Platform SDK Developer's Guide

Connecting to a Server

4/21/2025

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);
```