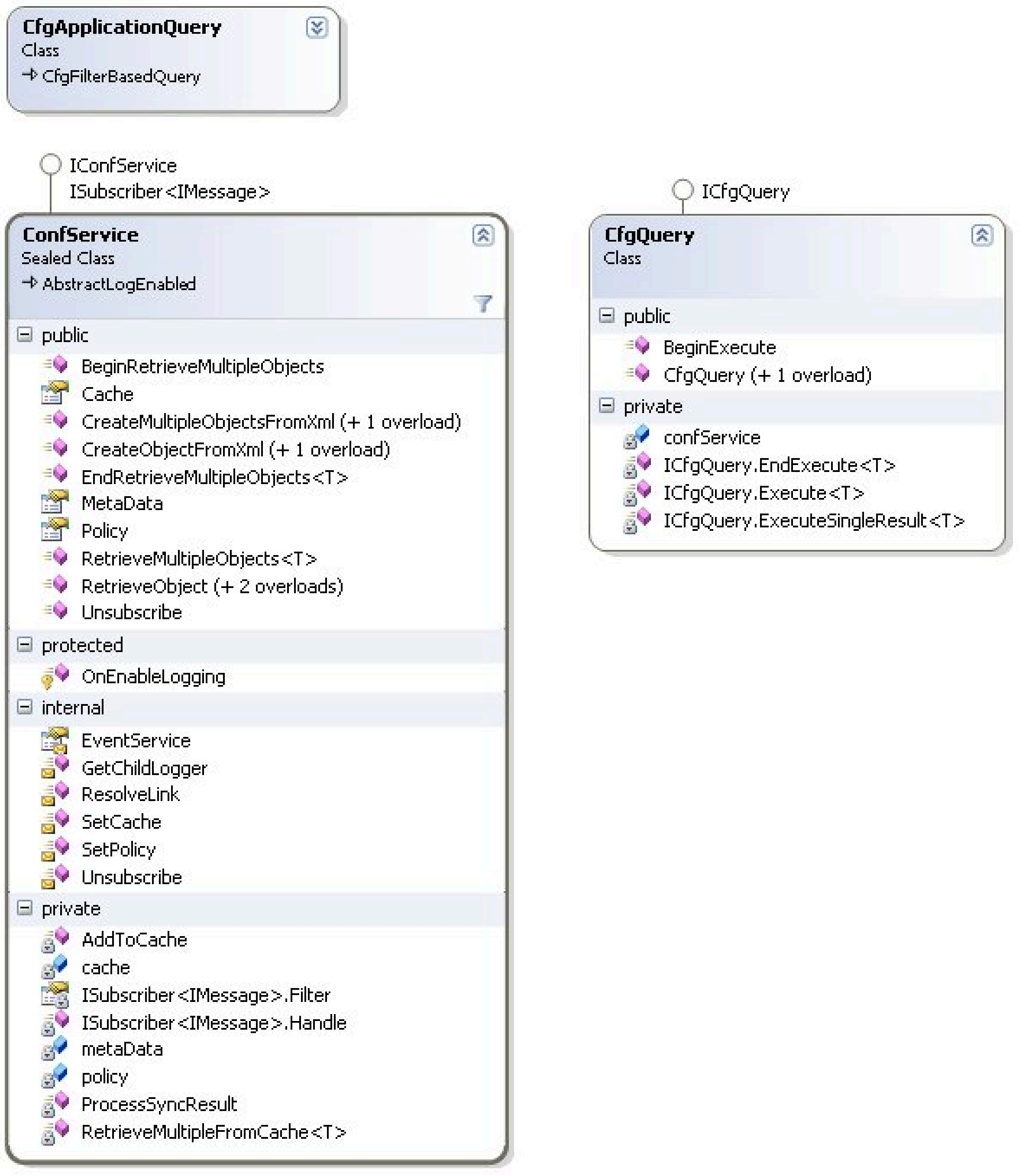
In addition, a new filter class will be added in order to allow the subscriber to filter the cache events. The `ConfCacheFilter` will implement the `MessageBroker`'s `IPredicate` interface, allowing for the filter to be passed during registration for events via `ISubscriptionService`. The `ConfCacheFilter`'s properties will specify the parameters by which the events will be filtered. Initially, the supported parameters will be object type, object DBID, and update type, allowing the user to filter events by one or a combination of these parameters assuming an AND relationship between the parameters specified.

The Configuration Object Model Application Block Interface

The following figures show the relationships among many of the classes that make up this application block.


ICfgObject
Interface
ICfgBase
public
Delete
ObjectDbid
ObjectType
Refresh
Save
Update
CfgApplication
Class
CfgObject
ICfgObject
ICloneable
CfgObject
Abstract Class
CfgBase
public
Clone
Delete
FolderId
ObjectDbid
ObjectPath
ObjectType
Refresh
RemoveAccount
RetrieveAccountPermissions
RetrievePermissions
Save
SetAccountPermissions (+ 1 overload)
Update
internal
Delete
Load
Refresh
Save
private
CreateLogger
CreateNewObject
objectPath
objectType
oldXmlData
parameters
UpdateCacheOnSave
UpdateSavedObject
CfgConnInfo
Class
CfgStructure
ICfgStructure
CfgStructure
Abstract Class
CfgBase
ConfService
Sealed Class
AbstractLogEnabled
public
BeginRetrieveMultipleObjects
Cache
CreateMultipleObjectsFromXml (+ 1 overload)
CreateObjectFromXml (+ 1 overload)
DeleteObject
EndRetrieveMultipleObjects<T>
MetaData
Policy
Protocol
RefreshObject
Register (+ 2 overloads)
RetrieveMultipleObjects<T>
RetrieveObject (+ 2 overloads)
SaveObject
Subscribe (+ 1 overload)
Unregister (+ 1 overload)
Unsubscribe
protected
OnEnableLogging
internal
ConfService
EventService
GetChildLogger
ResolveLink
SetCache
SetPolicy
Unsubscribe
private
AddToCache
cache
confProtocol
ISubscriber<IMessage>.Filter
ISubscriber<IMessage>.Handle
metaData
policy
ProcessSyncResult
RetrieveMultipleFromCache<T>


IConfService
ISubscriber<IMessage>
ConfService
Sealed Class
→ AbstractLogEnabled
public
BeginRetrieveMultipleObjects
Cache
CreateMultipleObjectsFromXml (+ 1 overload)
CreateObjectFromXml (+ 1 overload)
DeleteObject
EndRetrieveMultipleObjects<T>
MetaData
Policy
Protocol
RefreshObject
Register (+ 2 overloads)
RetrieveMultipleObjects<T>
RetrieveObject (+ 2 overloads)
SaveObject
Subscribe (+ 1 overload)
Unregister (+ 1 overload)
Unsubscribe
protected
OnEnableLogging
internal
ConfService
EventService
GetChildLogger
ResolveLink
SetCache
SetPolicy
Unsubscribe
private
AddToCache
cache
confProtocol
ISubscriber<IMessage>.Filter
ISubscriber<IMessage>.Handle
metaData
policy
ProcessSyncResult
RetrieveMultipleFromCache<T>
factory
CfgObjectFactory
Class
internal
BeginRetrieveMultipleObjects
CfgObjectFactory
EndRetrieveMultipleObjects<T>
RetrieveMultipleObjects<T>
RetrieveObject (+ 1 overload)
private
confProtocol
eventService
ISubscriptionService<ConfEvent>
ISubscriber<IMessage>
EventService
Class
public
AddRetrieveWait
Register (+ 2 overloads)
Unregister (+ 1 overload)
internal
EventService
Filter
RemoveRetrieveWait
Subscribe (+ 1 overload)
Unsubscribe
private
confProtocol
confProtocol_Closed
CreateDelta
ISubscriber<IMessage>.Filter
ISubscriber<IMessage>.Handle
messageFilter
messageWaitList
myBrokerService


CfgApplicationQuery
Class
CfgFilterBasedQuery
IConfService
ISubscriber<IMessage>
ConfService
Sealed Class
AbstractLogEnabled
public
BeginRetrieveMultipleObjects
Cache
CreateMultipleObjectsFromXml (+ 1 overload)
CreateObjectFromXml (+ 1 overload)
EndRetrieveMultipleObjects<T>
MetaData
Policy
RetrieveMultipleObjects<T>
RetrieveObject (+ 2 overloads)
Unsubscribe
protected
OnEnableLogging
internal
EventService
GetChildLogger
ResolveLink
SetCache
SetPolicy
Unsubscribe
private
AddToCache
cache
ISubscriber<IMessage>.Filter
ISubscriber<IMessage>.Handle
metaData
policy
ProcessSyncResult
RetrieveMultipleFromCache<T>
ICfgQuery
CfgQuery
Class
public
BeginExecute
CfgQuery (+ 1 overload)
private
confService
ICfgQuery.EndExecute<T>
ICfgQuery.Execute<T>
ICfgQuery.ExecuteSingleResult<T>

Using the Application Block

Installing the Configuration Object Model Application Block

Before you install the Configuration Object Model Application Block, it is important to review the software requirements established in the Genesys Supported Operating Environment Reference Manual.

Configuring the Configuration Object Model Application Block

In order to use the QuickStart application, you will need to set up the XML configuration file that comes with the application block. This file is located at `Quickstart\app.config`. This is what the contents look like:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Uri" value="tcp://yourhost:yourport"/>

    <add key="ClientName" value="StarterApp"/>

    <add key="ClientType" value="CFGAgentDesktop"/>

    <add key="UserName" value="default"/>

    <add key="Password" value="password"/>

  </appSettings>
</configuration>
```

Follow the instructions in the comments and save the file.

Building the Configuration Object Model Application Block

Tip

Starting with release 8.5.0, the common interfaces for COM Application Block and Message Broker have been moved to an individual `Genesyslab.Platform.ApplicationBlocks.Commons.dll` file.

The Platform SDK distribution includes a `Genesyslab.Platform.ApplicationBlocks.Commons.dll` file that you can use as is. This file is located in the `bin` directory at the root level of the Platform SDK directory. To build your own copy of this application block, follow the instructions below:

To build the Configuration Object Model Application Block:

1. Open the <Platform SDK Folder>\ApplicationBlocks\Com folder.
2. Double-click Com.sln.
3. Build the solution.

Using the QuickStart Application

The easiest way to start using the Configuration Object Model Application Block is to use the bundled QuickStart application. This application ships in the same folder as the application block.

To run the QuickStart application:

1. Open the `<Platform SDK Folder>\ApplicationBlocks\Com` folder.
2. Double-click `ComQuickStart.sln`.
3. Build the solution.
4. Find the executable for the QuickStart application, which will be at `<Platform SDK Folder>\ApplicationBlocks\Com\QuickStart\bin\Debug\ComQuickStart.exe`.
5. Double-click `ComQuickStart.exe`.

Editing Capacity Rules

The Configuration Object Model Application Block includes the `CapacityRuleHelper` class (introduced in release 8.1.4) which allows you to edit and update Capacity Rules. This helper class presents an XML representation for `CfgScript` objects of type `CapacityRule`, which can be updated and saved to edit existing Capacity Rules.

An example of how to edit capacity rules is provided below.

```
IConfService service = ConfServiceFactory.CreateConfService(protocol);
service.Protocol.Open();
CfgScriptQuery query = new CfgScriptQuery(service);
CfgScript script = service.RetrieveObject(query);
```

An instance of the `CapacityRuleHelper` class can now be created with static method `Create`. This method validates the input script object and can throw an exception (`CapacityRuleException`) if the object is not valid. Once the instance is created, the `XMLPresentation` property allows you access to Capacity Rules.

```
CapacityRuleHelper helper = CapacityRuleHelper.Create(script);
Document doc = helper.XMLPresentation;
// edit xml document here
```

The `XMLPresentation` property is able to be saved into the `CapacityRuleHelper` class instance. Once changes have been made to the XML document, apply your changes using the following code:

```
helper.XMLPresentation = doc;
helper.Script.Save();
service.Protocol.Close();
ConfServiceFactory.ReleaseConfService(service);
```

Introduction to the Configuration Layer Objects

Once you have reviewed the information in this section, you can look at the Configuration Layer Objects Reference Guide for detailed descriptions of available objects and enumerations.

The Genesys Configuration Layer is a database containing information about the objects in your contact center environment. You may need to get information about these objects. You may also want to add, update, or delete them. The Configuration Platform SDK gives you the means to do that.

This article contains information that is common to all of these Configuration Layer objects.

General Parameters

The following parameters are common to objects of all types. They will not be described again in the listings for individual objects.

- `DBID` — An identifier of this object in the Configuration Database. Generated by Configuration Server, it is unique within an object type. Identifiers of deleted objects are not used again. Read-only.
- `state` — Current object state. Mandatory. Refer to `CfgObjectState` in section Variable Types.

Tip

Change in the state of a parent object will cause the states of all its child objects to change accordingly. Configuration Server will provide a notification for each elementary change. Changing the state of a parent object will not be allowed unless the client application has privileges to change all of the child objects of this parent object.

- `userProperties` — In objects, a pointer to the list of user-defined properties. In delta objects, a pointer to a list of user-defined properties added to the existing list. Parameter `userProperties` has the following structure: Each key-value pair of the primary list (`TKVList *userProperties`) uses the key for the name of a user-defined section, and the value for a secondary list, that also has the `TKVList` structure and specifies the properties defined within that section. Each key-value pair of the secondary list uses the key for the name of a user-defined property, and the value for its current setting. User properties can be defined as variables of integer, character, or binary type. Names of sections must be unique within the primary list. Names of properties must be unique within the secondary list.

Tip

> Configuration Server is not concerned with logical meanings of user-defined sections, properties, or their values.

- `deletedUserProperties` — A pointer to the list of deleted user-defined properties. Has the same structure as parameter `userProperties` above. A user-defined property is deleted by specifying the name of the section that this property belongs to, and the name of the property itself with any value. A whole section is deleted by specifying the name of that section and an empty secondary list.

- `changedUserProperties` — A pointer to the list of user-defined properties whose values have been changed. Has the same structure as parameter `userProperties` above. A value of a user-defined property is changed by specifying the name of the section that this property belongs to, the name of the property itself, and a new value of that property.

- `flexibleProperties` — In objects, a pointer to the list of additional properties. In delta objects, a pointer to a list of user-defined properties added to the existing list. This parameter has the following structure: Each key-value pair of the primary list (`TKVList * flexibleProperties`) uses the key for the name of the section, and the value for a secondary list, that also has the `TKVList` structure and specifies either properties defined within that section or another section name. Each key-value pair of the secondary list uses the key for the name of a property, and the value for its current setting. Properties can be defined as variables of integer, character, or binary type or as the name of another list of properties. Names of sections must be unique within the primary list. Names of properties must be unique within the list. The data structure within the `flexibleProperties` property is object-type specific and hard-coded within Configuration Server. Each key-value in the `TKVList * flexibleProperties` is controlled and processed by Configuration Server only in the same manner as any other property in contrast with user-properties the contents of which are not Configuration Server concerned. If the structure of the property's `Extension` is not specified, the value is `NULL`. For more information, see the detailed object descriptions in this document.

Configuration Object Association

Configuration Objects can be associated with each other in a number of different ways that can be generally classified as follows:

- Parent-child relationship, where a child object cannot be created without a parent and will be deleted automatically if its parent object is deleted. Most of the object types will have an explicit reference to their parents which is marked with an asterisk in the specification below. For the object types that do not have such a reference, it is implied that their parent is the Service Provider (that is, the imaginary tenant with `DBID = 1`).
- Exclusive association, where an object cannot be associated in the same manner with more than one other object.
- Non-exclusive association, where an object can be associated in the same manner with more than one other object. Unless expressly noted otherwise, a reference to the DBID of another object without an asterisk indicates a non-exclusive assignment.

The parameters of all object-related structures are optional unless otherwise noted. However, all variables of character type must be initialized at the time an object is created. The variables of character type that are not mandatory may be initialized with an empty string (the recommended default value unless otherwise noted). The variables of character type that are mandatory may not be initialized with an empty string. Variables of character type may accept values of up to 255 symbols in length unless otherwise noted. The recommended default value for optional parameters of other types is `zero` or `NULL`, unless otherwise noted.

Filters

Filters are used to specify more precisely the kind of information that the client application is interested in. Filters reduce both volumes of data communicated by Configuration Server and data-processing efforts on the client side. Filters are structured as key-value pairs where the value of each key defines a certain condition of data selection. Filter keys are defined as variables of integer type unless otherwise noted.

> **Important**
> Although your application can use "and" to combine multiple filters when retrieving a set of matching configuration objects, specifying a DBID value as one of the filters causes all other filters in that request to be ignored. This is by design, as only a single configuration object can match the specified DBID value. However, this behavior could create unexpected results if your application intended to use filters as a method for checking whether a known configuration object also matches additional filter values.

Here is a list of common filter types:

- `folder_dbid` — A unique identifier of a folder. If specified, Configuration Server will return information only about objects of specific type located under specified folder. See also the description of the `ConfGetObjectInfo` function.
- delegate_dbid — A unique identifier of an account on behalf of which current query is to be executed. Produced result set will be calculated using a superposition of the registered account permissions and that passed in `delegate_dbid` filter. Must be used in conjunction with delegate_type filter in order to specify account type (`CFGPerson` or `CFGAccessGroup`).
- `delegate_type` — Object type of the account (`CFGPerson` or `CFGAccessGroup`) on behalf of which the current query is to be executed. Must be used in conjunction with `delegate_dbid`.
- `object_path` — A flag that causes Configuration Server to return a full path of the object in the folder hierarchy for every object in the result set. The path string will be returned in the `cfgDescription` field of the `CFGObjectInfo` event.
- `cmp_insensitive` — A flag that causes Configuration Server to perform case-insensitive comparison of string values in the filter. Supported from Configuration Server 7.2.000.00.
- `read_folder_dbid` — A flag that causes Configuration Server to return a Folder DBID for every object in the result set. The folder will be returned in the `cfgExtraInfo3` field of the `CFGObjectInfo` event. Supported from Configuration Server 7.2.000.00.

Stat Server

Stat Server tracks information about customer interaction networks (contact center, enterprise-wide, or multi-enterprise telephony and computer networks). It also converts the data accumulated for directory numbers (DNs), agents, agent groups, and non-telephony-specific object types, such as email and chat sessions, into statistically useful information, and passes these calculations to other software applications that request data. For example, Stat Server sends data to Universal Routing Server (URS), because Stat Server reports on agent availability. You can also use Stat Server's numerical statistical values as routing criteria.

Stat Server provides contact center managers with a wide range of information, allowing organizations to maximize the efficiency and flexibility of customer interaction networks. For more information about Stat Server, consult the Reporting Technical Reference 8.0 Overview and the Stat Server 8.5 User's Guide.

You can use the Platform SDK to write Java or .NET applications that gather statistical information from Stat Server. These applications may be fairly simple or quite advanced. This article shows how to implement the basic functions you will need to write a simple Statistics application.

A Typical Statistics Application

There are many ways in which you might need to use data from Stat Server, but in most cases, you will use three types of requests:

- `RequestOpenStatistic` and `RequestOpenStatisticEx` are used to ask Stat Server to start sending statistical information to your application. `RequestOpenStatistic` allows you to request information about a statistic that has already been defined in the Genesys Configuration Layer, while you can use `RequestOpenStatisticEx` to define your own statistics dynamically.
- You can use `RequestPeekStatistic` to get the value of a statistic that has already been opened using either `RequestOpenStatistic` or `RequestOpenStatisticEx`. Since it can take a while for certain types of statistical information to be sent to your application, this can be useful if you are writing an application—such as a wallboard application, for instance—for which you would like statistical values to be displayed immediately.
- Use `RequestCloseStatistic` to tell Stat Server that you no longer need information about a particular statistic.

Tip

When you use `RequestOpenStatistic` and `RequestOpenStatisticEx`, you have to specify a `ReferenceId`, which is a unique integer that allows Stat Server and your application to distinguish between different sets of statistical information. You must also enter this integer in the `StatisticId` field for any request that refers to the statistics generated on the basis of your Open request. For example, if you sent a request for "TotalNumberInboundCalls" for agent 001, you might give the `RequestOpenStatistic` a `ReferenceId` of 333001. A similar request for agent 002 might have a `ReferenceId` of `333002`. When you want to peek at the value of "TotalNumberInboundCalls" for agent 001, or close the statistic (or suspend or resume reporting on the statistic), you need to specify a `StatisticId` of `333001` for each of these requests.

Java

Connecting to Stat Server

As mentioned in the article on the architecture, the Platform SDKs uses a message-based architecture to connect to Genesys servers. This section describes how to connect to Stat Server, based on the material in the article on Connecting to a Server.

After you have set up your import statements, the first thing you need to do is create a `StatServerProtocol` object:

```
[Java]

StatServerProtocol statServerProtocol =
  new StatServerProtocol(
    new Endpoint(
      statServerEPName,
      host,
      port));
statServerProtocol.setClientName(clientName);
```

You can also configure your ADDP and warm standby settings at this point, following the example shown in the Connecting to a Server article.

Once your configuration is complete, open the connection to Stat Server:

```
[Java]

try {
        statServerProtocol.open();
} catch (InterruptedException e) {
        e.printStackTrace();
} catch (ProtocolException e) {
        e.printStackTrace();
}
```

Working with Statistics

The Stat Server application object in the Genesys Configuration Layer comes with many predefined statistics. You can also define your own statistics using the options tab of this application object. The Platform SDK allows you to get information about any of these statistics by using

RequestOpenStatistic. There may be times, however, when you want your application to be able to create new types of statistics dynamically. The Platform SDK also supports this, with the use of RequestOpenStatisticEx.

This section will show you how to use RequestOpenStatistic to get information on a predefined statistic. After that, we will give an example of how to use RequestOpenStatisticEx.

The first thing you need to do to use RequestOpenStatistic is to create the request:

```
[Java]

RequestOpenStatistic requestOpenStatistic
        = RequestOpenStatistic.create();
```

Now you need to describe the *statistics object*, that is, the object you are monitoring. This description consists of the object's Configuration Layer ID and object type, and the tenant ID and password:

```
[Java]

StatisticObject object = StatisticObject.create();
object.setObjectId("Analyst001");
object.setObjectType(StatisticObjectType.Agent);
object.setTenantName("Resources");
object.setTenantPassword("");
```

Next, you will specify the StatisticType property, which must correspond to the name of the statistic definition that appears in the options tab. In this case, we are asking for the total login time for an agent identified as "Analyst001":

```
[Java]

StatisticMetric metric = StatisticMetric.create();
metric.setStatisticType("TotalLoginTime");
```

Now you can specify the desired Notification settings. The Statistics Platform SDK supports four ways of gathering statistics:

1. NoNotification allows you to retrieve statistics when you want them.
2. Periodical means Stat Server reports on statistics based on the time period you request.
3. Immediate means Stat Server reports on statistics whenever a statistical value changes. For time-related statistics, Immediate means that Stat Server will report the current value whenever a statistical value changes, but it will also report that value periodically, using the specified notification frequency.
4. Reset means Stat Server reports the current value of a statistic right before setting the statistical value to zero (0).

In this case, we are interested in receiving statistics on a regular basis, so we have asked for a notification mode of Periodical, with updates every 5 seconds, using a GrowingWindow statistic interval. For more information on notification modes, see the section on Notification Modes in the Stat Server 8.5 User's Guide. For more information on statistic intervals, see the section on TimeProfiles in the same guide.

```
[Java]

Notification notification = Notification.create();
notification.setMode(NotificationMode.Periodical);
notification.setFrequency(5);
```

At this point, you can add the information about the statistic object and your notification settings to the request:

```
[Java]

requestOpenStatistic.setStatisticObject(object);
requestOpenStatistic.setStatisticMetric(metric);
requestOpenStatistic.setNotification(notification);
```

Before sending this request, you have to assign it an integer that uniquely identifies it, so that Stat Server and your application can easily distinguish it from other sets of statistical information. Note that you will also need to enter this integer in the `StatisticId` field for any subsequent requests that refer to the statistics generated on the basis of the Open request.

> **Tip**
>
> `ReferenceId` is a unique integer that is specified for identification of requested statistics. If no value is set then this property is assigned automatically just before sending a message; however, if the property has already been assigned then it will not be modified. If you specify this property on your own, you should guarantee its uniqueness. StatServer's behavior was corrected (starting from releases 8.1.000.44 and 8.1.200.14 in corresponding families) so that if two requests are sent with the same `ReferenceId` then an `EventError` message is returned for the second request.

```
[Java]

requestOpenStatistic.setReferenceId(2);
```

Now you can send the request:

```
[Java]

System.out.println("Sending:\n" + requestOpenStatistic);
statServerProtocol.send(requestOpenStatistic);
```

After Stat Server sends the `EventStatisticOpened` in response to this request, it will start sending `EventInfo` messages every 5 seconds. You need to set up an event handler to receive these messages, as discussed in the the Event Handling article.

This is what one such message might look like:

```
'EventInfo' ('2')
message attributes:
REQ_ID [int]    = 4
USER_REQ_ID [int] = -1
TM_SERVER [int] = 1244412448
TM_LENGTH [int] = 0
LONG_VALUE [int] = 0
VOID_VALUE [object] = AgentStatus {
        AgentId = Analyst001
        AgentStatus = 23
        Time = 1240840034
        PlaceStatus =   PlaceStatus = 23
        Time = 1240840034
        LoginId = LoggedOut
}
```

Creating Dynamic Statistics

As mentioned above, there may be times when you want to get statistical information that has not already been defined in the Configuration Layer. In cases like that, you can use `RequestOpenStatisticEx`. Before you do, however, you should make sure you understand several topics covered in the Reporting Technical Reference 8.0 Overview and the Stat Server 8.5 User's Guide, including the use of masks.

The first things you need to do in order to use `RequestOpenStatisticEx` are similar to what we did in the previous section. You will start by creating the request and specifying the statistic object and notification mode, which you will add to the request:

```
[Java]

RequestOpenStatisticEx request =
        RequestOpenStatisticEx.create();

StatisticObject object = StatisticObject.create();
object.setObjectId("Analyst001");
object.setObjectType(StatisticObjectType.Agent);
object.setTenantName("Resources");
object.setTenantPassword("");

Notification notification = Notification.create();
notification.setMode(NotificationMode.Immediate);

request.setNotification(notification);
request.setStatisticObject(object);
```

Now, instead of requesting a pre-defined statistic type, you need to set up your own masks, as described in the section on "Metrics: Their Composition and Definition" in the Reporting Technical Reference 8.0 Overview. The following mask and statistic metric settings give the Current State for the agent mentioned above:

```
[Java]

DnActionMask mainMask = ActionsMask.createDNActionsMask();
mainMask.setBit(DnActions.WaitForNextCall);
mainMask.setBit(DnActions.CallDialing);
mainMask.setBit(DnActions.CallRinging);
mainMask.setBit(DnActions.NotReadyForNextCall);
mainMask.setBit(DnActions.CallOnHold);
mainMask.setBit(DnActions.CallUnknown);
mainMask.setBit(DnActions.CallConsult);
mainMask.setBit(DnActions.CallInternal);
mainMask.setBit(DnActions.CallOutbound);
mainMask.setBit(DnActions.CallInbound);
mainMask.setBit(DnActions.LoggedOut);

DnActionMask relMask = ActionsMask.createDNActionsMask();

StatisticMetricEx metric = StatisticMetricEx.create();
metric.setCategory(StatisticCategory.CurrentState);
metric.setMainMask(mainMask);
metric.setRelativeMask(relMask);
metric.setSubject(StatisticSubject.DNStatus);

request.setStatisticMetricEx(metric);
```

Once you have set up the masks and the statistic metric, you can create a `ReferenceId` and send the request:

```
[Java]

request.setReferenceId(anIntThatYouSpecify);

System.out.println("Sending:\n" + request);
Message response = statServerProtocol.request(request);
System.out.println("Received:\n" + response);
```

Current Target State Events

You can use `RequestGetStatisticEx` and `RequestOpenStatisticEx` to set up the same type of current target state definitions that Universal Routing Server (URS) uses. (You can also set these up using Configuration Manager.) When this type of request has been sent, Stat Server sends some additional event types:

- EventCurrentTargetStateSnapshot
- EventCurrentTargetStateTargetUpdated
- EventCurrentTargetStateTargetAdded
- EventCurrentTargetStateTargetRemoved

The `Snapshot` event is returned in response to the open, while the `Updated` event is sent as state changes occur. In a situation where you open a CurrentTargetState-based statistic against an agent group, the `Added` and `Removed` messages occur when an agent is added to or removed from an agent group — it would behave in a similar fashion for place groups.

Here is the output from a typical request:

```
'EventCurrentTargetStateSnapshot' (17) attributes:
      TM_LENGTH [int] = 0
      USER_REQ_ID [int] = -1
      LONG_VALUE [int] = 0
      CURRENT_TARGET_STATE_INFO [CurrentTargetState] = CurrentTargetStateSnapshot (size=1) [
    [0] CurrentTargetStateInfo {
      AgentId = Analyst001
      AgentDbId = 101
      LoginId = null
      PlaceId = null
      PlaceDbId = 0
      Extensions = KVList:
'VOICE_MEDIA_STATUS' [int] = 0
'AGENT_VOICE_MEDIA_STATUS' [int] = 0
    }
  ]

      REQ_ID [int] = 5
      TM_SERVER [int] = 1245182089
```

Peeking at a Statistic

There may be times when you need to get immediate information on a statistic you have opened. For example, you may want to initialize a wallboard display. In that case, you can use `RequestPeekStatistic`. Note that Stat Server does not send a handshake event when you use this request, so you should use the send method rather than the request method when you use it. Note also that you need to use the `StatisticId` property to provide the `ReferenceId` of the `RequestOpenStatistic` or `RequestOpenStatisticEx` associated with the statistic you want information on:

Tip

If you use the request method on a `RequestPeekStatistic`, your request will time out and receive null, rather than retrieving the desired information from Stat Server.

```
[Java]

RequestPeekStatistic req = RequestPeekStatistic.create();
req.setStatisticId(2);

System.out.println("Sending:\n" + req);
statServerProtocol.send(req);
```

Suspending Notification

Because there are times when you do not need to collect information on a statistic for a while, the Platform SDK has requests that allow you to suspend and resume notification. These requests are like the peek request in that Stat Server does not send a handshake event when you use them, so you should use the send method rather than the request method when you use these requests. Note also that you need to use the `StatisticId` property of these requests to provide the `ReferenceId` of the `RequestOpenStatistic` or `RequestOpenStatisticEx` associated with the statistic you want information on. Here is how to suspend notification:

```
[Java]

RequestSuspendNotification req = RequestSuspendNotification.create();
req.setStatisticId(2);

System.out.println("Sending:\n" + req);
statServerProtocol.send(req);
```

Use code like this to resume notification:

```
[Java]

RequestResumeNotification req = RequestResumeNotification.create();
req.setStatisticId(2);

System.out.println("Sending:\n" + req);
statServerProtocol.send(req);
```

Closing the Statistic and the Connection

When you are finished communicating with Stat Server, you should close the statistics that you have opened and close the connection, in order to minimize resource utilization:

```
[Java]

RequestCloseStatistic req = RequestCloseStatistic.create();
req.setStatisticId(2);

System.out.println("Sending:\n" + req);
statServerProtocol.send(req);

...

statServerProtocol.beginClose();
```

.NET

Connecting to Stat Server

As mentioned in the article on the architecture, the Platform SDKs uses a message-based architecture to connect to Genesys servers. This section describes how to connect to Stat Server, based on the material in the article on Connecting to a Server.

After you have set up `using` statements, the first thing you need to do is create a `StatServerProtocol` object:

```
[C#]

StatServerProtocol statServerProtocol =
       new StatServerProtocol(new Endpoint(statServerUri));
statServerProtocol.ClientId = clientID;
statServerProtocol.ClientName = clientName;
```

You can also configure your ADDP and warm standby settings at this point, as described in the Connecting to a Server article.

Once you have finished configuring your protocol object, open the connection to Stat Server:

```
[C#]

statServerProtocol.Open();
```

Working with Statistics

The Stat Server application object in the Genesys Configuration Layer comes with many predefined statistics. You can also define your own statistics using the options tab of this application object. The

Platform SDK allows you to get information about any of these statistics by using `RequestOpenStatistic`. There may be times, however, when you want your application to be able to create new types of statistics dynamically. The Platform SDK also supports this, with the use of `RequestOpenStatisticEx`.

This section will show you how to use `RequestOpenStatistic` to get information on a predefined statistic. After that, we will give an example of how to use `RequestOpenStatisticEx`.

The first thing you need to do to use `RequestOpenStatistic` is to create the request:

```
[C#]

var requestOpenStatistic = RequestOpenStatistic.Create();
```

Now you need to describe the *statistics object*, that is, the object you are monitoring. This description consists of the object's Configuration Layer ID and object type, and the tenant ID and password:

```
[C#]

requestOpenStatistic.StatisticObject = StatisticObject.Create();
requestOpenStatistic.StatisticObject.ObjectId = "Analyst001";
requestOpenStatistic.StatisticObject.ObjectType = StatisticObjectType.Agent;
requestOpenStatistic.StatisticObject.TenantName = "Environment";
requestOpenStatistic.StatisticObject.TenantPassword = "";
```

Next, you will specify the `StatisticMetric` property for this statistic. A `StatisticMetric` contains information including the `StatisticType` (which must correspond to the name of the statistic definition that appears in the options tab), along with the required `TimeRangeLeft` and `TimeRangeRight` parameters.

In this case, we are asking for the total login time for an agent identified as "Analyst001":

```
[C#]

requestOpenStatistic.StatisticMetric = StatisticMetric.Create();
requestOpenStatistic.StatisticMetric.StatisticType = "TotalLoginTime";
requestOpenStatistic.StatisticMetric.TimeProfile = "Default";
// Note: if no time profile is provided, then the default is used automatically
```

Finally, specify the desired Notification settings. The Statistics Platform SDK supports four ways of gathering statistics:

- NoNotification allows you to retrieve statistics when you want them.
- Periodical means Stat Server reports on statistics based on the time period you request.
- Immediate means Stat Server reports on statistics whenever a statistical value changes. For time-related statistics, Immediate means that Stat Server will report the current value whenever a statistical value changes, but it will also report that value periodically, using the specified notification frequency.
- Reset means Stat Server reports the current value of a statistic right before setting the statistical value to zero (0).

In this case, we are interested in receiving statistics on a regular basis, so we have asked for a notification mode of Periodical, with updates every 5 seconds. For more information on notification modes, see the section on Notification Modes in Stat Server 8.5 User's Guide.

```
[C#]
```

```
requestOpenStatistic.Notification = Notification.Create();
requestOpenStatistic.Notification.Mode = NotificationMode.Periodical;

requestOpenStatistic.Notification.Frequency = 5; // seconds
```

Before sending this request, you have to assign it an integer that uniquely identifies it, so that Stat Server and your application can easily distinguish it from other sets of statistical information. Note that you will also need to enter this integer in the `StatisticId` field for any subsequent requests that refer to the statistics generated on the basis of the Open request.

> **Tip**
>
> `ReferenceId` is a unique integer that is specified for identification of requested statistics. If no value is set then this property is assigned automatically just before sending a message; however, if the property has already been assigned then it will not be modified. If you specify this property on your own, you should guarantee its uniqueness. StatServer's behavior was corrected (starting from releases 8.1.000.44 and 8.1.200.14 in corresponding families) so that if two requests are sent with the same `ReferenceId` then an `EventError` message is returned for the second request.

```
[C#]

requestOpenStatistic.ReferenceId = 3; // Must be unique and is included as StatisticId in
// Peek/Close for the stat
```

Now you can send the request:

```
[C#]

Console.WriteLine("Sending:\n{0}", requestOpenStatistic);
var response =
        statServerProtocol.Request(requestOpenStatistic);
Console.WriteLine("Received:\n{0}", response);

if (response == null || response.Id != EventStatisticOpened.MessageId)
{
        // Open failed, proper error handling goes here
        throw new Exception("RequestOpenStatistic failed.");
}

var @event = response as EventStatisticOpened;
```

After Stat Server sends the `EventStatisticOpened` in response to this request, it will start sending EventInfo messages every 5 seconds. You need to set up an event handler to receive these messages, as discussed in the the Event Handling article.

This is what one such message might look like:

```
[C#]

'EventInfo' ('2')
message attributes:
REQ_ID [int]    = 4
USER_REQ_ID [int] = -1
TM_SERVER [int] = 1244412448
```

```
TM_LENGTH [int] = 0
LONG_VALUE [int] = 0
VOID_VALUE [object] = AgentStatus {
        AgentId = Analyst001
        AgentStatus = 23
        Time = 1240840034
        PlaceStatus =   PlaceStatus = 23
        Time = 1240840034
        LoginId = LoggedOut
}
```

Creating Dynamic Statistics

As mentioned above, there may be times when you want to get statistical information that has not already been defined in the Configuration Layer. In cases like that, you can use `RequestOpenStatisticEx`. Before you do, however, you should make sure you understand several topics covered in the Reporting Technical Reference 8.0 Overview and the Stat Server 8.5 User's Guide, including the use of masks.

The first things you need to do in order to use `RequestOpenStatisticEx` are similar to what we did in the previous section. You will start by creating the request and specifying the statistic object and notification mode:

```
[C#]

var req = RequestOpenStatisticEx.Create();

req.StatisticObject = StatisticObject.Create();
req.StatisticObject.ObjectId = "Analyst001";
req.StatisticObject.ObjectType = StatisticObjectType.Agent;
req.StatisticObject.TenantName = "Resources";
req.StatisticObject.TenantPassword = "";

req.Notification = Notification.Create();
req.Notification.Mode = NotificationMode.Immediate;
req.Notification.Frequency = 15;
```

Now, instead of requesting a statistic type, you need to set up your own masks, as described in the section on "Metrics: Their Composition and Definition" in the Reporting Technical Reference 8.0 Overview. The following mask and statistic metric settings give the Current State for the agent mentioned above:

```
[C#]

var mainMask = ActionsMask.CreateDnActionMask();
mainMask.SetBit(DnActions.WaitForNextCall);
mainMask.SetBit(DnActions.CallDialing);
mainMask.SetBit(DnActions.CallRinging);
mainMask.SetBit(DnActions.NotReadyForNextCall);
mainMask.SetBit(DnActions.CallOnHold);
mainMask.SetBit(DnActions.CallUnknown);
mainMask.SetBit(DnActions.CallConsult);
mainMask.SetBit(DnActions.CallInternal);
mainMask.SetBit(DnActions.CallOutbound);
mainMask.SetBit(DnActions.CallInbound);
mainMask.SetBit(DnActions.LoggedOut);
```

```
var relMask = ActionsMask.CreateDnActionMask();

req.StatisticMetricEx = StatisticMetricEx.Create();
req.StatisticMetricEx.Category = StatisticCategory.CurrentState;
req.StatisticMetricEx.IntervalLength = 0;
req.StatisticMetricEx.MainMask = mainMask;
req.StatisticMetricEx.RelativeMask = relMask;
req.StatisticMetricEx.Subject = StatisticSubject.DNStatus;
```

Once you have set up the masks and the statistic metric, you can create a `ReferenceId` and send the request:

```
[C#]

req.ReferenceId = referenceIdFromRequestOpenStatistic;

Console.WriteLine("Sending:\n{0}", req);
var response =
        statServerProtocol.Request(req);
Console.WriteLine("Received:\n{0}", response);
```

Current Target State Events

You can use `RequestGetStatisticEx` and `RequestOpenStatisticEx` to set up the same type of current target state definitions that Universal Routing Server (URS) uses. (You can also set these up using Configuration Manager.) When this type of request has been sent, Stat Server sends some additional event types:

- `EventCurrentTargetStateSnapshot`
- `EventCurrentTargetStateTargetUpdated`
- `EventCurrentTargetStateTargetAdded`
- `EventCurrentTargetStateTargetRemoved`

The `Snapshot` event is returned in response to the open, while the `Updated` event is sent as state changes occur. In a situation where you open a `CurrentTargetState`-based statistic against an agent group, the `Added` and `Removed` messages occur when an agent is added to or removed from an agent group — it would behave in a similar fashion for place groups.

Here is the output from a typical request:

```
'EventCurrentTargetStateSnapshot' (17) attributes:
      TM_LENGTH [int] = 0
      USER_REQ_ID [int] = -1
      LONG_VALUE [int] = 0
      CURRENT_TARGET_STATE_INFO [CurrentTargetState] = CurrentTargetStateSnapshot (size=1) [
    [0] CurrentTargetStateInfo {
      AgentId = Analyst001
      AgentDbId = 101
      LoginId = null
      PlaceId = null
      PlaceDbId = 0
      Extensions = KVList:
'VOICE_MEDIA_STATUS' [int] = 0
'AGENT_VOICE_MEDIA_STATUS' [int] = 0
```

```
    }
  ]

      REQ_ID [int] = 5
      TM_SERVER [int] = 1245182089
```

Peeking at a Statistic

There may be times when you need to get immediate information on a statistic you have opened. For example, you may want to initialize a wallboard display. In that case, you can use `RequestPeekStatistic`. Note that Stat Server does not send a handshake event when you use this request, so you should use the Send method rather than the `Request` method when you use it. Note also that you need to use the `StatisticId` property to provide the `ReferenceId` of the `RequestOpenStatistic` or `RequestOpenStatisticEx` associated with the statistic you want information on:

Tip

If you use the `Request` method on a `RequestPeekStatistic`, your request will time out and receive null, rather than retrieving the desired information from Stat Server.

```
[C#]

var requestPeekStatistic = RequestPeekStatistic.Create();
requestPeekStatistic.StatisticId = 3;

Console.WriteLine("Sending:\n{0}", requestPeekStatistic);
statServerProtocol.Send(requestPeekStatistic);
```

Suspending Notification

Because there are times when you do not need to collect information on a statistic for a while, the Platform SDK has requests that allow you to suspend and resume notification. These requests are like the peek request in that Stat Server does not send a handshake event when you use them, so you should use the send method rather than the request method when you use these requests. Note also that you need to use the `StatisticId` property of these requests to provide the `ReferenceId` of the `RequestOpenStatistic` or `RequestOpenStatisticEx` associated with the statistic you want information on. Here is how to suspend notification:

```
[C#]

var requestSuspendNotification = RequestSuspendNotification.Create();
requestSuspendNotification.StatisticId = 3;

Console.WriteLine("Sending:\n{0}", requestSuspendNotification);
statServerProtocol.Send(requestSuspendNotification);
```

Use code like this to resume notification:

```
[C#]

var requestResumeNotification = RequestResumeNotification.Create();
requestResumeNotification.StatisticId = 3;

Console.WriteLine("Sending:\n{0}", requestResumeNotification);
statServerProtocol.Send(requestResumeNotification);
```

Closing the Statistic and the Connection

When you are finished communicating with Stat Server, you should close the statistics that you have opened and close the connection, in order to minimize resource utilization:

```
[C#]

var requestCloseStatistic = RequestCloseStatistic.Create();
requestCloseStatistic.StatisticId = 3;

Console.WriteLine("Sending:\n{0}", requestCloseStatistic);
statServerProtocol.Send(requestCloseStatistic);

...

statServerProtocol.BeginClose();
```

Custom Statistics: Getting Agent State for All Channels

When working with Stat Server, you can configure custom statistics that allow your applications to easily monitor information that might not otherwise be available.

In this article we will build a custom statistic that can be used to return the agent state for all channels, and then look at what the returned EventInfo message might look like.

Configuring a Custom Statistic

Custom statistics can be configured by updating the Stat Server application object in your Genesys environment. This means adding a new section to the application object, and then specifying values for a set of options inside that section that determine how statistics are formed and reported.

Important

For details on how to build your own custom statistics, refer to the Stat Server documentation.

To monitor the agent state for all channels, add the following section to your Stat Server application object:

```
[Custom_CurrentAgentDNState]
Objects=Agent
Category=CurrentState
MainMask=*
Subject=DNAction
```

Resulting EventInfo Object

Once your custom statistic is defined in the Stat Server application object, your application can receive `EventInfo` messages that give your application details about the agent state for all channels. Code from the *Working with Statistics* section of the Stat Server article shows how to subscribe to a statistic; the only change required is using the name defined as part of your custom statistic. (In the example above, the custom statistic name is `Custom_CurrentAgentDNState`.)

A sample `EventInfo` message is provided below for reference:

```
'EventInfo' (2) attributes:
VOID_VALUE [object] = ObjectValue: AgentStatus {
  AgentId = MCR_Agent0
  LoginId = 6000
  Status = 9
  Time = 1392641892
  Place = PlaceStatus {
    PlaceId = Place_6000_MCR
    PlaceStatus = 9
    Time = 1392641892
    DnStatuses = DnStatusesCollection (size=7) [
      [0] DnStatus {
        DN Id = 6000
        SwitchId = Simulator
        GSW DN TYPES = 1
        DN Status = 9
        Time = 1392641892
        Actions = DnActionCollection (size=3) [
          [0] DnAction {
            Action = Monitored
            Time = 1392641870
            ActionDataType = NoData
            ConnectionId = null
            DNIS = null
            ANI = null
            UserData = null
          }
          [1] DnAction {
            Action = LoggedIn
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
          }
          [2] DnAction {
            Action = AfterCallWork
            Time = 1392641892
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
          }
        ]
      }
      [1] DnStatus {
        DN Id = workitem
        SwitchId = null
        GSW DN TYPES = 0
        DN Status = 8
        Time = 1392641870
        Actions = DnActionCollection (size=2) [
          [0] DnAction {
            Action = LoggedIn
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "workitem"
          }
          [1] DnAction {
            Action = NotReadyForNextCall
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "workitem"
          }
        ]
      }
      [2] DnStatus {
        DN Id = email
        SwitchId = null
        GSW DN TYPES = 0
        DN Status = 8
        Time = 1392641870
        Actions = DnActionCollection (size=2) [
          [0] DnAction {
            Action = LoggedIn
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "email"
          }
          [1] DnAction {
            Action = NotReadyForNextCall
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "email"
          }
        ]
      }
      [3] DnStatus {
        DN Id = fax
        SwitchId = null
        GSW DN TYPES = 0
        DN Status = 8
        Time = 1392641870
        Actions = DnActionCollection (size=2) [
          [0] DnAction {
            Action = LoggedIn
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "fax"
          }
          [1] DnAction {
            Action = NotReadyForNextCall
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "fax"
          }
        ]
      }
      [4] DnStatus {
        DN Id = chat
        SwitchId = null
        GSW DN TYPES = 0
        DN Status = 8
        Time = 1392641870
        Actions = DnActionCollection (size=2) [
          [0] DnAction {
            Action = LoggedIn
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "chat"
          }
          [1] DnAction {
            Action = NotReadyForNextCall
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "chat"
          }
        ]
      }
      [5] DnStatus {
        DN Id = sms
        SwitchId = null
        GSW DN TYPES = 0
        DN Status = 8
        Time = 1392641870
        Actions = DnActionCollection (size=2) [
          [0] DnAction {
            Action = LoggedIn
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "sms"
          }
          [1] DnAction {
            Action = NotReadyForNextCall
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "sms"
          }
        ]
      }
      [6] DnStatus {
        DN Id = webform
        SwitchId = null
        GSW DN TYPES = 0
        DN Status = 8
        Time = 1392641870
        Actions = DnActionCollection (size=2) [
          [0] DnAction {
            Action = LoggedIn
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "webform"
          }
          [1] DnAction {
            Action = NotReadyForNextCall
            Time = 1392641870
            ActionDataType = CallData
            ConnectionId = 0000000000000000
            DNIS = null
            ANI = null
            UserData = KVList:
            'MediaType' [str] = "webform"
          }
        ]
      }
    ]
  }
}

TM_LENGTH [int] = 0
LONG_VALUE [int] = 0
USER_REQ_ID [int] = -1
TM_SERVER [int] = 1392641892
REQ_ID [int] = 520
```

Interaction Server

You can use the Open Media Platform SDK to write Java or .NET applications that handle third-party work items in conjunction with the Genesys Interaction Server. You can also use it to work with servers that implement the Genesys External Service Protocol.

This document shows how to implement the basic functions you will need to write simple Interaction Server–based email applications. The first application is a simple media server that submits a new third-party work item. The second application enables an agent to receive a third-party work item, accept it for processing, and mark it done.

Java

Setting Up Interaction Server Protocol Objects

The first thing you need to do to use the Open Media Platform SDK is instantiate a Protocol object. To do that, you must supply information about the server you want to connect with. This example uses an `InteractionServerProtocol` object, supplying its URI, but you can also use name, host, and port information:

```
[Java]

InteractionServerProtocol interactionServerProtocol =
        new InteractionServerProtocol(
                new Endpoint(
                        InteractionServerUri));
```

After instantiating the `InteractionServerProtocol` object, you need to open a connection to Interaction Server:

```
[Java]

interactionServerProtocol.open();
```

Creating a Simple Media Server

The Open Media Platform SDK makes it easy to write a simple server that can submit third-party work items to Interaction Server. To write one, start by entering configuration information:

```
[Java]

// Enter configuration information here:
private String interactionServerName = "<server name>";
private String interactionServerHost = "<host>";
private int interactionServerport = <port>;
private int tenantId = 101;
private String inboundQueue = "<queue>";
private String mediaType = "<media type>";
// End of configuration information.
```

Now you will need to set up a protocol object:

```
[Java]

interactionServerUri = new Uri("tcp://"
        + interactionServerHost + ":"
        + interactionServerport);
InteractionServerProtocol interactionServerProtocol =
        new InteractionServerProtocol(
        new Endpoint(interactionServerName, interactionServerUri));
```

Once you have set up the protocol object, you can tell it the name of your application and let it know that it is a media server:

```
[Java]

interactionServerProtocol.setClientName("EntityListener");
interactionServerProtocol.setClientType(
        InteractionClient.MediaServer);
```

At this point, you can add user data associated with the new interaction:

```
[Java]

KeyValueCollection userData =
        new KeyValueCollection();

userData.add("Subject",
        "New Interaction Created by a Custom Media Server");
```

Now you can open the protocol object, and prepare the interaction to be submitted:

```
[Java]

try
{
        interactionServerProtocol.open();

        RequestSubmit requestSubmit = RequestSubmit.create(
                inboundQueue,
                mediaType,
                "Inbound");
        requestSubmit.setTenantId(tenantId);
        requestSubmit.setInteractionSubtype("InboundNew");
        requestSubmit.setUserData(userData);
```

If you use the `Request` method, you will receive a synchronous response containing a message from Interaction Server:

```
[Java]

Message response =
    interactionServerProtocol.request(requestSubmit);
System.out.println("Response: " + response.messageName() + ".\n\n");
```

Closing the Connection

Finally, when you are finished communicating with Interaction Server, you should close the connection to minimize resource utilization:

```
[Java]

interactionServerProtocol.close();
```

.NET

Setting Up Interaction Server Protocol Objects

The first thing you need to do to use the Open Media Platform SDK is instantiate a Protocol object. To do that, you must supply information about the server you want to connect with. This example uses an `InteractionServerProtocol` object, supplying its URI, but you can also use name, host, and port information:

```
[C#]

InteractionServerProtocol interactionServerProtocol =
        new InteractionServerProtocol(
                new Endpoint(
                        InteractionServerUri));
```

After instantiating the `InteractionServerProtocol` object, you need to open a connection to Interaction Server:

```
[C#]

interactionServerProtocol.Open();
```

Creating a Simple Media Server

The Open Media Platform SDK makes it easy to write a simple server that can submit third-party work items to Interaction Server. To write one, start by entering configuration information:

```
[C#]

// Enter configuration information here:
private string interactionServerName = "<server name>";
private string interactionServerHost = "<host>";
private int interactionServerport = <port>;
private int tenantId = 101;
private string inboundQueue = "<queue>";
private string mediaType = "<media type>";
// End of configuration information.
```

Now you will need to set up a protocol object:

```
[C#]

interactionServerUri = new Uri("tcp://"
        + interactionServerHost + ":"
        + interactionServerport);
InteractionServerProtocol interactionServerProtocol =
        new InteractionServerProtocol(
        new Endpoint(interactionServerName, interactionServerUri));
```

Once you have set up the protocol object, you can tell it the name of your application and let it know that it is a media server:

```
[C#]

interactionServerProtocol.ClientName = "EntityListener";
interactionServerProtocol.ClientType =
        InteractionClient.MediaServer;
```

At this point, you can add user data associated with the new interaction:

```
[C#]

KeyValueCollection userData =
        new KeyValueCollection();

userData.Add("Subject",
        "New Interaction Created by a Custom Media Server");
```

Now you can open the protocol object, and prepare the interaction to be submitted:

```
[C#]

try
{
        interactionServerProtocol.Open();

        RequestSubmit requestSubmit = RequestSubmit.Create(
                inboundQueue,
                mediaType,
                "Inbound");
        requestSubmit.TenantId = tenantId;
        requestSubmit.InteractionSubtype = "InboundNew";
        requestSubmit.UserData = userData;
```

If you use the `Request` method, you will receive a synchronous response containing a message from Interaction Server:

```
[C#]

IMessage response =
    interactionServerProtocol.Request(requestSubmit);
LogAreaRichTextBox.Text = LogAreaRichTextBox.Text
    + "Response: " + response.Name + ".\n\n";
```

Closing the Connection

Finally, when you are finished communicating with Interaction Server, you should close the connection to minimize resource utilization:

```
[C#]

interactionServerProtocol.Close();
```

Additional Topics

As support for the Platform SDKs continues to grow, new topics and examples that illustrate best-practice approaches to common tasks are being added to the documentation. For more information about using the Open Media Platform SDK, including functional code snippets, please read the following topics:

- Creating an Email - This article discusses how to use the Open Media and Contacts Platform SDKs in conjunction to create outgoing email messages. You can also apply the concepts illustrated here to other types of Interactions.

Universal Contact Server

You can use the Contacts Platform SDK to write Java or .NET applications that interact with the Genesys Universal Contact Server (UCS). This allows you to create applications that work with contacts, interactions, and standard responses in a variety of ways - either to create a full-featured agent desktop, or a simple application that forwards email messages.

This document shows how to implement the basic functions you will need to write simple UCS-based applications.

When you are ready to write more complicated applications, take a look at the classes and methods described in the Platform SDK API Reference.

Java

Using the Contacts Protocols

Before using the Contacts Platform SDK, you should include `import` statements that allow access to the Platform SDK Commons and Contacts classes:

```
[Java]

import com.genesyslab.platform.commons.protocol.*;

import com.genesyslab.platform.contacts.protocol.*;
import com.genesyslab.platform.contacts.protocol.contactserver.*;
import com.genesyslab.platform.contacts.protocol.contactserver.events.*;
import com.genesyslab.platform.contacts.protocol.contactserver.requests.*;
```

Setting Up Universal Contact Server Protocol Objects

The first thing you need to do to use the Contacts Platform SDK is instantiate a `UniversalContactServerProtocol` object. To do that, you must supply information about the Universal Contact Server you want to connect with. This example uses the server's name, host, and port information, but you can also use just the URI of your Universal Contact Server:

```
[Java]

UniversalContactServerProtocol ucsConnection = new UniversalContactServerProtocol(new
Endpoint(universalContactServerURI));
```

It is a good practice to always set the application name at the same time that you instantiate a new protocol object. This application name will be used to identify where UCS requests came from.

This is also a good time to add event handlers to the protocol object. See the Event Handling section in this introductory material for code samples and details.

```
[Java]

// Set the ApplicationName property
ucsConnection.setClientName("IntroducingContactsPSDK");
```

After setting up your protocol object, the code to open a connection to the server is simple:

```
[Java]

ucsConnection.open();
```

Tip

Be sure to use proper error handling techniques in your code, especially when working with the protocol connection. To save space, these error handling steps are not shown in this example.

Inserting an Interaction

Now that the protocol connection is open, you are ready to start handling interactions. In this example, we will start by creating a new, outbound email interaction using the `RequestInsertInteraction` request.

Creating a new email interaction object takes a bit of planning. Before you can create and submit the request object, you need to create and configure the following objects:

- `InteractionAttributes` - Sets common attributes for this interaction, specifying details such as the media type and status. All interactions need these attributes to be configured.
- `EmailOutEntityAttributes` - Sets attributes that are specific to an outbound email interaction. For outbound email interactions, this includes the sending and receiving addresses. (The type of interaction you are creating will dictate which object to use here; for example, phone interactions require a `PhoneCallEntityAttributes` object instead of `EmailOutEntityAttributes`.)
- `InteractionContent` - Specifies the actual interaction content. This can be Text, MIME, StructuredText, or StructuredText with MIME content.

The following code snippet shows how each of these objects is configured for our simple outbound email example:

```
[Java]

// Set common interaction attributes
InteractionAttributes attributes = new InteractionAttributes();
attributes.setTenantId(101);
attributes.setMediaTypeId("email");
attributes.setTypeId("Outbound");
attributes.setSubtypeId("OutboundRedirect");
attributes.setStatus(Statuses.Pending);
attributes.setSubject(subjectLine);
attributes.setQueueName(queueName);
attributes.setEntityTypeId(EntityTypes.EmailOut);

// Set entity-specific attributes
EmailOutEntityAttributes outEntityAttributes = new EmailOutEntityAttributes();
outEntityAttributes.setFromAddress(fromAddress);
outEntityAttributes.setToAddresses(forwardAddress);

// Set interaction content
InteractionContent content = new InteractionContent();
content.setText("Email message text...");
```

Tip

The `InteractionAttributes` class stores the `StartDate` property in UTC format. If no value is provided, UCS uses the current date.

Once you have configured the attributes and content for the interaction, it is easy to create and submit the new request:

```
[Java]

// Create the new interaction request
RequestInsertInteraction request = RequestInsertInteraction.create();
request.setInteractionAttributes(attributes);
request.setEntityAttributes(outEntityAttributes);
request.setInteractionContent(content);

// Submit the request
EventInsertInteraction eventInsertIxn = (EventInsertInteraction)
ucsConnection.request(request);
```

Adding an Attachment

Now that you know how to create new email interactions, it is the perfect time to learn how to add attachments to existing interactions. The process for this is much easier than creating a new interaction; you just need to create the request and specify the attachment properties as shown in the code snippet below. Once the request is ready, submit it to your UCS protocol object.

```
[Java]

RequestAddDocument request = RequestAddDocument.create();
request.setInteractionId(eventInsertIxn.getInteractionId());
request.setDocumentId(strDocumentId);
request.setDescription(strDescription);
request.setMimeType(strMimeType);
request.setTheName(strName);
request.setTheSize(intSize);

EventAddDocument eventAddDocument = (EventAddDocument) ucsConnection.request(request);
```

Note that before adding an attachment, you need to have the Interaction ID available. In our example, the Interaction ID was returned as part of the `EventInsertInteraction` from the previous section. Otherwise we would need to submit a `RequestGetInteractionContent` request and then take the Interaction ID from the resulting event.

Getting an Interaction from UCS

Now that we have created a new Interaction and submitted it to UCS, what happens next? The final task we will cover in this introduction is how to return the Interaction and any of its attachments for processing.

The structure of `RequestGetInteractionContent` is very basic: set the Interaction ID you are looking for, and then use the `IncludeAttachments` and `IncludeBinaryContent` properties to specify what type of content you want to be returned. In this example, we will return the attachment created previously and store it in an `Attachment` object for later use.

```
[Java]

RequestGetInteractionContent request = RequestGetInteractionContent.create();
request.setInteractionId(interactionId);
request.setIncludeAttachments(true);

EventGetInteractionContent eventGetIxnContent = (EventGetInteractionContent)
ucsConnection.request(request);

String subject = eventGetIxnContent.getInteractionAttributes().getSubject();
String key = eventGetIxnContent.getInteractionAttributes().getId();
if (eventGetIxnContent.getAttachments() != null) {
        Attachment attachedFile = eventGetIxnContent.getAttachments().get(0);
}
```

Closing the Connection

Finally, when you are finished communicating with the server, you should close the connection and dispose of the object to minimize resource utilization:

```
[Java]

if (ucsConnection.getState() != ChannelState.Closed && ucsConnection.getState() !=
ChannelState.Closing)
        {
                ucsConnection.close();
        }
```

.NET

Using the Contacts Protocols

Before using the Contacts Platform SDK, you should include using statements that allow access to the Platform SDK Commons and Contacts namespaces:

```
[C#]

using Genesyslab.Platform.Commons.Protocols;
using Genesyslab.Platform.Contacts.Protocols;
using Genesyslab.Platform.Contacts.Protocols.ContactServer;
using Genesyslab.Platform.Contacts.Protocols.ContactServer.Requests;
using Genesyslab.Platform.Contacts.Protocols.ContactServer.Events;
```

Setting Up Universal Contact Server Protocol Objects

The first thing you need to do to use the Contacts Platform SDK is instantiate a `UniversalContactServerProtocol` object. To do that, you must supply information about the Universal Contact Server you want to connect with. This example uses the server's name, host, and port information, but you can also use just the URI of your Universal Contact Server:

```
[C#]

UniversalContactServerProtocol ucsConnection;
ucsConnection = new UniversalContactServerProtocol(new Endpoint("UCS", ucsHost, ucsPort));
```

It is a good practice to always set the application name at the same time that you instantiate a new protocol object. This application name will be used to identify where UCS requests came from.

This is also a good time to add event handlers to the protocol object. See the Event Handling article for details.

```
[C#]

// Set the ApplicationName property
ucsConnection.ClientName = "IntroducingContactsPSDK";

// Add event handlers
ucsConnection.Opened += new EventHandler(ucsConnection_Opened);
ucsConnection.Error += new EventHandler(ucsConnection_Error);
ucsConnection.Closed += new EventHandler(ucsConnection_Closed);
```

After setting up your protocol object, the code to open a connection to the server is simple:

```
[C#]

ucsConnection.Open();
```

Tip

Be sure to use proper error handling techniques in your code, especially when working with the protocol connection. To save space, these error handling steps are not shown in this example.

Inserting an Interaction

Now that the protocol connection is open, you are ready to start handling interactions. In this example, we will start by creating a new, outbound email interaction using the `RequestInsertInteraction` request.

Creating a new email interaction object takes a bit of planning. Before you can create and submit the request object, you need to create and configure the following objects:

- `InteractionAttributes` - Sets common attributes for this interaction, specifying details such as the media type and status. All interactions need these attributes to be configured.
- `EmailOutEntityAttributes` - Sets attributes that are specific to an outbound email interaction. For outbound email interactions, this includes the sending and receiving addresses. (The type of interaction you are creating will dictate which object to use here; for example, phone interactions require a `PhoneCallEntityAttributes` object instead of `EmailOutEntityAttributes`.)
- `InteractionContent` - Specifies the actual interaction content. This can be Text, MIME, StructuredText, or StructuredText with MIME content.

The following code snippet shows how each of these objects is configured for our simple outbound email example:

```
[C#]

// Set common interaction attributes
InteractionAttributes attributes = new InteractionAttributes();
attributes.TenantId = 101;
attributes.MediaTypeId = "email";
attributes.TypeId = "Outbound";
attributes.SubtypeId = "OutboundRedirect";
attributes.Status = new NullableStatuses(Statuses.Pending);
attributes.Subject = subjectLine;
attributes.QueueName = queueName;
attributes.EntityTypeId = new NullableEntityTypes(EntityTypes.EmailOut);

// Set entity-specific attributes
EmailOutEntityAttributes outEntityAttributes = new EmailOutEntityAttributes();
outEntityAttributes.FromAddress = fromAddress;
outEntityAttributes.ToAddresses = forwardAddress;

// Set interaction content
InteractionContent content = new InteractionContent();
content.Text = "Email message text...";
```

Tip

The `InteractionAttributes` class stores the `StartDate` property in UTC format. If no

> value is provided, UCS uses the current date.

Once you have configured the attributes and content for the interaction, it is easy to create and submit the new request:

```
[C#]

// Create the new interaction request
RequestInsertInteraction request = RequestInsertInteraction.Create();
request.InteractionAttributes = attributes;
request.EntityAttributes = outEntityAttributes;
request.InteractionContent = content;

// Submit the request
EventInsertInteraction eventInsertIxn = (EventInsertInteraction)
ucsConnection.Request(request);
```

Adding an Attachment

Now that you know how to create new email interactions, it is the perfect time to learn how to add attachments to existing interactions. The process for this is much easier than creating a new interaction; you just need to create the request and specify the attachment properties as shown in the code snippet below. Once the request is ready, submit it to your UCS protocol object.

```
[C#]

RequestAddDocument request = RequestAddDocument.Create();
request.InteractionId = eventInsertIxn.InteractionId;
request.DocumentId = strDocumentId;
request.Description = strDescription;
request.MimeType = strMimeType;
request.TheName = strName;
request.TheSize = intSize;

EventAddDocument eventAddDocument = (EventAddDocument) ucsConnection.Request(request);
```

Note that before adding an attachment, you need to have the Interaction ID available. In our example, the Interaction ID was returned as part of the `EventInsertInteraction` from the previous section. Otherwise we would need to submit a `RequestGetInteractionContent` request and then take the Interaction ID from the resulting event.

Getting an Interaction from UCS

Now that we have created a new Interaction and submitted it to UCS, what happens next? The final task we will cover in this introduction is how to return the Interaction and any of its attachments for processing.

The structure of `RequestGetInteractionContent` is very basic: set the Interaction ID you are looking for, and then use the `IncludeAttachments` and `IncludeBinaryContent` properties to specify what type of content you want to be returned. In this example, we will return the attachment created previously and store it in an `Attachment` object for later use.

```
[C#]

RequestGetInteractionContent request = RequestGetInteractionContent.Create();
request.InteractionId = eventInsertIxn.InteractionId;
request.IncludeAttachments = true;

EventGetInteractionContent eventGetIxnContent = (EventGetInteractionContent)
ucsConnection.Request(request);

String subject = eventGetIxnContent.InteractionAttributes.Subject;
String key = eventGetIxnContent.InteractionAttributes.Id;
if (eventGetIxnContent.Attachments != null)
{
        Attachment attachedFile = eventGetIxnContent.Attachments.Get(0);
}
```

Closing the Connection

Finally, when you are finished communicating with the server, you should close the connection and dispose of the object to minimize resource utilization:

```
[C#]

if (ucsConnection.State != ChannelState.Closed && ucsConnection.State != ChannelState.Closing)
        {
                ucsConnection.Close();
                ucsConnection.Dispose();
        }
```

Usage Tips

This section provides tips and recommended usage details for the Contacts SDK.

Getting Categories Efficiently

Introduced in release 8.0.0

In many cases, the `RequestGetAllCategories` message should not be used because it returns an excessive amount of attached information. Instead, this request can often be replaced by the following combination of services:

1. Call `GetRootCategories` - This returns only the root categories (without including sub-categories) with a limited amount of information attached: Name, Date, Type, Language
2. Manually filter categories - Decide your own filtering on root categories based on the attached information.
3. Call `GetCategory` using the root ID - Options can be passed to return a summary of sub-category or sub-Standard Response, making it possible to return a full tree without content.

4. Call `GetStandardResponse` - To get content, after filtering from `GetCategory` which Standard Response is interesting for your agent (again based on the category attributes).

This combination of services allows you to load the Knowledge Library in a more granular fashion. Prior to release 8.0.0, it was only possible to filter based on Language.

Historically, the `GetAllCategories` was intended for 3-tier servers such as Genesys Desktop. However, this request would cause the server to load the entire Category tree into memory in KeyValueCollection form when preparing the response to a client. In cases where a large Knowledge Library is mixed with a large number of `GetAllCategories` requests in parallel, consuming large amounts of memory.

Additional Topics

As support for the Platform SDKs continues to grow, new topics and examples that illustrate best-practice approaches to common tasks are being added to the documentation. For more information about using the Contacts Platform SDK, including functional code snippets, please read the following topics:

- Creating an Email - This article discusses how to use the Open Media and Contacts Platform SDKs in conjunction to create outgoing email messages.

Creating an Email

This article discusses the general process used to create email messages, and provides suggestions about how you should work with those protocols.

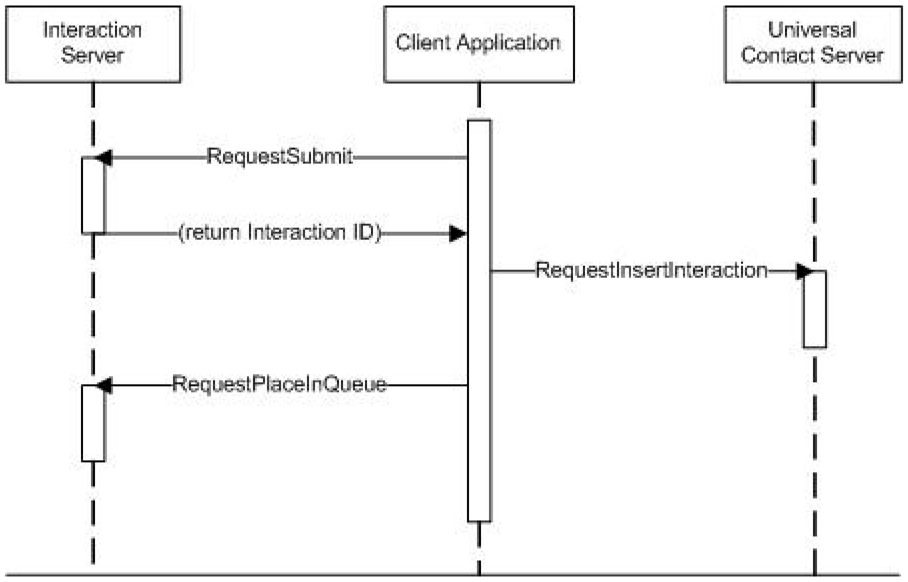
Java

Overview of Creating a New Email Message

To create a new email message, there are four basic steps you should follow:

1. Connect to Genesys Servers - Use the Protocol Manager Application Block to access the appropriate Genesys Servers.
2. Create a new Interaction - Request a new Interaction that will be used to manage the email message within Interaction Server.
3. Store email details in UCS - Once the Interaction is available, you can use the unique `InteractionId` that is returned to create a new UCS entry that contains details and contents for the email message.
4. Place the Interaction in the appropriate queue - When both parts of the email message have been stored, move the Interaction into the correct queue for processing.

A quick overview of these steps, and an outline of the key requests sent to Genesys servers, is shown below.



Tip

The order of the second and third steps can be reversed, if desired, as long as the final UCS entry contains the correct InteractionId value. In this case you would need to update the UCS entry after creating the new Interaction.

The following sections include code snippets that show one possible approach for handling each of these steps. The snippets have been simplified to focus only on code related to Genesys-specific functions.

Connecting to Genesys Servers

When creating and handling email interactions, it is important to remember how email messages are stored within the Genesys environment, and which Genesys servers you are interacting with.

Each email message is stored as two separate pieces: an Interaction, and an entry in the Universal Contact Server (UCS) database. The email is represented as an Interaction so that it can be sorted and processed using queues that have defined behavior. Even though emails are managed through Interaction Server, the actual contents and subject matter of each message must be stored in the UCS database. Any attempt to create or handle email messages will require access to both Genesys Servers: Interaction Server (using the Open Media protocol) and UCS (using the Contacts Platform SDK protocol).

Before writing your email application, some fairly standard code must be added to allow access to these Genesys servers. First, all necessary references and import statements must be added to your project. This includes the two specific protocols mentioned above, together with some common Genesys libraries and the Protocol Manager Application Block.

With those statements in place, we configure the Protocol Manager Application Block to handle communication with Genesys servers using the `ProtocolManagementServiceImpl` object, which is defined and configured as shown below.

```
[Java]

private InteractionServerProtocol interactionServerProtocol;
private UniversalContactServerProtocol contactServerProtocol;

public void connectToProtocols() throws URISyntaxException, ProtocolException
{
    Endpoint interactionServerEndpoint = new Endpoint(new URI("tcp://ixnServer:7005"));
    interactionServerProtocol = new InteractionServerProtocol(interactionServerEndpoint);
    interactionServerProtocol.setClientName("EmailSample");
    interactionServerProtocol.setClientType(InteractionClient.AgentApplication);

    Endpoint contactServerEndpoint = new Endpoint(new URI("tcp://ucsServer:7006"));
    contactServerProtocol = new UniversalContactServerProtocol(contactServerEndpoint);
    contactServerProtocol.setClientName("EmailSample");

    interactionServerProtocol.beginOpen();
    contactServerProtocol.beginOpen();
}
```

For more information about the Protocol Manager Application Block, see the Connecting to a Server article found in this guide.

Creating an Interaction

With connections to the Genesys servers established, we are ready to request a new Interaction that will represent our email message in Interaction Server. You accomplish this by creating a new `RequestSubmit`, setting a few parameters to indicate that this Interaction represents an email message, and then sending the request to Interaction Server with your `ProtocolManagementService` object.

```
[Java]

public void createInteraction(String ixnType, String ixnSubtype, String queue) throws
Exception
{
    RequestSubmit req = RequestSubmit.create();
    req.setInteractionType(ixnType);
    req.setInteractionSubtype(ixnSubtype);
    req.setQueue(queue);
    req.setMediaType("email");

    Message response = interactionServerProtocol.request(req);
    if(response == null || response.messageId() != EventAck.ID) {
          // For this sample, no error handling is implemented
          return;
    }

    EventAck event = (EventAck)response;
    mInteractionId = event.getExtension().getString("InteractionId");
}
```

A full list of properties that need to be set is included in the table below. Note that the `InteractionType` and `InteractionSubtype` properties must match existing business attributes, as specified in Configuration Server.

Property Name	Description
InteractionType	Interaction type for this email message. Must match an Interaction Type Business Attribute, as specified in Configuration Server.
InteractionSubtype	Interaction subtype for this email message. Must match an Interaction Subtype Business Attribute, as specified in Configuration Server.
Queue	Queue that this Interaction will be placed in initially. Must be defined in Configuration Server. When creating a new email Interaction, the initial queue should not process the message (because additional information needs to be stored in UCS first).
MediaType	Primary media type of the interaction that is being submitted to Interaction Server. Intended for Media Server.

Once a response is received from Interaction Server, you can confirm that an `EventAck` response was returned and that the request was processed successfully. If an `EventError` response is returned instead, then you will need to implement some error handling code.

It is also important to save and track the `InteractionId` value of the newly created Interaction. This

ID needs to be specified in UCS entries that hold details related to the email message, and is also required for moving the Interaction to an appropriate queue when you are ready to process the email. In this example we are storing the `InteractionId` value in a simple variable named `mInteractionId`, which is assumed to be defined for your project. In larger samples (or full projects), a more robust way of tracking and handling Interactions may be required.

Storing Email Details in UCS

With the ID of your newly created Interaction available, it is time to store details about the email you are sending in the UCS database.

There are three types of information that must be stored in the UCS database:

- Interaction Attributes - Define details about the related Interaction for this information.
- Entity Attributes - Define where the email message is coming from and going to. You will use `EmailOutEntityAttributes` for storing outbound email messages, and `EmailInEntityAttributes` for storing inbound email messages.
- Interaction Content - Define the actual contents of the email message, including the main text and any MIME attachments.

Creating and configuring a `RequestInsertInteraction` object with this information can be easily accomplished, as shown below.

```
[Java]

public void storeDetails(String ixnType, String ixnSubtype) throws Exception
{
    // Set Interaction Attributes
    InteractionAttributes ixnAttributes = new InteractionAttributes();
    ixnAttributes.setId(mInteractionId);
    ixnAttributes.setMediaTypeId("email");
    ixnAttributes.setTypeId(ixnType);
    ixnAttributes.setSubtypeId(ixnSubtype);
    ixnAttributes.setTenantId(101);
    ixnAttributes.setStatus(Statuses.Pending);
    ixnAttributes.setSubject("Sample email subject");
    ixnAttributes.setEntityTypeId(EntityTypes.EmailOut);

    // Set Entity Attributes
    EmailOutEntityAttributes entityAttributes = new EmailOutEntityAttributes();
    entityAttributes.setFromAddress("sending@email.com");
    entityAttributes.setToAddresses("receiving@email.com");
    entityAttributes.setCcAddresses("copying@email.com");
    ...

    // Set Interaction Content
    InteractionContent content = new InteractionContent();
    content.setText("This is the email body.");
    ...

    // Send the request
    RequestInsertInteraction req = new RequestInsertInteraction();
    req.setInteractionAttributes(ixnAttributes);
    req.setEntityAttributes(entityAttributes);
    req.setInteractionContent(content);

    contactServerProtocol.send(req);
}
```

A list of `InteractionAttributes` properties that need to be set for an email message is provided in the following table. The properties shown for `EmailOutEntityAttributes` and `InteractionContent` represent some of those most commonly used with email. Please check the documentation provided for each class to see a full list of available properties.

Interaction Attribute Name	Description
EntityTypeId	Indicates whether this is an outgoing or incoming email.
Id	Interaction ID of the related Interaction record, created earlier.
MediaTypeId	Primary media type of the Interaction you are submitting to Interaction Server. Intended for Media Server.
Subject	Subject line for this email message.
SubtypeId	Interaction subtype for this email message. Must match an Interaction Subtype Business Attribute, as specified in Configuration Server.
Status	Current status of the email message.
TenantId	ID of the Tenant where this email belongs.
TypeId	Interaction type for this email message. Must match an Interaction Type Business Attribute, as specified in Configuration Server.

Placing the Interaction in the Appropriate Queue

When an Interaction has been created to handle the email, and all content has been stored in the UCS database, you are free to begin processing the message as you would process any normal Interaction. This is accomplished by moving the Interaction that you created into the appropriate queue for email processing, as defined in Interaction Routing Designer.

```
[Java]

public void placeInQueue(String queue) throws Exception
{
    RequestPlaceInQueue req = RequestPlaceInQueue.create();
    req.setInteractionId(mInteractionId);
    req.setQueue(queue);

    interactionServerProtocol.send(req);
}
```

Replying to an Email Message

Replying to an existing email message follows the same basic process outlined above, but requires a few additional parameters to be set in your requests. These changes are described in the following subsections.

Changes to Creating an Interaction

When creating the Interaction, you need to specify one additional parameter before submitting your `RequestSubmit`. Take the `InteractionId` of the Interaction that represents the original email message, and use that value as the `ParentInteractionId` parameter in your request, as shown below:

```
[Java]

RequestSubmit req = RequestSubmit.create();

...

req.setParentInteractionId = parentInteractionId;
```

The following table describes these additional attributes.

Attribute Name	Description
ParentInteractionId	`InteractionId` of a parent email Interaction. Only set this value when replying to an existing email message.

Changes to Storing Email Details in UCS

When storing email details in UCS, you need to specify values for three additional interaction attributes before sending your `RequestInsertInteraction`. These attributes (shown in the code snippet below) provide a link between the parent entry in UCS and any related children, as well as specifying a common thread ID.

```
[Java]

InteractionAttributes ixnAttributes = new InteractionAttributes();

...

ixnAttributes.setParentId(parentInteractionId);
ixnAttributes.setCanBeParent(False);
ixnAttributes.setThreadId(parentThreadId);
```

The table below describes these additional attributes.

Attribute Name	Description
CanBeParent	Boolean value that indicates whether this message can be a parent.
ParentId	Interaction ID for the parent email Interaction.
ThreadId	Unique value that is shared between all UCS entries in an email conversation.

Other Considerations

Although this introduction to creating and handling email messages is not intended to be a comprehensive guide, it is useful to quickly introduce some other considerations and basic concepts regarding how requests are submitted and how errors should be handled.

- The first consideration to take into account is how you submit requests using the Protocol Management Application Block. In the code provided here, a simple send method is used to submit most requests without waiting for a response from the server. However, in more complicated samples or implementations you may need to process responses, or store and use values returned (such as the `InteractionId` in this example) once a request is processed.
  Please read the article on Event Handling provided in this document for a better understanding of how to handle incoming responses in both a synchronous and asynchronous fashion. This allows better error handling to be implemented if a request fails.
- A second consideration to be aware of is how records in Interaction Server and UCS are related when implementing error handling. If you have already created a new Interaction when your `RequestInsertInteraction` fails, then you will need to either resubmit the UCS record or delete the related Interaction by submitting a `RequestStopProcessing`. (If you reversed the steps shown here and created a UCS record first, then the same concept applies for removing that record when a new Interaction request fails.)

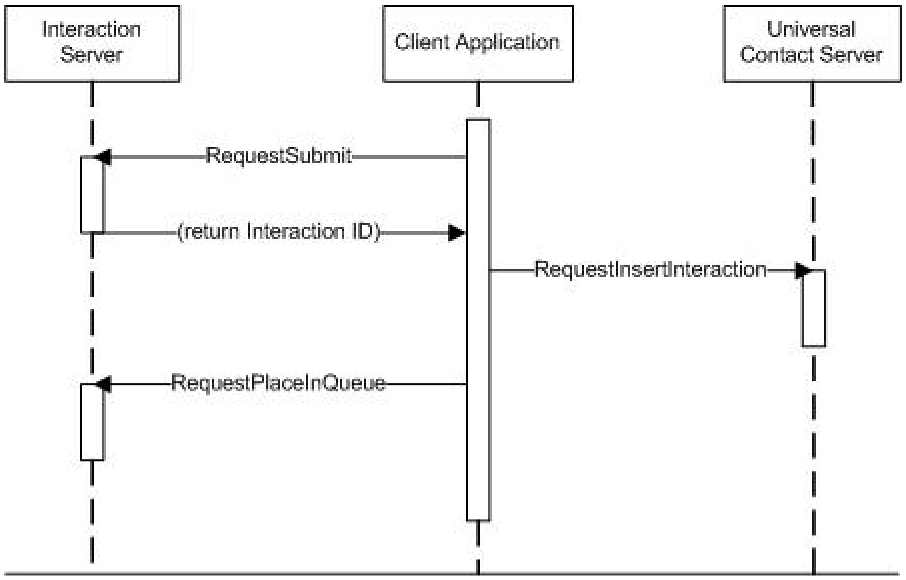
.NET

Overview of Creating a New Email Message

To create a new email message, there are four basic steps you should follow:

1. Connect to Genesys Servers - Use the Protocol Manager Application Block to access the appropriate Genesys Servers.
2. Create a new Interaction - Request a new Interaction that will be used to manage the email message within Interaction Server.
3. Store email details in UCS - Once the Interaction is available, you can use the unique InteractionId that is returned to create a new UCS entry that contains details and contents for the email message.
4. Place the Interaction in the appropriate queue - When both parts of the email message have been stored, move the Interaction into the correct queue for processing.

A quick overview of these steps, and an outline of the key requests sent to Genesys servers, is shown below.



Tip

The order of the second and third steps can be reversed, if desired, as long as the final UCS entry contains the correct `InteractionId` value. In this case you would need to update the UCS entry after creating the new Interaction.

The following sections include code snippets that show one possible approach for handling each of these steps. The snippets have been simplified to focus only on code related to Genesys-specific functions.

Connecting to Genesys Servers

When creating and handling email interactions, it is important to remember how email messages are stored within the Genesys environment, and which Genesys servers you are interacting with.

Each email message is stored as two separate pieces: an Interaction, and an entry in the Universal Contact Server (UCS) database. The email is represented as an Interaction so that it can be sorted and processed using queues that have defined behavior. Even though emails are managed through Interaction Server, the actual contents and subject matter of each message must be stored in the UCS database. Any attempt to create or handle email messages will require access to both Genesys Servers: Interaction Server (using the Open Media protocol) and UCS (using the Contacts Platform SDK protocol).

Before writing your email application, some fairly standard code must be added to allow access to these Genesys servers. First, all necessary references and using statements must be added to your project.

```
[C#]

private InteractionServerProtocol interactionServerProtocol;
private UniversalContactServerProtocol contactServerProtocol;
```

```
public void ConnectToProtocols()
{
        var interactionServerEndpoint = new Endpoint(new Uri("tcp://ixnServer:7005"));
        interactionServerProtocol = new InteractionServerProtocol(interactionServerEndpoint);
        interactionServerProtocol.ClientName = "EmailSample";
        interactionServerProtocol.ClientType = InteractionClient.AgentApplication;

        var contactServerEndpoint = new Endpoint(new Uri("tcp://ucsServer:7006"));
        contactServerProtocol = new UniversalContactServerProtocol(contactServerEndpoint);
        contactServerProtocol.ClientName = "EmailSample";

        interactionServerProtocol.BeginOpen();
        contactServerProtocol.BeginOpen();
}
```

Creating an Interaction

With connections to the Genesys servers established, we are ready to request a new Interaction that will represent our email message in Interaction Server. All you need to do to accomplish this is to create a new `RequestSubmit`, set a few parameters to indicate that this Interaction represents an email message, and then use your `InteractionServerProtocol` object to send that request to Interaction Server.

Unlike other requests shown in this article, `RequestSubmit` is sent using the `BeginRequest` method so that we can receive and process the response from Interaction Server.

```
[C#]

public void CreateInteraction(string ixnType, string ixnSubtype, string queue)
{
var req = RequestSubmit.Create();
req.InteractionType = ixnType;
req.InteractionSubtype = ixnSubtype;
req.MediaType = "email";
req.Queue = queue;

interactionServerProtocol.BeginRequest(req, new AsyncCallback(OnCreateInteractionComplete),
null);
}
```

A full list of properties that need to be set is included in the following table. Note that the `InteractionType` and `InteractionSubtype` properties must match existing business attributes, as specified in Configuration Server.

Property Name	Description
InteractionSubtype	Interaction subtype for this email message. Must match an Interaction Subtype Business Attribute, as specified in Configuration Server.
InteractionType	Interaction type for this email message. Must match an Interaction Type Business Attribute, as specified in Configuration Server.
MediaType	Primary media type of the interaction that is being submitted to Interaction Server. Intended for Media

Property Name	Description
	Server.
Queue	Queue that this Interaction will be placed in initially. Must be defined in Configuration Server. When creating a new email Interaction, the initial queue should not process the message (because additional information needs to be stored in UCS first).

Once a response is received from Interaction Server, you can confirm that an `EventAck` response was returned and that the request was processed successfully. If an `EventError` response is returned instead, then you will need to implement some error handling code.

You should also save and track the `InteractionId` value of the newly created Interaction. This ID needs to be specified in UCS entries that hold details related to the email message, and is also required for moving the Interaction to an appropriate queue when you are ready to process the email.

```
[C#]

private void OnCreateInteractionComplete(IAsyncResult result)
{
    var response = interactionServerProtocol.EndRequest(result);
    if (response == null || response.Id != EventAck.MessageId)
        // for this sample, no error handling is implemented
        return;

    var @event = response as EventAck;
    mInteractionId = (string)@event.Extension["InteractionId"];
}
```

In this example we are storing the `InteractionId` value in a simple variable named `mInteractionId`, which is assumed to be defined for your project. In larger samples (or full projects), a more robust way of tracking and handling Interactions may be required.

Storing Email Details in UCS

With the ID of your newly created Interaction available, it is time to store details about the email you are sending in the UCS database.

There are three types of information that must be stored in the UCS database:

- Interaction Attributes - Define details about the related Interaction for this information.
- Entity Attributes - Define where the email message is coming from and going to. You will use EmailOutEntityAttributes for storing outbound email messages, and EmailInEntityAttributes for storing inbound email messages.
- Interaction Content - Define the actual contents of the email message, including the main text and any MIME attachments.

Creating and configuring a `RequestInsertInteraction` object with this information can be easily accomplished, as shown below.

```
[C#]

public void StoreDetails(string ixnType, string ixnSubtype)
```

```
{
    var req = new RequestInsertInteraction();
    req.InteractionAttributes = new InteractionAttributes()
    {
        Id = mInteractionId,
        MediaTypeId = "email",
        TypeId = ixnType,
        SubtypeId = ixnSubtype,
        TenantId = 101,
        Status = new NullableStatuses(Statuses.Pending),
        Subject = "Sample email subject",
        EntityTypeId = new NullableEntityTypes(EntityTypes.EmailOut),
    };
    req.EntityAttributes = new EmailOutEntityAttributes()
    {
        FromAddress = "sending@email.com",
        ToAddresses = "receiving@email.com",
        CcAddresses = "copied@email.com",
        ...
    };
    req.InteractionContent = new InteractionContent()
    {
        Text = "This is the email body.",
       ...
    };
    contactServerProtocol.Send(req);
}
```

A list of `InteractionAttributes` properties that need to be set for an email message is provided in the following table. The properties shown for `EmailOutEntityAttributes` and `InteractionContent` represent some of those most commonly used with email. Please check the documentation provided for each class to see a full list of available properties.

Interaction Attribute Name	Description
EntityTypeId	Indicates whether this is an outgoing or incoming email.
Id	Interaction ID of the related Interaction record, created earlier.
MediaTypeId	Primary media type of the Interaction you are submitting to Interaction Server. Intended for Media Server.
Subject	Subject line for this email message.
SubtypeId	Interaction subtype for this email message. Must match an Interaction Subtype Business Attribute, as specified in Configuration Server.
Status	Current status of the email message.
TenantId	ID of the Tenant where this email belongs.
TypeId	Interaction type for this email message. Must match an Interaction Type Business Attribute, as specified in Configuration Server.

Placing the Interaction in the Appropriate Queue

When an Interaction has been created to handle the email, and all content has been stored in the

UCS database, you are free to begin processing the message as you would process any normal Interaction. This is accomplished by moving the Interaction that you created into the appropriate queue for email processing, as defined in Interaction Routing Designer.

```
[C#]

public void PlaceInQueue(string queue)
{
    var req = RequestPlaceInQueue.Create();
    req.InteractionId = mInteractionId;
    req.Queue = queue;

    interactionServerProtocol.Send(req);
}
```

Replying to an Email Message

Replying to an existing email message follows the same basic process outlined above, but requires a few additional parameters to be set in your requests. These changes are described in the following subsections.

Changes to Creating an Interaction

When creating the Interaction, you need to specify one additional parameter before submitting your `RequestSubmit`. Take the `InteractionId` of the Interaction that represents the original email message, and use that value as the `ParentInteractionId` parameter in your request, as shown below:

```
[C#]

var req = RequestSubmit.Create();
...
req.ParentInteractionId = parentInteractionId;
```

The following table describes these additional attributes.

Attribute Name	Description
ParentInteractionId	InteractionId of a parent email Interaction. Only set this value when replying to an existing email message.

Changes to Storing Email Details in UCS

When storing email details in UCS, you need to specify values for three additional interaction attributes before sending your `RequestInsertInteraction`. These attributes (shown in the code snippet below) provide a link between the parent entry in UCS and any related children, as well as specifying a common thread ID.

```
[C#]

var req = new RequestInsertInteraction();
...
```

```
req.InteractionAttributes.ParentId = parentInteractionId;
req.InteractionAttributes.CanBeParent = False;
req.InteractionAttributes.ThreadId = parentThreadId;
```

The following table describes these additional attributes.

Attribute Name	Description
CanBeParent	Boolean value that indicates whether this message can be a parent.
ParentId	Interaction ID for the parent email Interaction.
ThreadId	Unique value that is shared between all UCS entries in an email conversation.

Other Considerations

Although this introduction to creating and handling email messages is not intended to be a comprehensive guide, it is useful to quickly introduce some other considerations and basic concepts regarding how requests are submitted and how errors should be handled.

The first consideration to take into account is how you submit requests. In the code provided here, a simple Send method is used to submit most requests without waiting for a response from the server. However, for more complicated samples or implementations you should consider using the `BeginRequest` method with a callback handler instead.

Using `BeginRequest` allows requests to be submitted without waiting for a response, but provides the ability to confirm the result and response of each request. This allows better error handling to be implemented if a request fails. *Creating an Interaction* uses the `BeginRequest` method and a callback handler to capture the InteractionID that is returned.

A second consideration to be aware of is how records in Interaction Server and UCS are related when implementing error handling. If you have already created a new Interaction and then the `RequestInsertInteraction` fails, you need to either resubmit the UCS record or delete the related Interaction by submitting a `RequestStopProcessing`. (If you reversed those steps and created a UCS record first, then the same idea must be applied if the request to create a new Interaction fails.)

Chat

You can use the Web Media Platform SDK to write Java or .NET applications that use the Genesys Web Media Server's chat, email and voice callback protocols. These applications can range from the simple to the advanced.

This article shows how to implement the basic functions you will need to write a simple Web Media Server application. It provides code snippets to illustrate how the `FlexChat` protocol can be used to support a simple chat application.

Which Chat Protocol Should I Use?

One possible point of confusion when developing a Chat application is which protocol to use, because Platform SDK includes two different protocols: Basic Chat and Flex Chat.

Basic Chat is for agent applications. This protocol provides a persistent TCP connection for the duration of the chat session. With this protocol, a bi-directional connection is established only when required by the chat session, and is closed when the session is finished. Therefore the total number of connections is never larger than the number of active chat sessions. An unexpected loss of TCP connection is treated as a client disconnect by this protocol.

Flex Chat is specifically designed for customer-facing applications. This protocol is for one-directional messaging, with no push possible, and cannot be used to close the chat session directly.

Java

Importing the Web Media Protocols

Before using the Web Media Platform SDK, you will need to import the appropriate packages. Since we will be using the FlexChat protocol, we will use the following import statements:

```
[Java]
import com.genesyslab.platform.webmedia.protocol.*;
import com.genesyslab.platform.webmedia.protocol.flexchat.*;
import com.genesyslab.platform.webmedia.protocol.flexchat.events.*;
import com.genesyslab.platform.webmedia.protocol.flexchat.requests.*;
```

Setting Up Web Media Protocol Objects

When interacting with existing chat sessions, you will need to store session-specific details including a secure key and user ID. Additional objects that will be needed include a `FlexChatProtocol` object (for sending and receive messages) and a `FlexTranscript` object (used to store and interact with the chat transcript).

```
[Java]
private String mSecureKey = null;
private String mUserId = null;
private FlexTranscript mTranscript = null;
private FlexChatProtocol mFlexChatProtocol = null;
```

To use the Web Media Platform SDK, you first need to instantiate a protocol object by supplying information about the Web Media Server you want to connect with. This example specifies values for the name, host, and port values:

```
[Java]
mFlexChatProtocol = new FlexChatProtocol(new Endpoint("FlexChat", "<hostname>", <port>));
Thread mListenerThread = new ListenForEventsThread(mFlexChatProtocol);
```

Note that you have to provide a string when you create the `FlexChatProtocol` object. This string should be unique for each protocol used in your application. It might be a good idea to use the name of the server's application object from the configuration layer, which guarantees uniqueness as well as clearly identifying which server you are communicating with.

After instantiating the `FlexChatProtocol` object, you need to open the connection to the Web Media Server:

```
[Java]
mFlexChatProtocol.open();
```

Note that you should always use proper error handling techniques in your code, especially when working with the protocol connection. To save space, these error handling steps are not shown in this example.

Logging in to Chat Server

```
[Java]
// filter the request based on our configured application name
KeyValueCollection kvUserData = new KeyValueCollection();
kvUserData.addObject("FirstName", "John");
kvUserData.addObject("LastName", "Smith");
kvUserData.addObject("EmailAddress", "john.smith@email.com");
RequestLogin reqLogin = RequestLogin.create(strNickName, 0, kvUserData);
Message msg = mFlexChatProtocol.request(reqLogin);
```

After successfully logging in to Chat Server, a message is returned that includes some important information: the Secure Key and User ID values. You will use these values when sending messages to the Chat Server, so remember to keep track of them for later.

```
[Java]
if (msg != null && msg.messageId() == EventStatus.ID)
{
        EventStatus status = (EventStatus)msg;
        if (status.getRequestResult() == RequestResult.Success)
        {
                mSecureKey = status.getSecureKey();
                mUserId = status.getUserId();
        }
}
```

Updating a Chat Session

By creating a `RequestRefresh` object, you can either check for updates or send new text to an existing chat session. The following sample shows how to create a `RequestRefresh` object, send it to the Chat Server, and process the result.

```
[Java]
RequestRefresh reqRefresh = RequestRefresh.create(mUserId, mSecureKey,
        mTranscript.getLastPosition() + 1, MessageText.create("text", message));
Message msg = mFlexChatProtocol.request(reqRefresh);
if (msg != null && msg.messageId() == EventStatus.ID)
{
        EventStatus status = (EventStatus)msg;
        if (status.getRequestResult() == RequestResult.Success)
        {
                processTranscript(status.getFlexTranscript());
        }
}
```

Working with Restricted Characters

Due to server-side requirements, the XML-based Webmedia Platform SDK protocols (BasicChat, FlexChat, Callback and Email) do not support illegal characters in string values. See http://www.w3.org/TR/2000/REC-xml-20001006#NT-Char for the allowable character range.

The Platform SDK protocols do not change user data by default, but the following options are available if you want to replace illegal characters:

(1) Include code in your application to configure the protocol connection. For example:

```
[Java]
PropertyConfiguration conf = new PropertyConfiguration();
conf.setBoolean(WebmediaChannel.OPTION_NAME_REPLACE_ILLEGAL_UNICODE_CHARS, true);
// "replacement" value is optional: if it is not specified - illegal characters will be
removed
conf.setOption(WebmediaChannel.OPTION_NAME_ILLEGAL_UNICODE_CHARS_REPLACEMENT, "?");
BasicChatProtocol protocol = new BasicChatProtocol(new Endpoint("chatServer", HOST, PORT,
conf));
protocol.open();
```

(2) Set specific JVM properties for the client application using `webmediaprotocol.jar`. For example:

```
[Java]
"-Dcom.genesyslab.platform.WebMedia.BasicChat.replace-illegal-unicode-chars=true"
```

or

```
[Java]
"-Dcom.genesyslab.platform.WebMedia.BasicChat.replace-illegal-unicode-chars=true
-Dcom.genesyslab.platform.WebMedia.BasicChat.illegal-unicode-chars-replacement=?"
```

Using JVM system properties will affect all protocol connections for the specified Webmedia protocol. Using specific connection configuration values will only affect the specified protocol instance(s), and will take priority over JVM settings.

If no replacement character or string is specified, then illegal characters will be removed (that is, replaced with an empty string).

Values are extracted independently for the two methods listed above. If you enable character replacement using the `PropertyConfiguration` class without specifying a replacement value, but a replacement value is already specified through the JVM system properties, then characters will be replaced without verifying the enabling option in the JVM properties. It is recommended to use both options while writing connection configuration code.

Logging out of a Chat Session

When a client is ready to log out from the existing chat session, build a `RequestLogout` object and send it to the Chat Server.

```
[Java]
RequestLogout reqLogout = RequestLogout.create(mUserId, mSecureKey,
        mTranscript.getLastPosition());
Message msg = mFlexChatProtocol.request(reqLogout);
if (msg != null && msg.messageId() == EventStatus.ID)
{
        EventStatus status = (EventStatus)msg;
        if (status.getRequestResult() == RequestResult.Success)
        {
                processTranscript(status.getFlexTranscript());
        }
}
```

Disconnecting from a Chat Server

Finally, when you are finished communicating with the Chat Server, you should close the connection to minimize resource utilization.

```
[Java]
mFlexChatProtocol.close();
```

.NET

Using the Web Media Protocols

Before using the Web Media Platform SDK, you should include `using` statements that allow access to types from the Platform SDK Commons and Web Media namespaces. For the `FlexChat` protocol, we use the following statements:

```
[C#]
using Genesyslab.Platform.Commons.Collections;
using Genesyslab.Platform.Commons.Connection;
using Genesyslab.Platform.Commons.Protocols;

using Genesyslab.Platform.WebMedia.Protocols;
using Genesyslab.Platform.WebMedia.Protocols.FlexChat;
using Genesyslab.Platform.WebMedia.Protocols.FlexChat.Events;
using Genesyslab.Platform.WebMedia.Protocols.FlexChat.Requests;
```

Setting Up Web Media Protocol Objects

When interacting with existing chat sessions, you will need to store session-specific details including a secure key and user ID. Additional objects that will be needed include a `FlexChatProtocol` object (for sending and receive messages) and a `FlexTranscript` object (used to store and interact with the chat transcript).

```
[C#]
private string secureKey;
private string userId;
private FlexTranscript flexTranscript;
private FlexChatProtocol flexChatProtocol;
```

To use the Web Media Platform SDK, you first need to instantiate a `Protocol` object by supplying information about the Web Media Server you want to connect with. This example specifies values for the name, host, and port values:

```
[C#]
flexChatProtocol = new FlexChatProtocol(new Endpoint("Flex_Chat_Server", "<hostname>",
<port>));
```

Note that you have to provide a string when you create the `FlexChatProtocol` object. This string should be unique for each protocol used in your application. It might be a good idea to use the name of the server's application object from the configuration layer, which guarantees uniqueness as well as clearly identifying which server you are communicating with.

After instantiating the `FlexChatProtocol` object, you need to open the connection to the Web Media Server:

```
[C#]
flexChatProtocol.Open();
```

You should always use proper error handling techniques in your code, especially when working with the protocol connection. To save space, these error handling steps are not shown in this example.

Logging in to Chat Server

```
[C#]
// filter the request based on our configured application name
KeyValueCollection kvUserData = new KeyValueCollection();
```

```
kvUserData.Add("FirstName", "John");
kvUserData.Add("LastName", "Smith");
kvUserData.Add("EmailAddress", "john.smith@email.com");
RequestLogin reqLogin = RequestLogin.Create("reqLogin", 0, kvUserData);
EventStatus msg = this.flexChatProtocol.Request(reqLogin) as EventStatus;
```

After successfully logging in to Chat Server, a message is returned that includes some important information: the Secure Key and User ID values. You will use these values when sending messages to the Chat Server, so remember to keep track of them for later.

```
[C#]
if (msg != null && msg.Id == EventStatus.MessageId)
{
    if (msg.RequestResult == RequestResult.Success)
    {
        secureKey = msg.SecureKey;
        userId = msg.UserId;
    }
}
```

Updating a Chat Session

By creating a `RequestRefresh` object, you can either check for updates or send new text to an existing chat session. The following sample shows how to create a `RequestRefresh` object, send it to the Chat Server, and process the result.

```
[C#]
RequestRefresh reqRefresh = RequestRefresh.Create(
        userId, secureKey, flexTranscript.LastPosition + 1, MessageText.Create(""));
EventStatus msg = this.flexChatProtocol.Request(reqJoin) as EventStatus;
if (msg != null && msg.Id == EventStatus.MessageId)
{
        if (msg.RequestResult == RequestResult.Success)
        {
                ProcessTranscript(msg.FlexTranscript);
        }
}
```

Working with Restricted Characters

Due to server-side requirements, the XML-based Web Media Platform SDK protocols (`BasicChat`, `FlexChat`, `Callback` and `Email`) do not support illegal characters in string values. See http://www.w3.org/TR/2000/REC-xml-20001006#NT-Char for the allowable character range.

The Platform SDK protocols do not change user data by default, but if you want to replace illegal characters then you can include code in your application to configure the protocol connection. For example:

```
[C#]
// Note: to use the PropertyConfiguration class, ensure that your using
// statements include Genesyslab.Platform.Commons.Connection
PropertyConfiguration conf = new PropertyConfiguration();
conf.SetBoolean(WebmediaChannel.OptionNameReplaceIllegalUnicodeChars, true);
```

```
// "replacement" value is optional: if it is not specified - illegal characters will be
removed
conf.SetOption(WebmediaChannel.OptionNameIllegalUnicodeCharsReplacement, "?");
BasicChatProtocol protocol = new BasicChatProtocol(new Endpoint("chatServer", HOST, PORT,
conf));
protocol.Open();
```

Using specific connection configuration values in this manner will only affect the specified protocol instance(s).

If no replacement character or string is specified, then illegal characters will be removed (that is, replaced with an empty string).

Logging out of a Chat Session

When a client is ready to log out from the existing chat session, build a `RequestLogout` object and send it to the Chat Server.

```
[C#]
RequestLogout reqLogout = RequestLogout.Create(userId, secureKey,
flexTranscript.LastPosition);
IMessage msg = flexChatProtocol.Request(reqLogout);
if (msg != null && msg.Id == EventStatus.MessageId)
{
    if ((msg as EventStatus).RequestResult == RequestResult.Success)
    {
                  ProcessTranscript((msg as EventStatus).FlexTranscript);
    }
}
```

Disconnecting from a Chat Server

Finally, when you are finished communicating with the Chat Server, you should close the connection to minimize resource utilization.

```
[C#]
flexChatProtocol.Close();
```

E-Mail Server

The `EspEmailProtocol` allows communication to be established between a client application and an ESP-based E-mail Server.

Tip

Note that Platform SDK also provides the deprecated `EmailProtocol` class along with `EspEmailProtocol`. The `EmailProtocol` cannot be used with E-mail Servers of release 8.0.2 and higher. For newer email servers, use `EspEmailProtocol`.

E-mail Server interfaces with the enterprise mail server and the Genesys Web API Server, bringing in new email interactions and sending out replies or other outbound messages. For better understanding of email server, see the eServices documentation.

The `EspEmailProtocol` can be used in a web-based application, for example to submit new email interaction that user filled out via the web form. You can find advanced samples with `EspEmailProtocol` in the Web API Client Developer's Guide.

The sample below shows how to connect to E-mail Server and submit a new email interaction.

Java

Open Connection to Server

Before using the Web Media Platform SDK, you will need to import the appropriate packages. Since we will be using the `EmailServer` protocol, we will use the following `import` statements:

```
import com.genesyslab.platform.commons.protocol.Endpoint;
import com.genesyslab.platform.commons.protocol.Message;
import com.genesyslab.platform.webmedia.protocol.EspEmailProtocol;
import com.genesyslab.platform.webmedia.protocol.espemail.EmailAttachment;
import com.genesyslab.platform.webmedia.protocol.espemail.EmailAttachmentList;
import com.genesyslab.platform.webmedia.protocol.espemail.events.EventCreateWebEmailIn;
import com.genesyslab.platform.webmedia.protocol.espemail.events.EventError;
import com.genesyslab.platform.webmedia.protocol.espemail.requests.RequestCreateWebEmailIn;
```

Now create a Protocol object and open a connection to the server.

```
EspEmailProtocol emailProtocol = new EspEmailProtocol(new Endpoint(hostname, port));
emailProtocol.open();
```

Submit a New E-mail Interaction

Create a `RequestCreateWebEmailIn` message to provide the email message content and submit a new email interaction.

```
RequestCreateWebEmailIn req = RequestCreateWebEmailIn.create();
req.setFirstName(firstName);
req.setLastName(lastName);
req.setFromAddress(emailAddress);
req.setSubject(subject);
req.setText(body);
req.setMailbox(replayFrom);
```

It is possible to attach data to an email using `EmailAttachmentList` and `EmailAttachment` objects.

```
EmailAttachmentList attachmentList= new EmailAttachmentList();
EmailAttachment attachment = new EmailAttachment();
attachment.setContentType("image/png");
attachment.setFileName("picture.png");
attachment.setContent(content);
attachmentList.add(attachment);

req.setAttachments(attachmentList);
```

Send your request to the E-mail Server. If the operation is completed successfully, the server will respond with an `EventCreateWebEmailIn` message. This message contains the ID of the new interaction.

```
Message response = emailProtocol.request(req);
if (response instanceof EventCreateWebEmailIn) {
    EventCreateWebEmailIn eventAck = (EventCreateWebEmailIn)response;
    intearctionId = eventAck.getNewInteractionId();
}
else if (response instanceof EventError) {
    EventError eventError = (EventError) response;
    //handle error
}
```

Disconnecting from E-mail Server

Finally, when you are finished communicating with the E-mail Server, you should close the connection to minimize resource utilization.

```
emailProtocol.close();
```

.NET

Open Connection to Server

Before using the Web Media Platform SDK, you should include `using` statements that allow access to types from the Platform SDK Commons and Web Media namespaces. For the `EspEmail` protocol, we use the following statements:

```
using Genesyslab.Platform.Commons.Collections;
using Genesyslab.Platform.Commons.Connection;
using Genesyslab.Platform.Commons.Protocols;
using Genesyslab.Platform.WebMedia.Protocols;
using Genesyslab.Platform.WebMedia.Protocols.EspEmail;
using Genesyslab.Platform.WebMedia.Protocols.EspEmail.Requests;
using Genesyslab.Platform.WebMedia.Protocols.EspEmail.Events;
```

Now create a Protocol object and open a connection to the server.

```
EspEmailProtocol espProtocol = new EspEmailProtocol(new Endpoint(hostname, port));
emailProtocol.Open();
```

Submit a New E-mail Interaction

Create a `RequestCreateWebEmailIn` message to provide the email message content and submit a new email interaction.

```
RequestCreateWebEmailIn req = RequestCreateWebEmailIn.Create();
req.FirstName = firstName;
req.LastName = lastName;
req.FromAddress = emailAddress;
req.Subject = subject;
req.Text = body;
req.Mailbox = replayFrom;
```

It is possible to attach data to an email using `EmailAttachmentList` and `EmailAttachment` objects.

```
EmailAttachmentList attachmentList= new EmailAttachmentList();
EmailAttachment attachment = new EmailAttachment();
attachment.ContentType = "image/png";
attachment.FileName = fileName;
attachment.Content = File.ReadAllBytes(file);
attachmentList.Add(attachment);
req.Attachments = attachmentList;
```

Send your request to the E-mail Server. If the operation is completed successfully, the server will respond with an `EventCreateWebEmailIn` message. This message contains the ID of the new interaction.

```
IMessage response = emailProtocol.Request(req);

if (response is EventCreateWebEmailIn)
{
      EventCreateWebEmailIn eventAck = (EventCreateWebEmailIn)response;
      intearctionId = eventAck.NewInteractionId;
}
else if (response is EventError)
```

```
{
      EventError eventError = (EventError) response;
      //handle error
}
```

Disconnecting from E-mail Server

Finally, when you are finished communicating with the E-mail Server, you should close the connection to minimize resource utilization.

```
emailProtocol.Close();
```

Outbound

Java

You can use the Outbound Contact Platform SDK to write Java or .NET applications that work with the Genesys Outbound Contact Server. These applications can range from the simple to the advanced. This document shows how to implement the basic functions you will need to write a simple Outbound Contact application. It is organized to show the kind of structure you will probably use to write your own applications.

Setting Up an OutboundServerProtocol Object

The first thing you need to do to use the Outbound Contact Platform SDK is instantiate a `OutboundServerProtocol` object. To do that, you must supply information about the Outbound Contact Server you want to connect with. This example uses the URI of the server, but you can also use name, host, and port information:

```
[Java]

OutboundServerProtocol outboundServerProtocol =
        new OutboundServerProtocol(
                new Endpoint(
                        outboundServerUri));
```

After instantiating the protocol object, you need to open the connection to the server:

```
[Java]

outboundServerProtocol.open();
```

Closing the Connection

Finally, when you are finished communicating with the Outbound Contact Server, you should close the connection to minimize resource utilization:

```
[Java]

outboundServerProtocol.close();
```

.NET

You can use the Outbound Contact Platform SDK to write Java or .NET applications that work with the

Genesys Outbound Contact Server. These applications can range from the simple to the advanced. This document shows how to implement the basic functions you will need to write a simple Outbound Contact application. It is organized to show the kind of structure you will probably use to write your own applications.

Setting Up an OutboundServerProtocol Object

The first thing you need to do to use the Outbound Contact Platform SDK is instantiate a `OutboundServerProtocol` object. To do that, you must supply information about the Outbound Contact Server you want to connect with. This example uses the URI of the server, but you can also use name, host, and port information:

```
[C#]

OutboundServerProtocol outboundServerProtocol =
        new OutboundServerProtocol(
                new Endpoint(
                        outboundServerUri));
```

After instantiating the `OutboundServerProtocol` object, you need to open the connection to the Outbound Contact Server:

```
[C#]

outboundServerProtocol.Open();
```

Closing the Connection

Finally, when you are finished communicating with the Outbound Contact Server, you should close the connection to minimize resource utilization:

```
[C#]

outboundServerProtocol.Close();
```

Management Layer

You can use the Management Platform SDK to write Java or .NET applications that interact with the Genesys Message Server, Solution Control Server and Local Control Agents (LCAs). Most people will want to use this SDK to make their applications visible to the Genesys Management Layer so they can monitor them with Solution Control Server.

This document shows how to implement the basic functions you will need to write a simple voice application. It is organized to show the kind of structure you will probably use to write your own applications.

Java

Making Your Application Visible to the Genesys Management Layer

A Genesys Local Control Agent (LCA) runs on each host in the Genesys environment, enabling the Management Layer to monitor and control the applications running on that host. This section shows how to use the LCA running on your own host to make your application visible to the Genesys Management Layer.

Connecting to the Local Control Agent

The first step is to create a Local Control Agent Protocol instance, specifying the LCA port in an `Endpoint` object. This sample uses the default LCA port of 4999.

```
LocalControlAgentProtocol lcaProtocol = new LocalControlAgentProtocol(new
Endpoint("localhost", 4999));
```

Now you can configure the connection. Set the `applicationName` to the same value as the name of an application that you have set up in the Configuration Layer. Then set the application status to `Initializing`, and the execution mode to `Backup`.

```
lcaProtocol.setClientName(applicationName);
lcaProtocol.setControlStatus(ApplicationStatus.Initializing.asInteger());
lcaProtocol.setExecutionMode(ApplicationExecutionMode.Backup);
```

The next step is to set up a message handler to process events from LCA. See Event Handling articles for a better understanding of how messages and protocols should be managed. The code snippets below show how to handle events from LCA.

```
MessageHandler lcarMessageHandler = new MessageHandler() {
  public void onMessage(Message message) {
    System.out.println("Message received: \n"+message);
    //process message
  }
};
lcaProtocol.setMessageHandler(lcarMessageHandler);
```

Important

You need to know that your event-handling logic will be executed by using the protocol invoker. Please set the appropriate invoker for your application needs. For more information about the protocol invoker and how to set it, refer to Connecting to a Server.

Once you have finished configuring your protocol, you can open a connection to the LCA.

```
lcaProtocol.Open();
```

Updating the Application Status

When you need to update the status of your application, send a `RequestUpdateStatus`. Here is how to indicate that the application is running:

```
RequestUpdateStatus requestUpdateStatus = RequestUpdateStatus.create();
requestUpdateStatus.setApplicationName(lcaProtocol.getClientName());
requestUpdateStatus.setControlStatus(ApplicationStatus.Running.asInteger());
lcaProtocol.send(requestUpdateStatus);
```

The LCA does not return an event when you change the application status. So for this particular task, you will not need any more code.

Execution Mode and Event Handling

As mentioned, the LCA will not return an event when you change the application status. But when you change the execution mode — for example, from `Primary` to `Backup` — you will receive an `EventChangeExecutionMode`. Unlike most events you receive in the Platform SDK, this event *requires* a response from your application. If the Management Layer does not know that your application is expecting to work in `Primary` mode, for example, it cannot rely on the stability of the Genesys environment.

Important

If you do not respond within the configured timeout period, your application will be terminated by the Management Layer.

After receiving the `EventChangeExecutionMode`, your application must send a `ResponseExecutionModeChanged` to indicate to the Management Layer that you are now ready to run in the new execution mode.

In order to handle these events, you need to setup message handler for a LCA protocol object as shown above.

Implement your event-handling logic:

```
MessageHandler lcarMessageHandler = new MessageHandler() {
  public void onMessage(Message message) {
    switch(message.messageId()){
      case EventChangeExecutionMode.ID:
          OnEventChangeExecutionMode(message);
          break;
      // other messages
    }
  }
};
```

Here is a sample of the handler you might set up for the EventChangeExecutionMode. This handler includes your ResponseExecutionModeChanged:

```
private static void OnEventChangeExecutionMode(Message message)
{
  if(message instanceof EventChangeExecutionMode)
  {
     EventChangeExecutionMode eventChangeExecutionMode = (EventChangeExecutionMode)message;
    System.out.println("eventChangeExecutionMode received: \n"+eventChangeExecutionMode);

    ApplicationExecutionMode mode = eventChangeExecutionMode.getExecutionMode();

    ResponseExecutionModeChanged response =
ResponseExecutionModeChanged.create(mode);
    System.out.println("Sending response: " + response);
    try {
      lcaProtocol.send(response);
    } catch (ProtocolException e) {
      e.printStackTrace();
    }
  }
}
```

Tip

However, if your real application didn't successfully do the switchover, you can skip sending ResponseExecutionModeChanged. That way SCS will revert the switchover and put the application in Unknown status. That's convenient for admins as they can have alarms on that.

Closing the Connection

When you are finished, close the connection to the LCA:

```
lcaProtocol.Close();
```

Monitoring Your Application with Solution Control Server

Solution Control Server can be used to monitor applications running in the Genesys environment.

Here is how to obtain information about hosts and applications.

Connecting to Solution Control Server

Create a protocol instance and supply the necessary parameters. The `ClientName` is the name of a Solution Control application that has been set up in the Configuration Layer, while the `ClientId` is the DBID of that application.

```
SolutionControlServerProtocol  scsProtocol = new SolutionControlServerProtocol(new
Endpoint("host", port));

scsProtocol.setClientName(scsApplicationName);
scsProtocol.setClientId(scsApplicationDBid);
scsProtocol.setUserName(userName);
```

Setting Up Event Handling

You will need to set up some event handling code, since Solution Control Server will return `EventInfo` or `EventError` messages in response to your requests for information. The code for this is similar to the LCA-related code shown above.

```
MessageHandler scsrMessageHandler = new MessageHandler() {
  public void onMessage(Message message) {
    switch(message.messageId()){
      case EventInfo.ID:
          OnEventInfo(message);
          break;
      //Other events.
    }
  }
};
scsProtocol.setMessageHandler(scsrMessageHandler);

...

private static void OnEventInfo(Message message)
{
  System.out.println("Event info: \n"+message);
  //Handling logic here info.
}
```

Open Connection

Once you have configured your protocol, you can open your connection to the SCS:

```
scsProtocol.open();
```

Requesting Application Information

Here is how to request the status of an application, using its DBID:

```
RequestGetApplicationInfo requestGetApplicationInfo =
  RequestGetApplicationInfo.create(applicationDbid);

scsProtocol.send(requestGetApplicationInfo);
```

When you send this request, you will receive an `EventInfo` that includes the status of the application:

```
'EventInfo' ('1')
message attributes:
  attr_app_work_mode [int] = 0 [Primary]
  attr_client [int] = 660
  attr_ref_id [int] = 4
  attr_message [str] = "APP_STATUS_RUNNING"
  attr_obj_live_status [int] = 6 [Running]
  attr_cfg_obj_id [int] = 109
  attr_cfg_obj_type [int] = 9 [Application]
```

If you want to be notified when the status of an application changes, send a `RequestSubscribe`.

```
RequestSubscribe requestSubscribeApp = RequestSubscribe.create();
requestSubscribeApp.setControlObjectType(ControlObjectType.Application);
requestSubscribeApp.setControlObjectId(applicationDbid);
scsProtocol.send(requestSubscribeApp);
```

Whenever the application's status changes, you will receive an `EventInfo` that informs you of the new status.

Requesting Host Information

You can also request information about the status of a host. But in this case, you must issue a `RequestSubscribe` before you will receive any information about the host. Here is how:

```
RequestSubscribe requestSubscribeHost = RequestSubscribe.create();
requestSubscribeHost.setControlObjectType(ControlObjectType.Host);
requestSubscribeHost.setControlObjectId(hostDbid);
scsProtocol.send(requestSubscribeHost);

RequestGetHostInfo requestGetHostInfo = RequestGetHostInfo.create();
requestGetHostInfo.setControlObjectId(hostDbid);
scsProtocol.send(requestGetHostInfo);
```

If you just send the `RequestSubscribe`, you will be notified any time the host status changes. If you also send the `RequestGetHostInfo`, you will also receive an immediate notification of the host's status, whether it has changed or not. Here is a sample of the information you will receive.

```
'EventInfo' ('1')
message attributes:
  attr_client [int] = 660
  attr_ref_id [int] = 3
  attr_message [str] = "HOST_STATUS_RUNNING"
  attr_obj_live_status [int] = 2 [StopTransition]
  attr_cfg_obj_id [int] = 111
  attr_cfg_obj_type [int] = 10 [Host]
```

Once you have subscribed to a host, you can send a `RequestGetHostInfo` at any time to receive information about its status.

Closing the Connection

When you are finished, close the connection to Solution Control Server:

```
scsProtocol.close();
```

Sending a Log Message to Message Server

You can easily send log messages to Message Server using the Management Platform SDK. This sample shows how to log application events, raise alarms and view them in Solution Control Interface.

First you need to create the Protocol object:

```
MessageServerProtocol messageServerProtocol = new MessageServerProtocol( new Endpoint(new
URI(serverURI)));
```

Now you can configure the Protocol object and open the connection to Message Server. Specify the application type, application name, host on which the application is running, and application DBID. Note that the `CfgAppType` enum is defined in the Configuration Protocol package: `com.genesyslab.platform.configuration.protocol.types`

```
messageServerProtocol.setClientType(CfgAppType.CFGGenericServer.ordinal());
messageServerProtocol.setClientName ("Primary_Server_App");
messageServerProtocol.setClientHost (applicationHostName);
messageServerProtocol.setClientId(applicationDBID);
```

Now you can configure the Protocol object and open the connection to Message Server:

```
messageServerProtocol.open();
```

Create `RequestLogMessage` to log an application event. To raise an Alarm with this event, specify `requestLogMessage.EntryId` equal to the alarm detect ID. (There is more information about configuring Alarm conditions in Configuration Manager at the end of the article.)

```
RequestLogMessage requestLogMessage = RequestLogMessage.create();
requestLogMessage.setEntryId(9600);
requestLogMessage.setEntryCategory(LogCategory.Application);
requestLogMessage.setEntryText("Primary_Server_App out of service...");
requestLogMessage.setLevel(LogLevel.Alarm);
```

Once you have created the request, you can send the request to Message Server.

```
messageServerProtocol.send(requestLogMessage);
Thread.sleep(15000); //stop execution to view raised alarm in SCI
```

You can cancel an alarm after your application is restored. Specify the cancel alarm event ID and send that message to Message Server.

```
requestLogMessage = RequestLogMessage.create();
requestLogMessage.setEntryId(9601);
requestLogMessage.setEntryCategory(LogCategory.Application);
requestLogMessage.setEntryText("Primary_Server_App back in service...");
requestLogMessage.setLevel(LogLevel.Alarm);

messageServerProtocol.send(requestLogMessage);
```

When you are finished, you should close the connection:

```
messageServerProtocol.close();
```

Configuring Genesys Management Framework

You can view event logs and active alarms created by this code snippet in Solution Control Interface. However, Genesys Management Framework should be configured according to the list of required settings, below:

- Solution Control Server must be connected to Message Server. See the *Connections* tab in the Solution Control Server properties dialog.
- Solution Control Interface must be connected to Solution Control Server. See the *Connections* tab in the Solution Control Interface properties dialog.
- Solution Control Interface must be connected to the DataBase Access Point. See the *Connections* tab in the Solution Control Interface properties dialog.
- Message Server must be connected to the DataBase Access Point. See the *Connections* tab in the Message Server properties dialog.
- Message Server must have the `db_storage=true` property. See the *messages* section under *Options* of the Solution Control Interface properties dialog. This option is required to store messages in the database.
- The DataBase Access Point must be associated with DBServer. See the *General* tab of the DataBase Access Point. Check that the *DB Info* tab has the proper connection options to SQL Server.

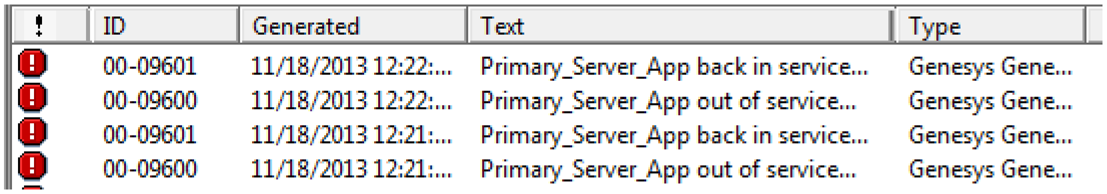
Configuring Alarm Conditions

You need to create the Alarm Condition that will trigger an Alarm for log events sent by the code snippet. To do this, open Configuration Manager, find the Alarm Conditions section and create a new Condition.

- On the *General* tab specify a condition name and description, then select `Major` for the category.
- On the *Detect Event* tab specify a Log event ID that will raise the alarm. This refers to the `RequestLogMessage.EntryId` value.
- On the *Detect Event* tab choose Select By Application for the Selection Mode and choose the application for which an event will be triggered.
- On the *Detect Event* tab specify a Log event ID that will cancel the alarm. This refers to the `RequestLogMessage.EntryId` value.

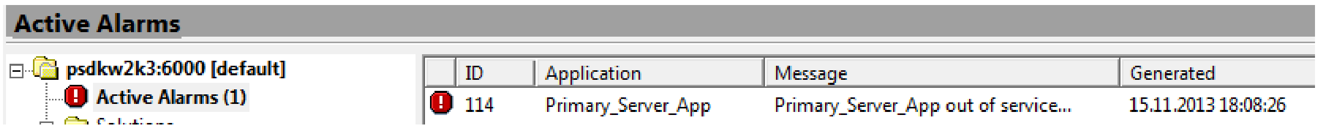
You can observe the results of running the application from Solution Control Interface.

Here is what the log messages look like in SCI:



!	ID	Generated	Text	Type
❗	00-09601	11/18/2013 12:22:...	Primary_Server_App back in service...	Genesys Gene...
❗	00-09600	11/18/2013 12:22:...	Primary_Server_App out of service...	Genesys Gene...
❗	00-09601	11/18/2013 12:21:...	Primary_Server_App back in service...	Genesys Gene...
❗	00-09600	11/18/2013 12:21:...	Primary_Server_App out of service...	Genesys Gene...

And here is what the alarm entry looks like while the alarm is active:



ID	Application	Message	Generated
114	Primary_Server_App	Primary_Server_App out of service...	15.11.2013 18:08:26

.NET

Making Your Application Visible to the Genesys Management Layer

A Genesys Local Control Agent (LCA) runs on each host in the Genesys environment, enabling the Management Layer to monitor and control the applications running on that host. This section shows how to use the LCA running on your own host to make your application visible to the Genesys Management Layer.

Connecting to the Local Control Agent

The first step is to create a Local Control Agent Protocol instance, specifying the LCA port in an `Endpoint` object. This sample uses the default LCA port of 4999.

```
LocalControlAgentProtocol lcaProtocol = new LocalControlAgentProtocol(new
Endpoint("localhost", 4999));
```

Now you can configure the connection. Set the `applicationName` to the same value as the name of an application that you have set up in the Configuration Layer. Then set the application status to `Initializing`, and the execution mode to Backup.

```
lcaProtocol.ClientName = applicationName;
lcaProtocol.ControlStatus = (int)ApplicationStatus.Initializing;
lcaProtocol.ExecutionMode = ApplicationExecutionMode.Backup;
```

The next step is to set up a message handler to process events from LCA. See Event Handling articles for a better understanding of how messages and protocols should be managed. The code snippets below show how to handle events from LCA.

```
private void OnLcaMessageReceived(object sender, EventArgs args)
{
  Console.WriteLine("New message: {0}",((MessageEventArgs)args).Message);
  // Message handling logic here.
}
. . .

lcaProtocol.Received += OnLcaMessageReceived;
```

Important

You need to know that your event-handling logic will be executed by using the protocol invoker. Please set the appropriate invoker for your application needs. For more information about the protocol invoker and how to set it, refer to Connecting to a Server.

Once you have configured your protocol, you can open your connection to the LCA:

```
lcaProtocol.Open();
```

Updating the Application Status

When you need to update the status of your application, send a `RequestUpdateStatus`. Here is how to indicate that the application is running:

```
RequestUpdateStatus requestUpdateStatus = RequestUpdateStatus.Create();
requestUpdateStatus.ApplicationName = lcaProtocol.ClientName;
requestUpdateStatus.ControlStatus = (int)ApplicationStatus.Running;
lcaProtocol.Send(requestUpdateStatus);
```

The LCA will not return an event when you change the application status. So for this particular task, you will not need any more code.

Execution Mode and Event Handling

As mentioned, the LCA will not return an event when you change the application status. But when Solution Control Server going to change your execution mode — for example, from `Primary` to `Backup` — you will receive an `EventChangeExecutionMode`. Unlike most events you receive in the Platform SDK, this event requires a response from your application. If the Management Layer does not know that your application is expecting to work in `Primary` mode, for example, it cannot rely on the stability of the Genesys environment.

Important

If you do not respond within the configured timeout period, your application will be terminated by the Management Layer.

After receiving the `EventChangeExecutionMode`, your application must send a `ResponseExecutionModeChanged` to indicate to the Management Layer that you are now ready to run in the new execution mode.

In order to handle these events, you need to setup message handler for a LCA protocol object as shown in the article above.

Implement your event-handling logic:

```
private void OnLcaMessageReceived(object sender, EventArgs args)
{
  IMessage message = ((MessageEventArgs)args).Message;
  switch (message.Id)
  {
    case EventChangeExecutionMode.MessageId:
        OnEventChangeExecutionMode(message);
        break;
        //  . . .
  }
}
```

Here is a sample of the handler you might set up for the EventChangeExecutionMode. This handler includes your ResponseExecutionModeChanged:

```
private void OnEventChangeExecutionMode(IMessage theMessage)
{
  EventChangeExecutionMode eventChangeExecutionMode  = theMessage as EventChangeExecutionMode;
  if (eventChangeExecutionMode != null)
  {
    ApplicationExecutionMode mode = eventChangeExecutionMode.ExecutionMode;
    ResponseExecutionModeChanged response = ResponseExecutionModeChanged.Create(mode);
    Console.WriteLine("Sending response: " + response);
    lcaProtocol.Send(response);
  }
}
```

Tip

However, if your real application did not successfully do the switchover, you can skip sending ResponseExecutionModeChanged. That way SCS will revert the switchover and put the application in Unknown status - a convenient event for administrators as they can have alarms trigger.

Closing the Connection

When you are finished, close the connection to the LCA:

```
lcaProtocol.Close();
```

Monitoring Your Application with Solution Control Server

Solution Control Server can be used to monitor applications running in the Genesys environment. Here is how to obtain information about hosts and applications.

Connecting to Solution Control Server

Create protocol instance and supply the necessary parameters. The ClientName is the name of a Solution Control application that has been set up in the Configuration Layer, while the ClientId is the DBID of that application:

```
var scsProtocol = new SolutionControlServerProtocol(new Endpoint("host", port));
scsProtocol.ClientName = applicationName;
scsProtocol.ClientId = applicationDBid;
scsProtocol.UserName = userName;
```

Once you have configured your protocol, you can open your connection to the SCS:

```
scsProtocol.Open();
```

Setting Up Event Handling

You will need to set up some event handling code, since Solution Control Server will return `EventInfo` or `EventError` messages in response to your requests for information. The code for this is similar to the LCA-related code shown above:

```
scsProtocol.Received += OnScsMessageReceived;
...

private void OnScsMessageReceived(object sender, EventArgs args)
{
  IMessage message = ((MessageEventArgs)args).Message;
  switch (message.Id)
  {
    case EventInfo.MessageId:
        OnEventInfo(message);
         break;
         //case ... other message
  }
}
...

private void OnEventInfo(IMessage theMessage)
{
  var eventInfo = theMessage as EventInfo;
  if (eventInfo != null)
  {
    Console.WriteLine("EventInfo received: \n{0}", eventInfo);
    // Handle this event
  }
}
```

Requesting Application Information

Here is how to request the status of an application, using its DBID:

```
var requestGetApplicationInfo = RequestGetApplicationInfo.Create(applicationDbid);
scsProtocol.Send(requestGetApplicationInfo);
```

When you send this request, you will receive an `EventInfo` that includes the status of the application:

```
'EventInfo' ('1')
message attributes:
  attr_app_work_mode [int] = 0 [Primary]
  attr_client [int] = 660
  attr_ref_id [int] = 4
  attr_message [str] = "APP_STATUS_RUNNING"
  attr_obj_live_status [int] = 6 [Running]
  attr_cfg_obj_id [int] = 109
  attr_cfg_obj_type [int] = 9 [Application]
```

If you want to be notified when the status of an application changes, send a `RequestSubscribe`.

```
RequestSubscribe requestSubscribeApp = RequestSubscribe.Create();
requestSubscribeApp.ControlObjectType = ControlObjectType.Application;
requestSubscribeApp.ControlObjectId = applicationDbid;
scsProtocol.Send(requestSubscribeApp);
```

Whenever the application's status changes, you will receive an `EventInfo` that informs you of the new status.

Requesting Host Information

You can also request information about the status of a host. But in this case, you must issue a `RequestSubscribe` before you will receive any information about the host. Here is how:

```
RequestSubscribe requestSubscribeHost = RequestSubscribe.Create();
requestSubscribeHost.ControlObjectType = ControlObjectType.Host;
requestSubscribeHost.ControlObjectId = HostDbid;
scsProtocol.Send(requestSubscribeHost);

RequestGetHostInfo requestGetHostInfo = RequestGetHostInfo.Create();
requestGetHostInfo.ControlObjectId = HostDbid;
scsProtocol.Send(requestGetHostInfo);
```

If you just send the `RequestSubscribe`, you will be notified any time the host status changes. If you also send the `RequestGetHostInfo`, you will also receive an immediate notification of the host's status, whether it has changed or not. Here is a sample of the information you will receive.

```
'EventInfo' ('1')
message attributes:
  attr_client [int] = 660
  attr_ref_id [int] = 3
  attr_message [str] = "HOST_STATUS_RUNNING"
  attr_obj_live_status [int] = 2 [StopTransition]
  attr_cfg_obj_id [int] = 111
  attr_cfg_obj_type [int] = 10 [Host]
```

Once you have subscribed to a host, you can send a `RequestGetHostInfo` at any time to receive information about its status.

Closing the Connection

When you are finished, close the connection to Solution Control Server:

```
scsProtocol.Close();
```

Sending a Log Message to Message Server

You can easily send log messages to Message Server using the Management Platform SDK. This sample shows how to log application events, raise alarms and view them in Solution Control Interface.

First you need to create the Protocol object:

```
MessageServerProtocol messageServerProtocol = new MessageServerProtocol( new Endpoint(new
Uri(serverURI)));
```

Now you can configure the Protocol object and open the connection to Message Server. Specify the application type, application name, host on which the application is running, and application DBID. Note that the `CfgAppType` enum is defined in the Configuration Protocol namespace: `Genesyslab.Platform.Configuration.Protocols.Types`

```
messageServerProtocol.ClientType = (int) CfgAppType.CFGGenericServer;
messageServerProtocol.ClientName = "Primary_Server_App";
messageServerProtocol.ClientHost = applicationHostName;
messageServerProtocol.ClientId = applicationDBID;
```

Now you can configure the Protocol object and open the connection to Message Server:

```
messageServerProtocol.Open();
```

Create `RequestLogMessage` to log an application event. To raise an Alarm with this event, specify `requestLogMessage.EntryId` equal to the alarm detect ID. (There is more information about configuring Alarm conditions in Configuration Manager at the end of the article.)

```
RequestLogMessage requestLogMessage = RequestLogMessage.Create();
requestLogMessage.EntryId = 9600;
requestLogMessage.EntryCategory = LogCategory.Application;
requestLogMessage.EntryText = "Primary_Server_App out of service...";
requestLogMessage.Level = LogLevel.Alarm;
```

Once you have created the request, you can send it to Message Server.

```
messageServerProtocol.Send(requestLogMessage);
Thread.Sleep(15000); //stop execution to view raised alarm in SCI
```

You can cancel an alarm after your application is restored. Specify the cancel alarm event ID and send that message to Message Server.

```
requestLogMessage = RequestLogMessage.Create();
requestLogMessage.EntryId = 9601;
requestLogMessage.EntryCategory = LogCategory.Application;
requestLogMessage.EntryText = "Primary_Server_App back in service...";
requestLogMessage.Level = LogLevel.Alarm;

messageServerProtocol.Send(requestLogMessage);
```

When you are finished, you should close the connection:

```
messageServerProtocol.Close();
```

Configuring Genesys Management Framework

You can view event logs and active alarms created by this code snippet in Solution Control Interface. However, Genesys Management Framework should be configured according to the list of required settings, below:

- Solution Control Server must be connected to Message Server. See the *Connections* tab in the Solution Control Server properties dialog.

- Solution Control Interface must be connected to Solution Control Server. See the *Connections* tab in the Solution Control Interface properties dialog.
- Solution Control Interface must be connected to the DataBase Access Point. See the *Connections* tab in the Solution Control Interface properties dialog.
- Message Server must be connected to the DataBase Access Point. See the *Connections* tab in the Message Server properties dialog.
- Message Server must have the `db_storage=true` property. See the *messages* section under *Options* of the Solution Control Interface properties dialog. This option is required to store messages in the database.
- The DataBase Access Point must be associated with DBServer. See the *General* tab of the DataBase Access Point. Check that the *DB Info* tab has the proper connection options to SQL Server.

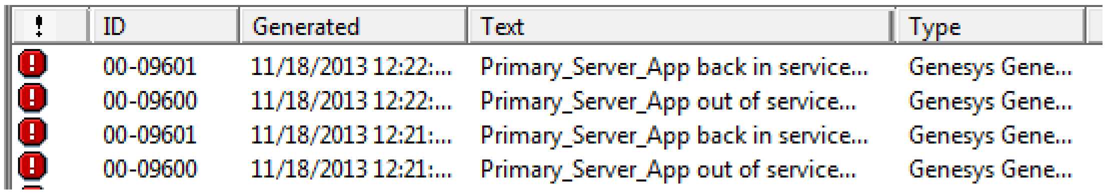
Configuring Alarm Conditions

You need to create the Alarm Condition that will trigger an Alarm for log events sent by the code snippet. To do this, open Configuration Manager, find the Alarm Conditions section and create a new Condition.

- On the *General* tab specify a condition name and description, then select `Major` for the category.
- On the *Detect Event* tab specify a Log event ID that will raise the alarm. This refers to the `RequestLogMessage.EntryId` value.
- On the *Detect Event* tab choose Select By Application for the Selection Mode and choose the application for which an event will be triggered.
- On the *Detect Event* tab specify a Log event ID that will cancel the alarm. This refers to the `RequestLogMessage.EntryId` value.

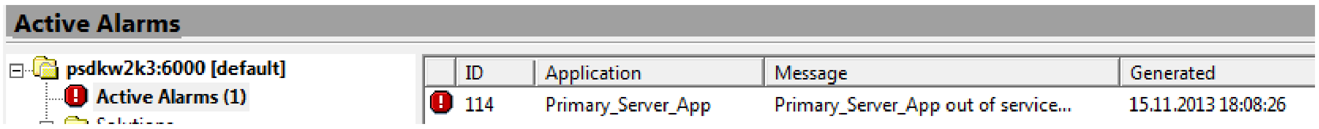
You can observe the results of running the application from Solution Control Interface.

Here is what the log messages look like in SCI:



!	ID	Generated	Text	Type
❗	00-09601	11/18/2013 12:22:...	Primary_Server_App back in service...	Genesys Gene...
❗	00-09600	11/18/2013 12:22:...	Primary_Server_App out of service...	Genesys Gene...
❗	00-09601	11/18/2013 12:21:...	Primary_Server_App back in service...	Genesys Gene...
❗	00-09600	11/18/2013 12:21:...	Primary_Server_App out of service...	Genesys Gene...

And here is what the alarm entry looks like while the alarm is active:



Active Alarms

psdkw2k3:6000 [default]
Active Alarms (1)

ID	Application	Message	Generated
114	Primary_Server_App	Primary_Server_App out of service...	15.11.2013 18:08:26

LCA Protocol Usage Samples

This page hosts code samples that showcase basic LCA Protocol usage for Java and .NET applications, and provides instructions about the deployment and configuration steps required to run the samples.

Java

Overview

The `SampleServerApp_Java_814.zip` file contains the minimal amount of code required for an application to connect to LCA and maintain this connection in accordance with the Management Framework expectations. Download the sample code below, and then follow the steps below to set up and run this sample.

Java Code Sample
SampleServerApp_Java_814.zip

Unpacking the ZIP file will give you access to the following folders:

- /source - Contains application project files with detailed comments. You can import this source code into Eclipse or some other IDE as a Maven project.
- /application - Contains pre-compiled binaries.

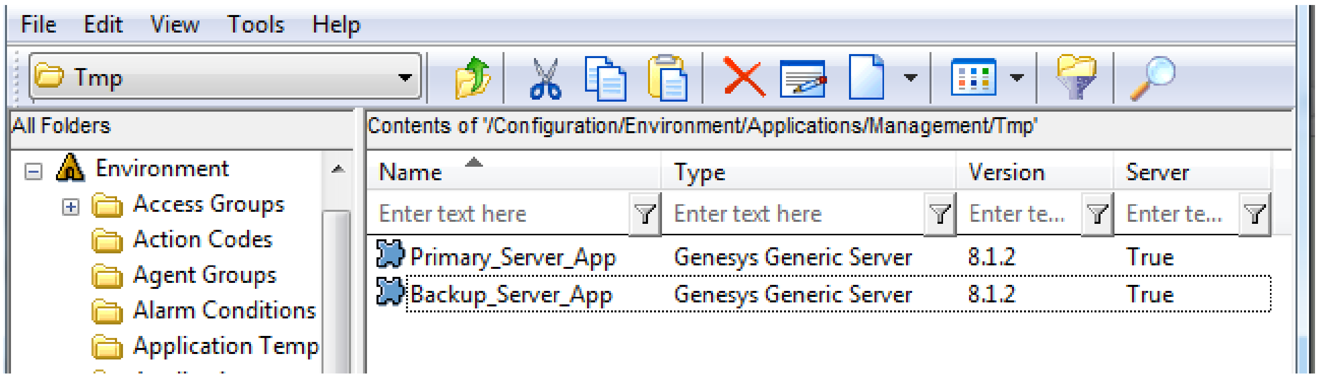
Before running this sample application, you need to complete two tasks:

1. Create and configure a pair of primary and backup applications in Genesys Management Framework.
2. Copy the binary files to correct locations, and adjust the application properties.

These tasks are described in more detail below.

Genesys Environment Configuration

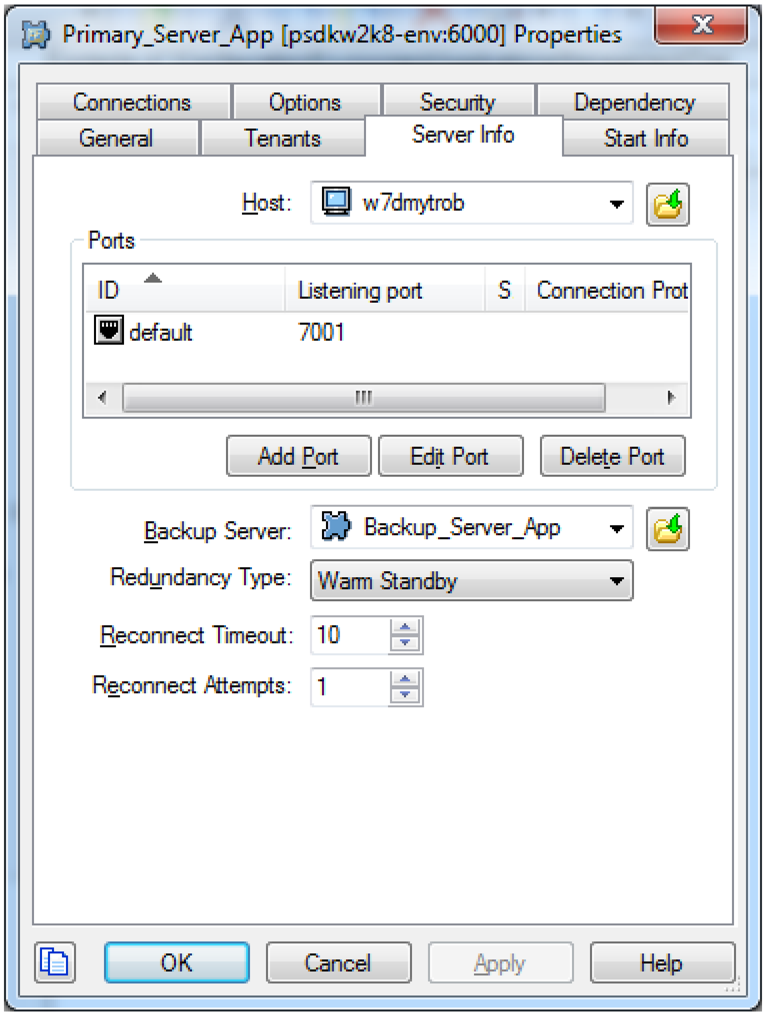
1. Open Configuration Manager and create two application in Genesys Management Framework: one for the primary server and a second for the backup server. Select `Genesys Generic Server` type for these applications.



Contents of '/Configuration/Environment/Applications/Management/Tmp'

Name	Type	Version	Server
Primary_Server_App	Genesys Generic Server	8.1.2	True
Backup_Server_App	Genesys Generic Server	8.1.2	True

2. Setup `Warm Standby` redundancy type for the `Primary_Server_App`, as shown below.

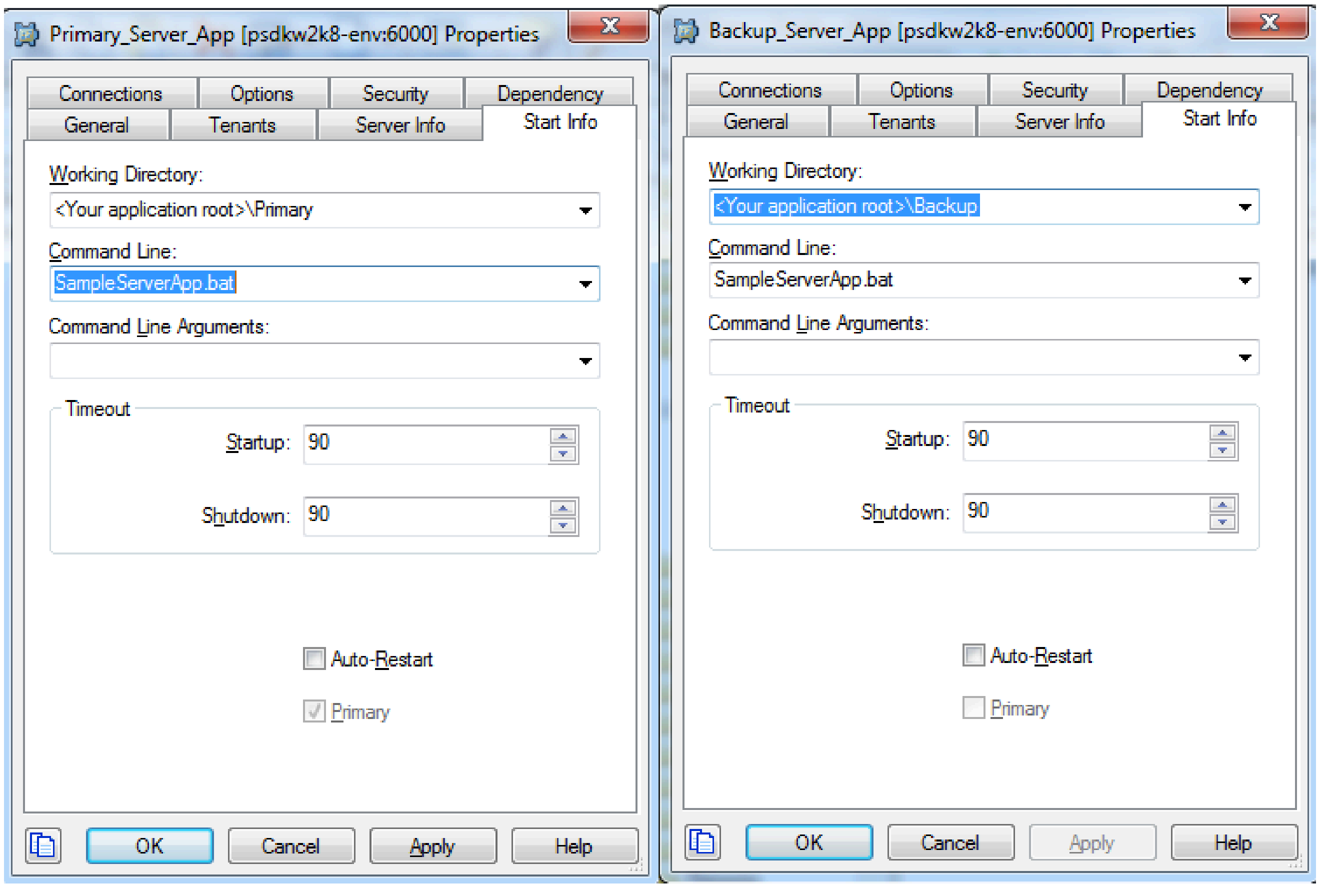


Host: w7dmytrob

ID	Listening port	S	Connection Prot
default	7001		

Backup Server: Backup_Server_App
Redundancy Type: Warm Standby
Reconnect Timeout: 10
Reconnect Attempts: 1

3. Open the `Start Info` tab to set up application start settings, as shown below.


Primary_Server_App [psdkw2k8-env:6000] Properties

Working Directory: <Your application root>\Primary
Command Line: SampleServerApp.bat
Command Line Arguments:
Timeout
Startup: 90
Shutdown: 90
[ ] Auto-Restart
[✓] Primary

Backup_Server_App [psdkw2k8-env:6000] Properties

Working Directory: <Your application root>\Backup
Command Line: SampleServerApp.bat
Command Line Arguments:
Timeout
Startup: 90
Shutdown: 90
[ ] Auto-Restart
[ ] Primary

Application Settings

1. Copy the content of the `/application` folder extracted from the ZIP file into the `<your application root>\Primary` and `<your application root>\Backup` folders.

2. Open `SampleServerApp.properties` file in the `Primary` folder.

3. Setup connection options to Configuration Server:

```
ConfigServerHost=host
ConfigServerPort=port
UserName=user name
Password=your password
ClientName=Primary_Server_App
```

4. Specify LCA port:

```
LCAPort=4999
```

5. Make similar settings for the Backup application, changing the `ClientName` property as shown below:

```
ClientName=Backup_Server_App
```

Run the Application

After these steps are complete, you can start Solution Control Interface and go to the applications status view. From here, you can start or stop the application, do switchover, etc.

The applications will write simple logs about their workflow and execution mode changes.

.NET

Overview

The `SampleServerApp_Net_814.zip` file contains the minimal amount of code required for an application to connect to LCA and maintain this connection in accordance with the Management Framework expectations. Download the sample code below, and then follow the steps below to set up and run this sample.

.NET Code Sample
SampleServerApp_Net_814.zip

Unpacking the ZIP file will give you access to a Visual Studio project with detailed comments in code.

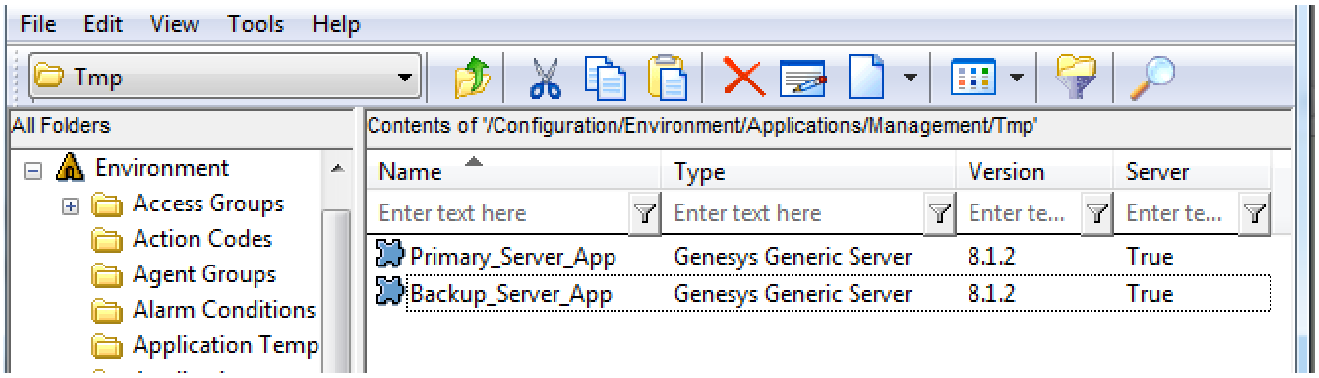
Before running this sample application, you need to build the project and then complete the following two tasks:

1. Create and configure a pair of primary and backup applications in Genesys Management Framework.
2. Copy the binary files to correct locations, and adjust the application properties.

These tasks are described in more detail below.

Genesys Environment Configuration

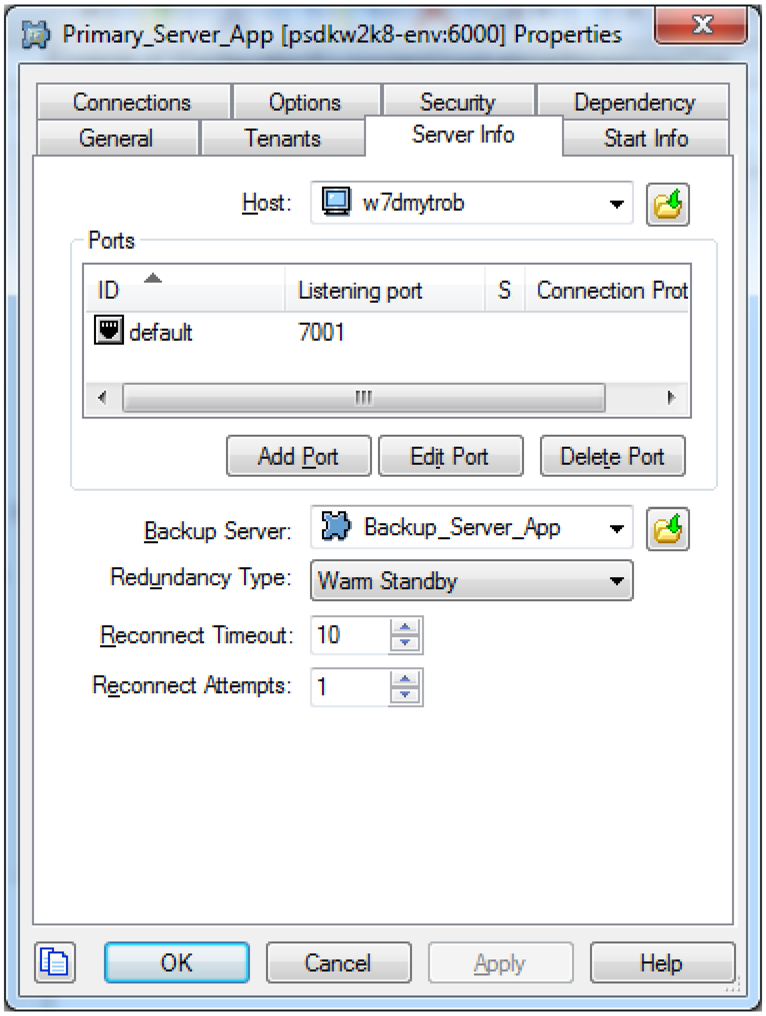
1. Open Configuration Manager and create two application in Genesys Management Framework: one for the primary server and a second for the backup server. Select `Genesys Generic Server` type for these applications.



Contents of '/Configuration/Environment/Applications/Management/Tmp'

Name	Type	Version	Server
Primary_Server_App	Genesys Generic Server	8.1.2	True
Backup_Server_App	Genesys Generic Server	8.1.2	True

2. Setup `Warm Standby` redundancy type for the `Primary_Server_App`, as shown below.



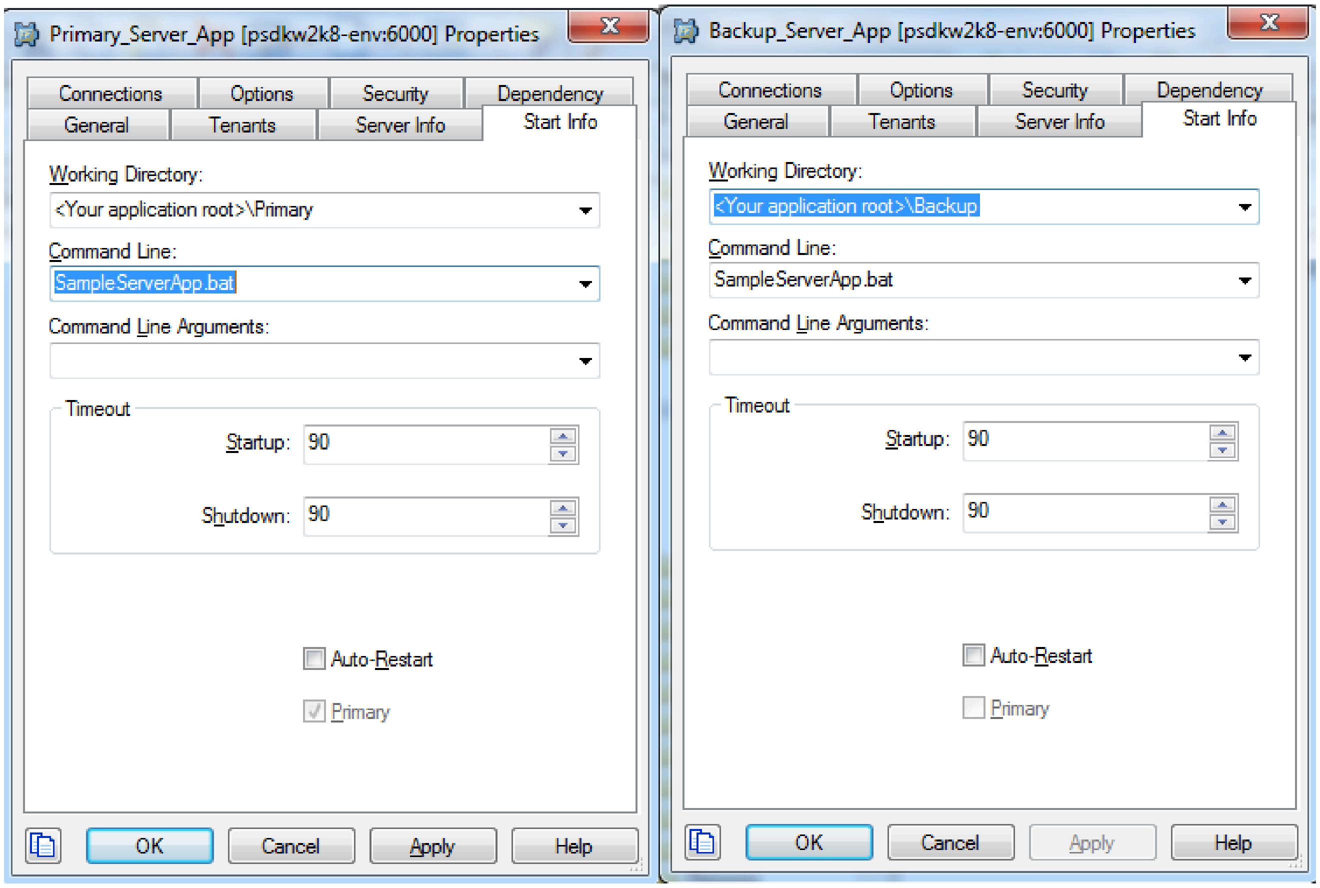
Host: w7dmytrob

Ports

ID	Listening port	S	Connection Prot
default	7001		

Backup Server: Backup_Server_App
Redundancy Type: Warm Standby
Reconnect Timeout: 10
Reconnect Attempts: 1

3. Open the `Start Info` tab to set up application start settings, as shown below.



Primary_Server_App [psdkw2k8-env:6000] Properties

Working Directory: <Your application root>\Primary

Command Line: SampleServerApp.bat

Command Line Arguments:

Timeout

Startup: 90

Shutdown: 90

[ ] Auto-Restart

[✓] Primary

Backup_Server_App [psdkw2k8-env:6000] Properties

Working Directory: <Your application root>\Backup

Command Line: SampleServerApp.bat

Command Line Arguments:

Timeout

Startup: 90

Shutdown: 90

[ ] Auto-Restart

[ ] Primary

Application Settings

1. Build the Visual Studio project that you extracted from the ZIP file and copy the results into the `<your application root>\Primary` and `<your application root>\Backup` folders.

2. Open `SampleServerApp.exe.config` file in the `Primary` folder.

3. Setup connection options to Configuration Server and Local Control Agent:

```
<appSettings>
  <add key="ConfigServerHost" value="host"/>
  <add key="ConfigServerPort" value="port"/>
  <add key="UserName" value="user name"/>
  <add key="Password" value="password"/>
  <add key="ClientName" value="Primary_Server_App"/>
  <add key="LCAPort" value="4999"/>
</appSettings>
```

5. Make similar settings for the Backup application, changing the `ClientName` property as shown below:

```
<add key="ClientName" value="Backup_Server_App"/>
```

Run the Application

After these steps are complete, you can start Solution Control Interface and go to the applications status view. From here, you can start or stop the application, do switchover, etc.

The applications will write simple logs about their workflow and execution mode changes.

LCA Hang-Up Detection Support

This page provides:

- an overview and list of requirements for the LCA Hang-Up Detection Support feature
- design details explaining how this feature works
- code examples showing how to implement LCA Hang-Up Detection Support in your applications

Introduction to LCA Hang-up Detection Support

Beginning with release 8.1, the Platform SDKs now allow user-developed application to include hang-up detection functionality.

The Genesys Management Layer relies on Local Control Agent (LCA) to monitor and control applications. An open connection between LCA and Genesys applications is typically used to determine which applications are running or stopped. However, if an application that has stopped responding still has a connection to LCA then it could appear to be running correctly - preventing Management Layer from switching over to a backup application or taking other actions to restore functionality.

Hang-up detection allows Local Control Agent (LCA) to detect unresponsive Genesys applications by checking for regular heartbeat messages. When an unresponsive application is found, pre-configured actions can be taken - including triggering alarms or restarting the application.

Tip

Hang-up detection functionality has been available in the Genesys Management Layer since release 8.0.1. For more information, refer to the Framework 8.0 Management Layer User's Guide. For details about related configuration options, refer to the Framework 8.0 Configuration Options Reference Manual.

Two levels of hang-up detection are available: implicit and explicit.

Implicit Hang-up Detection

The easiest form of hang-up detection to implement is implicit hang-up detection.

In this scenario, application status is monitored through the connection between your application and LCA. This functionality can be extended by adding a requirement that your application periodically interacts with LCA (either responding to ping request or sending its own heart-beat messages) as a necessary condition of application liveliness.

This simple form of hang-up detection can be implemented internally by using the

`LocalControlAgentProtocol` to connect to LCA. In this case, existing applications only need to be rebuilt with a version of `LocalControlAgentProtocol` that supports hang-up detection functionality - no coding changes are required - and given the appropriate configuration options in Genesys Management Layer.

Explicit Hang-up Detection

Explicit hang-up detection offers more robust protection from applications that may become unresponsive, but is also more complex.

The periodic interaction that is monitored by implicit hang-up detection only confirms that your application can interact with LCA. In most cases this means that the application is able to communicate with other apps and that the thread responsible for communicating with LCA is still active. However, multi-threaded applications may contain other threads that are blocked or have stopped responding without interrupting communication with LCA. Explicit hang-up detection allows you to determine when only part of your application hangs-up by monitoring individual threads in the application.

In addition to allowing your application to register (or unregister) individual threads to be monitored, explicit hang-up detection also allows your application to stop or delay the monitoring process. Threads that execute synchronous functions (which can block thread execution for some extended periods) or other features that prevent accurate monitoring should take advantage of this feature.

Feature Overview

- To maintain backwards compatibility, hang-up detection must be explicitly enabled in the application configuration.
- Implicit hang-up detection can be used for applications that do not require complex monitoring functionality. No code changes are required, just rebuild your application using the new version of `LocalControlAgentProtocol`.
- Explicit hang-up detection requires minimal application participation - enabling monitoring, registering and unregistering execution threads, and providing heartbeats. Most hang-up detection functionality is implemented within the Management Layer component, while all timing information (such as maximum allowed period between heartbeats) is configured through Genesys Management Layer.

Design Details

This section provides an overview of the main classes and interfaces used to add thread monitoring functionality for Explicit hang-up detection. Before using the classes and methods described here, be sure that you have implemented basic LCA Integration in your application using `LocalControlAgentProtocol`.

Although the details of thread monitoring implementation are slightly differently for Java and .NET, the basic idea is the same: to create and update a thread monitoring table that LCA can use to confirm the status of your application.

Note that for implicit hang-up detection you are only required to rebuild your application and make adjustments to the configuration options in Genesys Management Layer; the details described below are not required for simple application monitoring.

Thread Monitoring Table

The new thread monitoring functions described below allow `LocalControlAgentProtocol` to create and maintain a thread monitoring table within the application. This table tracks basic thread status.

Sample Thread Monitoring Table

OS Thread ID	Logical Thread ID	Thread Class	Heartbeat Counter	Flags
0	«main»	1	444345	active
1	«pool_1»	2	354354	suspend
2	«pool_2»	2	432432	deleted
3	«pool_3»	2	434323	active
4	«DB_store»	3	31212	active
....	....	....	....	....

Each row corresponds to a monitored thread. Columns of the table are:

- OS Thread ID—The OS-specific thread ID, used for thread identification during monitoring. OS thread ID is not passed by application but is received directly from system.
- Logical Thread ID – Application logical thread ID (or logical name, in Java). Used for logging and thread identification.
- Thread Class—Thread class integer. This value is only meaningful within the scope of the application; threads with the same thread class value in a different application can have different roles. Examples of thread classes might be the main loop thread, pool threads, or special threads (such as external authentication threads in ConfigServer).
- Heartbeat Counter—Cumulative counter of `Heartbeat()` calls made by the corresponding thread. Incrementing this value is the main way to indicate that the thread is still alive.

> **Tip**
> This value is initialized with a random value when the thread is registered for monitoring. This prevents incorrect hang-up detection if threads are created and terminated with high frequency, leading to repeating OS thread IDs.

- Flag—Special flags.
  - Suspended/Resumed—Corresponds to the state of thread monitoring.
  - Deleted—Used internally to notify LCA that a thread was unregistered from monitoring.

.NET Implementation

ThreadMonitoring Class

The `ThreadMonitoring` class is defined in the `Genesyslab.Diagnostics` namespace of Genesyslab.Core.dll. This class contains the following public static methods:

- `Register(int threadClass, string threadLogicId)`—enables monitoring for this thread
- `Unregister()`—removes this thread from monitoring
- `Heartbeat()`—increases heartbeat counter for this thread (indicating that thread is still alive)
- `SuspendMonitoring()`—suspend monitoring for this thread
- `ResumeMonitoring()`—resumes monitoring for this thread

Tip

Each method should be called from within the thread that is being monitored.

When a thread is registered for monitoring, the following parameters are included:

- `threadClass`—Any positive integer that represents the type of thread, allowing you to specify different monitoring settings for groups of threads within an application.
- `threadLogicId`—A logical, descriptive thread ID that is independent from thread ID provided by OS. This value is used for thread identification within LCA and for logging purposes. This ID should be unique within the application.

PerformanceCounter Constants

The following String constants (names) are defined in the `ThreadMonitoring` class:

```
public const string CategoryName = "Genesyslab PSDK .NET";
public const string HeartbeatCounterName = "Thread Heartbeat";
public const string StateCounterName = "Thread State";
public const string ProcessIdCounterName = "ProcessId";
public const string OsThreadIdCounterName = "OsThreadId";
```

The Platform SDK thread monitoring functionality uses these constants to manage PerformanceCounter values. In addition to these custom performance counters, you can also use standard ones, such as those defined in `Thread` category: "% Processor Time", "% User Time", etc.

See MSDN PerformanceCounter Class for details about performance counters.

Tip

Use of PerformanceCounters is optional, and is not required for LCA hang-up detection functionality.

Java Implementation

ThreadHeartbeatCounter class

The `ThreadHeartbeatCounter` class is defined in the `com.genesyslab.platform.commons.threading` package, located within commons.jar. This class is designed as a JMX (see JMX: Java Management Extensions) MBean and implements the public ThreadHeartbeatCounterMBean interface which is accessible through Java management framework.

There is no public constructor for the `ThreadHeartbeatCounter` class; each thread that you want to monitor should create its own instance with following static method:

```
public static ThreadHeartbeatCounter createThreadHeartbeatCounter(
            String threadLogicalName,
            int threadClass);
```

When a thread is registered for monitoring, the following parameters are included:

- `threadLogicalName`—A logical, descriptive thread name that is used to identify the thread within LCA and for logging purposes. This name should be unique within the application.
- `threadClass`—Any positive integer that represents the type of thread, allowing you to specify different monitoring settings for groups of threads within an application.

One key difference from thread monitoring using .NET is the need to create a monitoring object instance. The lifecycle of this object, including `MBeanServer` registration, is supported by the parent class `PSDKMBeanBase` and is shown in the five steps below:

1. Start monitoring a thread:

```
ThreadHeartbeatCounter monitor =
  ThreadHeartbeatCounter.createThreadHeartbeatCounter(
                threadId, threadClass);
monitor.initialize();
```

2. Notify LCA that thread is still alive (increase heartbeat counter):

```
monitor.alive();
```

3. Suspend monitoring of this thread:

```
monitor.setActive(false);
```

4. Resume monitoring of this thread:

```
monitor.setActive(true);
```

5. Finish monitoring and unregister this thread:

```
monitor.unregister();
```

> **Tip**
> Each of these methods must be called from within the thread that is being monitored.

Once a `ThreadHeartbeatCounter` object is unregistered, that instance cannot be reused. To begin monitoring that thread again (or any other) you first need to create a new instance of the thread monitoring object.

ThreadHeartbeatCounterMBean interface

The `ThreadHeartbeatCounterMBean` interface is intended to present an open API to the JMX MBean. This interface contains the following publicly accessible methods:

```
public long getThreadSystemId();
public String getLogicalName();
public int getThreadClass();
public void setThreadClass(int newThreadClass);
public int getHeartbeatCounter();
public void setActive(boolean isActive);
public boolean isActive();
```

These methods are "MBean client-side" methods and are used by LCA protocol to get actual information about the thread for the monitoring table. They also allow users to change the thread class and suspend or resume thread monitoring (using `setActive(false/true)`) of a particular thread at application runtime.

Handle Application "Graceful Stop" with the LCA Protocol

Graceful stop is operation that allow application to complete current request handling before actual stop in case when such handling can require significant (time greater than application stop timeout) time.

Two new states are related to this command:

- SUSPENDING – This state means that an application has understood the command from LCA, so that the application does NOT accept new requests (unless specified by Management Layer) and will complete current requests. This status should be reported by an application as the result of Suspend command from LCA. If the application does no support graceful stop then it can just ignore the Suspend command; no state changes should be reported in this case.
- SUSPENDED – This state means that an application has completed handling current requests and can be stopped without any impact to its client.

For applications which DO support graceful stop, the scenario is as follows:

1. SCI issues command "Stop application graceful"
2. SCS receives command, sets up a suspended state timer and sends the Suspend application command to LCA
3. LCA receives Suspend command and resends it to application (application receives `EventSuspendApplication`)
4. Since the application supports this feature, it reports SUSPENDING state with `RequestUpdateStatus` and start behave accordingly
5. SCS receives a status update through LCA and cancels the timer set at point (2).
6. From this point, the application has unlimited time to complete handling requests (of course, it can also be stopped by usual stop command)
7. When the application completes handling requests, it reports the status SUSPENDED (with `RequestUpdateStatus`)
8. SCS receive the status update through LCA and stops the application with the usual stop command (`EventChangeExecutionMode`)

For applications which do NOT support graceful stop, the scenario is as follows:

1. SCI issues the command: "Stop application graceful"
2. SCS receives the command, sets up a suspended state timer and sends the Suspend application command to LCA
3. LCA receives Suspend command and resends it to application
4. Since the application does not support this feature, the Suspend command is ignored
5. Suspended state timer set at point (2) expires in SCS. SCS determines that the application does not support Graceful stop.

6. SCS issues an ordinary stop command to application (`EventChangeExecutionMode`).

Please also note that message between SCI-SCS and SCS-LCA are not same.

If your application should support graceful stop please check:

- If application receives Suspend command from LCA.
- If application correctly report SUSPENDING/SUSPENDED states.
- If application can be stopped by usual stop command.

If your application should NOT support graceful stop please check:

- If application can be stopped by usual stop command (in this case Graceful stop is equal to usual Stop command with some delay)

Tip

This feature is new for 8.0 Genesys Management Framework, so, all the involved components (SCI, SCS, LCA) should be 8.0+ versions (checked with 8.0.3x).

Routing Server

Many types of interactions enter a modern contact center, and each of them can have many possible destinations. Universal Routing Server (URS) helps them get to the right place at the right time by enabling you to create customized *routing strategies* — sets of instructions that tell URS how to handle interactions. These routing strategies can be as simple or complex as you need them to be. URS uses routing strategies to send interactions from one *target* to another, as needed, until the interactions have been successfully processed.

The Routing Platform SDK allows you to write .NET applications that combine logic from your custom application with the router-based logic of URS, in order to solve many common interaction-related tasks.

This document tells you where you can go to get more information about URS. It also contains a brief overview of the features of the Routing Platform SDK, followed by code snippets that show how to implement the basic functions you will need to write applications that work with URS.

Universal Routing Server Overview

The best way to start learning about Universal Routing Server (URS) is by getting a copy of the Universal Routing 8.1 Reference Manual. This book tells you how to work with routing strategies, objects, functions, options, and statistics. It also includes a detailed list of *Related Documentation Resources*, which discusses other sources of information that can be useful when you are working with Genesys Universal Routing.

After becoming familiar with the information in the *Universal Routing Reference Manual* and related documentation, you can start using the routing API that is exposed by the Platform SDK. As you learn about Genesys routing, it is important to keep in mind that the main purpose of the Platform SDK routing API is to work as a complement to the complex capabilities already available from URS, not to act as a replacement. This API makes it easier to resolve difficult interaction-related tasks by combining the capabilities of URS with logic from your custom application.

To create routing strategies, you use either Genesys Composer, which lets you create SCXML-based strategies, or Interaction Routing Designer (IRD), which creates strategies in the Genesys IRL routing language. Once the URS environment is established, you then use the Platform SDK routing API to give your application control over which routing strategies are selected under a given set of circumstances or what criteria URS uses to choose a particular routing target. For example, your application can select statistics for URS to use in determining which agent group would be the best one to route a particular interaction to.

Two Types of Router API Usage

Platform SDK lets you use two different methodologies in working with URS. The first method involves a **standalone router**. When you use the standalone router method, all of the interaction processing logic, including media control, is handled by the router. This method can be used by calling `RequestLoadStrategy`.

The second method is called the **router-behind** API. This method can be used when you want your application to handle media control, such as attached data or treatments, rather than leaving that up to the router. With this method, the router is normally used only to select resources.

The code snippets in this article include some requests that work with standalone routers and some that work with the router-behind API.

Java

Connecting to Universal Routing Server

The Platform SDK uses a message-based architecture to connect to Genesys servers. The following code samples show how to connect to URS by using the native protocol object that is part of the Routing Platform SDK.

First set up `import` statements for the routing namespaces:

```
import com.genesyslab.platform.routing.protocol.routingserver.*;
import com.genesyslab.platform.routing.protocol.routingserver.requests.*;
```

After you have set up your import statements, you need to create a `RoutingServerProtocol` object:

```
RoutingServerProtocol protocol =
        new RoutingServerProtocol(
                new Endpoint(
                        name, host, port));
protocol.setClientName(clientName);
protocol.setClientType(clientType);
```

Then you can open your connection to URS:

```
protocol.open();
```

Message Handling

Once you have set up your connection to the server, your application needs to be able to send requests to and receive messages from URS.

Tip

This section describes one possible way to provide message handling. For more details and other options regarding message handling, refer to the full Event Handling article.

Messages can be received asynchronously by setting up a `MessageHandler` object, and then providing code to execute for each type of message that your application expects to respond to. Once this object is created, you can assign it to the protocol used to connect to the URS server, as shown below.

```
MessageHandler ursMessageHandler = new MessageHandler() {
        @Override
        public void onMessage(Message message) {
                // your event-handling code goes here
                switch (message.messageId()) {
                        case EventInfo.ID:
                                OnEventInfo(message);
                                break;
                        ...
                }
        }
};
protocol.setMessageHandler(ursMessageHandler);
```

For each message type you intend to respond to, your application must include:

- an additional section inside the switch block that checks to ensure the message ID matches
- a function that defines the desired behavior when reacting to that message type

In the example above, the message ID being checked for would match an incoming `EventInfo` message, while the resulting behavior (inside an `OnEventInfo` function that you would create) has yet to be defined.

Working with URS

As mentioned above, there are two basic methods for using the Platform SDK to work with URS. This section contains examples of both the standalone router and router-behind APIs.

Standalone Router

The Routing Platform SDK allows you to control which routing strategy is executed on a given routing point, while leaving everything else to the routing server. To use this methodology, which is known as "standalone router," issue a `RequestLoadStrategy` that specifies the routing point and the associated T-Server, and also the location of the routing strategy. Once the routing strategy has been loaded, all interactions arriving on the specified routing point will be processed with that strategy.

The following snippet shows how to do this:

```
RequestLoadStrategy requestLoadStrategy = RequestLoadStrategy.create();
requestLoadStrategy.setTServer("TheT-Server");
requestLoadStrategy.setRoutingPoint("TheRoutingPoint");
requestLoadStrategy.setPath("<Path to the strategy>");

protocol.send(requestLoadStrategy);
```

URS will respond to your request with an `EventInfo`, an example of which is shown here:

```
'EventInfo' (2) attributes:
                R_Message [str] = "ATTENTION: Strategy has been loaded from ooo-file."
                R_cdn_status [int] = 1 [Loaded]
                R_cdn [str] = "RP_sip1"
                R_ErrorCode [int] = 0 [NoError]
                R_tserver [str] = "TServerSip1"
                R_refID [int] = 1
                R_time [str] = "06/30/2011 10:00:29"
                R_path [str] = "<Path>"
```

You can use `RequestNotify` to check which routing points have been loaded:

```
RequestNotify requestNotify = RequestNotify.create();
protocol.send(requestNotify);
```

This request will also return an `EventInfo` similar to the one shown above.

When you want to stop using the routing strategy you have loaded, for example, if you want to start using a different one, you can issue a `RequestReleaseStrategy`:

```
RequestReleaseStrategy requestReleaseStrategy =
    RequestReleaseStrategy.create();
requestReleaseStrategy.setTServer("TheT-Server");
requestReleaseStrategy.setRoutingPoint("TheRoutingPoint");

Message response = protocol.request(requestReleaseStrategy);
```

Router-Behind API

The router-behind method allows your application code to handle media control. The following example shows how to execute a strategy using `RequestExecuteStrategy`. This request is different from `RequestLoadStrategy` in that it only executes a strategy one time, instead of associating a particular strategy with a routing point. To use `RequestExecuteStrategy`, specify the routing strategy you want to execute and the tenant (contact center) in whose environment the strategy is to be executed, as shown here:

```
RequestExecuteStrategy requestExecuteStrategy =
        RequestExecuteStrategy.create();
requestExecuteStrategy.setStrategy("TheRoutingStrategyName");
requestExecuteStrategy.setTenant("TheTenant");

protocol.send(requestExecuteStrategy);
```

It is important to remember that it can often take a considerable amount of time to process a routing strategy. If your request is correctly formatted and can be executed by URS, then you will immediately receive an `EventExecutionInProgress`. This is a very simple event that only returns the reference ID of your request, as you can see here:

```
'EventExecutionInProgress' ('199')
message attributes:
R_refID [int]   = 2
```

Once your request has successfully executed, you will receive an `EventExecutionAck`. Here is an example of the kind of output you might receive from an `EventExecutionAck`:

```
'EventExecutionAck' ('200')
message attributes:
R_result [bstr] = KVList:
      'DN' [str] = "701"
      'CUSTOMER_ID' [str] = "TenantForTest"
      'TARGET' [str] = "701_sip@StatServer1.A"
      'SWITCH' [str] = "SipSwitch"
      'NVQ' [int] = 1
      'PLACE' [str] = "701"
      'AGENT' [str] = "701_sip"
      'ACCESS' [str] = "701"
      'VQ' [str] = "1234"
      Context         = ComplexClass(OperationContext):
      UserData [bstr] = KVList:
             'ServiceObjective' [str] = ""
             'ServiceType' [str] = "default"
             'CBR-Interaction_cost' [str] = ""
             'RTargetTypeSelected' [str] = "0"
             'CBR-IT-path_DBIDs' [str] = ""
             'RVQDBID' [str] = ""
             'RTargetPlaceSelected' [str] = "701"
             'RTargetAgentSelected' [str] = "701_sip"
             'CBR-actual_volume' [str] = ""
             'RStrategyName' [str] = "##GetTarget"
             'RRequestedSkillCombination' [str] = ""
             'RTargetRuleSelected' [str] = ""
             'RStrategyDBID' [str] = ""
             'RRequestedSkills' [bstr] = KVList:
             'CustomerSegment' [str] = "default"
             'RTargetObjSelDBID' [str] = "984"
             'RTargetObjectSelected' [str] = "701_sip"
             'RTenant' [str] = "TenantForTest"
             'RVQID' [str] = ""
             'CBR-contract_DBIDs' [str] = ""
R_refID [int]   = 0
```

If you have any syntax errors, your request will not execute and you will receive an `EventError`. Here is an example of an `EventError`:

```
'EventError' (1) attributes:
                R_cdn_status [int] = 0 [Released]
                R_cdn [str] = ""
                R_ErrorCode [int] = 4 [NotAvailable]
                R_tserver [str] = ""
                R_refID [int] = 1
                R_time [str] = ""
                R_path [str] = "<Path>"
```

If, on the other hand, URS has a problem executing your request, you will receive an `EventExecutionError`, an example of which is shown here:

```
'EventExecutionError' ('201')
message attributes:
R_result [bstr] = KVList:
                                'Reason' [str] = "Rejected"
Context         = ComplexClass(OperationContext):
                UserData [bstr] = KVList:
                                                'PegRejected' [int] = 1
R_refID [int]   = 2
```

There may be times when you want URS to pick a routing target for you. You can use `RequestFindTarget` for that purpose. As shown in the sample below, you can use a statistic to aid in this selection:

```
RequestFindTarget requestFindTarget =
    RequestFindTarget.create();
```

```
requestFindTarget.setTenant("TheTenant");
requestFindTarget.setTargets("TheTargetList");
requestFindTarget.setTimeout(5);
requestFindTarget.setStatistic("TheStatistic");
requestFindTarget.setStatisticUsage(StatisticUsage.Max);
requestFindTarget.setVirtualQueue("TheQueue");
requestFindTarget.setPriority(1);
requestFindTarget.setMediaType("TheMediaType");

protocol.send(requestFindTarget);
```

You can also have URS fetch statistical information for you directly, in case you want to know more about the current conditions in your contact center, perhaps in preparation for a `RequestFindTarget`. The following example shows how to do this, using `RequestGetStatistic`:

```
RequestGetStatistic requestGetStatistic =
        RequestGetStatistic.create();
requestGetStatistic.setTenant("TheTenant");
requestGetStatistic.setTargets("TheTargetList");
requestGetStatistic.setStatistic("StatAgentsBusy");

protocol.send(requestGetStatistic);
```

Both `RequestFindTarget` and `RequestGetStatistic` return the same messages as `RequestExecuteStrategy`.

Closing the Connection

When you are finished communicating with URS, you should close the connection, in order to minimize resource utilization:

```
protocol.close();
```

.NET

Connecting to Universal Routing Server

The Platform SDK uses a message-based architecture to connect to Genesys servers. The following code samples show how to connect to URS by using the native protocol object that is part of the Routing Platform SDK.

First set up using statements for the routing namespaces:

```
using Genesyslab.Platform.Routing.Protocols;
using Genesyslab.Platform.Routing.Protocols.RoutingServer;
using Genesyslab.Platform.Routing.Protocols.RoutingServer.Events;
using Genesyslab.Platform.Routing.Protocols.RoutingServer.Requests;
```

After you have set up your using statements, you need to create a `RoutingServerProtocol` object:

```
RoutingServerProtocol protocol =
        new RoutingServerProtocol(
                new Endpoint(
                        name, host, port));
protocol.ClientName = clientName;
protocol.ClientType = clientType;
```

Then you can open your connection to URS:

```
protocol.Open();
```

Message Handling

Once you have set up your connection to the server, your application needs to be able to send requests to and receive messages from URS.

Tip

This section describes one possible way to provide message handling. For more details and other options regarding message handling, refer to the full Event Handling article.

Messages can be received asynchronously by subscribing to the `Received` .NET event:

```
protocol.Received += OnURSMessageReceived;
```

Once subscribed, your application must provide code to handle each type of message that is expected from the protocol object, as shown below.

```
void OnURSMessageReceived(object sender, EventArgs e)
{
        IMessage message = ((MessageEventArgs)e).Message;
        // your event-handling code goes here
        switch (message.Id)
        {
                case EventInfo.MessageId:
                        OnEventInfo(message);
                        break;
                ...
        }
}
```

For each message type you intend to respond to, your application must include:

- an additional section inside the switch block that checks to ensure the message ID matches
- a function that defines the desired behavior when reacting to that message type

In the example above, the message ID being checked for would match an incoming `EventInfo` message, while the resulting behavior (inside an `OnEventInfo` function that you would create) has yet to be defined.

Working with URS

As mentioned above, there are two basic methods for using the Platform SDK to work with URS. This section contains examples of both the standalone router and router-behind APIs.

Standalone Router

The Routing Platform SDK allows you to control which routing strategy is executed on a given routing point, while leaving everything else to the routing server. To use this "standalone router" methodology, issue a `RequestLoadStrategy` that specifies the routing point and the associated T-Server, and also the location of the routing strategy. Once the routing strategy has been loaded, all interactions arriving on the specified routing point will be processed with that strategy.

The following snippet shows how to do this:

```
RequestLoadStrategy requestLoadStrategy =
    RequestLoadStrategy.Create();
requestLoadStrategy.TServer = "TheT-Server";
requestLoadStrategy.RoutingPoint = "TheRoutingPoint";
requestLoadStrategy.Path = "<Path to the strategy>";

protocol.Send(requestLoadStrategy);
```

URS will respond to your request with an `EventInfo`, an example of which is shown here:

```
'EventInfo' (2) attributes:
                R_Message [str] = "ATTENTION: Strategy has been loaded from ooo-file."
                R_cdn_status [int] = 1 [Loaded]
                R_cdn [str] = "RP_sip1"
                R_ErrorCode [int] = 0 [NoError]
                R_tserver [str] = "TServerSip1"
                R_refID [int] = 1
                R_time [str] = "06/30/2011 10:00:29"
                R_path [str] = "<Path>"
```

You can use `RequestNotify` to check which strategies have been loaded to routing points:

```
RequestNotify requestNotify =
    RequestNotify.Create();

protocol.Send(requestNotify);
```

This request will also return an `EventInfo` similar to the one shown above.

When you want to stop using the routing strategy you have loaded, for example, if you want to start using a different one, you can issue a `RequestReleaseStrategy`:

```
RequestReleaseStrategy requestReleaseStrategy =
    RequestReleaseStrategy.Create();
requestReleaseStrategy.TServer = "TheT-Server";
requestReleaseStrategy.RoutingPoint = "TheRoutingPoint";

protocol.Send(requestReleaseStrategy);
```

Router-Behind API

The router-behind method allows your application code to handle media control. The following example shows how to execute a strategy using `RequestExecuteStrategy`. This request is different from `RequestLoadStrategy` in that it only executes a strategy one time, instead of associating a particular strategy with a routing point. To use `RequestExecuteStrategy`, specify the routing strategy you want to execute and the tenant (contact center) in whose environment the strategy is to be executed, as shown here:

```
RequestExecuteStrategy requestExecuteStrategy =
        RequestExecuteStrategy.Create();
requestExecuteStrategy.Strategy = "TheRoutingStrategyName";
requestExecuteStrategy.Tenant = "TheTenant";

protocol.Send(requestExecuteStrategy);
```

It is important to remember that it can often take a considerable amount of time to process a routing strategy. If your request is correctly formatted and can be executed by URS, then you will immediately receive an `EventExecutionInProgress`. This is a very simple event that only returns the reference ID of your request, as you can see here:

```
'EventExecutionInProgress' ('199')
message attributes:
R_refID [int]   = 2
```

Once your request has successfully executed, you will receive an `EventExecutionAck`. Here is an example of the kind of output you might receive from an `EventExecutionAck`:

```
'EventExecutionAck' ('200')
message attributes:
R_result [bstr] = KVList:
                                'DN' [str] = "701"
                                'CUSTOMER_ID' [str] = "TenantForTest"
                                'TARGET' [str] = "701_sip@StatServer1.A"
                                'SWITCH' [str] = "SipSwitch"
                                'NVQ' [int] = 1
                                'PLACE' [str] = "701"
                                'AGENT' [str] = "701_sip"
                                'ACCESS' [str] = "701"
                                'VQ' [str] = "1234"
Context         = ComplexClass(OperationContext):
                UserData [bstr] = KVList:
                                                'ServiceObjective' [str] = ""
                                                'ServiceType' [str] = "default"
                                                'CBR-Interaction_cost' [str] = ""
                                                'RTargetTypeSelected' [str] = "0"
                                                'CBR-IT-path_DBIDs' [str] = ""
                                                'RVQDBID' [str] = ""
                                                'RTargetPlaceSelected' [str] = "701"
                                                'RTargetAgentSelected' [str] = "701_sip"
                                                'CBR-actual_volume' [str] = ""
                                                'RStrategyName' [str] = "##GetTarget"
                                                'RRequestedSkillCombination' [str] = ""
                                                'RTargetRuleSelected' [str] = ""
                                                'RStrategyDBID' [str] = ""
                                                'RRequestedSkills' [bstr] = KVList:

                                                'CustomerSegment' [str] = "default"
                                                'RTargetObjSelDBID' [str] = "984"
                                                'RTargetObjectSelected' [str] = "701_sip"
                                                'RTenant' [str] = "TenantForTest"
                                                'RVQID' [str] = ""
                                                'CBR-contract_DBIDs' [str] = ""
R_refID [int]   = 0
```

If you have any syntax errors, your request will not execute and you will receive an `EventError`. Here is an example of an `EventError`:

```
'EventError' (1) attributes:
                R_cdn_status [int] = 0 [Released]
                R_cdn [str] = ""
                R_ErrorCode [int] = 4 [NotAvailable]
                R_tserver [str] = ""
                R_refID [int] = 1
                R_time [str] = ""
                R_path [str] = "<Path>"
```

If, on the other hand, URS has a problem executing your request, you will receive an `EventExecutionError`, an example of which is shown here:

```
'EventExecutionError' ('201')
message attributes:
R_result [bstr] = KVList:
                                'Reason' [str] = "Rejected"
Context         = ComplexClass(OperationContext):
                UserData [bstr] = KVList:
                                                'PegRejected' [int] = 1
R_refID [int]   = 2
```

There may be times when you want URS to pick a routing target for you. You can use `RequestFindTarget` for that purpose. As shown in the sample below, you can use a statistic to aid in this selection:

```
RequestFindTarget requestFindTarget =
    RequestFindTarget.Create();
requestFindTarget.Tenant = "TheTenant";
requestFindTarget.Targets = "TheTargetList";
requestFindTarget.Timeout = 5;
requestFindTarget.Statistic = "TheStatistic";
requestFindTarget.StatisticUsage = StatisticUsage.Max;
requestFindTarget.VirtualQueue = "TheQueue";
requestFindTarget.Priority = 1;
requestFindTarget.MediaType = "TheMediaType";

protocol.Send(requestFindTarget);
```

You can also have URS fetch statistical information for you directly, in case you want to know more about the current conditions in your contact center, perhaps in preparation for a `RequestFindTarget`. The following example shows how to do this, using `RequestGetStatistic`:

```
RequestGetStatistic requestGetStatistic =
    RequestGetStatistic.Create();
requestGetStatistic.Tenant = "TheTenant";
requestGetStatistic.Targets = "TheTargetList";
requestGetStatistic.Statistic = "StatAgentsBusy";

protocol.Send(requestGetStatistic);
```

Both `RequestFindTarget` and `RequestGetStatistic` return the same messages as `RequestExecuteStrategy`.

Closing the Connection

When you are finished communicating with URS, you should close the connection, in order to minimize resource utilization:

```
protocol.Close();
```

Component Overviews

Component Overviews

- Using the Log Library

Using the Log Library

Java

The purpose of the Platform SDK Log Library is to present an easy-to-use API for logging messages in custom-built applications. Depending on how you configure your logger, you can quickly and easily have log messages of different verbose levels written to any of the following targets:

- Genesys Message Server
- Console
- Specified log files

This document provides some key considerations about how to configure and use this component, as well as code examples that will help you work with the Platform SDK Log Library in your own projects.

Introduction to Loggers

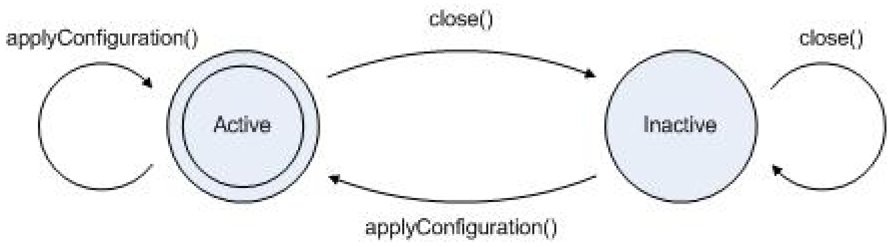
When working with custom Genesys loggers, the first step is to understand the basic process of creating and maintaining your logger. How do you create a logger instance? What configuration options should you use, and how and when can those options be changed? What is required to clean up once the logger is no longer useful?

Luckily, the main functions and lifecycle of a logger are easy to understand. The following list outlines the basic process required to create and maintain your customized logger. For more detailed information, check the *Lifecycle of a Logger section*, or the specific code examples related to *Creating a Logger*, *Customizing your Logger*, *Using your Logger*, or *Cleaning Up Your Code*.

1. Use the `LoggerFactory` to create a `RootLogger` instance.
2. Reconfigure the default `RootLogger` settings, if desired.
   - Create a `LogConfiguration` instance to enable and configure log targets. Depending on the setting you assign, log messages can be sent to the Console, to a Genesys Message Server, or to one or more user-defined log files.
   - `LoggerPolicy` property gives you control over how log messages are created and formatted, or allows you to overwrite `MessageServerProtocol` properties.
3. Use your logger for logging messages.
4. Dispose of the `RootLogger` instance when it is no longer needed. (You can also close the logger if it will be reused in the future.)

Lifecycle of a Logger

There are two possible states for a logger, as shown in the lifecycle diagram below.



Your logger begins in the active state once it is created. If you did not specify any configuration options during creation, then all messages with a verbose level at least equal to `VerboseLevel.Trace` are logged to the Console by default.

You can use the `applyConfiguration` method to change logger configuration settings from either an active or inactive state:

- If the logger is active when you call this method, then all messages being processed will be handled before the logger is stopped and reconfigured. Note that any file targets (from both the old and new configurations) are not closed automatically when the logger is reconfigured, although file targets *can* be closed and a new log file segment started if the new `Segmentation` settings require this.
- If the logger is inactive when you call this method, then it is activated after the new configuration settings are applied.

You can use the `close` method to make the logger inactive without disposing of it. All messages being processed when this method is called are processed before the logger is stopped. Once the logger is inactive, no further messages are processed until after the `applyConfiguration` method is called.

Important

If your logger is connected to Message Server, the logger does not manage the lifecycle of the `MessageServerProtocol` instance. You must manage and close that connection manually.

Creating a Default Logger

This section provides simple code examples that show how to quickly create a logger with the default configuration and use it as part of your application.

Creating a Logger

As always, the starting point is ensuring that you have the necessary Platform SDK libraries referenced and declared in your project. For logging functionality described in this article, that includes the following packages:

- import com.genesyslab.platform.commons.log.*;
- import com.genesyslab.platform.commons.collections.KeyValueCollection;
- import com.genesyslab.platform.management.protocol.messageserver.LogLevel;
- import com.genesyslab.platform.logging.*;
- import com.genesyslab.platform.logging.configuration.*;
- import com.genesyslab.platform.logging.utilities.*;
- import com.genesyslab.platform.logging.runtime.LoggerException;

Once your project is properly configured and coding about to begin, the first task is to create an instance of the `RootLogger` class. This is easy to accomplish with help from the `LoggerFactory` - the only information you need to provide is a logger name that can be used later to configure targets for filtering logging output.

```
[Java]

try{
        // Create a logger instance:
        RootLogger logger = new LoggerFactory("myLoggerName").getRootLogger();
}
catch(LoggerException e){
        // Handle exceptions...
}
```

The default behavior for this logger is to send all messages of `Trace` verbose and higher to the Console. You can change this behavior by using a `LogConfiguration` instance to change configuration settings and then applying those values to the `RootLogger` instance, as shown below, but for now we will accept the default values.

Using your Logger

With a logger created and ready for use, the next step is to generate some custom log messages and ensure that your logger is working correctly.

One way to generate log messages is with the `write` method. Depending on the parameters used with this method, message formatting can either be provided by templates extracted from LMS files, or through the settings that you configure in a `LogEntry` instance. For the example below, the only formatting come from the `LogEntry` parameter.

```
[Java]

LogEntry logEntry;
logEntry = new LogEntry("Sample Internal message.");
logEntry.setId(CommonMessage.GCTI_INTERNAL.getId());
logger.write(logEntry);

logEntry.setMessage("Sample Debug message.");
logEntry.setId(CommonMessage.GCTI_DEBUG.getId());
logger.write(logEntry);
```

You can also generate log messages by using one of the methods listed in the following table.

Message Level	Available Methods
Debug	• debug(Object arg0) • debug(Object arg0, Throwable arg1) • debugFormat(String arg0, object arg1)
Info	• info(Object arg0) • info(Object arg0, Throwable arg1) • infoFormat(String arg0, object arg1)
Interaction	• warn(Object arg0) • warn(Object arg0, Throwable arg1) • warnFormat(String arg0, object arg1)
Error	• error(Object arg0) • error(Object arg0, Throwable arg1) • errorFormat(String arg0, object arg1)
Alarm	• fatalError(Object arg0) • fatalError(Object arg0, Throwable arg1) • fatalErrorFormat(String arg0, object arg1)

These methods do not use any external templates or formatting, relying entirely on the information passed into them. In the examples below, the messages are logged at `Info` and `Debug` verbose levels, without any changes or formatting.

```
[Java]

logger.info("Sample Info message.");
logger.debug("Sample Debug message.");
```

Cleaning Up Your Code

Once you have finished logging messages with your logger, there are two options available: you can close the logger if you want it to be available for reuse later, or dispose of the logger if your application doesn't need it any longer. (Note that you do not have to close a logger before disposing of it.)

Once closed, a logger remains in an inactive state until either the `ApplyConfiguration` method is called or you dispose of the object, as shown in the *Lifecycle of a Logger* diagram above.

```
[Java]
```

```
// closing the logger
logger.close();
// disposing of the logger
logger = null;
```

Customizing your Logger

Now that you know how to create and use a generic logger, it is time to look at some of the configuration options available to alter the behavior of your logger.

The `LogConfiguration` class allows you change application details, specify targets (including Genesys Message Server) for your log messages, and adjust the verbose level you want to report on. You can apply these changes to either a new logger that is created with the `LogFactory`, or to an existing logger by using the `ApplyConfiguration` method.

Tip

`MessageHeaderFormat` property in the `LogConfiguration` class has no effect on records in the Message Server Database due to message server work specificity.

Creating a LogConfiguration to Specify Targets and Verbose Levels

The first step to configuring the settings for your logger is creating an instance of the `LogConfigurationImpl` class and setting some basic parameters that describe your application.

```
[Java]

LogConfigurationImpl logConfigImpl = new LogConfigurationImpl();
logConfigImpl.setApplicationHost("myHostname");
logConfigImpl.setApplicationName("myApplication");
logConfigImpl.setApplicationId(10);
logConfigImpl.setApplicationType(20);
logConfigImpl.setVerbose(VerboseLevel.ALL);
```

Additional `LogConfiguration` properties that can be configured to specify the name of an application-specific LMS file (`MessageFile`) and whether timestamps should use local or UTC format (`TimeUsage`). These steps aren't shown here for brevity; refer to the API Reference for details.

Tip

If logging to the network, timestamps for log entries always use UTC format to avoid confusion. In this case the `TimeUsage` setting specified by your `LogConfiguration` is ignored.

The next step is to assign this implementation to an actual `LogConfiguration` instance. Once you do that, you can specify the target locations where log messages will be sent and the verbose levels accepted by each individual target. (Only messages with a level greater than or equal to the verbose setting will be logged.)

```
[Java]

LogConfiguration config = logConfigImpl;

// configure logging to console
config.getTargets().getConsole().setEnabled(true);
config.getTargets().getConsole().setVerbose(VerboseLevel.TRACE);

// configure logging to system events log
config.getTargets().getNetwork().setEnabled(true);
config.getTargets().getNetwork().setVerbose(VerboseLevel.STANDARD);
```

Adding files to your logger requires one extra step: creating and configuring a FileConfiguration instance that provides details about each log file to be used. For example:

```
[Java]

// add logging to Log file "Log\fulllog" - for all messages
FileConfiguration file = new FileConfigurationImpl(true, VerboseLevel.ALL, "Log/fulllogfile");
file.setMessageHeaderFormat(MessageHeaderFormat.FULL);
config.getTargets().getFiles().add(file);

// add logging to Log file "Log\infolog" - for Info (and higher) messages
file = new FileConfigurationImpl(true, VerboseLevel.TRACE, "Log/infologfile");
file.setMessageHeaderFormat(MessageHeaderFormat.SHORT);
config.getTargets().getFiles().add(file);
```

Warning

Each file added as a target must have a unique name. If two or more items are added to the file collection with the same name, only one file target will be created with the lowest specified verbose level. Other settings will be taken from one of the items using the same filename, but there is no way to predict which item will be used.

In the example above, the first line of code ensures that your logger will process messages for all verbose levels - but each target location has its own setting afterwards that specifies what level of messages can be logged by that target. You also can enable or disable individual logging targets by changing and then reapplying the settings in the `LogConfiguration` instance.

Once you have created and configured the `LogConfiguration` instance, all that remains is to apply those settings to your logger. The following code shows how you can apply these settings to either a new Logger instance, or an already existing logger.

```
[Java]

// applying new configuration to an existing logger
logger.applyConfiguration(config);
```

For more information about using `ApplyConfiguration`, see the *Lifecycle of a Logger* section above and the API Reference entry for that method.

Alternative Ways to Create a LogConfiguration

Another way to create a `LogConfiguration` instance is by parsing a `KeyValueCollection` that contains the appropriate settings. A brief code example of how to accomplish this is provided below.

```
[Java]

KeyValueCollection kvConfig = new KeyValueCollection();
// verbose level of logger will be VerboseLevel.All
kvConfig.addString("verbose","all");

// enable output of info (and higher) messages to console
kvConfig.addString("trace","stdout");

// add file target for debug debug output
kvConfig.addString("debug","Log/dbglogfile");

// Parse the created keyValueCollection. Messages generated during parsing are logged to
Console.
LogConfiguration config = LogConfigurationFactory.parse(kvConfig, (ILogger)new
Log4JLoggerFactoryImpl ());
```

Finally, you can also create a `LogConfiguration` by parsing an `org.w3c.dom.Element` that contains the appropriate settings. This Element can be created manually, or obtained from a `CfgApplication` object.

Dealing with Sensitive Log Data

There are two optional filters included as part of the common Platform SDK functionality that can be used to handle senstive log data. These are not part of the Log Library but are discussed here to help ensure sensitive data is properly considered and handled in any custom applications involving logging.

- Hiding Data in Logs - The first option to protect sensitive data is to prevent it from being printed to log files at all.
- Adding Predefined Prefix/Postfix Strings - The second option does not hide the sensitive information directly, but adds user-defined strings around values for selected key-value pairs. This makes it easy for you to locate and removed sensitive data in case log files need to be shared or distributed for any reason.

For more information about these filters, refer to the `KeyValueOutputFilter` documentation in this API Reference.

Hiding Data in Logs

In the code except below, the `KeyValuePrinter` class is used to hide any value of a key-value pair where the key is "Password":

```
[Java]

KeyValueCollection kvOptions = new KeyValueCollection();
KeyValueCollection kvData = new KeyValueCollection();
kvData.addString("Password", KeyValuePrinter.HIDE_FILTER_NAME);
```

```
KeyValuePrinter hidePrinter = new KeyValuePrinter(kvOptions, kvData);
KeyValuePrinter.setDefaultPrinter(hidePrinter);

KeyValueCollection col = new KeyValueCollection();
col.addString("Password", "secretPassword");
```

As result, the KeyValueCollection log output will have the "secretPassword" value printed as "*****". Values for other keys will display as usual.

Adding Predefined Prefix/Postfix Strings

The `PrefixPostfixFilter` class is designed to give you the ability to wrap parts of the log with predefined prefix/postfix strings. This makes it possible to easily filter out sensitive information from an already-printed log file when such a necessity arises.

In the code except below, the KeyValuePrinter is set to wrap "Password" key-value pairs in the "<###" (prefix), "###>" (postfix) strings:

```
[Java]

KeyValueCollection kvData = new KeyValueCollection();
KeyValueCollection kvPPfilter = new KeyValueCollection();
KeyValueCollection kvPPOptions = new KeyValueCollection();
kvPPfilter.addString(KeyValuePrinter.CUSTOM_FILTER_TYPE, "PrefixPostfixFilter");
kvPPOptions.addString(PrefixPostfixFilter.KEY_PREFIX_STRING, "<###");
kvPPOptions.addString(PrefixPostfixFilter.VALUE_POSTFIX_STRING, "###>");
kvPPOptions.addString(PrefixPostfixFilter.KEY_POSTFIX_STRING, ">");
kvPPOptions.addString(PrefixPostfixFilter.VALUE_PREFIX_STRING, "<");
kvPPfilter.addList(KeyValuePrinter.CUSTOM_FILTER_OPTIONS, kvPPOptions);
kvData.addList("Password", kvPPfilter);
KeyValuePrinter.setDefaultPrinter(
        new KeyValuePrinter(new KeyValueCollection(), kvData));

KeyValueCollection col = new KeyValueCollection();
col.addString("test", "secretPassword");
```

As result, the `KeyValueCollection` log output will have the "Password-secretPassword" key-value printed as "<###Password-secretPassword###>", leaving all other key-values printed as normal.

.NET

The purpose of the Platform SDK Log Library is to present an easy-to-use API for logging messages in custom-built applications. Depending on how you configure your logger, you can quickly and easily have log messages of different verbose levels written to any of the following targets:

- Genesys Message Server
- Console
- .NET Trace
- Windows System Log (Application Log only)
- Specified log files

This document provides some key considerations about how to configure and use this component, as well as code examples that will help you work with the Platform SDK Log Library for .NET in your own projects.

Introduction to Loggers

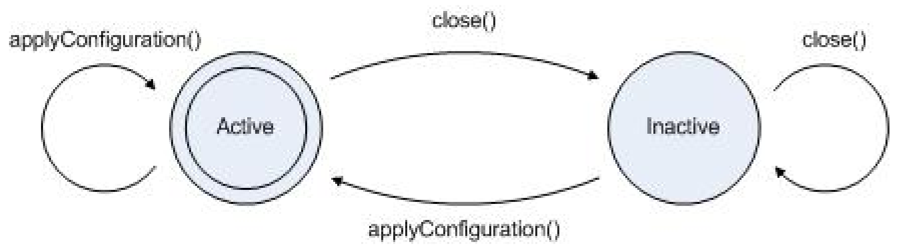
When working with custom Genesys loggers, the first step is to understand the basic process of creating and maintaining your logger. How do you create a logger instance? What configuration options should you use, and how and when can those options be changed? What is required to clean up once the logger is no longer useful?

Luckily, the main functions and lifecycle of a logger are easy to understand. The following list outlines the basic process required to create and maintain your customized logger. For more detailed information, check the *Lifecycle of a Logger section*, or the specific code examples related to *Creating a Logger*, *Customizing your Logger*, *Using your Logger*, or *Cleaning Up Your Code*.

1. Use the LoggerFactory to create an ILogger instance.
2. Reconfigure the default ILogger settings, if desired.
    - NetworkProtocol property allows you to specify a MessageSeverProtocol instance. This will let your ILogger instance send messages to Genesys Message Server.
    - LoggerPolicy property gives you control over how log messages are created and formatted, or allows you to overwrite MessageServerProtocol properties.
    - LogConfiguration class allows you to configure other aspects of your ILogger instance, which are applied with the ApplyConfiguration method.
3. Use the logger for logging messages.
4. Dispose of the ILogger instance when it is no longer needed. (You can also close the logger if it will be reused in the future.)

Lifecycle of a Logger

There are two possible states for a logger, as shown in the lifecycle diagram below.



Your logger begins in the active state once it is created. If you did not specify any configuration options during creation, then all messages with a verbose level at least equal to `VerboseLevel.Trace` are logged to the Console by default.

You can use the `ApplyConfiguration` method to change logger configuration settings from either an active or inactive state:

- If the logger is active when you call this method, then all messages being processed will be handled before the logger is stopped and reconfigured. Note that any file targets (from both the old and new configurations) are not closed automatically when the logger is reconfigured, although file targets can be closed and a new log file segment started if the new `Segmentation` settings require this.
- If the logger is inactive when you call this method, then it is activated after the new configuration settings are applied.

You can use the `Close` method to make the logger inactive without disposing of it. All messages being processed when this method is called are processed before the logger is stopped. Once the logger is inactive, no further messages are processed until after the `ApplyConfiguration` method is called.

Important
If your logger is connected to Message Server, the logger does not manage the lifecycle of the `MessageServerProtocol` instance. You must manage and close that connection manually.

Creating a Default Logger

This section provides simple code examples that show how to quickly create a logger with the default configuration and use it as part of your application.

Creating a Logger

As always, the starting point is ensuring that you have the necessary Platform SDK libraries referenced and declared in your project. For logging functionality, that includes the following namespaces:

- `Genesyslab.Platform.Commons.Logging`
- `Genesyslab.Platform.Logging`
- `Genesyslab.Platform.Logging.Configuration`
- `Genesyslab.Platform.Logging.Utilities`

Once your project is properly configured and coding about to begin, the first task is to create an instance of the `ILogger` class. This is easy to accomplish with help from the `LoggerFactory` - the only information you need to provide is a logger name that can be used later to configure targets for filtering logging output.

```
[C#]

IRootLogger logger = LoggerFactory.CreateRootLogger("myLoggerName");
```

The default behavior for this logger is to send all messages of `Trace` verbose and higher to the

Console. You can change this behavior by using the `ILogConfiguration` interface to pass configuration settings into the `LoggerFactory`, as shown below, but for this example we will accept the default values.

Using your Logger

Now that your logger is created and ready for use, the next step is to generate some custom log messages and ensure that the logger is working correctly.

One way to generate log messages is with the `Write` method. Message formatting is provided either by templates extracted from LMS files or directly from a `LogEntry` parameter, depending on what information you pass into the method. For the example below, LMS file templates provide formatting.

```
[C#]

//log the message with standard id: "9999|STANDARD|GCTI_INTERNAL|Internal error '%s' occurred"
//formatting template is extracted from LMS file
logger.Write((int)CommonMessage.GCTI_INTERNAL, "Sample Internal message.");

//log the message with standard id: "9900|DEBUG|GCTI_DEBUG|%s"
//formatting template is extracted from LMS file
logger.Write((int)CommonMessage.GCTI_DEBUG, "Sample Debug message.");
```

You can also generate log messages by using one of the methods listed in the following table.

Message Level	Available Methods
Debug	• Debug(object message) • Debug(object message, Exception exception) • DebugFormat(string format, params object [] args)
Info	• Info(object message) • Info(object message, Exception exception) • InfoFormat(string format, params object [] args)
Interaction	• Warn(object message) • Warn(object message, Exception exception) • WarnFormat(string format, params object [] args)
Error	• Error(object message) • Error(object message, Exception exception) • ErrorFormat(string format, params object [] args)

Message Level	Available Methods
Alarm	• FatalError(object message) • FatalError(object message, Exception exception) • FatalErrorFormat(string format, params object [] args)

These methods do not use any external templates or formatting, relying entirely on the information passed into them. In the examples below, the messages are logged at `Info` and `Debug` verbose levels, without any changes or formatting.

```
[C#]

logger.Info("Sample Info message.");
logger.Debug("Sample Debug message.");
```

Cleaning Up Your Code

Once you have finished logging messages with your logger, there are two options available: you can close the logger if you want it to be available for reuse later, or dispose of the logger if your application doesn't need it any longer. (Note that you do not have to close a logger before disposing of it.)

Once closed, a logger remains in an inactive state until either the `ApplyConfiguration` or `Dispose` method is called, as shown in the lifecycle diagram above.

```
[C#]

//closing the logger
logger.Close();
...
//disposing of the logger
logger.Dispose();
```

Customizing your Logger

Now that you know how to create and use a generic logger, it is time to look at some of the configuration options available to alter the behavior of your logger.

The `LogConfiguration` class allows you change application details, specify targets (including Genesys Message Server) for your log messages, and adjust the verbose level you want to report on. You can apply these changes to either a new logger that is created with the `LogFactory`, or to an existing logger by using the `ApplyConfiguration` method.

Tip

The `setMessageHeaderFormat` method has no effect on records in the Message Server

Database due to message server work specificity.

Creating a LogConfiguration to Specify Targets and Verbose Levels

The first step to configuring the settings for your logger is creating an instance of the `LogConfiguration` class and setting some basic parameters that describe your application.

```
[C#]

LogConfiguration config = new LogConfiguration
{
    ApplicationHost = "myHostname",
    ApplicationName = "myApplication",
    ApplicationId = 10,
    ApplicationType = 20
};
```

Additional `LogConfiguration` properties that can be configured to specify the name of an application-specific LMS file (`MessageFile`) and whether timestamps should use local or UTC format (`TimeUsage`). These steps aren't shown here for brevity; refer to the API Reference for details.

Tip

If logging to the network, as described in *Logging Messages to Genesys Message Server*, timestamps for log entries always use UTC format to avoid confusion. In this case the `TimeUsage` setting specified by your `LogConfiguration` is ignored.

The next step is to specify the target locations where log messages are recorded, and to configure the verbose levels for the logger and for individual targets. (Only messages with a level greater than or equal to the verbose setting will be logged.)

```
[C#]

config.Verbose = VerboseLevel.All;

//configure logging to console
config.Targets.Console.IsEnabled = true;
config.Targets.Console.Verbose = VerboseLevel.Trace;

//configure logging to system events log
config.Targets.System.IsEnabled = true;
config.Targets.System.Verbose = VerboseLevel.Standard;

//add logging to Log file "Log\fulllog" - for all messages
config.Targets.Files.Add(new FileConfiguration(true, VerboseLevel.All, "Log/fulllogfile"));
//add logging to Log file "Log\infolog" - for Info (and higher) messages
config.Targets.Files.Add(new FileConfiguration(true, VerboseLevel.Trace, "Log/infologfile"));
```

In the example above, the first line of code ensures that your logger will process messages for all verbose levels - but each target location has its own setting afterwards that specifies what level of messages can be logged by that target. You also can enable or disable individual logging targets by changing and then reapplying the settings in the LogConfiguration.

Warning

Each file added as a target must have a unique name. If two or more items are added to the file collection with the same name, only one file target will be created with the lowest specified verbose level. Other settings will be taken from one of the items using the same filename, but there is no way to predict which item will be used.

Once you have created and configured the LogConfiguration instance, all that remains is to apply those settings to your logger. The following code shows how you can apply these settings to either a new Logger instance, or an already existing logger.

```
[C#]

//applying new configuration to an existing logger
logger.ApplyConfiguration(config);
...
//apply new configuration to a new logger when it is created
IRootLogger newlogger = LoggerFactory.CreateRootLogger("NewLoggerName", config);
```

For more information about using ApplyConfiguration, see the logger lifecycle section above and the API Reference entry for that method.

Alternative Ways to Create a LogConfiguration

Another way to create a LogConfiguration is by parsing a KeyValueCollection that contains the appropriate settings. A brief code example of how to accomplish this is provided below.

```
[C#]

KeyValueCollection kvConfig = new KeyValueCollection();
//verbose level of logger will be VerboseLevel.All
kvConfig.Add("verbose","all");
//enable output of info (and higher) messages to console
kvConfig.Add("trace","stdout");
//add file target for debug debug output
kvConfig.Add("debug","Log/dbglogfile");
//Parse the created keyValueCollection. Messages generated during parsing are logged to
Console.
LogConfiguration config = LogConfigurationFactory.Parse(kvConfig, new ConsoleLogger());
```

Finally, you can also create a LogConfiguration by parsing an XElement that contains the appropriate settings, as shown below.

```
[C#]

XElement xElementConfig =
        new XElement("CfgApplication",
                new XElement("options",
                        new XElement("list_pair",
                                new XAttribute("key","log"),
                                        XElement.Parse("<str_pair key=\"verbose\" value =
\"all\"/>"),
                                        XElement.Parse("<str_pair key=\"trace\" value =
\"stdout\"/>"),
                                        XElement.Parse("<str_pair key=\"debug\" value = \"Log/
dbglogfile\"/>"),
                        )
                )
        );
LogConfiguration config = LogConfigurationFactory.Parse(xElementConfig, new ConsoleLogger());
```

Although the `XElement` can be created manually (as shown above), it is much more likely that it will be obtained from a `CfgApplication` object. The following code example illustrates how this can be done.

```
[C#]

ConfService confservice=null;
//...
//initializing the ConfService
//...
CfgApplication cfgApp = confservice.RetrieveObject<CfgApplication>(
        new CfgApplicationQuery{Name = "Sample Application"});
XElement xElementConfig = cfgApp.ToXml();
LogConfiguration config = LogConfigurationFactory.Parse(xElementConfig, new ConsoleLogger());
```

Logging Messages to Genesys Message Server

Creating a connection with Genesys Message Server is similar to setting other targets for your logger, but contains a couple of additional steps. Several new settings are required to determine how your logger handles buffering and spooling when sending log messages over the network. Once that is complete, you also have to create (and manage) a protocol object that connects to Message Server.

The following example shows how this can be accomplished. For details and additional information about the properties being configured, refer to the appropriate API Reference entries.

```
[C#]

LogConfiguration config = new LogConfiguration {Verbose = VerboseLevel.All};
config.Targets.Network.IsEnabled = true;
config.Targets.Network.Verbose = VerboseLevel.All;
config.Targets.Network.Buffering = Buffering.On|Buffering.KeepOnProtocolChange;
config.Targets.Network.SpoolFile = "temp/spool";

//create and open connection to message server
MessageServerProtocol protocol = new MessageServerProtocol(
        new Endpoint(myApplication, myHostname, myPort));
protocol.Open();

IRootLogger logger = LoggerFactory.CreateRootLogger("mySample");
logger.NetworkProtocol = protocol;
logger.ApplyConfiguration(config);
```

It is important to remember that any connection created to Message Server is not managed automatically by the Logger lifecycle. You are responsible to manage and dispose of the connection manually.

Dealing with Sensitive Log Data

There are two optional filters included as part of the common Platform SDK functionality that can be used to handle sensitive log data. These are not part of the Log Library for .NET, but are discussed here to help ensure sensitive data is properly considered and handled in any custom applications involving logging.

- Hiding Data in Logs - The first option to protect sensitive data is to prevent it from being printed to log files at all.
- Adding Predefined Prefix/Postfix Strings - The second option does not hide the sensitive information directly, but adds user-defined strings around values for selected key-value pairs. This makes it easy for you to locate and removed sensitive data in case log files need to be shared or distributed for any reason.

Hiding Data in Logs

In the code except below, the `KeyValuePrinter` class is used to hide any value of a key-value pair where the key is "Password":

```
[C#]

KeyValueCollection kvOptions = new KeyValueCollection();
KeyValueCollection kvData = new KeyValueCollection();
kvData["Password"] = KeyValuePrinter.HideFilterName;
KeyValuePrinter hidePrinter = new KeyValuePrinter(kvOptions, kvData);
KeyValuePrinter.DefaultPrinter = hidePrinter;

KeyValueCollection col = new KeyValueCollection();
col["Password"] = "secretPassword";
```

As result, the `KeyValueCollection` log output will have the "secretPassword" value printed as "*****". Values for other keys will display as usual.

Adding Predefined Prefix/Postfix Strings

The `PrefixPostfixFilter` class is designed to give you the ability to wrap parts of the log with predefined prefix/postfix strings. This makes it possible to easily filter out sensitive information from an already-printed log file when such a necessity arises.

In the code except below, the `KeyValuePrinter` is set to wrap "Password" key-value pairs in the "<###" (prefix), "###>" (postfix) strings:

```
[C#]

KeyValueCollection kvData = new KeyValueCollection();
KeyValueCollection kvPPfilter = new KeyValueCollection();
KeyValueCollection kvPPOptions = new KeyValueCollection();
kvPPfilter[KeyValuePrinter.CustomFilterType] = typeof(PrefixPostfixFilter).FullName;
kvPPOptions[PrefixPostfixFilter.KeyPrefixString] = "<###";
kvPPOptions[PrefixPostfixFilter.ValuePrefixString] = "<";
kvPPOptions[PrefixPostfixFilter.ValuePostfixString] = "###>";
kvPPOptions[PrefixPostfixFilter.KeyPostfixString] = ">";
kvPPfilter[KeyValuePrinter.CustomFilterOptions] = kvPPOptions;
kvData["Password "] = kvPPfilter;
```

```
KeyValuePrinter.DefaultPrinter = new KeyValuePrinter(new KeyValueCollection(), kvData);
KeyValueCollection col = new KeyValueCollection();
col["Password"] = "myPassword";
```

As result, the `KeyValueCollection` log output will have the "Password-secretPassword" key-value printed as "<###Password-secretPassword###>", leaving all other key-values printed as normal.

Migration Overview

This article provides an overview of any migration information you might require if coming to the Platform SDK 8.5.0 release after starting development with an earlier version.

Deprecated Application Blocks

Starting with release 8.5.0, the deprecated Message Broker Application Block and Protocol Manager Application Block should not be used in any current development. Removal of these Application Blocks simplifies the API, making code clearer and more reliable and smoothing the Platform SDK learning curve.

Both application blocks should be removed from any existing applications where possible. The following articles provide migration guidelines for this change:

- Migration from Message Broker Application Block Usage
- Migration from Protocol Manager Application Block Usage

If not possible to remove the Application Blocks from your code, then it is possible to adopt the full source code (which is available) for the application block into your project as a last resort.

Migration from Message Broker Application Block Usage

Introduction

Starting with release 8.5.0, use of the Message Broker Application Block is no longer recommended. This application block is now considered as legacy component and has been deprecated.

This article provides an overview of how to migrate existing applications, and outlines how behavior that was previously handled by the application block should now be implemented.

> **Tip**
> If you choose to continue using the Message Broker Application Block, please note that the common interfaces for COM Application Block and Message Broker have been moved to an individual file (`commonsappblock.jar` for Java, `Genesyslab.Platform.ApplicationBlocks.Commons.dll` for .NET) starting with release 8.5.0.

Functional Aspects of the Message Broker Application Block

SubscriptionService + Subscriber Pattern

The broker service is the main component of the application block. It contains several facade service implementations including: `BrokerService`, `EventBrokerService`, `EventReceivingBrokerService`, `RequestBrokerService` and `RequestReceivingBrokerService`.

Subscription Filters

Each `Subscriber` has its own events filter (using the `Predicate<T>` interface). The event broker service applies incoming events to all of a registered Subscriber's filters.

The application block contains several predefined `Message` filters for use with protocol message brokers. For instance, there are:

- `MessageFilter`: it may filter protocol messages by `ProtocolDescription`, `ProtocolId`, or `EndpointName`. It also has ability to be negated.
- `MessageIdFilter`: an extension of `MessageFilter` with the ability to filter specific `MessageId`.
- `MessageNameFilter`: an extension of `MessageFilter` with the ability to filter specific `MessageName`.

- `MessageRangeFilter`: an extension of `MessageFilter` with the ability to filter specific set of `MessageIds`.

The application block also contains helping classes to make composite filters: `AndPredicate` and `OrPredicate`.

Synchronous and Asynchronous Execution

- The application block contains a general purpose synchronous `BrokerService`<T> implementation. It does not use additional threads, does not have any queues, and executes generic events (of type <T>) publishing immediately.
- Asynchronous generic broker service `AsyncBrokerService`<T>. The difference between this broker and the synchronous broker is that events are published with the specified invoker (one or more other threads).
- Old type asynchronous broker services: `EventBrokerService` and `RequestBrokerService`. These services use dedicated internal thread to get events from intermediate queue and pass to invoker.
- `EventReceivingBrokerService` and `RequestReceivingBrokerService`. Compared to the old style brokers, these do not use the additional threads and replace the intermediate queues (implementing receiving interfaces).

Functional Replacements

COM Application Block ConfService Initialization

The first place where you may need to update application code to not use the Message Broker Application Block is with old-style initialization of `ConfService`.

Initialization of *ConfService* with Message Broker usage may look like following:

[+] Java Code Sample

```
Endpoint csEndpoint = new Endpoint(csEPName, csHost, csPort);
ConfServerProtocol csProtocol = new ConfServerProtocol(csEndpoint);
csProtocol.setClientName(clientName);
csProtocol.setClientApplicationType(clientType);
csProtocol.setUserName(username);
csProtocol.setUserPassword(password);
csProtocol.open();

EventBrokerService msgBroker = BrokerServiceFactory.CreateEventBroker(csProtocol);

IConfService confService = ConfServiceFactory.createConfService(csProtocol, msgBroker);
```

[+] .NET Code Sample

```
Endpoint csEndpoint = new Endpoint(csEPName, csHost, csPort);
ConfServerProtocol csProtocol = new ConfServerProtocol(csEndpoint);
csProtocol.ClientName = clientName;
csProtocol.ClientApplicationType = clientType;
csProtocol.UserName =userName;
csProtocol.UserPassword=password;
csProtocol.Open();

EventBrokerService msgBroker = BrokerServiceFactory.CreateEventBroker(csProtocol);
IConfService confService = ConfServiceFactory.CreateConfService(csProtocol, msgBroker);
```

The new initialization approach would be following:

[+] Java Code Sample

```
Endpoint csEndpoint = new Endpoint(csEPName, csHost, csPort);
ConfServerProtocol csProtocol = new ConfServerProtocol(csEndpoint);
csProtocol.setClientName(clientName);
csProtocol.setClientApplicationType(clientType);
csProtocol.setUserName(username);
csProtocol.setUserPassword(password);

IConfService confService = ConfServiceFactory.createConfService(csProtocol);

csProtocol.open();
```

[+] .NET Code Sample

```
Endpoint csEndpoint = new Endpoint(csEPName, csHost, csPort);
ConfServerProtocol csProtocol = new ConfServerProtocol(csEndpoint);
csProtocol.ClientName = clientName;
csProtocol.ClientApplicationType = clientType;
csProtocol.UserName =userName;
csProtocol.UserPassword=password;

IConfService confService = ConfServiceFactory.CreateConfService(csProtocol);
csProtocol.Open();
```

This change eliminates the redundant internal messages queue and the redundant thread.

Tip

Protocol `open()` has to be done after `ConfService` creation. By this way `ConfService` initializes its own internal instance of `MessageHandler`, so, the protocol has to be closed to allow it. And if the protocol instance has initialized custom `MessageHandler`, it will be overridden with the `ConfServices`' internal one.

If there is a need to receive asynchronous protocol messages from `ConfServerProtocol` and `ConfService` pair, your application may use `ConfService.setUserMessageHandler(MessageHandler)` instead of a subscription on the message broker:

[+] Java Code Sample

```
MessageHandler msgHandler = new MessageHandler() {
    public void onMessage(Message message) {
        // do something with incoming async protocol message
    }
};
confService.setUserMessageHandler(msgHandler);
```

[+] .NET Code Sample

```
confService.Protocol.Received += (sender, e) =>
{
   var args = e as MessageEventArgs;
   if ((args != null) && (args.Message!=null))
   {
     // do something with incoming async protocol message
   }
};
```

The message handling method `MessageHandler.onMessage(message)` will be executed using the protocol invoker thread.

Message Broker Component

Broker (Subscribers-Side Replacement)

The most common part of different types of broker services is functionality for message/event passing to service `Subscriber`'s.

So, when we have some broker service instance with initialization of several subscribers like this:

```
broker.register(subscriber1);
broker.register(subscriber2);
broker.register(subscriber3);
```

it would be replaced with function like:

[+] Java Code Sample

```
void doNotifySubscribers(final Message message) {
    if (<subscriber1 filter on 'message'>) {
        // do subscriber1 handling of 'message'
    }
    if (<subscriber2 filter on 'message'>) {
        // do subscriber2 handling of 'message'
    }
    if (<subscriber3 filter on 'message'>) {
        // do subscriber3 handling of 'message'
    }
}
```

[+] .NET Code Sample

```
void doNotifySubscribers(IMessage message) {
    if (<subscriber1 filter on 'message'>) {
        // do subscriber1 handling of 'message'
    }
    if (<subscriber2 filter on 'message'>) {
        // do subscriber2 handling of 'message'
    }
    if (<subscriber3 filter on 'message'>) {
        // do subscriber3 handling of 'message'
    }
}
protocol.Received += (sender, e) =>
{
   var args = e as MessageEventArgs;
   if ((args != null) && (args.Message!=null))
   {
     doNotifySubscribers(args.Message)
   }
};
```

In most cases it is possible to optimize such a function to do not execute all the filters for all incoming messages, but use "if {} else if {} ...", "switch(<>) {}", or, even do not use explicit filtering taking into account specifics of the broker instance like expected set of incoming messages, their types, etc.

Subscribers Filters Replacement

Message Broker Application Block contains several predefined filters for protocol messages filtering.

The messages filters provide several properties for filtering. Each of the properties corresponds to specific attribute of protocol message. So, if some property is initialized and is not null, then it is to be applied for incoming messages filtering.

For example, here is a sample filter logic:

```
Action<Message> action = new Action<Message>() {
    public void handle(final Message message) {
        // do something with 'message'
    }
};
brokerService.register(action, new MessageIdFilter(
        ConfServerProtocol.PROTOCOL_DESCRIPTION, EventObjectUpdated.ID));
```

Such broker would be changed to something like:

[+] Java Code Sample

```
void doNotifySubscribers(final Message message) {
    if (ConfServerProtocol.PROTOCOL_DESCRIPTION.equals(message.getProtocolDescription())
            && (message.messageId() == EventObjectUpdated.ID)) {
        // do something with 'message'
    }
}
```

[+] .NET Code Sample

```
void doNotifySubscribers(IMessage message)
{
  if (ConfServerProtocol.Description.Equals(message.ProtocolDescription) && (message.Id ==
EventObjectUpdated.MessageId))
  {
    // do something with 'message'
  }
}
```

Broker (Service-Side Replacement)

The other side of a broker component is an entrance of messages/events for notification.

On this side we may have several different cases of broker service usage:

It may be a general broker for general event type like `BrokerService<T>`, `AsyncBrokerService<T>`, or some other broker type with explicit events publishing with `broker.publish(event);`.

In this case `broker.publish(event);` may be simply replaced with direct call to the newly created `doNotifySubscribers(event);`.

Usage of `EventReceivingBrokerService` as `MessageReceiver` or `MessageHandler` would be replaced with direct `MessageHandler`:

```
EventReceivingBrokerService evBroker = new EventReceivingBrokerService();
evBroker.register(subscriber1);
evBroker.register(subscriber2);
protocol.setMessageHandler(evBroker);
protocol.open();
```

would be changed to:

[+] Java Code Sample

```
protocol.setMessageHandler(new MessageHandler() {
    public void onMessage(final Message message) {
        if (<subscriber1 filter on 'message'>) {
            // do subscriber1 handling of 'message'
        }
        if (<subscriber2 filter on 'message'>) {
            // do subscriber2 handling of 'message'
        }
    }
});
protocol.open();
```

[+] .NET Code Sample

```
private void OnReceivedHandler(object sender, EventArgs e)
{
  var args = e as MessageEventArgs;
  if ((args != null) && (args.Message != null))
  {
        if (<subscriber1 filter on 'message'>) {
            // do subscriber1 handling of 'message'
        }
        if (<subscriber2 filter on 'message'>) {
            // do subscriber2 handling of 'message'
        }
  }
}


protocol.Received += OnReceivedHandler;
protocol.Open();
```

Custom `MessageHandler` may be shared between several protocols connections just like

`EventReceivingBrokerService`.

Usage of old style "EventBrokerService"

It's a special case of broker service which is based on intermediate messages queue, and it uses extra thread to synchronously read messages from the queue and pass them for handling.

For example:

```
EventBrokerService broker = BrokerServieFactory.CreateEventBroker(protocol);

// or:

EventBrokerService broker = new EventBrokerService(protocol);
broker.activate();
```

Replacing of such kind of broker with `MessageHandler` (see above) eliminates redundant messages queue and the redundant thread.

Request Broker Component

Request broker service is a kind of message broker for handling of clients requests on `ServerChannel` side.

There are two types of request broker services: `RequestBrokerService` and `RequestReceivingBrokerService`.

Actual recommendation for requests handling logic on `ServerChannel` is to do not use any broker service, but explicitly handle incoming requests in accordance to application specific architecture (without additional shared queue).

It may be done with custom implementation of request receiver:

[+] Java Code Sample

```
RequestReceiverSupport rqReceiver = new RequestReceiverSupport() {
    public void processRequest(final RequestContext incomingRequest) {
        // ! pass some task to do something with 'incomingRequest' in separated thread or
thread pool !
        //   like "executor.execute(new RequestHandlerTask(incomingRequest));"
    }
    public RequestContext receiveRequest() throws InterruptedException, IllegalStateException
{
        throw new <Exception>("requests are handled asynchronously");
    }
    // ... implementation for other RequestReceiverSupport methods goes here ...
};
serverChannel.setReceiver(rqReceiver);
serverChannel.open();
```

[+] .NET Code Sample

```
private class CustomRequestReceiver : IRequestReceiverSupport
{
  public IRequestContext ReceiveRequest(TimeSpan timeout)
  {
    throw new InvalidOperationException("requests are handled asynchronously");
  }
  public void ProcessRequest(IRequestContext request)
  { // ! pass some task to do something with 'incomingRequest' in separated thread or thread
pool !
    // for example:
    ThreadPool.QueueUserWorkItem(delegate(object state){
      var context = state as IRequestContext;
      // process request context
    }, request);
  }
  // ... implementation for other RequestReceiverSupport methods goes here ...
}

serverChannel.SetReceiver(new CustomRequestReceiver());
serverChannel.Open();
```

Migration from Protocol Manager Application Block Usage

Introduction

Starting with release 8.5.0, use of the Protocol Manager Application Block is no longer recommended. This application block is now considered a legacy component and has been deprecated.

This article provides an overview of how to migrate existing applications, and outlines how behavior that was previously handled by the application block should now be implemented.

Functional Aspects of the Protocol Manager Application Block

Protocols configurations helper classes:

- Protocols handshake options
- Connection configuration related options
- `WarmStandby` related options

Protocol Management Service functionality:

- `MessageReceiver` sharing feature
- `ChannelListener`'s sharing feature
- Protocols `WarmStandby` initialization feature
- Bulk `BeginOpen`/`BeginClose` functions

Functional Replacements

The usual Protocol Manager Application Block usage scenario is to help with the initialization of multiple protocol connections. Protocol Management configuration classes contain following parts:

- protocol handshake options,
- typified connection configuration options,
- `WarmStandby` options.

Connection Configuration Feature

[+] Java Code Sample

```
ProtocolManagementServiceImpl pmService = new ProtocolManagementServiceImpl();

TServerConfiguration config = new TServerConfiguration("t-server");

config.setUri(host, port); // - Target server host/port

config.setUseAddp(true); // - ConnectionConfiguration typified options like "UseAddp",
"AddpClientTimeout", etc
config.setAddpServerTimeout(addpServerTimeout);
config.setAddpClientTimeout(addpClientTimeout);

config.setFaultTolerance(FaultToleranceMode.WarmStandby);
config.setWarmStandbyUri(hostBackup, portBackup); // - Backup server host/port
config.setWarmStandbyAttempts((short) 3); // - WarmStandby typified options like
"WarmStandbyAttempts", etc
config.setWarmStandbyTimeout(2000);

config.setClientName(clientName); // - Protocol handshake typified options like "ClientName",
etc

Protocol protocol = pmService.register(config);
pmService.beginOpen();
```

[+] .NET Code Sample

```
var pmService = new ProtocolManagementService();
var config = new TServerConfiguration("t-server")
{
   Uri = new Uri("tcp://host:port/"), // - Target server host/port
   UseAddp = true,                    // - ConnectionConfiguration typified options like
"UseAddp", "AddpClientTimeout", etc
   AddpServerTimeout = addpServerTimeout,
   AddpClientTimeout = addpClientTimeout,
   FaultTolerance = FaultToleranceMode.WarmStandby,
   WarmStandbyUri = new Uri("tcp://backupHost:backupPort/"), // - Backup server host/port
   WarmStandbyAttempts = 3,                                  // - WarmStandby typified
options like "WarmStandbyAttempts", etc
   WarmStandbyTimeout = 2000,
   ClientName = clientName        // - Protocol handshake typified options like "ClientName",
etc
 };

IProtocol protocol = pmService.Register(config);
pmService.BeginOpen();
```

Last PSDK versions contain extended ConnectionConfiguration interfaces with set of typified properties for connections configuration. So, generally speaking, now we can initialize all of these options right with protocol connection initialization and it is not needed to have additional intermediate (duplicating) configuration structures.

Elimination of Protocol Management Service may look like:

[+] Java Code Sample

```
PropertyConfiguration connConf = new PropertyConfiguration();
```

```
connConf.setUseAddp(true); // - ConnectionConfiguration typified options like "UseAddp",
"AddpClientTimeout", etc
connConf.setAddpServerTimeout(addpServerTimeout);
connConf.setAddpClientTimeout(addpClientTimeout);


Endpoint endpoint = new Endpoint(epName1, host, port, connConf); // - Target server host/port
are here
Endpoint endpointBackup = new Endpoint(epName2, hostBackup, portBackup, connConf); // -
Backup server host/port are here

TServerProtocol protocol = new TServerProtocol(endpoint);
protocol.setClientName(clientName); // - Protocol handshake typified options like
"ClientName", etc

WarmStandbyConfiguration wsConf = new WarmStandbyConfiguration(endpoint, endpointBackup);
wsConf.setAttempts((short) 3); // - WarmStandby typified options like "WarmStandbyAttempts",
etc
wsConf.setTimeout(2000);
WarmStandbyService wsService = new WarmStandbyService(protocol);
wsService.applyConfiguration(wsConfig);
wsService.start();
protocol.beginOpen();
```

[+] .NET Code Sample

```
// - ConnectionConfiguration typified options like "UseAddp", "AddpClientTimeout", etc
PropertyConfiguration connConf = new PropertyConfiguration()
{
   UseAddp = true,
   AddpServerTimeout = addpServerTimeout,
   AddpClientTimeout = addpClientTimeout
};

var endpoint = new Endpoint(epName1, host, port, connConf); // - Target server host/port are
here
var endpointBackup = new Endpoint(epName2, hostBackup, portBackup, connConf); // - Backup
server host/port are here

var protocol = new TServerProtocol(endpoint);
protocol.ClientName = clientName; // - Protocol handshake typified options like "ClientName",
etc
var wsConf = new WarmStandbyConfiguration(endpoint, endpointBackup);
wsConf.Attempts = 3; // - WarmStandby typified options like "WarmStandbyAttempts", etc
wsConf.Timeout = 2000;

WarmStandbyService wsService = new WarmStandbyService(protocol);
wsService.ApplyConfiguration(wsConf);
wsService.Start();
protocol.BeginOpen();
```

MessageReceiver Sharing Feature

The Protocol Manager Service instance unconditionally uses its own instance of `MessageReceiver` for all of its registered protocols.

So, it is not possible to use `protocol.receive()` on protocols instances created with `ProtocolManagementService`. It may be done with `pmServiceImpl.getReceiver().receive()`. This method returns asynchronous incoming message from shared queue of all protocols registered with this `pmServiceImpl`.

Actually, `MessageReceiver` mechanism is deprecated and, usually, it is not effective to use single handler/queue to process event messages from different protocols connections.

Application logic will be more clear and supportable when each protocol has own specific event handling logic.

So, the recommendation is to use specific `MessageHandler` for protocol instance where it is required. For example:

[+] Java Code Sample

```
protocol.setMessageHandler(new MessageHandler() {
        public void onMessage(final Message message) {
            // do fast event message procession or pass it to some other executor for handling
        }
});
```

[+] .NET Code Sample

```
protocol.Received += (sender, args) =>
{
   IMessage message = ((MessageEventArgs) args).Message;
   // do fast event message procession or pass it to some other executor for handling
};
```

ChannelListener Events Concentration

Protocol Management service supports notifications to set of clients `ChannelListeners`.

To migrate out of Protocol Manager Application Block usage, `ChannelListeners` may be added directly to protocol connections.

By the way, in most cases situation like "publish channel events from all of the N connections to all of the M listeners" is a kind of an application design issue, though, it is possible to be realized.

WarmStandby Service Initialization

Protocol Manager service instance does initialize `WarmStandby` service internally if given protocol configuration contains `WarmStandby` related options/values.

[+] Java Code Sample

```
...
config.setFaultTolerance(FaultToleranceMode.WarmStandby);

config.setWarmStandbyUri(hostBackup, portBackup);
config.setWarmStandbyAttempts((short) 3);
config.setWarmStandbyTimeout(2000);
...
```

[+] .NET Code Sample

```
var config = new ConfServerConfiguration("confserver")
{
   ...
   FaultTolerance = FaultToleranceMode.WarmStandby,
   WarmStandbyUri = new Uri("tcp://backupHost:backupPort/"),
   WarmStandbyAttempts = 3,
   WarmStandbyTimeout = 2000,
   ...
 };
```

The recommendation is to create and initialize it explicitly. Initialization schema was mentioned above. Initialization may look like:

[+] Java Code Sample

```
Endpoint confEPprimary = ...;
Endpoint confEPbackup = ...;

ConfServerProtocol cfgProtocol = new ConfServerProtocol(confEPprimary);
cfgProtocol.setClientName(appName);
cfgProtocol.setClientApplicationType(appType);
cfgProtocol.setUserName(username);
cfgProtocol.setUserPassword(password);

WarmStandbyConfiguration wsConfig = new WarmStandbyConfiguration(confEPprimary, confEPbackup);
wsConfig.setTimeout(2000);
wsConfig.setAttempts((short) 3);
WarmStandbyService wsService = new WarmStandbyService(cfgProtocol);
wsService.applyConfiguration(wsConfig);
wsService.start();

cfgProtocol.beginOpen();
```

[+] .NET Code Sample

```
Endpoint confEPprimary = ...;
Endpoint confEPbackup = ...;

ConfServerProtocol cfgProtocol = new ConfServerProtocol(confEPprimary);
cfgProtocol.ClientName = appName;
cfgProtocol.ClientApplicationType = appType;
cfgProtocol.UserName = username;
cfgProtocol.UserPassword = password;

WarmStandbyConfiguration wsConfig = new WarmStandbyConfiguration(confEPprimary, confEPbackup);
wsConfig.Timeout = 2000;
wsConfig.Attempts = 3;
WarmStandbyService wsService = new WarmStandbyService(cfgProtocol);
wsService.ApplyConfiguration(wsConfig);
wsService.Start();

cfgProtocol.BeginOpen();
```

For more details, see WarmStandby Application Block documentation.

Bulk Open/Close Functions

Actually, `pmService.beginOpen()` means `protocol.beginOpen()` for all protocols registered in given

PM Service.

Legacy Topics

Topics in this section are no longer applicable for new development, but are maintained here for backwards compatibility.

- Using the Message Broker Application Block
- Event Handling Using the Message Broker Application Block
- Using the Protocol Manager Application Block
- Connecting to a Server Using the Protocol Manager Application Block
- Legacy Warm Standby Application Block Description

Using the Message Broker Application Block

Deprecation Notice: This application block is considered a legacy product starting with release 8.1.1. Documentation is provided for backwards compatibility, but new development should consider using the improved method of message handling.

Important

This application block is a reusable production-quality component. It has been designed using industry best practices and provided with source code so it can be used "as is," extended, or tailored if you need to.

Please see the License Agreement for details.

The Message Broker Application Block makes it easy for your applications to handle events in an efficient way.

Java

Installing the Message Broker Application Block

To work with the Message Broker Application Block, you must ensure that your system meets the software requirements established in the Genesys Supported Operating Environment Reference Guide.

Building the Message Broker Application Block

Tip

Starting with release 8.5.0, the common interfaces for COM Application Block and Message Broker have been moved to an individual `commonsappblock.jar` file.

To build the Message Broker Application Block:

1. Open the `<Platform SDK Folder>\applicationblocks\messagebroker` folder.
2. Run either `build.bat` or `build.sh`, depending on your platform.

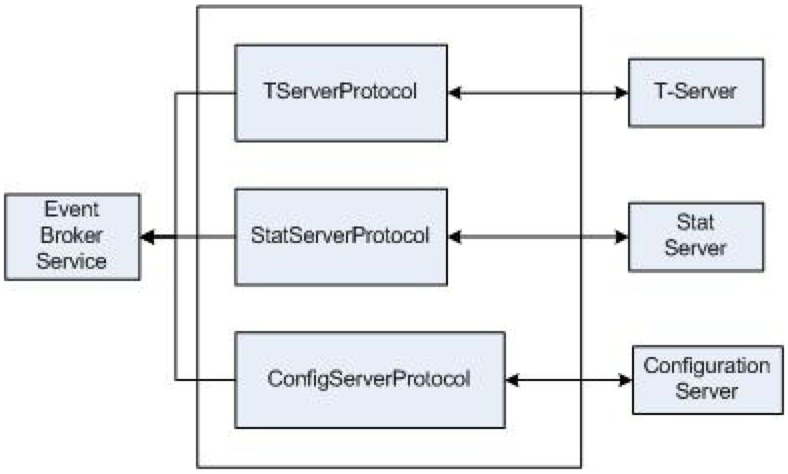
This will create the `commonsappblock.jar` file, located within the `<Platform SDK Folder>\applicationblocks\messagebroker\dist\lib` directory.

Working with the Message Broker Application Block

You can find basic information on how to use the Message Broker Application Block in the article on Event Handling Using the Message Broker Application Block.

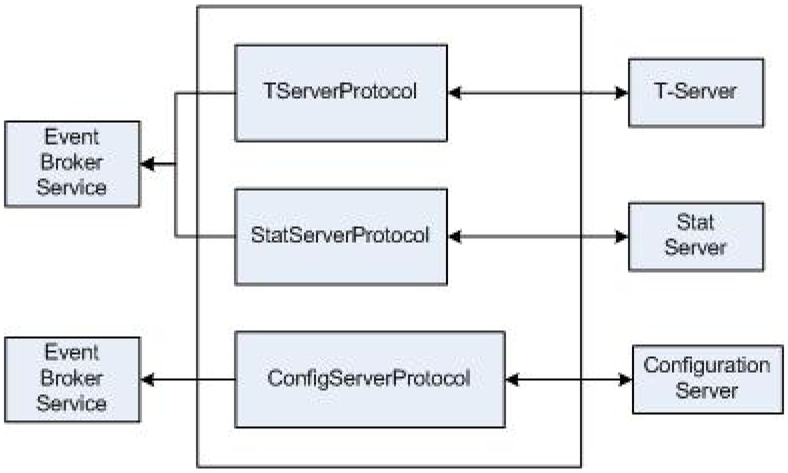
Configuring Message Broker

When you first work with Message Broker, you will probably use a single instance of `EventBrokerService`. This means that all messages coming into your application will first pass through this single instance, as shown in below. Note that configuration diagrams used here do not show the Protocol Manager Application Block, in order to focus on the architecture of Message Broker.

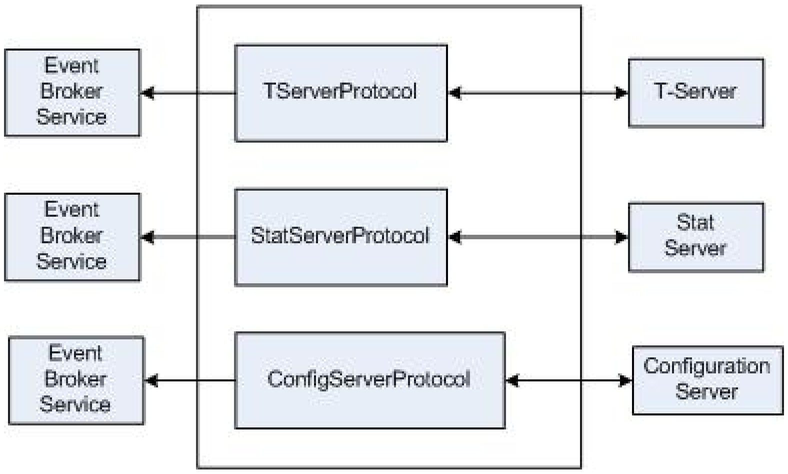


However, there may be high-traffic scenarios that require multiple instances of Message Broker. This might happen if you have one or more servers whose events use so much of Message Broker’s processing time that events from other servers must wait for an unacceptable amount of time. In that case, you could dedicate an instance of `EventBrokerService` to the appropriate server.

For example, you may have a scenario in which you frequently receive large volumes of statistics. To handle that situation, you could dedicate an `EventBrokerService` instance to Stat Server. In other situations, you might regularly receive large amounts of Configuration Layer data from Configuration Server. You could handle this in a similar way by giving Configuration Server its own instance of `EventBrokerService`, as shown here:



Sometimes you may have large message volumes for each server, in which case you could use a separate instance of `EventBrokerService` for each server, as shown here.



Using Message Filters

Message Broker comes with several types of message filters. You can filter on individual messages using `MessageIdFilter` or `MessageNameFilter`. In most cases you will want to use `MessageIdFilter`, as it is more efficient than `MessageNameFilter`. You can also use a `MessageRangeFilter` to filter on several messages at a time.

As shown in the article on Event Handling Using the Message Broker Application Block, you can specify these filters when you register an event handler with the Event Broker Service. Here is a sample of how to set up a `MessageIdFilter`:

```
[Java]

eventBrokerService.register(new StatPackageOpenedHandler(),
        packageEvents);
```

There may be times when you want to process several events in the same event handler. In such cases, you can use a `MessageRangeFilter`, which will direct all of these events to that handler. Here is a sample of how to set up the filter:

```
[Java]

int[] messageRange = new int[] {EventPackageOpened.ID, EventPackageClosed.ID};
MessageRangeFilter packageStatusEvents = new MessageRangeFilter
        (messageRange);
eventBrokerService.register(new StatPackageStatusChangedHandler(),
        packageStatusEvents);
```

Your event handler might look something like this:

```
[Java]

class StatPackageStatusChangedHandler implements Action {

        public void handle(Message obj) {
                // Common processing goes here...
                if (obj.messageId() == EventPackageOpened.ID) {
                        // EventPackageOpened processing goes here...
                } else {
                        // EventPackageClosed processing goes here...
                }
        }
}
```

Some servers use events that have the same name as events used by another server. One example is `EventError`, which is used by just about every server except Stat Server. The Event Handling Using the Message Broker Application Block article shows how to use a Protocol Description object to filter events by server type in order to avoid confusion when handling these events.

There also may be times when you have several instances of a given server in your environment and you want to filter by a specific one. To do this, first specify an `Endpoint` for that server, using a name for the server in the `Endpoint` constructor:

```
[Java]

String statServer1EndpointName = "StatServer1";
Endpoint statServer1Endpoint =
        new Endpoint(statServer1EndpointName, statServer1Uri);
```

Now create the filter:

```
[Java]

MessageIdFilter statServer1EndpointFilter =
        new MessageIdFilter(EventPackageOpened.ID);
```

And set the EndpointName in the filter:

```
[Java]

statServer1EndpointFilter.setEndpointName(statServer1EndpointName);
```

When you register this filter, the handler you specify will only receive messages that were sent from the instance you mentioned above:

```
[Java]

eventBrokerService.register(new StatPackageOpenedHandler_StatServer1(),
        statServer1EndpointFilter);
```

Architecture and Design

The Message Broker Application Block is designed to make it easy for your applications to handle events in an efficient way.

Message Broker allows you to set up individual classes to handle specific events coming from Genesys servers. It receives all of the events from the servers you specify, and sends each one to the appropriate handler class. Message Broker is a high-performance way to hide the complexity of event-driven programming — so you can focus on other areas of your application.

> **Tip**
>
> Message Broker has been designed for use with the Protocol Manager Application Block. Protocol Manager is another high-performance component that makes it easy for your applications to connect to Genesys servers. You can find basic information on how to use the Protocol Manager Application Block in the article on Connecting to a Server.

The Message Broker Application Block Architecture

The Message Broker Application Block uses a service-based API that enables you to write individual methods that handle one or more events.

For example, you might want to handle every occurrence of `EventAgentLogin` with a specific dedicated method, while there might be other events that you wish to send to a common event-handling method. Message Broker allows you write these methods and register them with an event broker that manages them for you.

Message Filters

Message Broker uses *message filters* to identify specific messages, assign them to specified methods, and route them accordingly.

Design Patterns

This section gives an overview of the design patterns used in the Message Broker Application Block.

Publish/Subscribe Pattern

There are many occasions when one class (the subscriber) needs to be notified when something changes in another class (the publisher). The Message Broker Application Block use the Publish/ Subscribe pattern to inform the client application when events arrive from the server.

Factory Method Pattern

It is common practice for a class to include constructors that enable clients of the class instantiate it. There are times, however, when a client may need to instantiate one of several different classes. In some of these situations, the client should not need to decide which class is being created. In this case, a Factory Method pattern is used. The Factory Method pattern lets a class developer define the interface for creating an object, while retaining control of which class to instantiate.

How To Properly Manage the EventBrokerService Lifecycle

Unfortunately, a commonly encountered problem is that users create `EventBrokerService` but do not dispose of it properly. `EventBrokerService` exclusively uses an invoker thread to run an infinite cycle with `MessageReceiver.receive()` and incoming messages handling logic. `EventBroker` is created by user code, so it should be disposed by user code as well. Useful methods are `MessageBrokerService.deactivate()` and `MessageBrokerService.dispose()`.

In PSDK 8.1 this class is deprecated and a new one is added to resolve the problem with thread waiting: `EventReceivingBrokerService`. This new class implements the `MessageReceiver` interface and may be used as external receiver for Platform SDK protocols. In this case, we have no intermediate redundant queue and incoming messages are delivered from protocol(s) to handler(s) directly. This class still requires async invoker to execute messages handling, but in this case the invoker is called once per incoming message, so it's thread is not blocked during the `.receive()` operation.

So, `EventReceivingBrokerService` does not need `.dispose()` and is GC friendly.

Tip
A similar change has been made to `RequestBrokerService`.

Also note that the `Invoker` instance still represents a "costly" resource (thread) and is managed by user code, so proper attention (allocation/deallocation) is required.

Q: Does it matter if the event broker service is created by the `BrokerServiceFactory` or not?

A: Actually, `BrokerServiceFactory` just creates and activates the corresponding broker instance. So if a broker is created by a call to the factory, it must be disposed of by user code in accordance to its usage there.

.NET

Installing the Message Broker Application Block

To work with the Message Broker Application Block, you must ensure that your system meets the software requirements established in the Genesys Supported Operating Environment Reference Guide.

Building the Message Broker Application Block

Tip

Starting with release 8.5.0, the common interfaces for COM Application Block and Message Broker have been moved to an individual `Genesyslab.Platform.ApplicationBlocks.Commons.dll` file.

The Platform SDK distribution includes a `Genesyslab.Platform.ApplicationBlocks.Commons.dll` file that you can use as is. This file is located in the `bin` directory at the root level of the Platform SDK directory. To build your own copy of this application block, follow the instructions below:

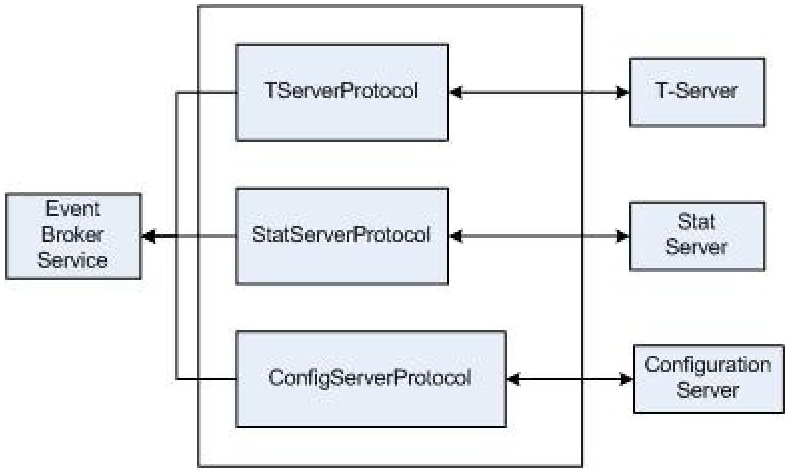
1. Open the `<Platform SDK Folder>\ApplicationBlocks\MessageBroker` folder.
2. Double-click `MessageBroker.sln`.
3. Build the solution.

Working with the Message Broker Application Block

You can find basic information on how to use the Message Broker Application Block in the article on Event Handling Using the Message Broker Application Block.

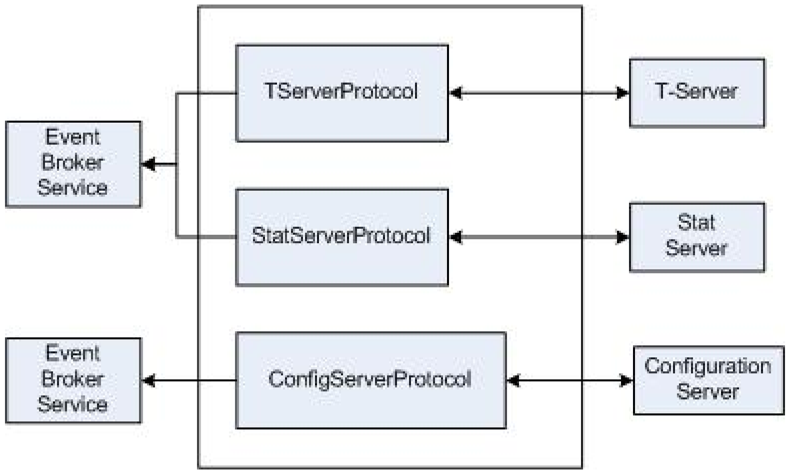
Configuring Message Broker

When you first work with Message Broker, you will probably use a single instance of `EventBrokerService`. This means that all messages coming into your application will first pass through this single instance, as shown in the figure below. Note that the following configuration diagrams do not show the Protocol Manager Application Block, in order to focus on the architecture of Message Broker.

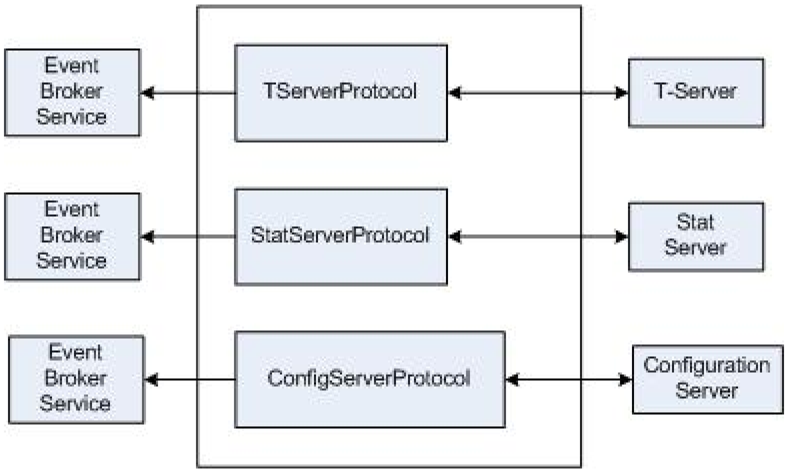


However, there may be high-traffic scenarios that require multiple instances of Message Broker. This might happen if you have one or more servers whose events use so much of Message Broker's processing time that events from other servers must wait for an unacceptable amount of time. In that case, you could dedicate an instance of `EventBrokerService` to the appropriate server.

For example, you may have a scenario in which you frequently receive large volumes of statistics. To handle that situation, you could dedicate an `EventBrokerService` instance to Stat Server. In other situations, you might regularly receive large amounts of Configuration Layer data from Configuration Server. You could handle this in a similar way by giving Configuration Server its own instance of `EventBrokerService`, as shown in the following figure:



Sometimes you may have large message volumes for each server, in which case you could use a separate instance of `EventBrokerService` for each server, as shown here.



Using Message Filters

Message Broker comes with several types of message filters. You can filter on individual messages using `MessageIdFilter` or `MessageNameFilter`. In most cases you will want to use `MessageIdFilter`, as it is more efficient than `MessageNameFilter`. You can also use a `MessageRangeFilter` to filter on several messages at a time.

As shown in the article on Event Handling Using the Message Broker Application Block in the beginning of this guide, you can specify these filters when you register an event handler with the Event Broker Service. Here is a sample of how to set up a `MessageIdFilter`:

```
[C#]

eventBrokerService.Register(this.OnEventPackageClosed,
        new MessageIdFilter(EventPackageClosed.MessageId));
```

There may be times when you want to process several events in the same event handler. In such cases, you can use a `MessageRangeFilter`, which will direct all of these events to that handler. Here is a sample of how to set up the filter:

```
[C#]

eventBrokerService.Register(this.OnEventPackageStatusChanged, new MessageRangeFilter(new
int[]   {
        EventPackageOpened.MessageId, EventPackageClosed.MessageId}));
```

Your event handler might look something like this:

```
[C#]

private void OnEventPackageStatusChanged(IMessage theMessage)
{
        // Common processing goes here...
        if (theMessage.Id == EventPackageOpened.MessageId)
        {
                // EventPackageOpened processing goes here...
        }
        else
        {
                // EventPackageClosed processing goes here...
        }
}
```

Some servers use events that have the same name as events used by another server. One example is `EventError`, which is used by just about every server except Stat Server. The Event Handling Using the Message Broker Application Block article shows how to use a Protocol Description object to filter events by server type in order to avoid confusion when handling these events.

There also may be times when you have several instances of a given server in your environment and you want to filter by a specific one. To do this, first specify an `Endpoint` for that server, using a name for the server in the `Endpoint` constructor:

```
[C#]

string statServer1EndpointName = "StatServer1";
Endpoint statServer1Endpoint =
        new Endpoint(statServer1EndpointName, statServer1Uri);
```

Now create the filter:

```
[C#]

MessageIdFilter statServer1EndpointFilter =
        new MessageIdFilter(EventPackageOpened.MessageId);
```

And set the `EndpointName` property of the filter:

```
[C#]

statServer1EndpointFilter.EndpointName = statServer1EndpointName;
```

When you register this filter, the handler you specify will only receive messages that were sent from the instance you mentioned above:

```
[C#]

eventBrokerService.Register(
        this.OnEventPackageOpened_StatServer1, statServer1EndpointFilter);
```

Architecture and Design

The Message Broker Application Block is designed to make it easy for your applications to handle events in an efficient way.

Message Broker allows you to set up individual classes to handle specific events coming from Genesys servers. It receives all of the events from the servers you specify, and sends each one to the appropriate handler class. Message Broker is a high-performance way to hide the complexity of event-driven programming — so you can focus on other areas of your application.

Tip

Message Broker has been designed for use with the Protocol Manager Application Block. Protocol Manager is another high-performance component that makes it easy for your applications to connect to Genesys servers. You can find basic information on how to use the Protocol Manager Application Block in the article on Connecting to a Server Using the Protocol Manager Application Block.

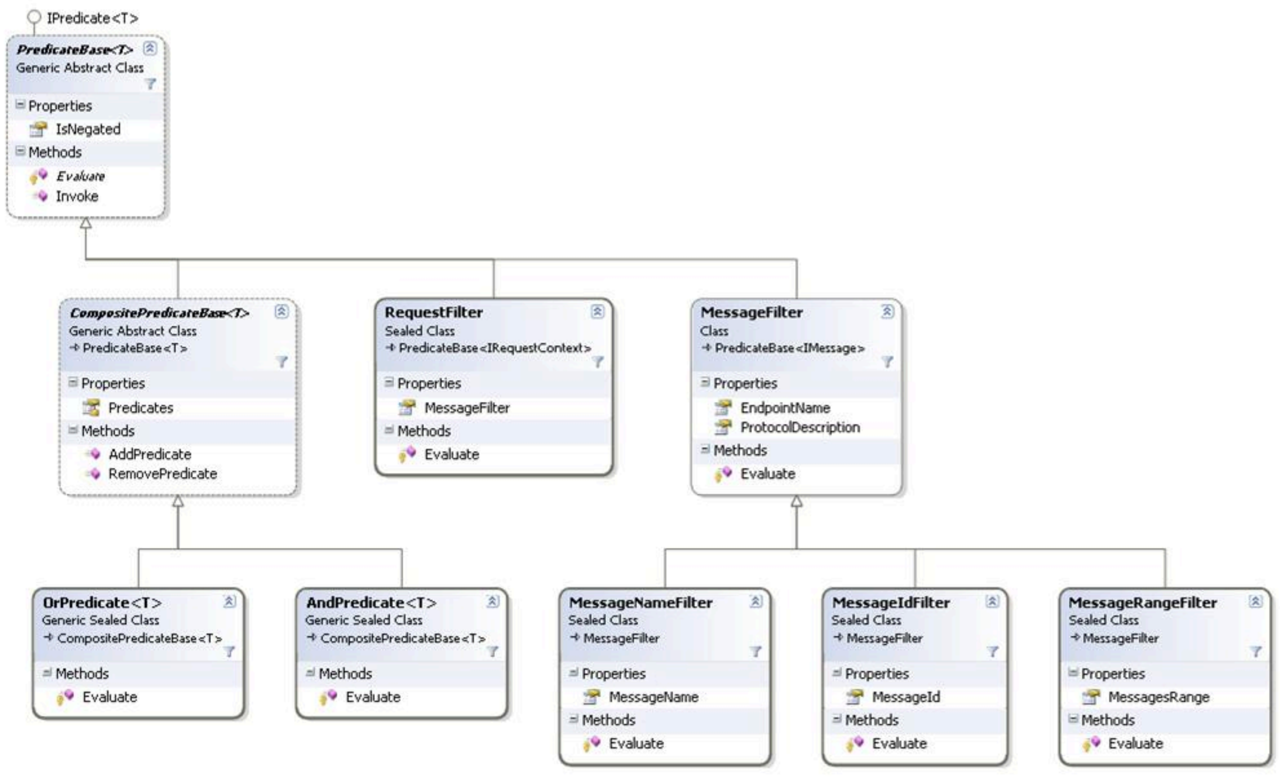
The Message Broker Application Block Architecture

The Message Broker Application Block uses a service-based API that enables you to write individual methods that handle one or more events.

For example, you might want to handle every occurrence of `EventAgentLogin` with a specific dedicated method, while there might be other events that you wish to send to a common event-handling method. Message Broker allows you write these methods and register them with an event broker that manages them for you.

Message Filters

Message Broker uses *message filters* to identify specific messages, assign them to specified methods, and route them accordingly. These message filters are shown in greater detail in the figure below.


IPredicate<T>
PredicateBase<T> Generic Abstract Class
Properties
IsNegated
Methods
Evaluate
Invoke
CompositePredicateBase<T> Generic Abstract Class → PredicateBase<T>
Properties
Predicates
Methods
AddPredicate
RemovePredicate
RequestFilter Sealed Class → PredicateBase<IRequestContext>
Properties
MessageFilter
Methods
Evaluate
MessageFilter Class → PredicateBase<IMessage>
Properties
EndpointName
ProtocolDescription
Methods
Evaluate
OrPredicate<T> Generic Sealed Class → CompositePredicateBase<T>
Methods
Evaluate
AndPredicate<T> Generic Sealed Class → CompositePredicateBase<T>
Methods
Evaluate
MessageNameFilter Sealed Class → MessageFilter
Properties
MessageName
Methods
Evaluate
MessageIdFilter Sealed Class → MessageFilter
Properties
MessageId
Methods
Evaluate
MessageRangeFilter Sealed Class → MessageFilter
Properties
MessagesRange
Methods
Evaluate

Design Patterns

This section gives an overview of the design patterns used in the Message Broker Application Block.

Publish/Subscribe Pattern

There are many occasions when one class (the subscriber) needs to be notified when something changes in another class (the publisher). Message Broker uses the Publish/Subscribe pattern to inform the client application when events arrive from the server.

Factory Method Pattern

It is common practice for a class to include constructors that enable clients of the class instantiate it. There are times, however, when a client may need to instantiate one of several different classes. In some of these situations, the client should not need to decide which class is being created. In this case, a Factory Method pattern is used. The Factory Method pattern lets a class developer define the interface for creating an object, while retaining control of which class to instantiate.

Event Handling Using the Message Broker Application Block

Important

The Message Broker Application Block is considered a legacy product as of release 8.1.1 due to changes to the default event-receiving mechanism. Documentation related to this application block is retained for backwards compatibility. For information about event handling *without* use of the deprecated Message Broker Application Block, refer to the Event Handling article.

Once you have connected to a server using the Protocol Manager Application Block, much of the work of your application will be to send messages to that server and then handle the events you receive from it.

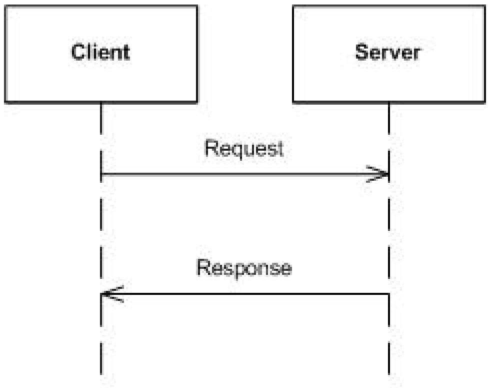
Genesys recommends that you use the **Message Broker Application Block** for most of your event handling needs. This article shows how to send and receive simple synchronous events without using Message Broker and then discusses how to use Message Broker for asynchronous event handling.

Tip

It is important to determine whether your application needs to use synchronous or asynchronous messages. In general, you will probably use only one or the other type in your application. If you decide to use synchronous messages, you must make sure that your code handles all of the messages you receive from your servers. For example, if you send a `RequestReadObjects` message to Configuration Server, you will receive several `EventObjectsRead` messages, followed by an `EventObjectsSent` message. If your application does not handle all of these messages, it will not work properly.

The messages you send to a server are in the form of requests. For example, you may send a request to log in an agent or to gather statistics. You might also send a request to update a configuration object, or to shut down an application.

In each of these cases, the server will respond with an event message, as shown below.



Some of the requests you send may best be handled with a synchronous response, while others may best be handled asynchronously. Let’s talk about synchronous requests first.

Java

Synchronous Requests

Sometimes you might want a synchronous response to your request. For example, if you are using the Open Media Platform SDK, you may want to log in an agent. To do this, you need to let the server know that you want to log in. And then you need to wait for confirmation that your login was successful.

The first thing you need to do is to create a login request, as shown here:

```
[Java]

RequestAgentLogin requestAgentLogin =
        RequestAgentLogin.create(
                tenantId,
                placeId,
                reason);
```

This version of `RequestAgentLogin.Create` specifies most of the information you will need in order to perform the login, but there is one more piece of data required. Here is how to add it:

```
[Java]

requestAgentLogin.setMediaList(mediaList);
```

Once you have created the request and set all required properties, you can make a synchronous request by using the request method of your ProtocolManagementService object, like this:

```
[Java]

Message response = null;
response = protocolManagementServiceImpl.getProtocol("Interaction_Server_App")
        .request(requestAgentLogin);
```

Tip

For information on how to use the ProtocolManagementServiceImpl class of the Protocol Manager Block to communicate with a Genesys server, see the article on Connecting to a Server.

There are two important things to understand when you use the `request` method:

- When you execute this method call, the calling thread will be blocked until it has received a response from the server.
- This method call will only return one message from the server. If the server returns subsequent messages in response to this request, you must process them separately. This can happen in the example of sending a `RequestReadObjects` message to Configuration Server, as mentioned at the beginning of this article.

The response from the server will come in the form of a `Message`. This is the interface implemented by all events in the Platform SDK. Some types of requests will be answered by an event that is specific to the request, while others may receive a more generic response of `EventAck`, which simply acknowledges that your request was successful. If a request fails, the server will send an `EventError`.

A successful `RequestAgentLogin` will receive an `EventAck`, while an unsuccessful one will receive an `EventError`. You can use a switch statement to test which response you received, as outlined here:

```
[Java]

switch(response.messageId())
{
        case EventAck.ID:
                OnEventAck(response);
        case EventError.ID:
                OnEventError(response);
        ...
}
```

Using Message Broker to Handle Asynchronous Requests

There are times when you need to receive asynchronous responses from a server.

First of all, some requests to a server can result in multiple events. For example, if you send a `RequestReadObjects` message, which is used to read objects from the Genesys Configuration Layer, Configuration Server may send more than one `EventObjectsRead` messages in response, depending on whether there is too much data to be handled by a single `EventObjectsRead`.

In other cases, events may be unsolicited. To continue with our example, once you have received all of the `EventObjectsRead` messages, Configuration Server will also send an `EventObjectsSent`, which confirms that it has completed your request.

To make an asynchronous request, you would use the send method of your `ProtocolManagementServiceImpl` class. For example, you might need to fetch information about some objects in the Genesys Configuration Layer. Here is how to set up a `RequestReadObjects`, followed by the send:

```
[Java]

KeyValueCollection filterKey = new KeyValueCollection();
filterKey.addObject("switch_dbid", 113);
filterKey.addObject("dn_type", CfgDNType.CFGExtension.asInteger());
RequestReadObjects requestReadObjects = RequestReadObjects.create(
        CfgObjectType.CfgDN.asInteger(), filterKey);
protocolManagementServiceImpl.getProtocol("Config_Server_App")
        .send(requestReadObjects);
```

This snippet is searching for all DNs that have a type of `Extension` and are associated with the switch that has a database ID of 113.

There are several ways to handle the response from the server, but Genesys recommends that you use the Message Broker Application Block, which is included with the Platform SDK. Message Broker allows you to set up individual classes to handle specific events. It receives the events from the servers you are working with, and sends them to the appropriate handler class. Message Broker is a high-performance way to hide the complexity of event-driven programming — so you can focus on other areas of your application.

To use the Message Broker Application Block, add the following .jar file to the classpath for your application:

- `messagebrokerappblock.jar`

This .jar file was precompiled using the default Application Block code, and can be located at: `<Platform SDK Folder>\lib`.

Tip

You can also view or modify the Message Broker Application Block source code. To do this, open the Message Broker Java source files that were installed with the Platform SDK. The Java source files for this project are located at: `<Platform SDK Folder>\applicationblocks\messagebroker\src\java`. If you make any changes to the project, you will have to run Ant (or use the `build.bat` file for this Application Block) to rebuild the .jar archive listed above. After you run Ant, add the resulting .jar to your classpath.

Now you can add the appropriate `import` statements to your source code. For example:

```
[Java]

import com.genesyslab.platform.applicationblocks.commons.broker.*;
```

In order to use the Message Broker Application Block, you need to create an `EventBrokerService` object to handle the events your application receives. Since you are using the Protocol Manager Application Block to connect to your servers, as shown in the section on Connecting to a Server, you should specify the `ProtocolManagementServiceImpl` object in the `EventBrokerService` constructor:

```
[Java]
```

```
EventBrokerService mEventBrokerService = new EventBrokerService(
        (MessageReceiverSupport) protocolManagementServiceImpl
                .getReceiver());
```

You also need to set up the appropriate filters for your event handlers and register the handlers with the `EventBrokerService`. This allows that service to determine which classes will be used for event-handling. Note that you should register these classes before you open the connection to the server. Otherwise, the server might send events before you are ready to handle them. The sample below shows how to filter on Message ID, which is an integer associated with a particular message:

```
[Java]

mEventBrokerService.register(new ConfObjectsReadHandler(),
        new MessageIdFilter(EventObjectsRead.ID));
mEventBrokerService.register(new ConfObjectsSentHandler(),
        new MessageIdFilter(EventObjectsSent.ID));
mEventBrokerService.register(new StatPackageInfoHandler(),
        new MessageIdFilter(EventPackageInfo.ID));
```

Once you have registered your event-handling classes, you can activate the `EventBrokerService` and open the connection to your server. In the following snippet, connections are being opened to both Configuration Server and Stat Server:

```
[Java]

mEventBrokerService.activate();

protocolManagementServiceImpl.getProtocol("Config_Server_App")
        .open();
protocolManagementServiceImpl.getProtocol("Stat_Server_App").open();
```

At this point, you are ready to set up classes to handle the events you have received from the server. Here is a simple class that handles the `EventObjectsRead` messages:

```
[Java]

class ConfObjectsReadHandler implements Action {

        public void handle(Message obj) {
                EventObjectsRead objectsRead = (EventObjectsRead) obj;
                // Add processing here...
        }
}
```

As mentioned earlier, once Configuration Server has sent all of the information you requested, it will let you know it has finished by sending an `EventObjectsSent` message. Note that this handler has a structure that is similar to the one for `EventObjectsRead`:

```
[Java]

class ConfObjectsSentHandler implements Action {

        public void handle(Message obj) {
                EventObjectsSent objectsSent = (EventObjectsSent) obj;
                // Add processing here...
        }
}
```

`Message Broker only routes non-null messages of the type you specify to your message`

```
handlers. For example, if you send a RequestReadObjects and no objects in the
Configuration Layer meet your filtering criteria, you will not receive an
EventObjectsRead. In that case, you will only receive an EventObjectsSent. Therefore,
you do not need to check for a null message in your EventObjectsRead handler.|2
```

The `EventPackageInfo` handler also has a similar structure, but in this case, we show how to print information about the statistics contained in the requested package:

```
[Java]

class StatPackageInfoHandler implements Action {

  public void handle(Message obj) {
    EventPackageInfo eventPackageInfo = (EventPackageInfo) obj;
    if (eventPackageInfo != null)
    {
      int statisticsCount = eventPackageInfo.getStatistics().getCount();
      StatisticsCollection statisticsCollection = eventPackageInfo.getStatistics();

      for (int i = 0; i < statisticsCount; i++)
      {
        Statistic statistic = statisticsCollection.getStatistic(i);

        System.out.println("\nStatistic Metric is: " +
             statistic.getMetric().toString());
        System.out.println("Statistic Object is: " +
            statistic.getObject());
        System.out.println("Statistic IntValue is: " +
            statistic.getIntValue());
        System.out.println("Statistic StringValue is: " +
            statistic.getStringValue());
        System.out.println("Statistic ObjectValue is: " +
            statistic.getObjectValue());
        System.out.println("Statistic ExtendedValue is: " +
            statistic.getExtendedValue());
         System.out.println("Statistic Tenant is: " +
            statistic.getObject().getTenant());
        System.out.println("Statistic Type is: " +
            statistic.getObject().getType());
        System.out.println("Statistic Id is: " +
            statistic.getObject().getId());
        System.out.println("Statistic TimeProfile is: " +
            statistic.getMetric().getTimeProfile());
         System.out.println("Statistic StatisticType is: " +
            statistic.getMetric().getStatisticType());
        System.out.println("Statistic TimeRange is: " +
            statistic.getMetric().getTimeRange());
        }
      }
    }
  }
```

Filtering Messages by Server

Each server in the Genesys environment makes use of a particular set of events that corresponds to the tasks of that server. For example, Configuration Server sends `EventObjectsRead` and `EventObjectsSent` messages, among others, while Stat Server's events include `EventPackageInfo` and `EventPackageOpened`. Although your applications can identify each of these events by name, it is more efficient to use the ID field associated with an event, which you specify as an `int`. You can do this by using a `MessageIdFilter`, as shown here:

```
[Java]

mEventBrokerService.register(new ConfEventErrorHandler(),
        new MessageIdFilter(EventError.ID));
```

However, the integer used for the Message ID of, say, a Configuration Server message, could be same as the integer used for a completely different message on another server. This could lead to problems if your application works with messages from more than one server. For example, if a multi-server application includes a handler that processes a specific type of message from the first server and that message has an ID of 12, any messages from the other servers that also have a Message ID of 12 will be sent by your `MessageIdFilter` to the same handler.

Fortunately, the Platform SDK allows you to filter messages on a server-by-server basis in addition to filtering on `MessageId`. Here is how to set up a Protocol Description object that allows you to specify that you want some of your handlers to work only with events that are coming from Configuration Server:

```
[Java]

ConfServerProtocol confServerProtocol = (ConfServerProtocol)
        protocolManagementServiceImpl.getProtocol("Config_Server_App");
ProtocolDescription configProtocolDescription = null;
if (confServerProtocol != null)
{
        configProtocolDescription =
                confServerProtocol.getProtocolDescription();
}
```

Once you have set up this Protocol Description, you can use it to indicate that you only want to process events associated with that server, in addition to specifying which event or events you want each handler to process:

```
[Java]

mEventBrokerService.register(new ConfEventErrorHandler(),
        new MessageIdFilter(configProtocolDescription, EventError.ID));
```

You are now ready to open the connection to Configuration Server:

```
[Java]

protocolManagementServiceImpl.
        getProtocol("Config_Server_App").open();
```

Using One Handler for Multiple Events

There may be times when you would like to use a single event handler for more than one event. In that case, you can create the handler and then register the appropriate events with it. For example, you might create a handler for both `EventObjectsRead` and `EventObjectsSent`:

```
[Java]

        class ConfEventHandler implements Action {
                ...
        }
```

You might use a `case` statement inside the handler, in order to process each event appropriately. In any case, once you have set up this handler, all you need to do is register both events with it, as shown here:

```
[Java]

mEventBrokerService.register(new ConfEventHandler(),
        new MessageIdFilter(configProtocolDescription, EventObjectsRead.ID));
mEventBrokerService.register(new ConfEventHandler(),
        new MessageIdFilter(configProtocolDescription, EventObjectsSent.ID));
```

These are the basics of how to use the Message Broker Application Block. For more information, see the Using the Message Broker Application Block article.

.NET

Synchronous Requests

Sometimes you might want a synchronous response to your request. For example, if you are using the Open Media Platform SDK, you may want to log in an agent. To do this, you need to let the server know that you want to log in. And then you need to wait for confirmation that your login was successful.

The first thing you need to do is to create a login request, as shown here:

```
[C#]

RequestAgentLogin requestAgentLogin =
        RequestAgentLogin.Create(
                tenantId,
                placeId,
                reason);
```

This version of `RequestAgentLogin.Create` specifies most of the information you will need in order to perform the login, but there is one more piece of data required. Here is how to add it:

```
[C#]

requestAgentLogin.MediaList = mediaList;
```

Once you have created the request and set all required properties, you can make a synchronous request by using the `Request` method of your `ProtocolManagementService` object, like this:

```
[C#]

IMessage response =
        protocolManagementService["InteractionServer"].
            Request(requestAgentLogin);
```

Tip

> For information on how to use the `ProtocolManagementService` class of the Protocol Manager Application Block to communicate with a Genesys server, see the article on Connecting to a Server Using the Protocol Manager Application Block.

There are two important things to understand when you use the Request method:

- When you execute this method call, the calling thread will be blocked until it has received a response from the server.
- This method call will only return one message from the server. If the server returns subsequent messages in response to this request, you must process them separately. This can happen in the example of sending a `RequestReadObjects` message to Configuration Server, as mentioned at the beginning of this article.

The response from the server will come in the form of an `IMessage`. This is the interface implemented by all events in the Platform SDK. Some types of requests will be answered by an event that is specific to the request, while others may receive a more generic response of `EventAck`, which simply acknowledges that your request was successful. If a request fails, the server will send an `EventError`.

A successful `RequestAgentLogin` will receive an `EventAck`, while an unsuccessful one will receive an `EventError`. You can use a switch statement to test which response you received, as outlined here:

```
[C#]

switch(response.Id)
{
        case EventAck.MessageId:
                OnEventAck(response);
        case EventError.MessageId:
                OnEventError(response);
        ...
}
```

Using Message Broker to Handle Asynchronous Requests

There are times when you need to receive asynchronous responses from a server.

First of all, some requests to a server can result in multiple events. For example, if you send a `RequestReadObjects` message, which is used to read objects from the Genesys Configuration Layer, Configuration Server may send more than one `EventObjectsRead` messages in response, depending on whether there is too much data to be handled by a single `EventObjectsRead`.

In other cases, events may be unsolicited. To continue with our example, once you have received all of the `EventObjectsRead` messages, Configuration Server will also send an `EventObjectsSent`, which confirms that it has completed your request.

To make an asynchronous request, you would use the Send method of your `ProtocolManagementService` class. Here is how to set up a `RequestReadObjects`, followed by the Send:

```
[C#]

KeyValueCollection filterKey = new KeyValueCollection();
filterKey.Add("switch_dbid", 113);
filterKey.Add("dn_type", (int) CfgDNType.Extension);
RequestReadObjects requestReadObjects =
        RequestReadObjects.Create(
                (int) CfgObjectType.CFGDN,
                filterKey);
protocolManagementService["ConfigServer"].Send(requestReadObjects);
```

This snippet is searching for all DNs that have a type of `Extension` and are associated with the switch that has a database ID of 113.

There are several ways to handle the response from the server, but Genesys recommends that you use the Message Broker Application Block, which is included with the Platform SDK. Message Broker allows you to set up individual handlers for specific events. It receives the events from the servers you are working with, and sends them to the appropriate handler. Message Broker is a high-performance way to hide the complexity of event-driven programming — so you can focus on other areas of your application.

To use the Message Broker Application Block, open the Solution Explorer for your application project and add a reference to the following file:

- Genesyslab.Platform.ApplicationBlocks.Commons.Broker.dll

This dll file is precompiled using the default Application Block code, and can be located at: `<Platform SDK Folder>\Bin`.

Tip

You can also view or modify the Message Broker Application Block source code. To do this, open the Message Broker Visual Studio project that was installed with the Platform SDK. The solution file for this project is located at: `<Platform SDK Folder>\ApplicationBlocks\MessageBroker`. If you make any changes to the project, you will have to rebuild the .dll file listed above.

Once you have added the reference, you can add a using statement to your source code:

```
[C#]

using Genesyslab.Platform.ApplicationBlocks.Commons.Broker;
```

In order to use the Message Broker Application Block, you need to create an `EventBrokerService` object to handle the events your application receives. Declare this object with your other fields:

```
[C#]

EventBrokerService eventBrokerService;
```

Then you can set up the `EventBrokerService` to receive events from the Protocol Manager Application Block's `ProtocolManagementService` class, which you are using to connect to your servers, as shown in the section on Connecting to a Server:

```
[C#]

eventBrokerService = new EventBrokerService(protocolManagementService.Receiver);
```

Now you are ready to set up your event handlers.

Note that there are two ways to do this. In 7.5, when Message Broker was introduced, you needed to use attributes to filter the events you wanted processed by a particular handler. Starting in 7.6, you can still do it that way, but you can also set up your filters in the statement that registers an event handler with the Event Broker service, rather than using attributes that are associated with the handler itself. This new method may perform better than the old way, but we will show you how to use both.

Using Event Handlers Without Attributes

Let us start by setting up a couple of event handlers. First, here is a simple handler for the `EventError` message:

```
[C#]

private void OnConfEventError(IMessage theMessage)
{
        EventError eventError = theMessage as EventError;
        /// Add processing here...
}
```

And here is one for the `EventObjectsRead` message:

```
[C#]

private void OnConfEventObjectsRead(IMessage theMessage)
{
        EventObjectsRead objectsRead = theMessage as EventObjectsRead;
        /// Add processing here...
}
```

As mentioned earlier, once Configuration Server has sent all of the information you requested, it will let you know it has finished by sending an `EventObjectsSent` message. Here is a handler for that:

```
[C#]

private void OnConfEventObjectsSent(IMessage theMessage)
{
        EventObjectsSent objectsSent = theMessage as EventObjectsSent;
        /// Add processing here...
}
```

Now you can set up the appropriate filters for your event handlers and register the handlers with the `EventBrokerService`. This allows that service to determine which classes will be used for event-handling. Note that you should register these handlers before you open the connection to the server. Otherwise, the server might send events before you are ready to handle them. The sample below shows how to filter on Message ID, which is an integer associated with a particular message:

```
[C#]

eventBrokerService.Register(
        this.OnConfEventError,
        new MessageIdFilter(EventError.MessageId));
```

```
eventBrokerService.Register(
        this.OnConfEventObjectsRead,
        new MessageIdFilter(EventObjectsRead.MessageId));
eventBrokerService.Register(
        this.OnConfEventObjectsSent,
        new MessageIdFilter(EventObjectsSent.MessageId));
```

`Message Broker only routes non-null messages of the type you specify to your message handlers. For example, if you send a RequestReadObjects and no objects in the Configuration Layer meet your filtering criteria, you will not receive an EventObjectsRead. In that case, you will only receive an EventObjectsSent. Therefore, you do not need to check for a null message in your EventObjectsRead handler.`

Filtering Messages by Server

Each server in the Genesys environment makes use of a particular set of events that corresponds to the tasks of that server. For example, Configuration Server sends `EventObjectsRead` and `EventObjectsSent` messages, among others, while Stat Server's events include `EventPackageInfo` and `EventPackageOpened`. Although your applications can identify each of these events by name, it is more efficient to use the ID field associated with an event, which you specify as an int. You can do this by using a `MessageIdFilter`, as shown here:

```
[C#]

eventBrokerService.Register(this.OnConfEventError);
```

However, the integer used for the Message ID of, say, a Configuration Server message, could be same as the integer used for a completely different message on another server. This could lead to problems if your application works with messages from more than one server. For example, if a multi-server application includes a handler that processes a specific type of message from the first server and that message has an ID of 12, any messages from the other servers that also have a Message ID of 12 will be sent by your `MessageIdFilter` to the same handler.

Fortunately, the Platform SDK allows you to filter messages on a server-by-server basis in addition to filtering on `MessageId`. Here is how to set up a Protocol Description object that allows you to specify that you want some of your handlers to work only with events that are coming from Configuration Server:

```
[C#]

ConfServerProtocol confServerProtocol =
        protocolManagementService["Config_Server_App"]
                as ConfServerProtocol;
ProtocolDescription configProtocolDescription = null;
if (confServerProtocol != null)
{
        configProtocolDescription =
                confServerProtocol.ProtocolDescription;
}
```

Once you have set up this Protocol Description, you can use it to indicate that you only want to process events associated with that server, in addition to specifying which event or events you want each handler to process:

```
[C#]

eventBrokerService.Register(
        this.OnConfEventError,
                new MessageIdFilter(
                        configProtocolDescription,
                        EventError.MessageId));
eventBrokerService.Register(
        this.OnConfEventObjectsRead,
                new MessageIdFilter(
                         configProtocolDescription,
                        EventObjectsRead.MessageId));
eventBrokerService.Register(
        this.OnConfEventObjectsSent,
                new MessageIdFilter(
                        configProtocolDescription,
                        EventObjectsSent.MessageId));
```

You are now ready to open the connection to Configuration Server:

```
[C#]

protocolManagementService["Config_Server_App"].Open();
```

Using One Handler for Multiple Events

There may be times when you would like to use a single event handler for more than one event. In that case, you can create the handler and then register the appropriate events with it. For example, you might create a handler for both `EventObjectsRead` and `EventObjectsSent`:

```
[C#]

private void OnConfEvents (IMessage theMessage) {
        ...
}
```

You might use a case statement inside the handler, in order to process each event appropriately. In any case, once you have set up this handler, all you need to do is register both events with it, as shown here:

```
[C#]

eventBrokerService.Register(
        this.OnConfEvents,
                new MessageIdFilter(
                        configProtocolDescription,
                        EventObjectsRead.MessageId));
eventBrokerService.Register(
        this.OnConfEvents,
                new MessageIdFilter(
                        configProtocolDescription,
                        EventObjectsSent.MessageId));
```

Using Attributes with Your Event Handlers

As mentioned above, you can also use attributes to filter your event handlers. It is important to note that this may not perform as well as the method outlined above, but in case you would like to use attributes in your application, here is how to proceed.

When you use attributes, you have to specify the name of the protocol object you are using, and the name of the SDK it is part of, as shown here:

```
[C#]

private const string protocolName = "ConfServer";
private const string sdkName = "Configuration";
```

These values can be determined by accessing the `ProtocolDescription.ProtocolName` and `ProtocolDescription.SdkName` properties of your protocol object. They are also provided in the following table.

SDK	SdkName	Protocol Object	ProtocolName
Configuration Platform SDK	Configuration	ConfServerProtocol	ConfServer
Contacts Platform SDK	Contacts	UniversalContactServerProtocol	ContactServer
Management Platform SDK	Management	• LocalControlAgentProtocol • MessageServerProtocol • SolutionControlServerProtocol	• LocalControlAgent • MessageServer • SolutionControlServer
Open Media Platform SDK	OpenMedia	• InteractionServerProtocol • ExternalServiceProtocol	• InteractionServer • ExternalService
Outbound Contact Platform SDK	Outbound	OutboundServerProtocol	OutboundServer
Routing Platform SDK	Routing	• RoutingServerProtocol • UrsCustomProtocol	• RoutingServer • CustomServer
Statistics Platform SDK	Reporting	StatServerProtocol	StatServer
Voice Platform SDK	Voice	TServerProtocol	TServer
Web Media Platform SDK	WebMedia	• BasicChatProtocol • FlexChatProtocol • EmailProtocol • EspEmailProtocol • CallbackProtocol	• BasicChat • FlexChat • Email • EspEmail • Callback

Table 1: Platform SDK SdkName and ProtocolName Values

You also need to register the methods you will handle your events with. This allows the `EventBrokerService` to determine which methods will be used for event-handling. When registering for event handlers that use attributes, you only specify the name of the event-handling method. In this case, you need to handle three different events. Note that you should register these methods before you open the connection to the server, as shown here. Otherwise, the server might send events before you are ready to handle them:

```
[C#]
```

```
eventBrokerService.Register(this.OnConfEventObjectsRead);
eventBrokerService.Register(this.OnConfEventObjectsSent);
eventBrokerService.Register(this.OnConfEventError);
protocolManagementService["Config_Server_App"].Open();
```

At this point, you are ready to set up methods to handle the events you have received from the server. Here is a simple method that handles the `EventError` message:

```
[C#]

[MessageIdFilter(EventError.MessageId, ProtocolName = "ConfServer", SdkName =
"Configuration")]
private void OnConfEventError(IMessage theMessage)
{
        EventError eventError = theMessage as EventError;
        /// Add processing here...
}
```

Notice that there is a `MessageIdFilter` attribute right before the method body. This attribute indicates that all `EventError` messages for the Configuration Platform SDK's Configuration protocol will be handled by this method.

The attributes and methods for `EventObjects`Read have a similar structure:

```
[C#]

[MessageIdFilter(EventObjectsRead.MessageId, ProtocolName = "ConfServer", SdkName =
"Configuration")]
private void OnConfEventObjectsRead(IMessage theMessage)
{
        EventObjectsRead objectsRead = theMessage as EventObjectsRead;
        /// Add processing here...
}
```

And so do the attributes and methods for `EventObjectsSent`:

```
 [C#]

[MessageIdFilter(EventObjectsSent.MessageId, ProtocolName = "ConfServer", SdkName =
"Configuration")]
private void OnConfEventObjectsSent(IMessage theMessage)
{
        //protocolManagementService["Config_Server_App"].Close();
        EventObjectsSent objectsSent = theMessage as EventObjectsSent;
        /// Add processing here...

}
```

If you want to process more than one event with a single handler, you can set up multiple attributes for that handler, like this:

```
[C#]

[MessageIdFilter(EventObjectsRead.MessageId, ProtocolName = "ConfServer", SdkName =
"Configuration")]
[MessageIdFilter(EventObjectsSent.MessageId, ProtocolName = "ConfServer", SdkName =
"Configuration")]
```

```
private void OnConfEvents (IMessage theMessage) {
        ...
}
```

These are the basics of how to use the Message Broker Application Block. For more information, see the Using the Message Broker Application Block article.

Using the Protocol Manager Application Block

Deprecation Notice: This application block is considered a legacy product staring with release 8.1.1. Documentation is provided for backwards compatibility, but new development should consider using the improved method of connecting to servers.

> **Important**
>
> This application block is a reusable production-quality component. It has been designed using industry best practices and provided with source code so it can be used "as is," extended, or tailored if you need to.
>
> Please see the License Agreement for details.

One of the two main functions of the Platform SDK is to enable your applications to establish and maintain connections with Genesys servers. The Protocol Manager Application Block provides unified management of server protocol objects. It takes care of opening and closing connections to many different servers, as well as reconfiguration of high availability connections.

Java

Installing the Protocol Manager Application Block

Before you install the Protocol Manager Application Block, it is important to review the software requirements and the structure of the software distribution.

Building the Protocol Manager Application Block

To build the Protocol Manager Application Block:

1. Open the `<Platform SDK Folder>\applicationblocks\protocolmanager` folder.
2. Run either `build.bat` or `build.sh`, depending on your platform.

This will create the `protocolmanagerappblock.jar` file, located within the `<Platform SDK Folder>\applicationblocks\protocolmanager\dist\lib` directory.

Working with the Protocol Manager Application Block

You can find basic information on how to use the Protocol Manager Application Block in the article on Connecting to a Server Using the Protocol Manager Application Block.

Configuring ADDP

To enable ADDP, set the `UseAddp` property of your Configuration object to `true`. You can also set server and client timeout intervals, as shown here:

```
[Java]

statServerConfiguration.setUseAddp(true);
statServerConfiguration.setAddpServerTimeout(10);
statServerConfiguration.setAddpClientTimeout(10);
```

Tip

To avoid connection exceptions in the scenario where a client has configured ADDP but the server has not, "ADDP" is included as a default value for the "protocol" key in the `configure()` method of the `ServerChannel` class.

Configuring Warm Standby

Enable warm standby in your application by setting your Configuration object's `FaultTolerance` property to `FaultToleranceMode.WarmStandby`, as shown here. You can also configure the backup server's URI, the timeout interval, and the number of times your application will attempt to contact the primary server before switching to the backup:

```
[Java]

statServerConfiguration
                .setFaultTolerance(FaultToleranceMode.WarmStandby);
statServerConfiguration.setWarmStandbyTimeout(10);
statServerConfiguration.setWarmStandbyAttempts((short) 5);
try {
        statServerConfiguration.setWarmStandbyUri(new URI("tcp://"
                + statServerBackupHost
                + ":"
                + statServerBackupPort));
} catch (URISyntaxException e) {
        e.printStackTrace();
}
```

High-Performance Message Parsing

The Platform SDK exposes the protocols of supported Genesys servers as an API. This means you can write .NET and Java applications that communicate with these servers in their native protocols.

Every message you receive from a Genesys server is formatted in some way. Most Genesys servers use binary protocols, while some use XML-based protocols. When your application receives one of these messages, it parses the message and places it in the message queue for the appropriate protocol.

By default, the Platform SDK uses a single thread for all of this message parsing. Since this parsing can be time-consuming in certain cases, some applications may face serious performance issues. For example, some applications may receive lots of large binary-format messages, such as some of the statistics messages generated by Stat Server, while others might need to parse messages in non-binary formats, such as the XML format used to communicate with Genesys Multimedia (or e-Services) servers.

This section gives an example of how you can modify Protocol Manager to selectively enable multi-threaded parsing of incoming messages, in order to work around these kinds of performance issues. It is important to stress that you must take a careful look at which kind of multi-threading options to pursue in your applications, since your needs are specific to your situation.

> **Tip**
> Your application may also face other performance bottlenecks. For example, you may need more than one instance of the Message Broker Application Block if you handle large numbers of messages. For more information on how to configure Message Broker for high-performance situations, see the Message Broker Application Block Guide.

This example shows how to call `com.genesyslab.platform.commons.threading.DefaultInvoker`, which uses `SingleThreadInvoker` behind the scenes. As mentioned, you need to determine whether this is the right solution for your application.

The main thing to take from this example is how to set up an invoker interface, so that you can use another invoker if `DefaultInvoker` doesn't meet your needs. For example, Genesys also supplies `com.genesyslab.platform.commons.threading.SingleThreadInvoker`, which assigns a single dedicated thread to each protocol that enables it in your application. This may be useful in some cases where you have to parse XML messages.

The enhancement shown here will only require small changes to two of the classes in Protocol Manager, namely `ProtocolConfiguration` and `ProtocolFacility`.

To get started, let's declare a new multi-threaded parsing property in the `ProtocolConfiguration` class. In this example, the property is called `useMultiThreadedMessageParsing`. It is declared right after some ADDP and Warm Standby declarations:

```
[Java]

private boolean useAddp;
private FaultToleranceMode faultTolerance;
private Boolean useMultiThreadedMessageParsing;
```

Now you can code the getter and setter methods for the property itself, as shown here:

```
[Java]

public Boolean getUseMultiThreadedMessageParsing()
{
        return useMultiThreadedMessageParsing;
}

public void setUseMultiThreadedMessageParsing(Boolean value)
{
        useMultiThreadedMessageParsing = value;
}
```

Once you have made these changes, add an if statement to the `ApplyChannelConfiguration` method of the `ProtocolFacility` class so that your applications can selectively enable this property:

```
[Java]

private void applyChannelConfiguration(
        ProtocolConfiguration conf, ProtocolInstance instance)
{
        if (conf.getUri() != null)
        {
                instance.getProtocol().setEndpoint(
                        new Endpoint(conf.getName(), conf.getUri()));
        }

        if (conf.getUseMultiThreadedMessageParsing() != null &&
                        conf.getUseMultiThreadedMessageParsing().booleanValue())
        {
                instance.getProtocol().
                        setConnectionInvoker(DefaultInvoker.getSingletonInstance());
        }
...
```

Enabling `UseMultiThreadedMessageParsing` now calls `DefaultInvoker`.

To enable multi-threaded parsing, set the `useMultiThreadedMessageParsing` property of your Configuration object to true. Here is how to enable the new property for Stat Server messages:

```
[Java]

statServerConfiguration.setUseMultiThreadedMessageParsing(true);
```

Receiving Copies of Synchronous Server Messages

Most of the time, when you send a synchronous message to a server, you are satisfied to receive the response synchronously. But there can be situations where you want to receive a copy of the response asynchronously, as well. This section shows how to do that.

As in the previous section, this enhancement will only require small changes to the `ProtocolConfiguration` and `ProtocolFacility` classes.

To get started, let's declare a new `copyResponse` property in the `ProtocolConfiguration` class. You can put this declaration right after the `useMultiThreadedMessageParsing` declaration we created in the previous section:

```
[Java]

private boolean useAddp;
private FaultToleranceMode faultTolerance;
private Boolean useMultiThreadedMessageParsing;
private Boolean copyResponse;
```

Now you can code the getter and setter methods for the property itself, as shown here:

```
[Java]

public Boolean getCopyResponse()
{
        return copyResponse;
}

public void setCopyResponse(Boolean value)
{
        copyResponse = value;
}
```

It might be a good idea to let anyone using Protocol Manager know whether this property is enabled. One way to do this is to add it to the `toString` method in this class:

```
[Java]

public String toString()
{
        StringBuilder sb = new StringBuilder();
        .
        .
        .
        sb.append(MessageFormat.format(
                    "AddpClientTimeout: {0}\n", addpClientTimeout));
        sb.append(MessageFormat.format(
                    "AddpServerTimeout: {0}\n", addpServerTimeout));
        sb.append(MessageFormat.format(
                "CopyResponse: {0}\n", copyResponse));
        ...
```

Once you have made these changes, add an if statement to the `applyChannelConfiguration` method of the `ProtocolFacility` class so that your applications can selectively enable this property:

```
[Java]

private void applyChannelConfiguration(
        ProtocolConfiguration conf, ProtocolInstance instance)
{
        if (conf.getUri() != null)
        {
                instance.getProtocol().setEndpoint(
                        new Endpoint(conf.getName(), conf.getUri()));
        }

        if (conf.getCopyResponse() != null)
        {
                instance.getProtocol().setCopyResponse(
                        conf.getCopyResponse());
        }
...
```

To receive a copy of synchronous server messages, set the `CopyResponse` property of your Configuration object to true. Here is how to enable the new property for Stat Server messages:

```
[Java]

statServerConfiguration.setCopyResponse(true);
```

Supporting New Protocols

When the Platform SDK was first developed, it supported many, but not all, of the servers in the Genesys environment. As the SDK has matured, support has been added for more servers. As you might expect, a given version of the Protocol Manager Application Block only supports those servers that were supported by the Platform SDK at the time of its release. Since you may want to work with a server that is not currently supported by Protocol Manager, it can be helpful to know how add support for that server.

This section shows how the Protocol Manager Application Block supports the Stat Server Protocol. You can use it as a guide if you need to add support for other servers or protocols.

Adding support for the Stat Server Protocol involved three basic steps:

1. Create a new subclass of `ProtocolConfiguration` called `StatServerConfiguration`.
2. Create a new subclass of `ProtocolFacility` called `StatServerFacility`.
3. Add a statement to the `initialize` method of `ProtocolManagementServiceImpl` that associates `StatServerFacility` with `StatServerProtocol`.

The StatServerConfiguration Class

Here is the code for `StatServerConfiguration`:

```
[Java]

package com.genesyslab.platform.applicationblocks.commons.protocols;
import com.genesyslab.platform.reporting.protocol.StatServerProtocol;
import java.text.MessageFormat;

public final class StatServerConfiguration extends ProtocolConfiguration
{

    private String clientName;
    private Integer clientId;

    public StatServerConfiguration(String name)
    {
        super(name, StatServerProtocol.class);
    }

     public Integer getClientId()
    {
        return clientId;
    }

    public void setClientId(Integer clientId)
    {
        this.clientId = clientId;
    }

    public String getClientName()
    {
        return clientName;
    }

    public void setClientName(String clientName)
    {
        this.clientName = clientName;
    }

    public String toString()
    {
        StringBuilder sb = new StringBuilder(super.toString());

        sb.append(MessageFormat.format("ClientName: {0}\n", clientName));
        sb.append(MessageFormat.format("ClientId: {0}\n", this.clientId));

        return sb.toString();
    }
}
```

As you can see, this class imports the protocol object, but you will also need to use `MessageFormat` when we create the `toString()` method, so there must be an import statement for that class, as well:

```
[Java]

import com.genesyslab.platform.reporting.protocol.StatServerProtocol;
import java.text.MessageFormat;
```

Here are the class declaration and the field and constructor declarations. Stat Server requires client name and ID, so these must both be present in `StatServerConfiguration`:

```
[Java]

public final class StatServerConfiguration extends ProtocolConfiguration
{

    private String clientName;
    private Integer clientId;

    public StatServerConfiguration(String name)
    {
        super(name, StatServerProtocol.class);
    }
```

Here are the getter and setter methods for the client name and ID:

```
[Java]

public Integer getClientId()
{
        return clientId;
}
```

```
public void setClientId(Integer clientId)
{
        this.clientId = clientId;
}

public String getClientName()
{
        return clientName;
}

public void setClientName(String clientName)
{
        this.clientName = clientName;
}
```

And finally, the `toString()` method:

```
[Java]

public String toString()
{
        StringBuilder sb = new StringBuilder(super.toString());

        sb.append(MessageFormat.format("ClientName: {0}\n", clientName));
        sb.append(MessageFormat.format("ClientId: {0}\n", this.clientId));

        return sb.toString();
}
```

The StatServerFacility Class

Now we can take a look at the `StatServerFacility` class. Once again, we will start with the code for the entire class:

```
[Java]

package com.genesyslab.platform.applicationblocks.commons.protocols;

import com.genesyslab.platform.commons.protocol.Endpoint;
import com.genesyslab.platform.commons.protocol.Protocol;
import com.genesyslab.platform.reporting.protocol.StatServerProtocol;
import java.net.URI;

public final class StatServerFacility extends ProtocolFacility
{

    public void applyConfiguration(
                ProtocolInstance instance, ProtocolConfiguration conf)
    {
        super.applyConfiguration(instance, conf);
        StatServerConfiguration statConf = (StatServerConfiguration)conf;
        StatServerProtocol statProtocol =
                    (StatServerProtocol) instance.getProtocol();

/*
        if (statConf.getClientName() != null)
        {
            statProtocol.setClientName(statConf.getClientName());
        }
*/
```

```
        if (statConf.getClientId() != null)
        {
            statProtocol.setClientId(statConf.getClientId());
        }
    }

    public Protocol createProtocol(String name, URI uri)
    {
        return new StatServerProtocol(new Endpoint(name, uri));
    }
}
```

This class needs the following import statements:

```
[Java]

import com.genesyslab.platform.commons.protocol.Endpoint;
import com.genesyslab.platform.commons.protocol.Protocol;
import com.genesyslab.platform.reporting.protocol.StatServerProtocol;
import java.net.URI;
```

Here is how to declare the class:

```
[Java]

public final class StatServerFacility extends ProtocolFacility
```

There are two methods in this class. The first is `applyConfiguration`:

```
[Java]

public void applyConfiguration(
                        ProtocolInstance instance, ProtocolConfiguration conf)
{
        super.applyConfiguration(instance, conf);
        StatServerConfiguration statConf = (StatServerConfiguration)conf;
        StatServerProtocol statProtocol =
                                (StatServerProtocol) instance.getProtocol();

/*
        if (statConf.getClientName() != null)
        {
                statProtocol.setClientName(statConf.getClientName());
        }
*/
        if (statConf.getClientId() != null)
        {
                statProtocol.setClientId(statConf.getClientId());
        }
}
```

The second method is `createProtocol`:

```
[Java]

public Protocol createProtocol(String name, URI uri)
{
        return new StatServerProtocol(new Endpoint(name, uri));
}
```

Updating ProtocolManagementServiceImpl

To complete this enhancement, a single line of code was added to the initialize method of `ProtocolManagementServiceImpl`:

```
[Java]

private void Initialize()
{
        this.facilities.Add(typeof(ConfServerProtocol), new ConfServerFacility());
        this.facilities.Add(typeof(TServerProtocol), new TServerFacility());
        this.facilities.Add(typeof(InteractionServerProtocol), new
InteractionServerFacility());
        this.facilities.Add(typeof(StatServerProtocol), new StatServerFacility());
        this.facilities.Add(typeof(OutboundServerProtocol), new OutboundServerFacility());
        this.facilities.Add(typeof(LocalControlAgentProtocol), new LcaFacility());
        this.facilities.Add(typeof(SolutionControlServerProtocol), new ScsFacility());
        this.facilities.Add(typeof(MessageServerProtocol), new MessageServerFacility());
}
```

Architecture and Design

The Protocol Manager Application Block uses a service-based API. You can use this API to open and close your connection with Genesys servers and to dynamically reconfigure the parameters for a given protocol. Protocol Manager also includes built-in warm standby capabilities.

Protocol Manager uses a `ServerConfiguration` object to describe each server it manages.

.NET

Installing the Protocol Manager Application Block

Before you install the Protocol Manager Application Block, it is important to review the software requirements and the structure of the software distribution.

Building the Protocol Manager Application Block

The Platform SDK distribution includes a Genesyslab.Platform.ApplicationBlocks.Commons.Protocols.dll file that you can use as is. This file is located in the bin directory at the root level of the Platform SDK directory. To build your own copy of this application block, follow the instructions below:

1. Open the <Platform SDK Folder>\ApplicationBlocks\ProtocolManager folder.
2. Double-click ProtocolManager.sln.
3. Build the solution.

Working with the Protocol Manager Application Block

You can find basic information on how to use the Protocol Manager Application Block in the article on Connecting to a Server Using the Protocol Manager Application Block at the beginning of this guide.

Configuring ADDP

To enable ADDP, set the `UseAddp` property of your Configuration object to true. You can also set server and client timeout intervals, as shown here:

```
[C#]

statServerConfiguration.UseAddp = true;
statServerConfiguration.AddpServerTimeout = 10;
statServerConfiguration.AddpClientTimeout = 10;
```

Configuring Warm Standby

Hot standby is not designed to handle situations where both the primary and backup servers are down. It is also not designed to connect to your backup server if the primary server was down when you initiated your connection. However, in cases like these, warm standby will attempt to connect. In fact, warm standby will keep trying one server and then the other, until it does connect. Because of this, you will probably want to enable warm standby in your applications, even if you are already using hot standby.

You can enable warm standby in your application by setting your Configuration object's `FaultTolerance` property to `FaultToleranceMode.WarmStandby`, as shown here. You can also configure the backup server's URI, the timeout interval, and the number of times your application will attempt to contact the primary server before switching to the backup:

```
[C#]

statServerConfiguration.FaultTolerance = FaultToleranceMode.WarmStandby;
statServerConfiguration.WarmStandbyTimeout = 5000;
statServerConfiguration.WarmStandbyAttempts = 5;
statServerConfiguration.WarmStandbyUri = statServerBackupUri;
```

High-Performance Message Parsing

The Platform SDK exposes the protocols of supported Genesys servers as an API. This means you can write .NET and Java applications that communicate with these servers in their native protocols.

Every message you receive from a Genesys server is formatted in some way. Most Genesys servers use binary protocols, while some use XML-based protocols. When your application receives one of these messages, it parses the message and places it in the message queue for the appropriate protocol.

By default, the Platform SDK uses a single thread for all of this message parsing. Since this parsing can be time-consuming in certain cases, some applications may face serious performance issues. For example, some applications may receive lots of large binary-format messages, such as some of the statistics messages generated by Stat Server, while others might need to parse messages in non-binary formats, such as the XML format used to communicate with Genesys Multimedia (or e-Services) servers.

This section gives an example of how you can modify Protocol Manager to selectively enable multi-threaded parsing of incoming messages, in order to work around these kinds of performance issues. It is important to stress that you must take a careful look at which kind of multi-threading options to pursue in your applications, since your needs are specific to your situation.

> Tip
>
> Your application may also face other performance bottlenecks. For example, you may need more than one instance of the Message Broker Application Block if you handle large numbers of messages. For more information on how to configure Message Broker for high-performance situations, see the Using the Message Broker Application Block.

This example shows how to call `Genesyslab.Platform.Commons.Threading.DefaultInvoker`, which uses the .NET thread pool for your message parsing needs. As mentioned, you need to determine whether this is the right solution for your application, since, for example, the .NET thread pool may be heavily used for other tasks.

The main thing to take from this example is how to set up an invoker interface, so that you can use another invoker if `DefaultInvoker` doesn't meet your needs. For example, Genesys also supplies `Genesyslab.Platform.Commons.Threading.SingleThreadInvoker`, which assigns a single dedicated thread to each protocol that enables it in your application. This may be useful in some cases where you have to parse XML messages.

The enhancement shown here will only require small changes to two of the classes in Protocol Manager, namely `ProtocolConfiguration` and `ProtocolFacility`.

To get started, let's declare a new multi-threaded parsing property in the `ProtocolConfiguration` class. In this example, the property is called `useMultiThreadedMessageParsing`. It is nullable and is declared right after some ADDP and Warm Standby declarations:

```
[C#]

private bool? useAddp;
private FaultToleranceMode? faultTolerance;
private string addpTrace;
private bool? useMultiThreadedMessageParsing;
```

Now you can code the property itself, as shown here:

```
[C#]

public bool? UseMultiThreadedMessageParsing
{
        get { return this.useMultiThreadedMessageParsing; }
        set { this.useMultiThreadedMessageParsing = value; }
}
```

Once you have made these changes, add an if statement to the `ApplyChannelConfiguration` method of the `ProtocolFacility` class so that your applications can selectively enable this property:

```
[C#]

private void ApplyChannelConfiguration(ProtocolInstance entry, ProtocolConfiguration conf)
{
        if( conf.Uri != null )
        {
                entry.Protocol.Endpoint = new Endpoint(conf.Name, conf.Uri);
        }

        if (conf.UseMultiThreadedMessageParsing != null &&
conf.UseMultiThreadedMessageParsing.Value)
                {
                        entry.Protocol.SetConnectionInvoker(DefaultInvoker.InvokerSingleton);
        }
...
```

Enabling `UseMultiThreadedMessageParsing` now calls `DefaultInvoker`, which uses the .NET thread pool, as mentioned above.

To enable multi-threaded parsing, set the `UseMultiThreadedMessageParsing` property of your Configuration object to true. Here is how to enable the new property for Stat Server messages:

```
[C#]

statServerConfiguration.UseMultiThreadedMessageParsing = true;
```

Receiving Copies of Synchronous Server Messages

Most of the time, when you send a synchronous message to a server, you are satisfied to receive the response synchronously. But there can be situations where you want to receive a copy of the response asynchronously, as well. This section shows how to do that.

As in the previous section, this enhancement will only require small changes to the `ProtocolConfiguration` and `ProtocolFacility` classes.

To get started, let's declare a new `copyResponse` property in the `ProtocolConfiguration` class. You can put this declaration right after the `useMultiThreadedMessageParsing` declaration we created in the previous section:

```
[C#]

private bool? useAddp;
private FaultToleranceMode? faultTolerance;
private string addpTrace;
private bool? useMultiThreadedMessageParsing;
private bool? copyResponse;
```

Now you can code the property itself, as shown here:

```
[C#]
```

```
public bool? CopyResponse
{
        get { return this.copyResponse; }
        set { this.copyResponse = value; }
}
```

It might be a good idea to let anyone using Protocol Manager know whether this property is enabled. One way to do this is to add it to the `ToString` method overrides in this class:

```
[C#]

public override string ToString()
{
        StringBuilder sb = new StringBuilder();
        .
        .
        .
        sb.AppendFormat("AddpClientTimeout: {0}\n", this.addpClientTimeout.ToString());
        sb.AppendFormat("AddpServerTimeout: {0}\n", this.addpServerTimeout.ToString());
        sb.AppendFormat("CopyResponse: {0}\n", this.copyResponse.ToString());
        ...
```

Once you have made these changes, add an if statement to the `ApplyChannelConfiguration` method of the `ProtocolFacility` class so that your applications can selectively enable this property:

```
[C#]

private void ApplyChannelConfiguration(ProtocolInstance entry, ProtocolConfiguration conf)
{
        if( conf.Uri != null )
        {
                entry.Protocol.Endpoint = new Endpoint(conf.Name, conf.Uri);
        }

        if (conf.CopyResponse != null)
{
        entry.Protocol.CopyResponse = conf.CopyResponse.Value;
}
...
```

To receive a copy of synchronous server messages, set the `CopyResponse` property of your Configuration object to true. Here is how to enable the new property for Stat Server messages:

```
[C#]

statServerConfiguration.CopyResponse = true;
```

Supporting New Protocols

When the Platform SDK was first developed, it supported many, but not all, of the servers in the Genesys environment. As the SDK has matured, support has been added for more servers. As you might expect, a given version of the Protocol Manager Application Block only supports those servers that were supported by the Platform SDK at the time of its release. Since you may want to work with a server that is not currently supported by Protocol Manager, it can be helpful to know how add support for that server.

For example, early versions of Protocol Manager were developed before the Platform SDK supported

Universal Contact Server (UCS). This section shows how to add UCS support to the Protocol Manager Application Block. You can also use these instructions as a guide if you need to add support for other servers.

This enhancement involves three basic steps:

- Create a new subclass of `ProtocolConfiguration`. We will call this class `ContactServerConfiguration`.
- Create a new subclass of `ProtocolFacility` called `ContactServerFacility`.
- Add a statement to the `Initialize` method of `ProtocolManagementService` that associates the new `ContactServerFacility` class with `UniversalContactServerProtocol`.

Creating a ContactServerConfiguration Class

We will use the `StatServerConfiguration` class as a template for the new `ContactServerConfiguration` class. Here is the code for `StatServerConfiguration`:

```
[C#]

using System;
using System.Text;

using Genesyslab.Platform.Reporting.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    public sealed class StatServerConfiguration : ProtocolConfiguration
    {
        #region Fields

        private string clientName;
        private int? clientId;

        #endregion Fields

        public StatServerConfiguration(string name)
            : base(name, typeof(StatServerProtocol))
        {
        }

        #region Properties

                public string ClientName
                {
                        get { return this.clientName; }
                        set { this.clientName = value; }
                }

                public int? ClientId
                {
                        get { return this.clientId; }
                        set { this.clientId = value; }
                }

                #endregion Properties

        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();
            sb.Append(base.ToString());

            sb.AppendFormat("ClientName: {0}\n", this.clientName);
            sb.AppendFormat("ClientId: {0}\n", this.clientId.ToString());

            return sb.ToString();
        }
    }
}
```

To get started, make a copy of `StatServerConfiguration.cs` and call it `ContactServerConfiguration.cs`. Rename the Platform SDK using statement and the class name, as shown here:

```
[C#]

using System;
using System.Text;
using Genesyslab.Platform.Contacts.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    public sealed class ContactServerConfiguration : ProtocolConfiguration
    {
    ...
```

The connection parameters required by Stat Server are different from those used by UCS. Instead of `clientName` and `clientId`, UCS requires `applicationName`. Like `clientName`, `applicationName` is of type string. One fairly simple way to modify this class is to delete all references to `clientId` and rename the references to `clientName` to `applicationName`. Make sure to retain the capitalization in the property name, which should become `ApplicationName`.

```
[C#]

using System;
using System.Text;
using Genesyslab.Platform.Contacts.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    public sealed class ContactServerConfiguration : ProtocolConfiguration
    {
        #region Fields

        private string applicationName;
        ~~private int? clientId;~~

        #endregion Fields

        ...

        #region Properties

                public string ApplicationName
                {
                        get { return this.applicationName; }
                        set { this.applicationName = value; }
                }

                ~~public int? ClientId~~
                ~~{~~
                        ~~get { return this.clientId; }~~
                        ~~set { this.clientId = value; }~~
                ~~}~~

                #endregion Properties

        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();
            sb.Append(base.ToString());

            sb.AppendFormat("applicationName: {0}\n", this.applicationName);
            ~~sb.AppendFormat("ClientId: {0}\n", this.clientId.ToString());~~

            return sb.ToString();
        }
    }
}
```

The constructor also needs to be renamed. This code:

```
[C#]

public StatServerConfiguration(string name)
        : base(name, typeof(StatServerProtocol))
{
}
```

should be replaced with this:

```
[C#]

public ContactServerConfiguration(string name)
        : base(name, typeof(UniversalContactServerProtocol))
{
}
```

When you have made all of these changes, your new class should look like this:

```
[C#]

using System;
using System.Text;
using Genesyslab.Platform.Contacts.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    public sealed class ContactServerConfiguration : ProtocolConfiguration
    {
        #region Fields

        private string applicationName;

        #endregion Fields

        public ContactServerConfiguration(string name)
            : base(name, typeof(UniversalContactServerProtocol))
        {
        }
        #region Properties
        public string ApplicationName
        {
            get { return this.applicationName; }
            set { this.applicationName = value; }
        }

        #endregion Properties

        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();
            sb.Append(base.ToString());

            sb.AppendFormat("ApplicationName: {0}\n", this.applicationName);

            return sb.ToString();
        }

    }
}
```

Creating a ContactServerFacility Class

The next step is to create a copy of `StatServerFacility.cs` and name it `ContactServerFacility.cs`. Here is what the `StatServerFacility` class looks like:

```
[C#]

using System;
using System.Text;

using Genesyslab.Platform.Commons.Collections;
using Genesyslab.Platform.Commons.Protocols;
using Genesyslab.Platform.Reporting.Protocols;
using Genesyslab.Platform.Commons.Logging;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    internal sealed class StatServerFacility : ProtocolFacility
    {
        public override void ApplyConfiguration(ProtocolInstance entry, ProtocolConfiguration
conf, ILogger logger)
        {
            base.ApplyConfiguration(entry, conf, logger);

            StatServerConfiguration statConf = (StatServerConfiguration)conf;
            StatServerProtocol statProtocol = (StatServerProtocol)entry.Protocol;

            if (statConf.ClientName != null)
            {
                statProtocol.ClientName = statProtocol.ClientName;
            }
            if (statConf.ClientId != null)
            {
                statProtocol.ClientId = statConf.ClientId.Value;
            }
        }

        public override ClientChannel CreateProtocol(string name, Uri uri)
        {
            return new StatServerProtocol(new Endpoint(name, uri));
        }
    }
}
```

Start by renaming the `using` statement and the class name:

```
[C#]

using System;
using Genesyslab.Platform.Commons.Logging;
using Genesyslab.Platform.Commons.Protocols;
using Genesyslab.Platform.Contacts.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    internal sealed class ContactServerFacility : ProtocolFacility
    ...
```

Rename `statConf` and `statProtocol`, giving them the correct configuration and protocol types:

```
[C#]

ContactServerConfiguration ucsConf = (ContactServerConfiguration)conf;
UniversalContactServerProtocol ucsProtocol =
        (UniversalContactServerProtocol)entry.Protocol;
```

And delete the references to `ClientId`:

[C#]

~~if (statConf.ClientId != null)~~
~~{~~
        ~~statProtocol.ClientId = statConf.ClientId.Value;~~

Now you can rename `ClientName` to `ApplicationName`:

```
[C#]

if (ucsConf.ApplicationName != null)
{
        ucsProtocol.ApplicationName = ucsConf.ApplicationName;
}
```

When you are finished, you will have a new class that looks like this:

```
[C#]

using System;
using Genesyslab.Platform.Commons.Logging;
using Genesyslab.Platform.Commons.Protocols;
using Genesyslab.Platform.Contacts.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    internal sealed class ContactServerFacility : ProtocolFacility
    {
        public override void ApplyConfiguration(ProtocolInstance entry, ProtocolConfiguration
conf, ILogger logger)
        {
            base.ApplyConfiguration(entry, conf, logger);
            ContactServerConfiguration ucsConf = (ContactServerConfiguration)conf;
            UniversalContactServerProtocol ucsProtocol =
(UniversalContactServerProtocol)entry.Protocol;

            if (ucsConf.ApplicationName != null)
            {
                ucsProtocol.ApplicationName = ucsConf.ApplicationName;
            }
        }

        public override ClientChannel CreateProtocol(string name, Uri uri)
        {
            return new UniversalContactServerProtocol(new Endpoint(name, uri));
        }
    }
}
```

Updating ProtocolManagementService

To complete this enhancement, add a single line of code to the `Initialize` method of `ProtocolManagementService`:

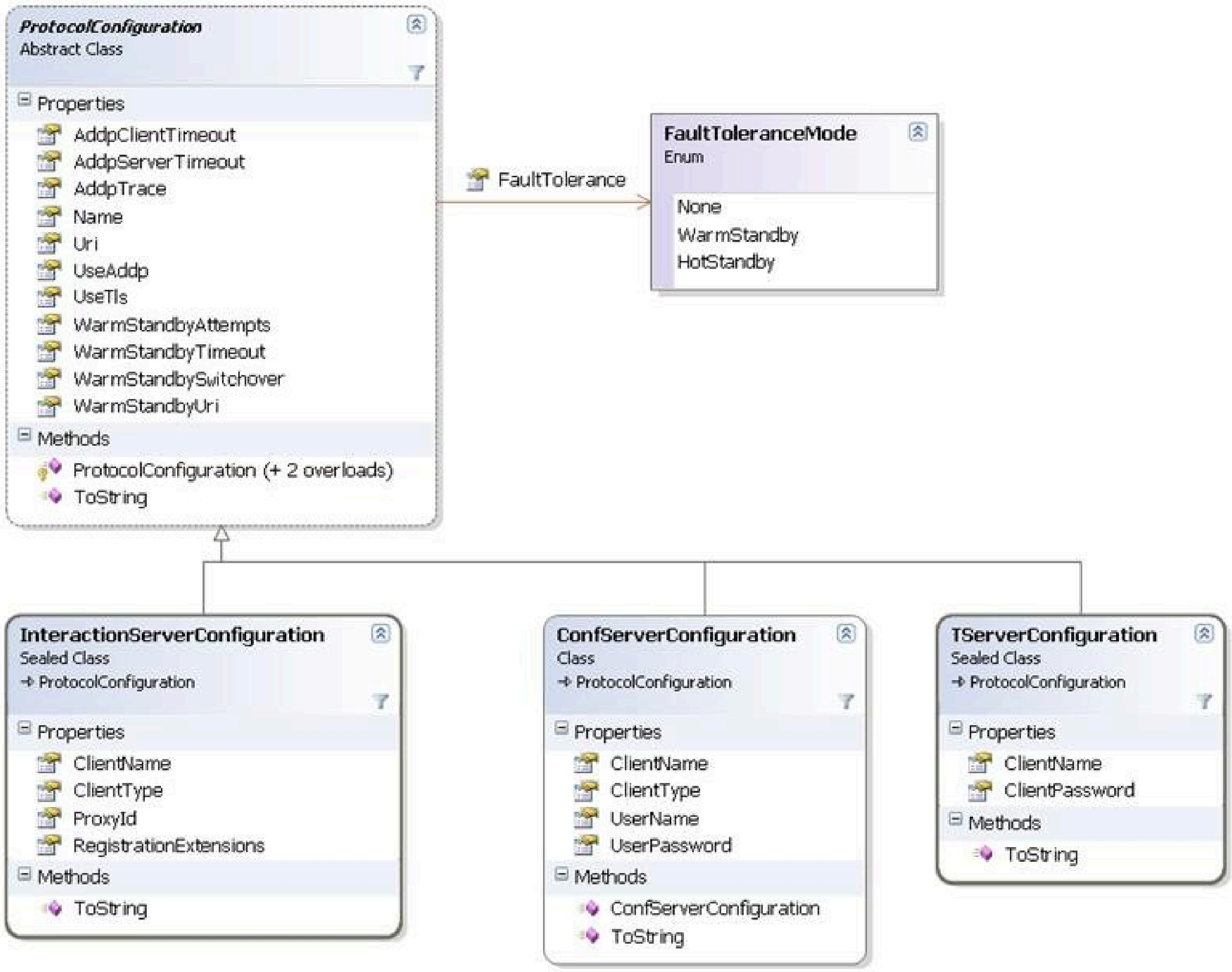
```
[C#]

private void Initialize()
{
        this.facilities.Add(typeof(ConfServerProtocol), new ConfServerFacility());
        this.facilities.Add(typeof(TServerProtocol), new TServerFacility());
        this.facilities.Add(typeof(InteractionServerProtocol), new
InteractionServerFacility());
        this.facilities.Add(typeof(StatServerProtocol), new StatServerFacility());
        this.facilities.Add(typeof(OutboundServerProtocol), new OutboundServerFacility());
        this.facilities.Add(typeof(LocalControlAgentProtocol), new LcaFacility());
        this.facilities.Add(typeof(SolutionControlServerProtocol), new ScsFacility());
        this.facilities.Add(typeof(MessageServerProtocol), new MessageServerFacility());
        this.facilities.Add(typeof(UniversalContactServerProtocol), new
ContactServerFacility());
}
```

Your copy of Protocol Manager now works with Universal Contact Server!

Architecture and Design

The Protocol Manager Application Block uses a service-based API. You can use this API to open and close your connection with Genesys servers and to dynamically reconfigure the parameters for a given protocol. Protocol Manager also includes built-in warm standby capabilities.

Protocol Manager uses a `ServerConfiguration` object to describe each server it manages. The figure below gives examples of the structure of some of these objects.



Tip

Any protocol can be reconfigured dynamically.

Connecting to a Server Using the Protocol Manager Application Block

> **Important**
>
> The Protocol Manager Application Block is considered a legacy product as of release 8.1.1 due to improvements in the configuration of core protocol classes. Documentation related to this application block is retained for backwards compatibility. For information about connecting to Genesys servers *without* use of the Protocol Manager Application Block, refer to the Connecting to a Server article.

The applications you write with the Platform SDK will need to communicate with one or more Genesys servers. So the first thing you need to do is create a connection with these servers. Genesys recommends that you use the **Protocol Manager Application Block** to do this. Protocol Manager is designed for high-performance communication with Genesys servers. It also includes built-in support for warm standby.

Once you have connected to a server, you will be sending and receiving messages to and from this server. The next article shows how to use the Message Broker Application Block for efficient event handling using the Message Broker Application Block.

> **Tip**
>
> Protocol Manager may not support all of the servers you need to use in your application. For information about how to update Protocol Manager to communicate with these servers, see the Using the Protocol Manager Application Block article.

Java

To use the Protocol Manager Application Block, add the following file to your classpath:

- `protocolmanagerappblock.jar`

This jar file was precompiled using the default Application Block code, and can be located at: `<Platform SDK Folder>\lib`.

> **Tip**

> You can also view or modify the Protocol Manager Application Block source code. To do this, open the Protocol Manager Java source files that were installed with the Platform SDK. The Java source files for this project are located at: `<Platform SDK Folder>\applicationblocks\protocolmanager\src\java`. If you make any changes to the project, you will have to run Ant (or use the `build.bat` file for this Application Block) to rebuild the jar archive listed above. After you run Ant, add the resulting jar to your classpath.

Now you can add `import` statements to your source code. For example:

```
[Java]

import com.genesyslab.platform.applicationblocks.commons.protocols.*;
import com.genesyslab.platform.applicationblocks.warmstandby.*;
```

You will also have to add additional JAR archives to your classpath and add `import` statements to your project for each specific protocol you are working with. The steps are not explicitly described here because the archives and classes required will vary depending on which SDKs and protocols you plan to use.

In order to use the Protocol Manager, you need to create a `ProtocolManagementServiceImpl` object. This object manages all of your server connections. Declare it with your other fields:

```
[Java]

ProtocolManagementServiceImpl protocolManagementServiceImpl;
```

Then you can initialize the service object inside the appropriate method body:

```
[Java]

protocolManagementServiceImpl =
        new ProtocolManagementServiceImpl();
```

You are now ready to create an object that will be used to specify how to communicate with the server. For example, if you are working with Configuration Server, you will set up a `ConfServerConfiguration` object:

[Java]

ConfServerConfiguration confServerConfiguration = new
ConfServerConfiguration("Config_Server_App");

Note that you have to provide a string when you create the `ConfServerConfiguration` object. This string should be unique for each protocol used in your application. It might be a good idea to use the name of the server's application object from the configuration layer, which guarantees uniqueness as well as clearly identifying which server you are communicating with.

After setting up the `ConfServerConfiguration` object, you need to specify the URI of the Configuration Server you want to communicate with, as well as a few other necessary pieces of information:

```
[Java]
```

```
try {
        confServerConfiguration.setUri(
                new URI("tcp://" + confServerHost + ":" + confServerPort));
} catch (URISyntaxException e) {
        e.printStackTrace();
}
confServerConfiguration.setClientApplicationType(CfgAppType.CFGSCE);
confServerConfiguration.setClientName(clientName);
confServerConfiguration.setUserName(userName);
confServerConfiguration.setUserPassword(password);
```

At this point, you can register your `ConfServerConfiguration` object with Protocol Manager:

```
[Java]

protocolManagementServiceImpl.register(confServerConfiguration);
```

Now you can tell Protocol Manager to open the connection to your server:

```
[Java]

try {
        protocolManagementServiceImpl.getProtocol("Config_Server_App").open();
} catch (ProtocolException e) {
        e.printStackTrace();
} catch (IllegalStateException e) {
        e.printStackTrace();
} catch (InterruptedException e) {
        e.printStackTrace();
}
```

You may want to set up a connection to more than one server. To do that, you could repeat the steps outlined above. Here is an example of how you might do that in order to add a connection to Stat Server:

```
[Java]

StatServerConfiguration statServerConfiguration = new StatServerConfiguration(
                "Stat_Server_App");
try {
        statServerConfiguration.setUri(new URI(statServerUri));
} catch (URISyntaxException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
}
protocolManagementServiceImpl.register(statServerConfiguration);
.
.
.
// Add this line to the try block for the Configuration Server open()
protocolManagementServiceImpl.getProtocol("Stat_Server_App").open();
```

In some cases, you may want to use the `beginOpen()` method instead of using the `open()` method. `beginOpen()` will open all of your connections with a single method call. However, unlike `open()`, `beginOpen()` is asynchronous. This means you will need to make sure you have received the `onChannelOpened` event before you send any messages. Otherwise, you might be trying to use a connection that does not yet exist.

In order to use `beginOpen()`, you need to implement the `ChannelListener` interface:

```
[Java]

import com.genesyslab.platform.commons.protocol.ChannelListener;
.
.
.
public class YourApplication
        implements ChannelListener, ...
```

You will also need to add a channel listener after you register your `ServerConfiguration` objects:

```
[Java]

protocolManagementServiceImpl.register(confServerConfiguration);
protocolManagementServiceImpl.register(statServerConfiguration);
.
.
.
protocolManagementServiceImpl.addChannelListener(this);
```

Now you can add a method to handle the `OnChannelOpened` event:

```
[Java]

public void onChannelOpened(EventObject event) {
        if ( event.getSource() instanceof ClientChannel )        {
                ClientChannel channel = (ClientChannel)event.getSource();

                if ( channel instanceof ConfServerProtocol ) {
                        // Work with Configuration Server messages...
                }
                else if ( channel instanceof StatServerProtocol ) {
                        // Work with Stat Server messages...
                }
        }
}
```

Having done that, you can remove these lines from the `try` block:

```
[Java]

protocolManagementServiceImpl.getProtocol("Config_Server_App").open();
protocolManagementServiceImpl.getProtocol("Stat_Server_App").open();
```

And replace them with this one:

```
[Java]

protocolManagementServiceImpl.beginOpen();
```

However, if you want to issue an asynchronous open for a specific protocol, you can invoke `beginOpen` for that protocol, like this:

```
[Java]

protocolManagementServiceImpl.getProtocol("Config_Server_App").beginOpen();
protocolManagementServiceImpl.getProtocol("Stat_Server_App").beginOpen();
```

Tip

When using the `beginOpen()` method, make sure that your code waits for the `onChannelOpened` event to fire before attempting to send or receive messages.

Once you have opened your connection, you can send and receive messages, as shown in the article on Event Handling. But before getting to that, please note that when you have finished communicating with your servers, you should close the connection, like this:

```
[Java]

protocolManagementServiceImpl.beginClose();
```

Or like this:

```
[Java]

protocolManagementServiceImpl.getProtocol("Config_Server_App")
        .close();
protocolManagementServiceImpl.getProtocol("Stat_Server_App")
        .close();
```

Or like this:

```
[Java]

protocolManagementServiceImpl.getProtocol("Config_Server_App")
                .beginClose();
protocolManagementServiceImpl.getProtocol("Stat_Server_App")
                .beginClose();
```

This introduction has only covered the most basic features of the Protocol Manager Application Block. Consult the Protocol Manager Application Block Guide for more information on how to use Protocol Manager, including the following topics:

- Configuring ADDP
- Configuring Warm Standby
- High-Performance Message Parsing
- Supporting New Protocols

To learn how to send and receive messages, go to the article on Event Handling Using the Message Broker Application Block.

.NET

To use the Protocol Manager Application Block, open the Solution Explorer for your application project and add references to the following files:

- `Genesyslab.Platform.ApplicationBlocks.Commons.Protocols.dll`

- `Genesyslab.Platform.ApplicationBlocks.WarmStandby.dll`

These dll files are precompiled using the default Application Block code, and can be located at: `<Platform SDK Folder>\Bin`.

> **Tip**
> You can also view or modify the Protocol Manager Application Block source code. To do this, open the Protocol Manager Visual Studio project that was installed with the Platform SDK. The solution file for this project is located at: `<Platform SDK Folder>\ApplicationBlocks\ProtocolManager`. If you make any changes to the project, you will have to rebuild the two .dll files listed above.

Once you have added the references, you can add using statements to your source code:

```
[C#]

using Genesyslab.Platform.ApplicationBlocks.Commons.Protocols;
using Genesyslab.Platform.ApplicationBlocks.WarmStandby;
```

You will also have to reference additional libraries and add using statements to your project for each specific protocol you are working with. The steps are not explicitly described here because the files and namespaces required will vary depending on which SDKs and protocols you plan to use.

In order to use the Protocol Manager, you now need to create a `ProtocolManagementService` object. This object manages all of your server connections. Declare it with your other fields:

```
[C#]

ProtocolManagementService protocolManagementService;
```

Then you can initialize the service object inside the appropriate method body:

```
[C#]

protocolManagementService =
        new ProtocolManagementService();
```

You are now ready to create an object that will be used to specify how to communicate with the server. For example, if you are working with Configuration Server, you will set up a `ConfServerConfiguration` object:

```
[C#]

ConfServerConfiguration confServerConfiguration =
        new ConfServerConfiguration("Config_Server_App");
```

Note that you have to provide a string when you create the `ConfServerConfiguration` object. This string should be unique for each protocol used in your application. It might be a good idea to use the name of the server's application object from the configuration layer, which guarantees uniqueness as well as clearly identifying which server you are communicating with.

After setting up the `ConfServerConfiguration` object, you need to specify the URI of the

Configuration Server you want to communicate with, as well as a few other necessary pieces of information:

```
[C#]

confServerConfiguration.Uri =
        new Uri("tcp://" + confServerHost + ":" + confServerPort);
confServerConfiguration.ClientApplicationType = CfgAppType.CFGSCE;
confServerConfiguration.ClientName = clientName;
confServerConfiguration.UserName = userName;
confServerConfiguration.UserPassword = password;
```

At this point, you can register your `ConfServerConfiguration` object with Protocol Manager:

```
[C#]

protocolManagementService.Register(confServerConfiguration);
```

Now you can tell Protocol Manager to open the connection to your server:

```
[C#]

protocolManagementService["Config_Server_App"].Open();
```

You may want to set up a connection to more than one server. To do that, you could repeat the steps outlined above. Here is an example of how you might do that in order to add a connection to Stat Server:

```
[C#]

StatServerConfiguration statServerConfiguration = new
StatServerConfiguration("Stat_Server_App");
statServerConfiguration.Uri = statServerUri;
protocolManagementService.Register(statServerConfiguration);
.
.
.
protocolManagementService["Stat_Server_App"].Open();
```

In some cases, you may want to use the `BeginOpen()` method instead of using the `Open()` method. `BeginOpen()` will open all of your connections with a single method call. However, unlike `Open()`, `BeginOpen()` is asynchronous. This means you will need to make sure you have received the `Opened` event before you send any messages. Otherwise, you might be trying to use a connection that does not yet exist.

Once you have set up an event handler for the `Opened` event, you can remove these lines from your code:

```
[C#]

protocolManagementService["Config_Server_App"].Open();
protocolManagementService["Stat_Server_App"].Open();
```

And replace them with this one:

```
[C#]

protocolManagementService.BeginOpen();
```

However, if you want to issue an asynchronous open for a specific protocol, you can invoke BeginOpen for that protocol, like this:

```
[C#]

protocolManagementService["Config_Server_App"].BeginOpen();
protocolManagementService["Stat_Server_App"].BeginOpen();
```

Tip

When using the BeginOpen() method, make sure that your code waits for the Opened event to fire before attempting to send or receive messages.

Once you have opened your connection, you can send and receive messages, as shown in the article on Event Handling Using the Message Broker Application Block. But before getting to that, please note that when you have finished communicating with your servers, you should close the connection, like this:

```
[C#]

protocolManagementService.BeginClose();
```

Or like this:

```
[C#]

protocolManagementService["Config_Server_App"].Close();
protocolManagementService["Stat_Server_App"].Close();
```

Or like this:

```
[C#]

protocolManagementService["Config_Server_App"].BeginClose();
protocolManagementService["Stat_Server_App"].BeginClose();
```

This introduction has only covered the most basic features of the Protocol Manager Application Block. Consult Using the Protocol Manager Application Block for more information on how to use Protocol Manager, including the following topics:

- Configuring ADDP
- Configuring Warm Standby
- High-Performance Message Parsing
- Supporting New Protocols

To learn how to send and receive messages, go to the article on Event Handling Using the Message Broker Application Block.

Transport Layer Security (TLS) Support

Platform SDK now supports Transport Layer Security (TLS). This section contains two sample configurations, but it is important to understand your environment and its unique requirements before using this new support. You should refer to the appropriate server manual to configure TLS on your server. You should also refer to Part 3 of the Genesys 8.0 Security Deployment Guide, "Communications Integrity—Transport Layer Security".

The first sample configuration shows a situation where the client application specifies the name of a server-based certificate:

```
[C#]

SolutionControlServerProtocol scsProtocol
        = new SolutionControlServerProtocol(myEndpoint);
KeyValueCollection kvCollection = new KeyValueCollection();
kvCollection[CommonConnection.TlsKey] = 1;
kvCollection[CommonConnection.CertificateNameKey] = "name";
KeyValueConfiguration kvConfig = new KeyValueConfiguration(kvCollection);
scsProtocol.Configure(kvConfig);
```

In this sample configuration, "name" is the name of the certificate, which is located in the certificate store on the server and used in the TLS configuration of the port/application/server in the Genesys Configuration Layer.

The second sample configuration shows a client application using its own client certificate to authenticate on the server:

```
[C#]

SolutionControlServerProtocol scsProtocol
        = new SolutionControlServerProtocol(myEndpoint);
KeyValueCollection kvCollection = new KeyValueCollection();
kvCollection[CommonConnection.TlsKey] = 1;
kvCollection[CommonConnection.CertificateKey] = @"c:\directory\certificate.p12";
kvCollection[CommonConnection.CertificatePwdKey] = "password";
KeyValueConfiguration kvConfig = new KeyValueConfiguration(kvCollection);
scsProtocol.Configure(kvConfig);
```

In this sample configuration, `CommonConnection.CertificateKey` is the path to the certificate file located on the client machine, while `CommonConnection.CertificatePwdKey` is the password which will be used to open the certificate file, if it is password protected.

Legacy Warm Standby Application Block Description

Deprecation Notice	Deprecated In
Significant changes were made to the Warm Standby Application Block with release 8.5.101.x. This documentation is maintained for backwards compatibility. Guidelines about using the **new warm standby implementation** should be followed for any new development.	**Java:** 8.5.101.06 **.NET:** 8.5.101.06

Important

This application block is a reusable production-quality component. It has been designed using industry best practices and provided with source code so it can be used "as is," extended, or tailored if you need to.

Please see the License Agreement for details.

This article examines the architecture and design of the Warm Standby Application Block, which enables developers to switch to a backup server in case their primary server fails without needing to guarantee the integrity of existing interactions. The application block also gives details about how to set up the QuickStart application that ships with this application block.

Java

Architecture and Design

Many contact center environments require redundant backup servers that are able to take over quickly if a primary server fails. In this situation, the primary server operates in active mode, accepting connections and exchanging messages with clients. The backup server, on the other hand, is in standby mode. If the primary server fails, the backup server switches to active mode, assuming the role and behavior of the primary server.

There are two standby modes: *warm standby* and *hot standby*. The main difference between them is that warm standby mode does not ensure the continuation of interactions in progress when a failure occurs, while hot standby mode does.

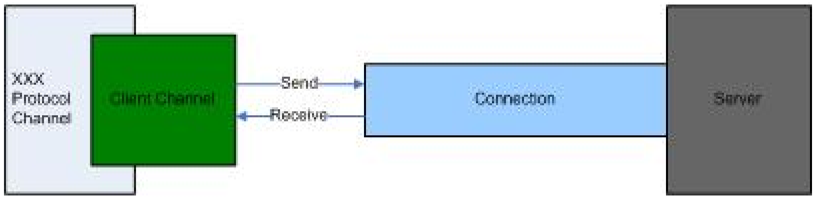
The Client Channel Architecture

Since the Warm Standby Application Block is designed to be used in the context of a Client Channel architecture, it is important to understand that architecture before talking about the application block itself.

To start with, this architecture consists of three functional components:

- A connection
- A client channel
- A protocol channel

These components are shown in the following figure.



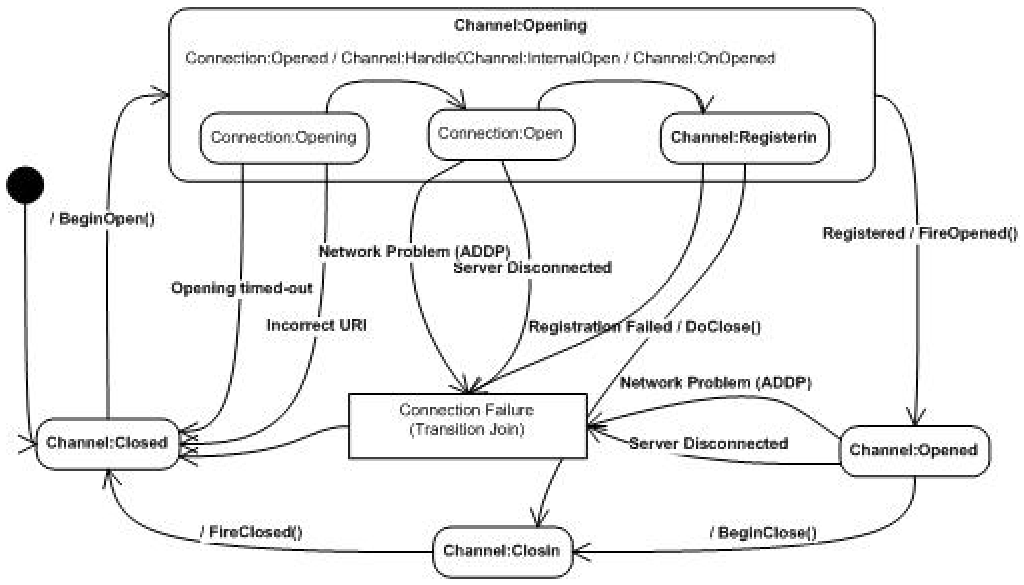
The *connection* controls all necessary TCP/IP connection activities, while the client channel contains the protocol- and server-independent channel functionality that is common for a protocol channel. Finally, the *protocol channel* controls all of the client channel activities that are dependent on the protocol and the server.

Client Channel State

The state of a client channel is based on the state of the corresponding connection. There are four major states:

- Opening (Registration)
- Opened
- Closing
- Closed

The figure below shows a detailed client channel state diagram.



In addition to establishing a TCP/IP connection, several activities may take place when a client channel opens. These activities can include things like:

- A preliminary exchange of messages with the server, which is known as registration
- Reading the client channel's locally stored configuration information

You can often determine the cause of a client channel failure by checking the state of the client channel just before it closed. There are exceptions to this rule, however, such as a registration failure, which is protocol-specific.

Client Channel Failure Scenarios

There are several common client channel failure scenarios:

Client Channel Failure Scenarios

Scenario	Description	Source States	Condition	Target State	Protocol-Dependent
Opening Timed Out	Channel tries to open connection to non-existing URI	Opening	Connection opening timeout	Closed	No
Wrong URI	Channel tries to open connection to non-existing URI	Opening	Incorrect URI exception	Closed	No
Connection Problem	Channel connection detects a connection	Opened Opening	Server disconnected	Closed	No

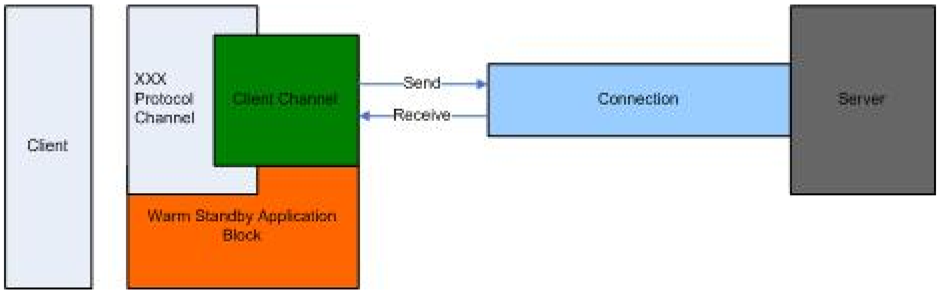
Scenario	Description	Source States	Condition	Target State	Protocol-Dependent
	problem				
Network Problem (ADDP)	Channel connection detects a network problem (ADDP)	Opened Opening	Network problem (ADDP)	Closed	No
Wrong Server or Protocol	Channel tries to open connection with an incorrect server or protocol	Opening	Registration Failed/ ProtocolException	Closing	Yes
Registration Failure	One of the channel registration steps failed	Opening	Registration Failed/ ProtocolException	Closing	Yes

Note that the first four scenarios, *Opening timed-out, Wrong URI, Connection Problem,* and *Network Problem* happen with the connection (TCP/IP) component. They do not involve protocol- or server-specific elements, whether in terms of failure-specific data or in terms of channel recovery actions and data.

The *Wrong Server* or *Protocol* and *Registration Failure* scenarios are protocol- or server-dependent and can be different for each type of protocol channel.

Application Block Architecture

The Warm Standby Application Block's functionality is based on intercepting the channel's transition from a non-closed state to the Closed state. As you can see in the following figure, the application block is able to pick up this information because it sits between the client and protocol channels.



Upon receiving the channel's `Closed` event, the application block uses diagnostic information to determine why the channel has closed. This diagnostic information is necessary to determine what actions, if any, the application block should take to restore the channel's connectivity to the server.

The Warm Standby Application Block can take several different steps to recover channel connectivity. These steps are:

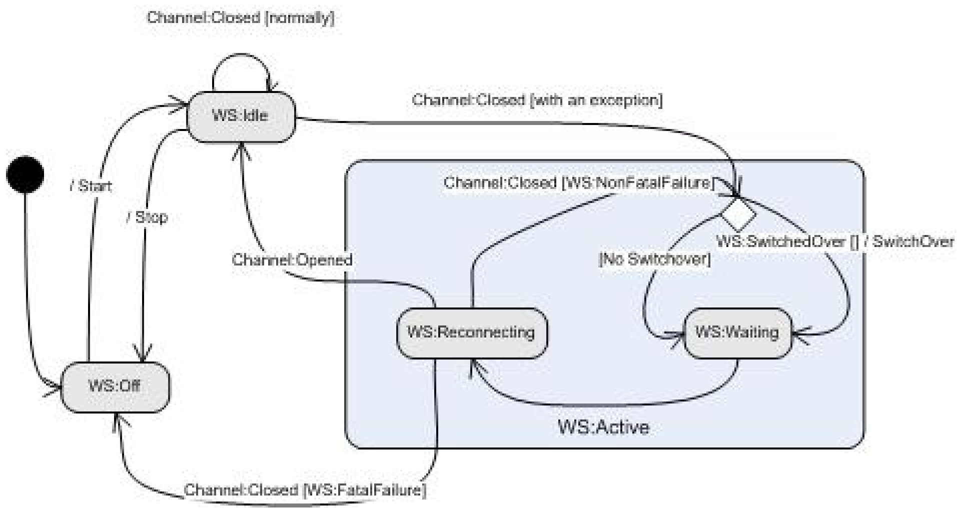
- Do nothing (close the channel by request of the user application)
- Attempt to open the channel without switching over its connectivity configuration from primary to backup
- Attempt to open the channel, switching its connectivity configuration from primary to backup
- Deactivate, in case of a fatal failure

Any application block activity will be followed by a corresponding event generated by the application block. These events will provide user applications with the opportunity to monitor and react to all of the application block's activities and failures

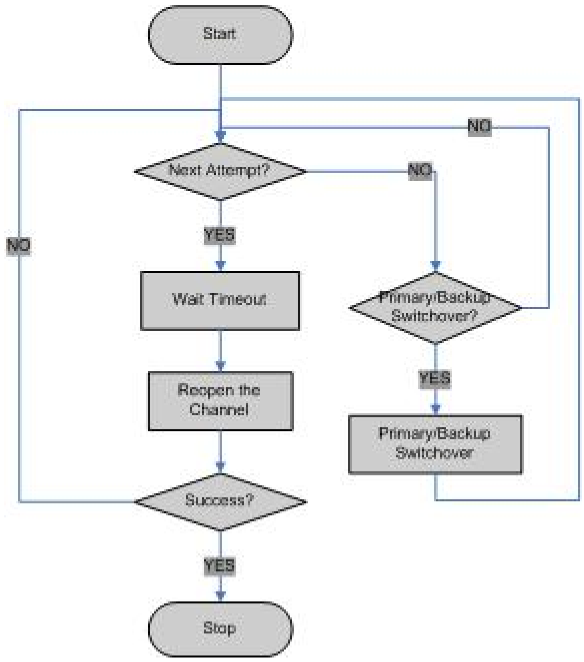
To control channel connectivity with a warm standby mechanism, the user application should activate the Warm Standby Application Block instance that is responsible for handling the particular channel's connectivity failure and recovery.

Warm Standby Application Block Algorithm

The Warm Standby Application Block has 4 states, as shown below.


Channel:Closed [normally]
WS:Idle
Channel:Closed [with an exception]
/ Start
/ Stop
Channel:Closed [WS:NonFatalFailure]
WS:SwitchedOver [] / SwitchOver
[No Switchover]
Channel:Opened
WS:Reconnecting
WS:Waiting
WS:Off
WS:Active
Channel:Closed [WS:FatalFailure]

As soon as a channel's Warm Standby Application Block is activated, it goes into the idle state, waiting for the channel's Closed event. When the channel issues a `Closed` event, the application block checks to see if the channel was closed due to a connectivity failure. If so, the application block instance starts the channel connectivity recovery procedure, as shown below.



Here is the procedure for the Warm Standby Application Block:

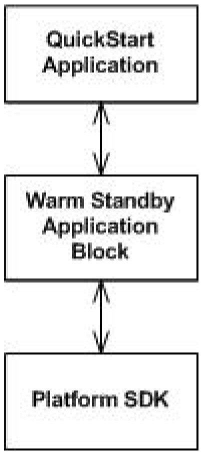
- The user should activate the Warm Standby Application Block for every channel he or she intends to work with.
- In the active state, the application block waits for the channel's `Closed` event.
- On receiving the channel's `Closed` event, the application block activates the channel connectivity recovery procedure.

Application Block Components

The Warm Standby Application Block distribution consists of two main components:

1. The application block itself, which provides an interface that you can use to integrate it into different GUI applications.
2. A sample application, the *WarmStandbyQuickStart* application, which is built on the Warm Standby Application Block

As shown below, the application block itself runs on top of the Platform SDK, while the QuickStart application runs on top of the application block.



The Warm Standby Application Block Interface

The Warm Standby Application Block consists of the following classes:

- `WarmStandbyService`
- `WarmStandbyConfiguration`

The `WarmStandbyService` class monitors and controls the connectivity of the channel it is responsible for, while the `WarmStandbyConfiguration` class handles all the parameters that are needed for the proper functioning of the warm standby process.

> **Tip**
>
> Starting with release 8.1.1, default behavior for the `WarmStandbyService` connection restoration includes the following improvements to provide improved performance:
>
> - Following a switchover or the first reconnection attempt, `WarmStandbyService` no longer waits for a timeout to occur.
> - Check backup server availability by performing a fast first switchover.

> **Tip**
>
> Starting with release 8.1.4, users can also modify the timeout value used during fast reconnect to a backup server if a live connection is terminated.

User applications can subscribe to the controlled channel's `Closed` and `Opened` events in order to monitor and handle channel connectivity.

`WarmStandbyService`'s `StateChanged` event is fired on any change of state in `WarmStandby`, providing the means for a user application to monitor state changes and to control the application block's activities.

Using the Warm Standby Application Block

Before you install the Warm Standby Application Block, it is important to review the software requirements and the structure of the software distribution.

Building the Warm Standby Application Block

To build the Warm Standby Application Block:

1. Open the *<Platform SDK Folder>\applicationblocks\warmstandby* folder.
2. Run either *build.bat* or *build.sh*, depending on your platform.

Tip

You may need to edit the path specified in the quickstart file by adding quotation marks if your ANT_HOME environment variable contains spaces.

This build file will create the *warmstandbyappblock.jar* file, located within the *<Platform SDK Folder>\applicationblocks\warmstandby\dist\lib* directory.

Now you are ready to add the appropriate import statements to your source code and start using the Warm Standby Application Block:

```
[Java]
import com.genesyslab.platform.applicationblocks.warmstandby.*;
```

Using the QuickStart Application

The easiest way to start using the Warm Standby Application Block is to use the bundled QuickStart application. This application ships in the same folder as the application block.

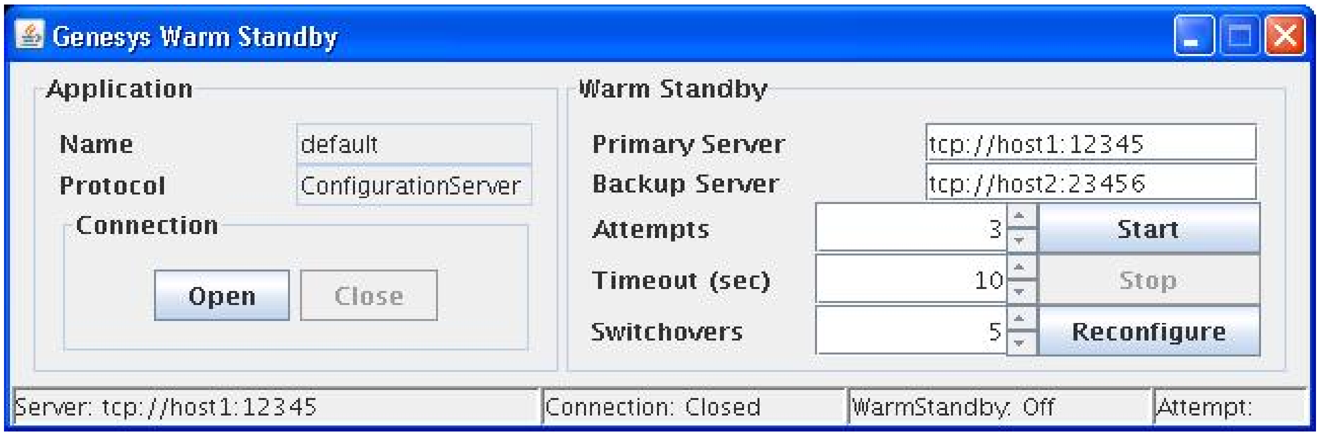
To run the QuickStart application:

1. Open the *\ApplicationBlocks\WarmStandby\quickstart* folder.
2. Run either *quickstart.bat* or *quickstart.sh*, depending on your platform.

Important

You may need to edit the path specified in the quickstart file by adding quotation marks if your ANT_HOME environment variable contains spaces.2

After you start the application, you will see the user interface shown below.



Name	default
Protocol	ConfigurationServer
Primary Server	tcp://host1:12345
Backup Server	tcp://host2:23456
Attempts	3
Timeout (sec)	10
Switchovers	5

```
Server: tcp://host1:12345 | Connection: Closed | WarmStandby: Off | Attempt:
```

On startup, the QuickStart application uses values specified by the *quickstart.properties* configuration file. You can change these values either by editing that file or by overwriting them after running the user interface.

This form has two main sections. The left side enables you to set up a connection for the application indicated in the Name field, using the protocol specified in the Protocol field. To open the connection, press the *Open* button. Press the *Close* button to close it.

The right side of the form lets you specify primary and backup servers. It also lets you specify the number of times the warm standby mechanism will try to contact the primary server, and what the timeout value should be for each attempt.

Once you have the desired values, you can press the *Start* button to turn on the warm standby feature. If you would like to change the configuration after warm standby is turned on, simply modify the configuration information and press the Reconfigure button. The warm standby configuration will be changed dynamically.

.NET

Architecture and Design

Many contact center environments require redundant backup servers that are able to take over quickly if a primary server fails. In this situation, the primary server operates in active mode, accepting connections and exchanging messages with clients. The backup server, on the other hand, is in standby mode. If the primary server fails, the backup server switches to active mode, assuming the role and behavior of the primary server.

There are two standby modes: *warm standby* and *hot standby*. The main difference between them is that warm standby mode does not ensure the continuation of interactions in progress when a failure occurs, while hot standby mode does.

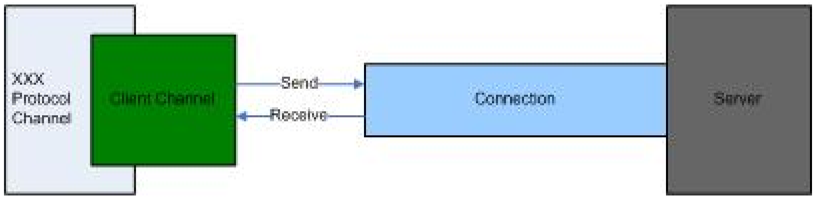
The Client Channel Architecture

Since the Warm Standby Application Block is designed to be used in the context of a Client Channel architecture, it is important to understand that architecture before talking about the application block itself.

To start with, this architecture consists of three functional components:

- A connection
- A client channel
- A protocol channel

These components are shown in the following figure.



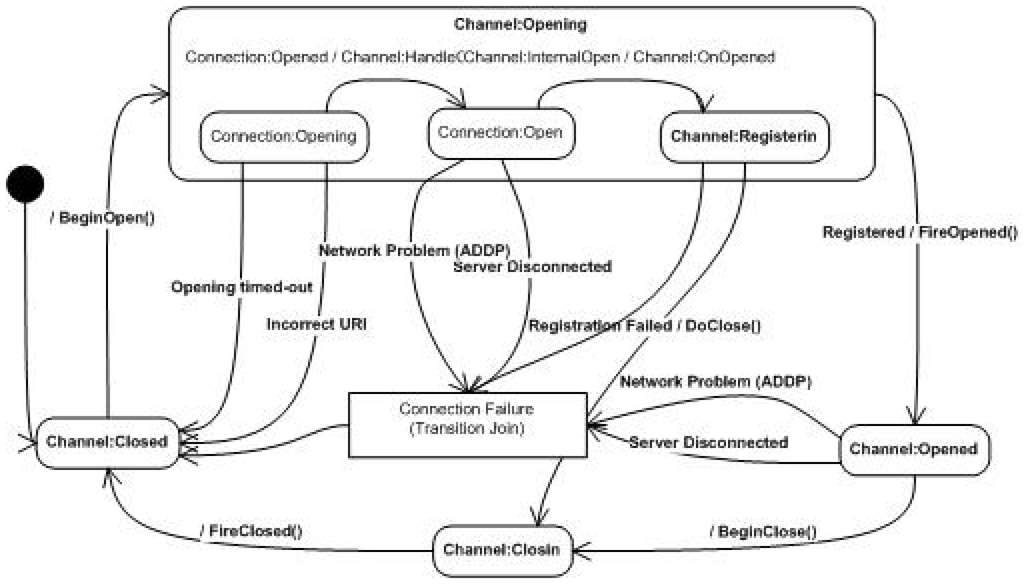
The *connection* controls all necessary TCP/IP connection activities, while the client channel contains the protocol- and server-independent channel functionality that is common for a protocol channel. Finally, the *protocol channel* controls all of the client channel activities that are dependent on the protocol and the server.

Client Channel State

The state of a client channel is based on the state of the corresponding connection. There are four major states:

- Opening (Registration)
- Opened
- Closing
- Closed

The figure below shows a detailed client channel state diagram.



In addition to establishing a TCP/IP connection, several activities may take place when a client channel opens. These activities can include things like:

- A preliminary exchange of messages with the server, which is known as registration
- Reading the client channel’s locally stored configuration information

You can often determine the cause of a client channel failure by checking the state of the client channel just before it closed. There are exceptions to this rule, however, such as a registration failure, which is protocol-specific.

Client Channel Failure Scenarios

There are several common client channel failure scenarios:

Client Channel Failure Scenarios

Scenario	Description	Source States	Condition	Target State	Protocol-Dependent
Opening Timed Out	Channel tries to open connection to non-existing URI	Opening	Connection opening timeout	Closed	No
Wrong URI	Channel tries to open connection to non-existing URI	Opening	Incorrect URI exception	Closed	No
Connection Problem	Channel connection detects a connection	Opened Opening	Server disconnected	Closed	No

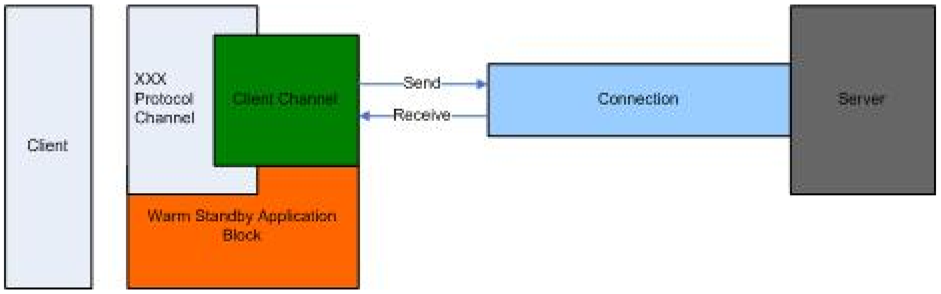
Scenario	Description	Source States	Condition	Target State	Protocol-Dependent
	problem				
Network Problem (ADDP)	Channel connection detects a network problem (ADDP)	Opened Opening	Network problem (ADDP)	Closed	No
Wrong Server or Protocol	Channel tries to open connection with an incorrect server or protocol	Opening	Registration Failed/ ProtocolException	Closing	Yes
Registration Failure	One of the channel registration steps failed	Opening	Registration Failed/ ProtocolException	Closing	Yes

Note that the first four scenarios, *Opening timed-out, Wrong URI, Connection Problem,* and *Network Problem* happen with the connection (TCP/IP) component. They do not involve protocol- or server-specific elements, whether in terms of failure-specific data or in terms of channel recovery actions and data.

The *Wrong Server* or *Protocol* and *Registration Failure* scenarios are protocol- or server-dependent and can be different for each type of protocol channel.

Application Block Architecture

The Warm Standby Application Block's functionality is based on intercepting the channel's transition from a non-closed state to the Closed state. As you can see in the following figure, the application block is able to pick up this information because it sits between the client and protocol channels.



Upon receiving the channel's `Closed` event, the application block uses diagnostic information to determine why the channel has closed. This diagnostic information is necessary to determine what actions, if any, the application block should take to restore the channel's connectivity to the server.

The Warm Standby Application Block can take several different steps to recover channel connectivity. These steps are:

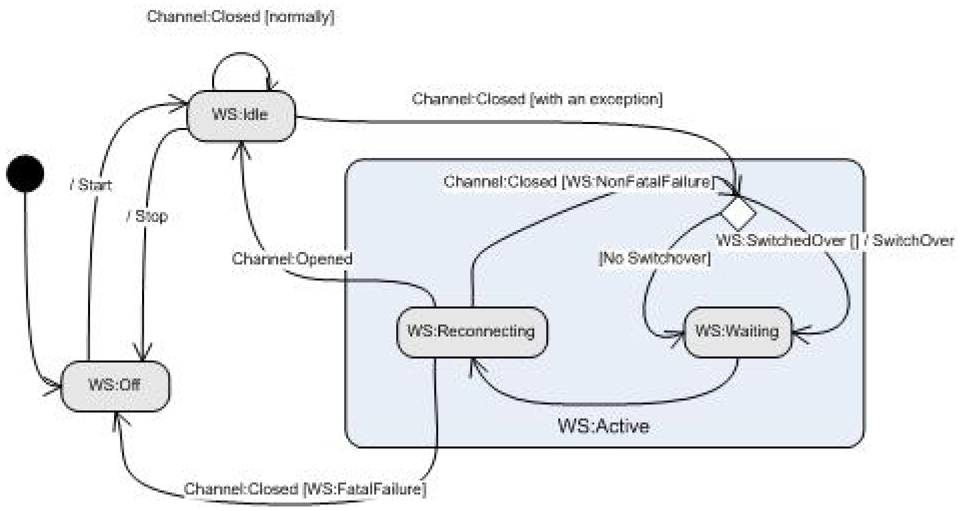
- Do nothing (close the channel by request of the user application)
- Attempt to open the channel without switching over its connectivity configuration from primary to backup
- Attempt to open the channel, switching its connectivity configuration from primary to backup
- Deactivate, in case of a fatal failure

Any application block activity will be followed by a corresponding event generated by the application block. These events will provide user applications with the opportunity to monitor and react to all of the application block's activities and failures

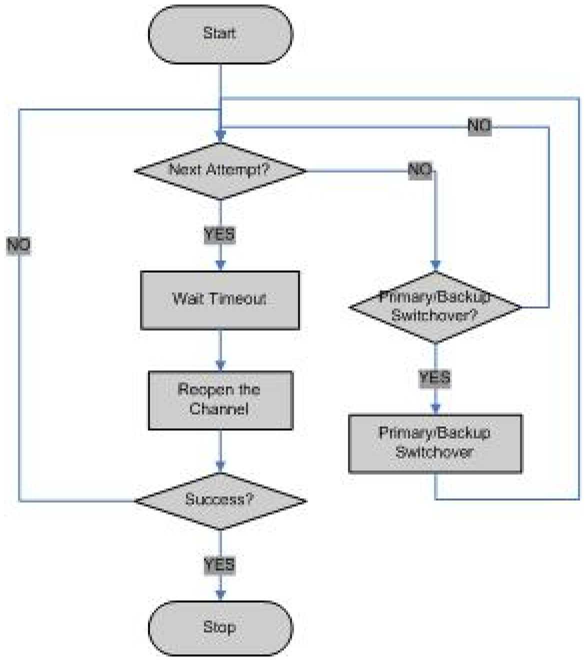
To control channel connectivity with a warm standby mechanism, the user application should activate the Warm Standby Application Block instance that is responsible for handling the particular channel's connectivity failure and recovery.

Warm Standby Application Block Algorithm

The Warm Standby Application Block has 4 states, as shown below.


Channel:Closed [normally]
WS:Idle
Channel:Closed [with an exception]
/ Start
/ Stop
Channel:Closed [WS:NonFatalFailure]
WS:SwitchedOver [] / SwitchOver
[No Switchover]
Channel:Opened
WS:Reconnecting
WS:Waiting
WS:Off
WS:Active
Channel:Closed [WS:FatalFailure]

As soon as a channel's Warm Standby Application Block is activated, it goes into the idle state, waiting for the channel's Closed event. When the channel issues a `Closed` event, the application block checks to see if the channel was closed due to a connectivity failure. If so, the application block instance starts the channel connectivity recovery procedure, as shown below.



Here is the procedure for the Warm Standby Application Block:

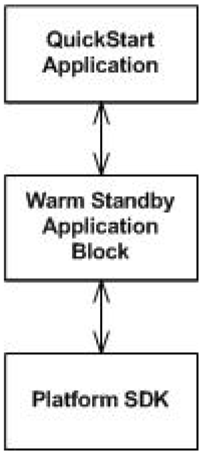
- The user should activate the Warm Standby Application Block for every channel he or she intends to work with.
- In the active state, the application block waits for the channel's `Closed` event.
- On receiving the channel's `Closed` event, the application block activates the channel connectivity recovery procedure.

Application Block Components

The Warm Standby Application Block distribution consists of two main components:

1. The application block itself, which provides an interface that you can use to integrate it into different GUI applications.
2. A sample application, the *WarmStandbyQuickStart* application, which is built on the Warm Standby Application Block

As shown below, the application block itself runs on top of the Platform SDK, while the QuickStart application runs on top of the application block.

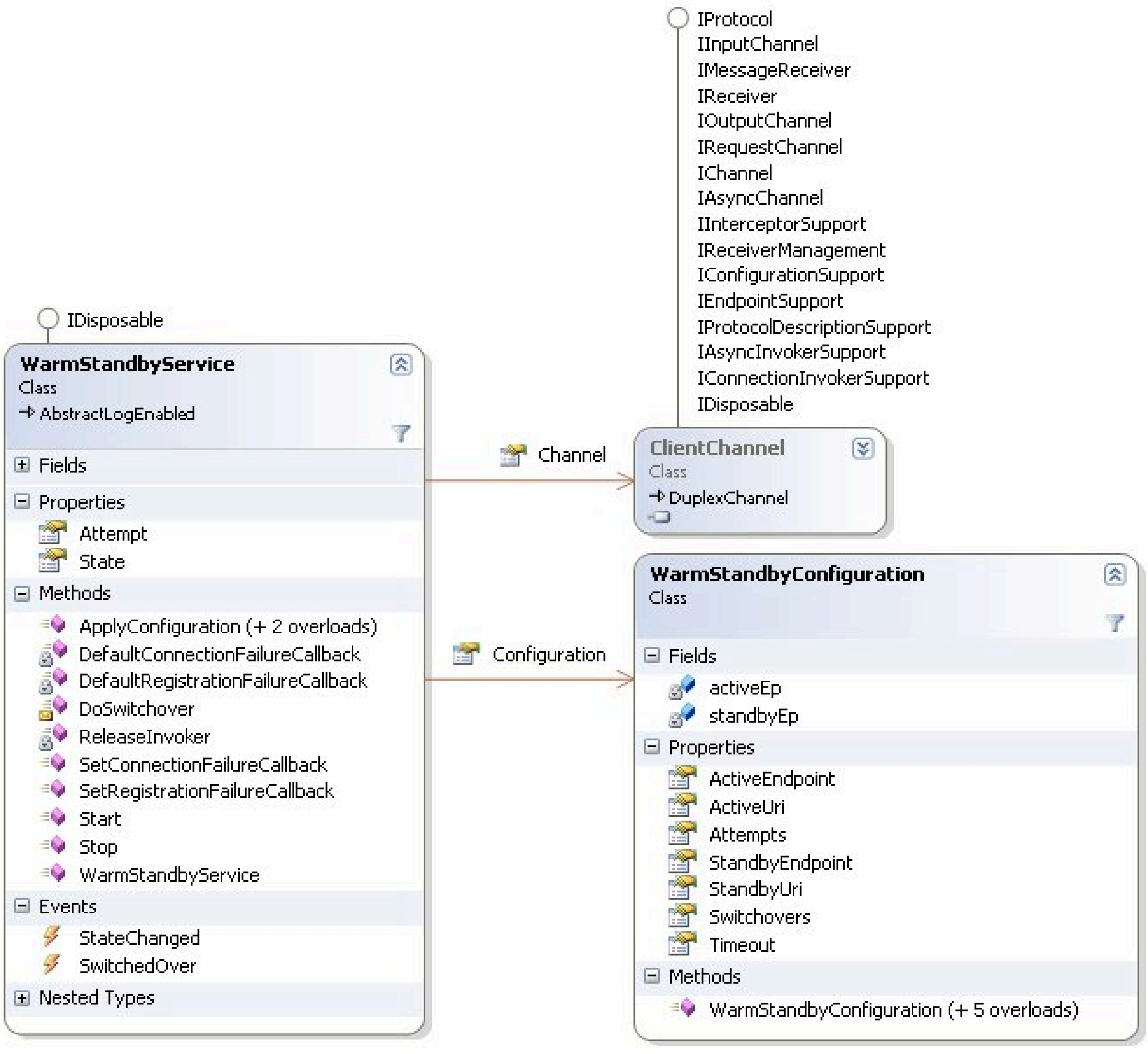


The Warm Standby Application Block Interface

The Warm Standby Application Block consists of the following classes:

- `WarmStandbyService`
- `WarmStandbyConfiguration`

These classes are shown in greater detail below.



The `WarmStandbyService` class monitors and controls the connectivity of the channel it is responsible for, while the `WarmStandbyConfiguration` class handles all the parameters that are needed for the proper functioning of the warm standby process.

Tip

Starting with release 8.1.1, default behavior for the `WarmStandbyService` connection restoration includes the following improvements to provide improved performance:

- Following a switchover or the first reconnection attempt, `WarmStandbyService` no longer waits for a timeout to occur.

- Check backup server availability by performing a fast first switchover.

Tip

Starting with release 8.1.4, users can also modify the timeout value used during fast reconnect to a backup server if a live connection is terminated.

User applications can subscribe to the controlled channel’s `Closed` and `Opened` events in order to monitor and handle channel connectivity.

`WarmStandbyService`'s `StateChanged` event is fired on any change of state in `WarmStandby`, providing the means for a user application to monitor state changes and to control the application block's activities.

Using the Warm Standby Application Block

Before you install the Warm Standby Application Block, it is important to review the software requirements and the structure of the software distribution.

Configuring the Warm Standby Application Block

In order to use the QuickStart application, you need to set up the XML configuration file that comes with the application block. This file is located at *Quickstart\app.config*. This is what the contents look like:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
        <configSections>

        </configSections>
        <WarmStandbyQuickStart>
                <Channel
                        ClientType="19"
                        ProtocolName="ConfigurationServer"
                        ClientName="default"
                />
                <WarmStandby
                        PrimaryServer="tcp://hostname:9999"
                        BackupServer="tcp://hostname:9999"
                        Attempts="3"
                        Timeout="10"
                        Switchovers="3"
                />
                <ConfServer
                        UserName="default"
                        UserPassword="password"
                />
        </WarmStandbyQuickStart>

</configuration>
```

Follow the instructions in the comments and save the file.

Building the Warm Standby Application Block

The Platform SDK distribution includes a *Genesyslab.Platform.ApplicationBlocks.WarmStandby.dll* file that you can use as is. This file is located in the bin directory at the root level of the Platform SDK directory. To build your own copy of this application block, follow the instructions below:

To build the Warm Standby Application Block:

1. Open the *<Platform SDK Folder>\ApplicationBlocks\WarmStandby* folder.
2. Double-click *WarmStandby.sln*.
3. Build the solution.

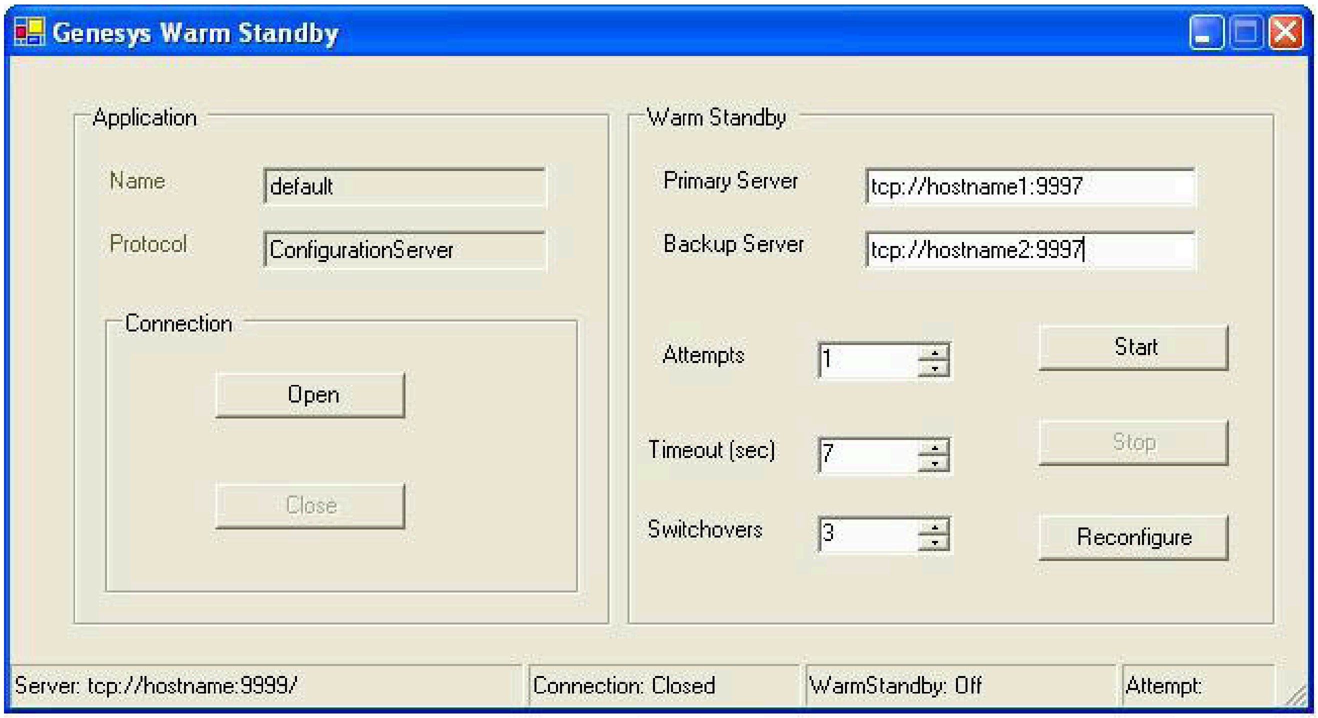
Using the QuickStart Application

The easiest way to start using the Warm Standby Application Block is to use the bundled QuickStart application. This application ships in the same folder as the application block.

To run the QuickStart application:

1. Open the *<Platform SDK Folder>\ApplicationBlocks\WarmStandby* folder.
2. Double-click *WarmStandbyQuickStart.sln*.
3. Build the solution.
4. Find the executable for the QuickStart application, which will be at *<Platform SDK Folder>\ApplicationBlocks\WarmStandby\QuickStart\bin\Debug\WarmStandbyQuickStart.exe*.
5. Double-click *WarmStandbyQuickStart.exe*.

After you start the application, you will see the user interface shown below.



Name	default
Protocol	ConfigurationServer
Primary Server	tcp://hostname1:9997
Backup Server	tcp://hostname2:9997
Attempts	1
Timeout (sec)	7
Switchovers	3

Server: tcp://hostname:9999/ | Connection: Closed | WarmStandby: Off | Attempt:

This form has two main sections. The left side enables you to set up a connection for the application indicated in the *Name* field, using the protocol specified in the *Protocol* field. To open the connection, press the *Open* button. Press the *Close* button to close it.

The right side of the form lets you specify primary and backup servers. It also lets you specify the number of times the warm standby mechanism will try to contact the primary server, and what the timeout value should be for each attempt. On startup, these values are picked up from the configuration file, but you can change them in the user interface.

Once you have the desired values, you can press the *Start* button to turn on the warm standby feature. If you would like to change the configuration after warm standby is turned on, simply modify the configuration information and press the Reconfigure button. The warm standby configuration will be changed dynamically.