Platform SDK Developer's Guide

Platform SDK 8.1.2

12/29/2021

Table of Contents

Welcome to the Developer's Guide!

This document introduces you to the tools and examples provided to help you get started with Platform SDK development.

Developer articles for the Platform SDK are divided broadly into the following categories:

- Introductory and Feature-Specific Topics describe general topics and common SDK functionality that all Platform SDK developers should be familiar with.
- Application Blocks provide production-ready blocks of code you should leverage, and modify if necessary, when creating applications.
- Server-Specific SDK Protocols work directly with Genesys servers using message-based requests and events.
- Library Components offer additional features and functionality such as logging.

For additional information about the Platform SDKs, please check the introductory materials provided as part of the Platform SDK API Reference for your release.

Introductory and Feature-Specific Topics

- Lazy Parsing of Message Attributes
- Platform SDK Implementation of TLS
- Setting Up Logging in Platform SDK

Server-Specific SDK Protocols

Platform SDK Protocol	Genesys Server(s)	Related Documentation
Configuration Platform SDK	Configuration Server	
Contacts Platform SDK	Universal Contact Server	
Management Platform SDK	Message Server Solution Control Server Local Control Agent	LCA Hang-Up Detection Support
Open Media Platform SDK	Interaction Server	
Outbound Contact Platform SDK	Outbound Contact Server	
Routing Platform SDK	Universal Routing Server Custom Server	
Statistics Platform SDK	Stat Server	
Voice Platform SDK	T-Servers	
Web Media Platform SDK	Chat Server	

Platform SDK Protocol	Genesys Server(s)	Related Documentation
	E-Mail Server Java Callback Server	

Library Components

Library Component	Releated Documentation
Platform SDK Log Library	
Platform SDK Switch Policy Library	Using the Switch Policy Library

Application Blocks

Application Block	Related Documentation
Application Template Application Block (Java)	
Configuration Object Model Application Block	
Message Broker Application Block	
Protocol Manager Application Block	
Warm Standby Application Block	Using the Warm Standby Application Block

New Content by Release

This section provides a quick outline of developer content based on the release where that information first became relevant, or where it where last updated.

Release 8.1.1 New Features:

- Platform SDK Implementation of TLS
- Platform SDK for Java JAR files now provide a valid OSGi manifest and can be used as bundles in OSGi containers (like Equinox and Karaf) without additional modifications. The manifest exposes public Platform SDK APIs so that OSGi consumer services can use the same variety of Platform SDK functionality as non-OSGi clients.

Release 8.1.0

New Features:

- Lazy Parsing of Message Attributes
- LCA Hang-Up Detection Support

Release 8.0

- Please refer to developer information provided as part of the introductory material in the Platform SDK

API Reference for this release.

Additional Resources

The following page contains reference materials that may be useful when developing applications with Platform SDK.

- Platform SDK Resources

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.
- By default, the lazy parsing feature is turned off.

System Requirements

Platform SDK for .NET:

- Configuration SDK protocol release 8.1 or later<ref name="ConfObject">**Note:** Currently, lazy parsing is only used with the `EventObjectsRead.ConfObject` property of the Configuration Platform SDK.</ref>
- .NET Framework 3.5
- Visual Studio 2008 (required for .NET project files)

Platform SDK for Java:

- Configuration SDK protocol release 8.1 or later<ref name="ConfObject"/>
- J2SE 5.0 or Java 6 SE runtime

<references/>

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. For illustration purposes, .NET code snippets are provided.

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).

Note: 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:

```
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 (as shown below):

```
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` function — 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>".

Platform SDK Implementation of TLS

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

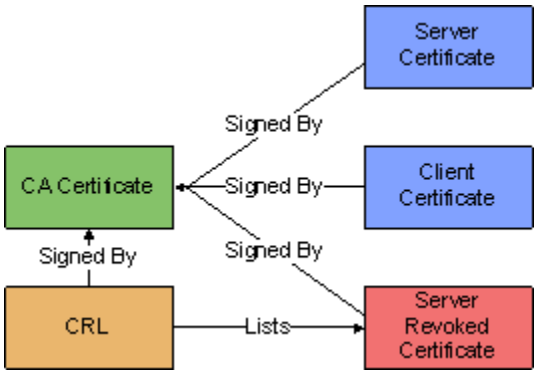
Transport Layer Security (TLS) technology uses public key cryptography, where the key required to encrypt and decrypt information is divided into two parts: a public key and a private key. These parts are reciprocal in the sense that data encrypted using a private key can be decrypted with the public 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:



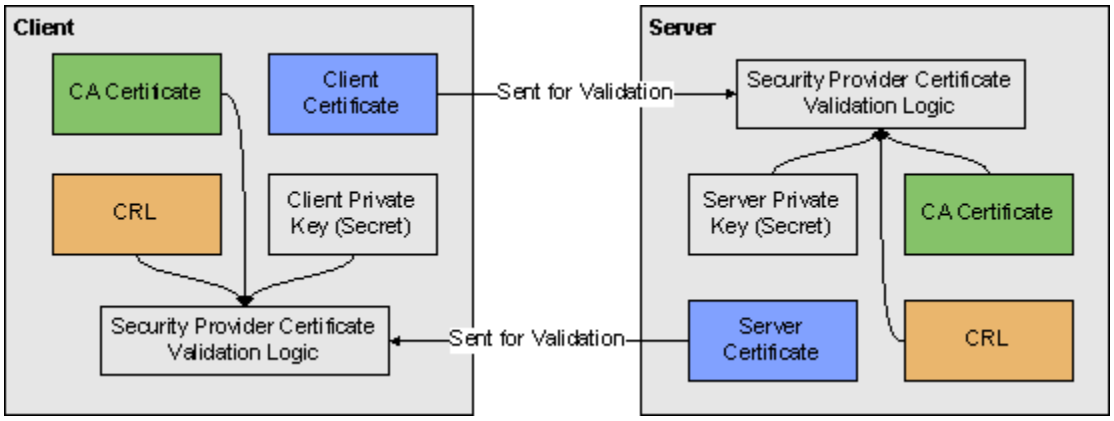
Certificate Usage

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:



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 Previous Versions of Platform SDK

Platform SDK for Java

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$$w0rd", "pa$$w0rd");
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", 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.

Platform SDK for .Net

There were no significant changes to interfaces for the .NET version of Platform SDK 8.1.1. In this

case, the same code would work for 8.1.0 and 8.1.1 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);
```

Known Issues

For more details about the known issues listed here, refer to Using and Configuring Security Providers.

- Java 5: MSCAPI provider is not supported.
- Java 6:
  - MSCAPI provider is only supported in 32-bit version since update 27: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6931562.
  - MSCAPI provider is only supported in 64-bit version since update 38: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=2215540.
  - CRLs located in WCS are ignored, please use CRLs as files.
- 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 of programmatic setting of password to private key stored in WCS. Regardless of password returned by CallbackHandler; if private key is protected with confirmation prompt or password prompt, user will be shown OS popup dialog.

Quick Start

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.

Platform SDK for Java

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

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, Host, Port, 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.

Using the Platform SDK Commons Library

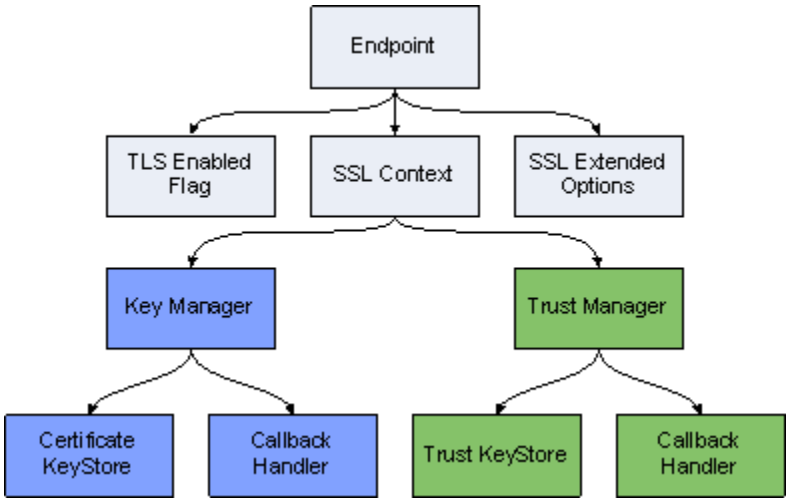
Using the Platform SDK Commons Library to Configure TLS

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.

Note: 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 e-mail 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. `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	

Methods

TLSConfiguration Methods List

Signature	Description
SSLContext createSslContext()	A shortcut for `TLSConfigurationHelper.createSslContext`

Signature	Description
	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)	

Signature	Description
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”.
```

Using the Application Template Application Block

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.

Note: 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:

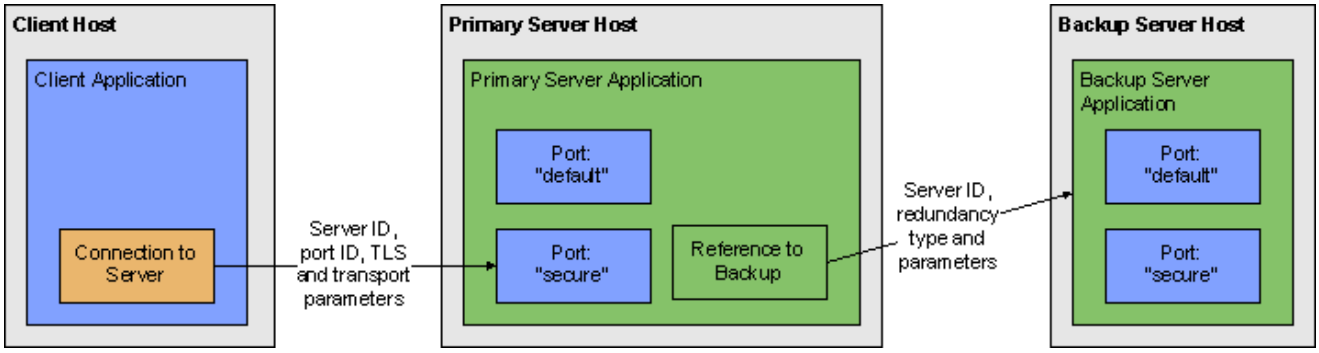
```
String serverAppName = "<my-app-name>";
String portID = "secure";
CfgApplication cfgApplication = confService.retrieveObject(
        CfgApplication.class, new CfgApplicationQuery(appName));
GCOMApplicationConfiguration appConfig =
        new GCOMApplicationConfiguration(cfgApplication);
IGApplicationConfiguration.IGPortInfo 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);
...
```

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.

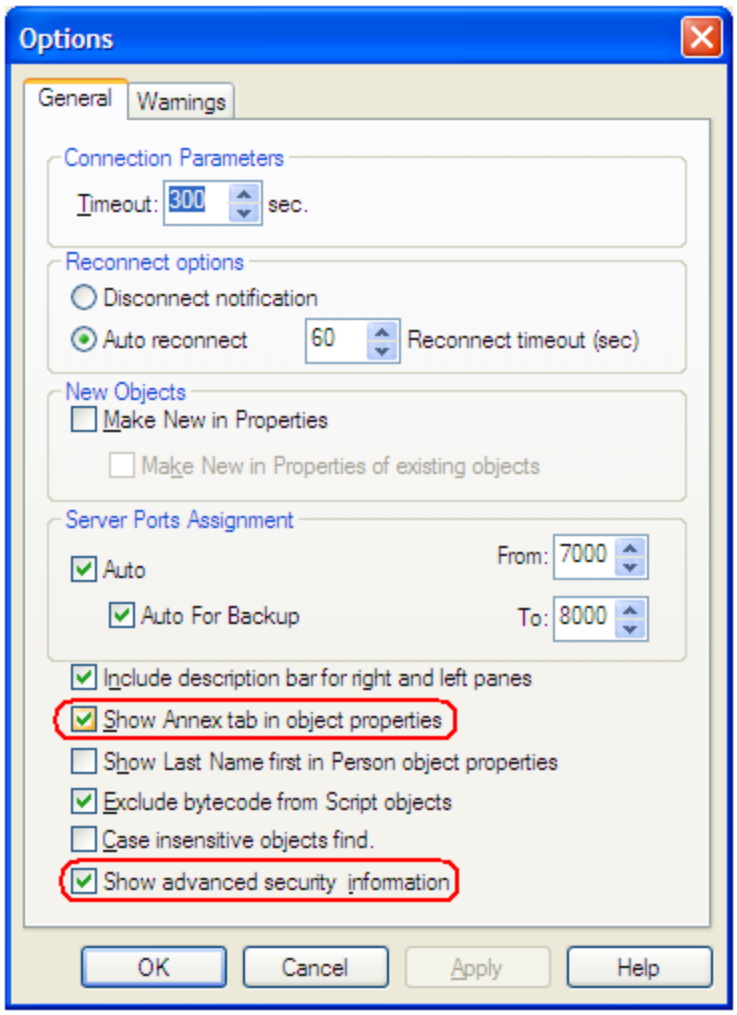
Application type	Configuration Objects Used, in Order of Precedence
	2. Application of the client. 3. Host where client application resides. 4. Port of the target server that client connects to.[1]
Server	1. Port of the server application. 2. Application of the server. 3. Host where the server application 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.



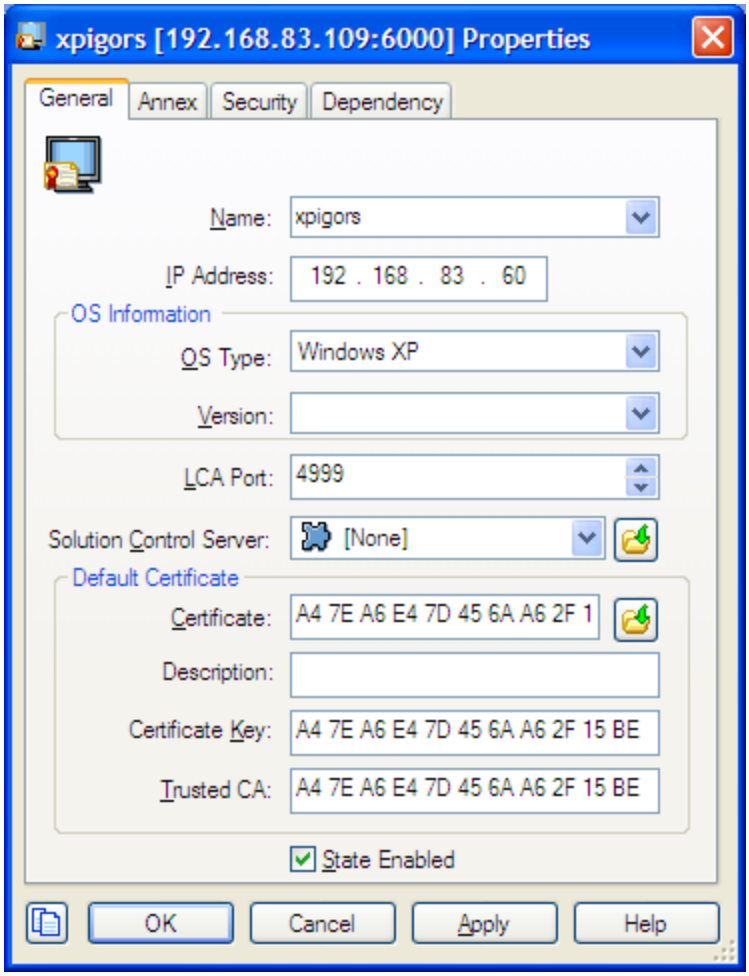
Application Objects

Host Object

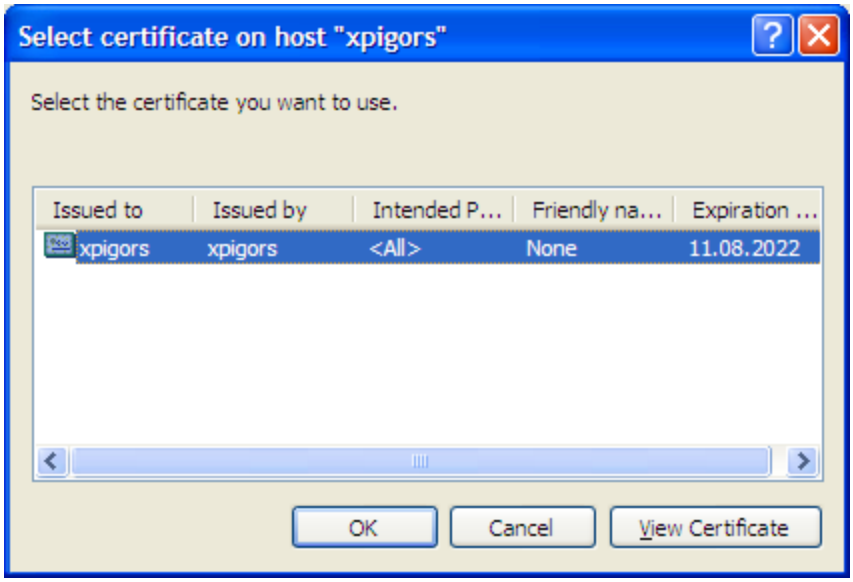
The properties window for a Host object includes most common TLS parameters on the General tab:

- Certificate
- Certificate Key
- Trusted CA

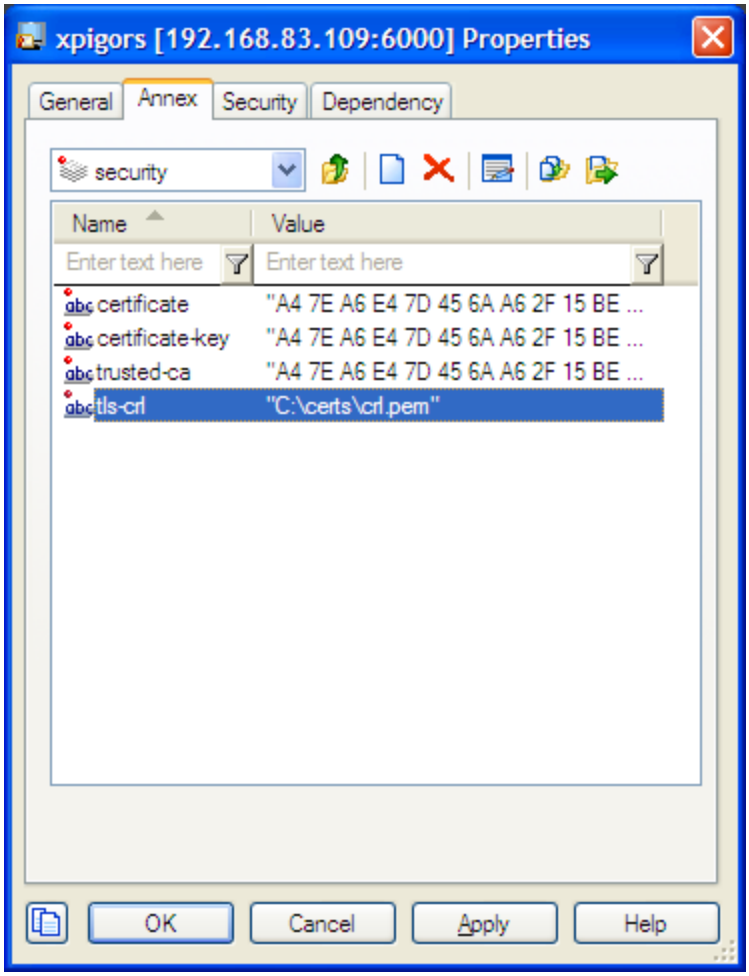
These fields allow copy/paste operations, so they can be set manually by copying and pasting the "Thumbprint" field values from certificates in Windows Certificate Services (WCS) into the related field in Configuration Manager.



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.

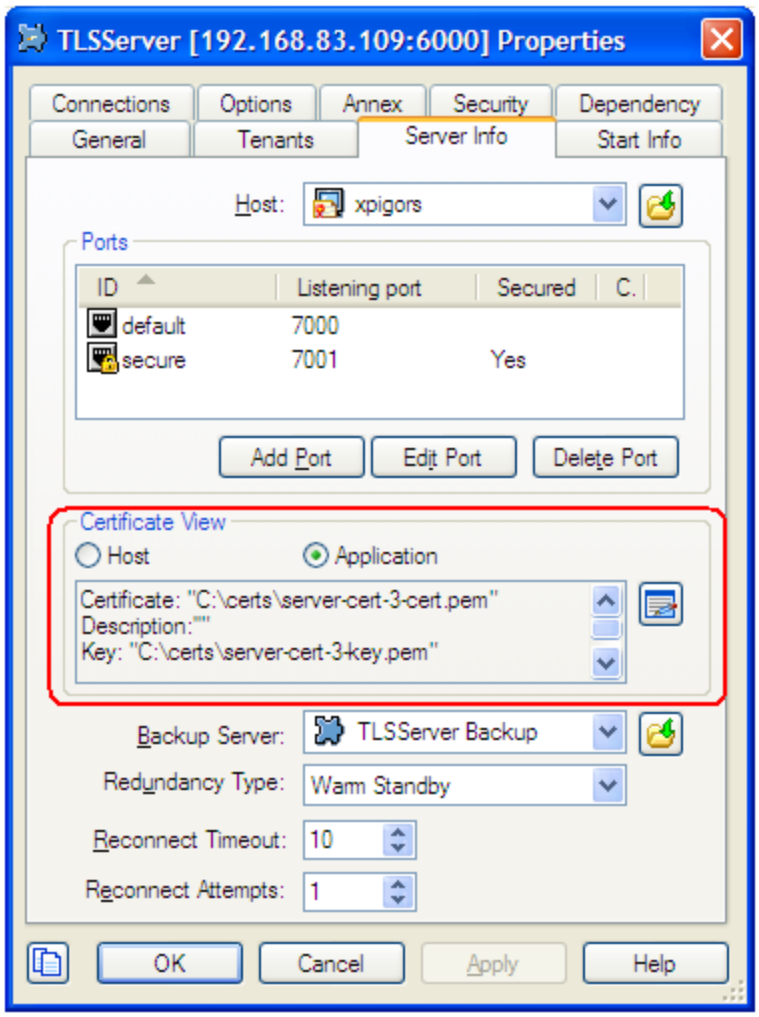


The Annex tab contains a security section that holds TLS settings for this object. Any change made to TLS-related fields on the General tab are mirrored between the Annex tab automatically. You can also specify additional TLS parameters here that aren't reflected on the General tab.

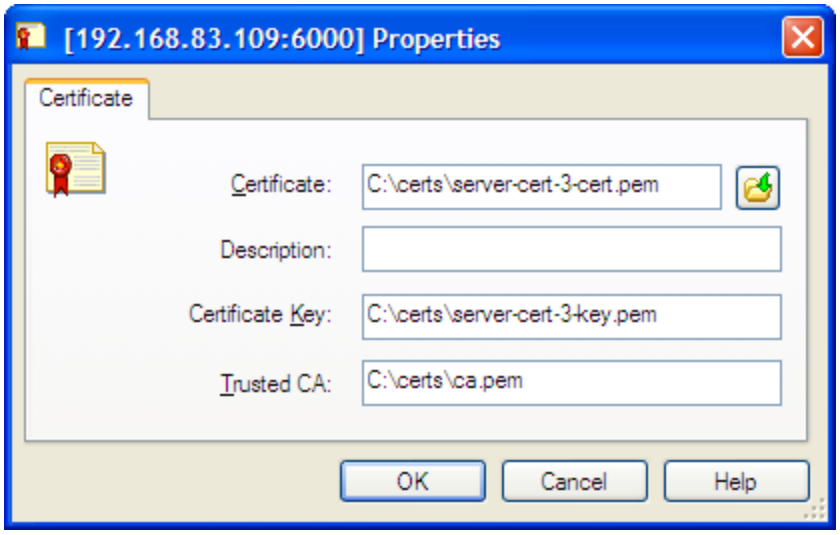


Server Application Object

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.

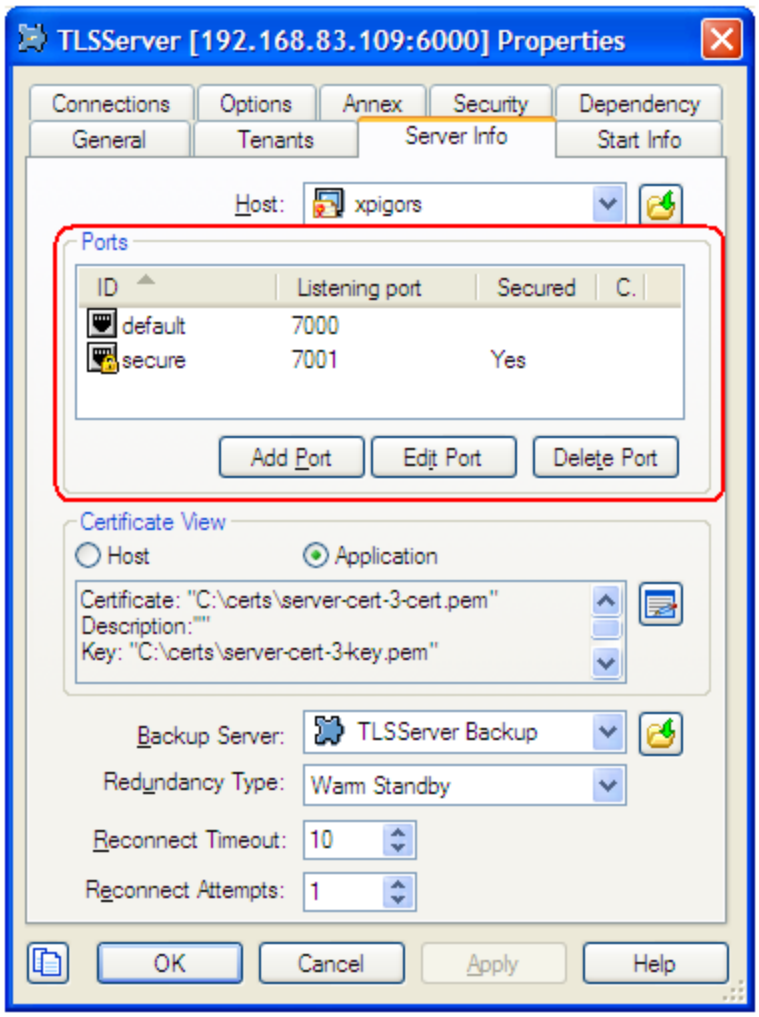


If using application-specific TLS parameters, use the button next to the certificate information field to open a certificate selection window where you can choose from a list of certificates installed for the Local Computer account or manually enter certificate information:



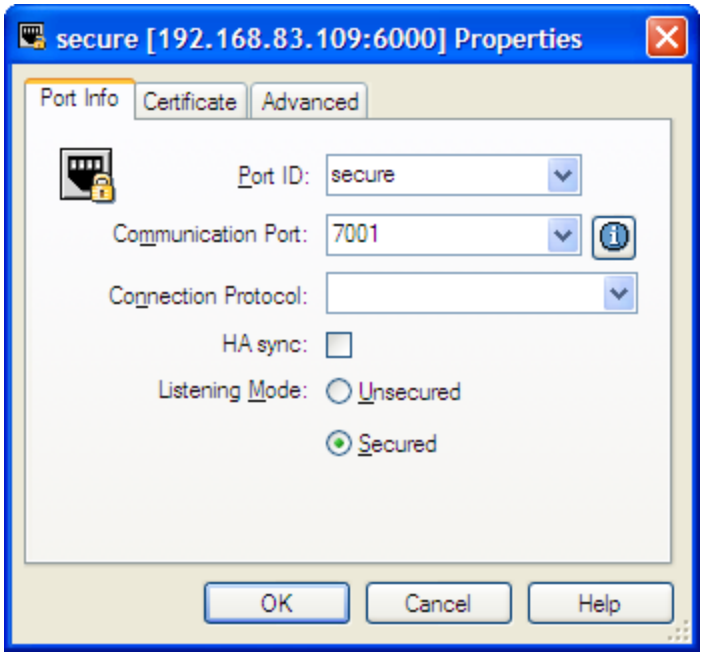
Port Object

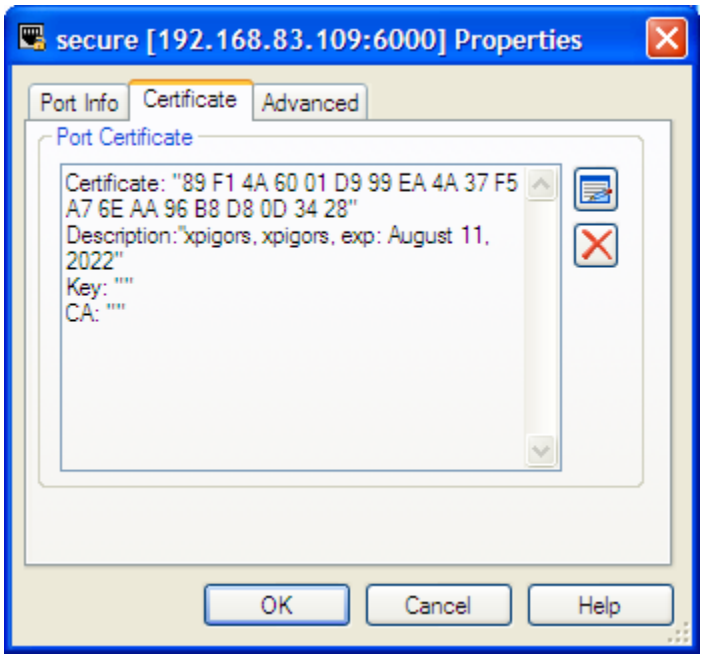
For port objects, TLS-related fields are located on the *Server Info* tab of the properties window. You can see here whether a port is secured (TLS-enabled) or not, and have the option to edit existing ports to update TLS parameters or to add new ports.

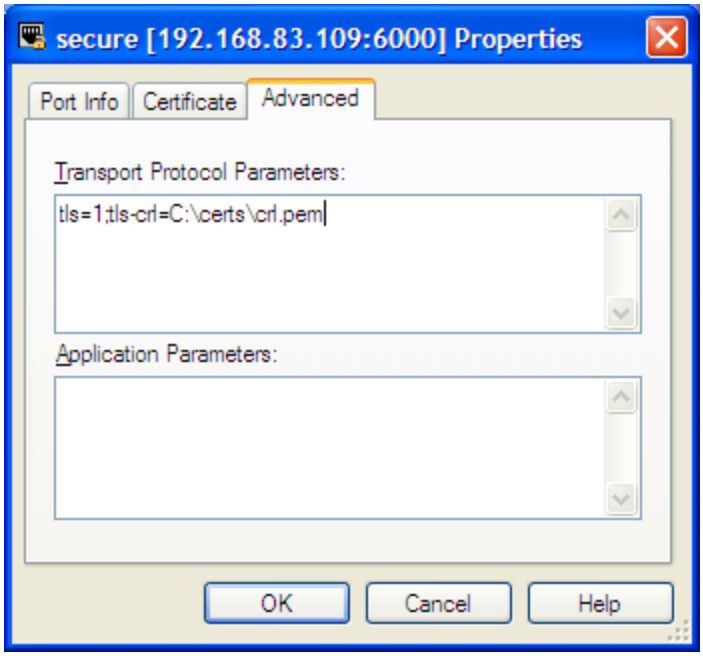


When adding or editing a port, TLS parameters are specified on the following tabs:

- *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.

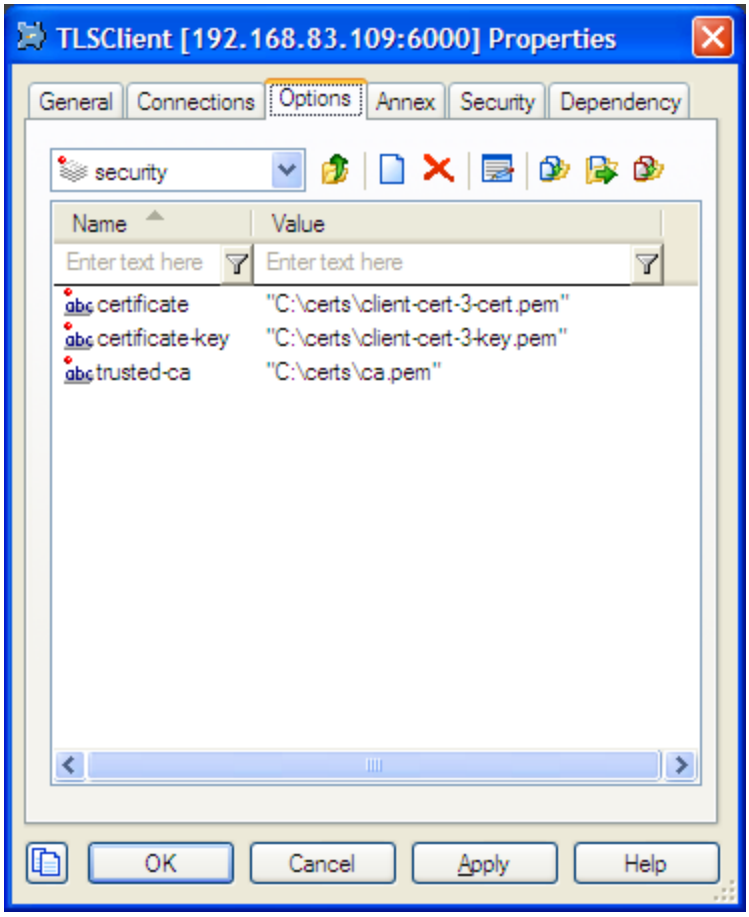

secure [192.168.83.109:6000] Properties
Port Info
Certificate
Advanced
Port ID:
secure
Communication Port:
7001
Connection Protocol:
HA sync:
Listening Mode:
Unsecured
Secured
OK
Cancel
Help

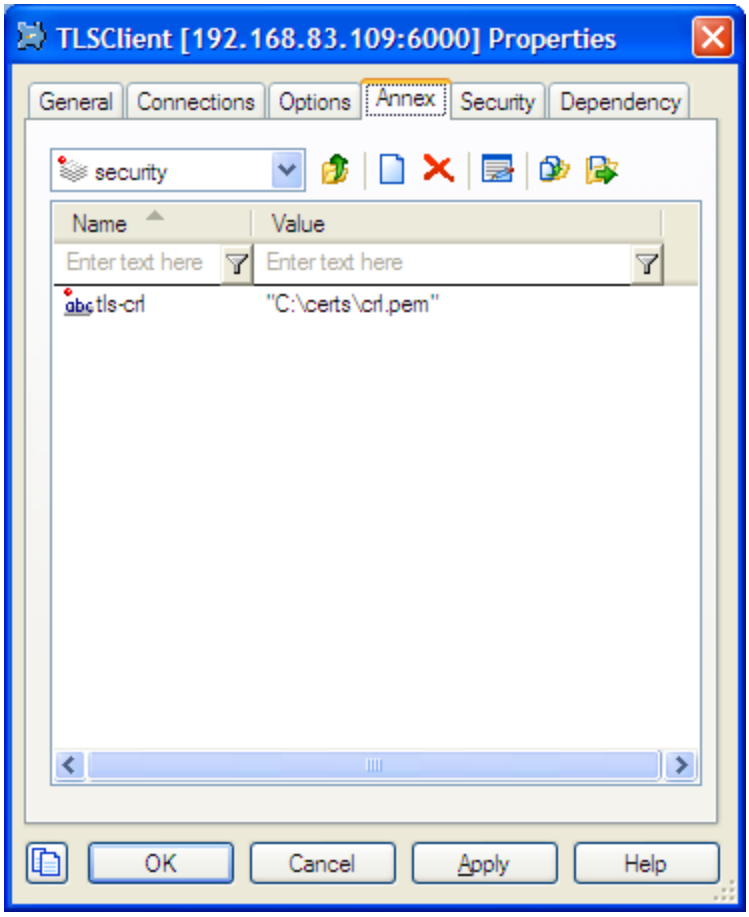

secure [192.168.83.109:6000] Properties
Port Info
Certificate
Advanced
Port Certificate
Certificate: "89 F1 4A 60 01 D9 99 EA 4A 37 F5 A7 6E AA 96 B8 D8 0D 34 28"
Description:"xpigors, xpigors, exp: August 11, 2022"
Key: ""
CA: ""
OK
Cancel
Help



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.

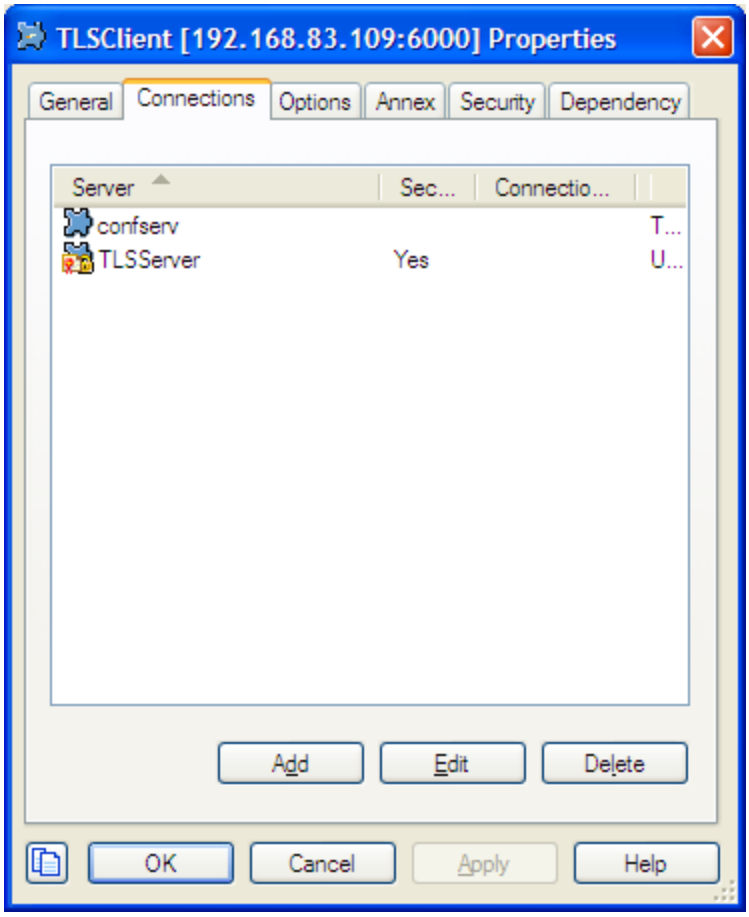

TLSClient [192.168.83.109:6000] Properties
General
Connections
Options
Annex
Security
Dependency
security
Name
Value
Enter text here
Enter text here
certificate
"C:\certs\client-cert-3-cert.pem"
certificate-key
"C:\certs\client-cert-3-key.pem"
trusted-ca
"C:\certs\ca.pem"
OK
Cancel
Apply
Help

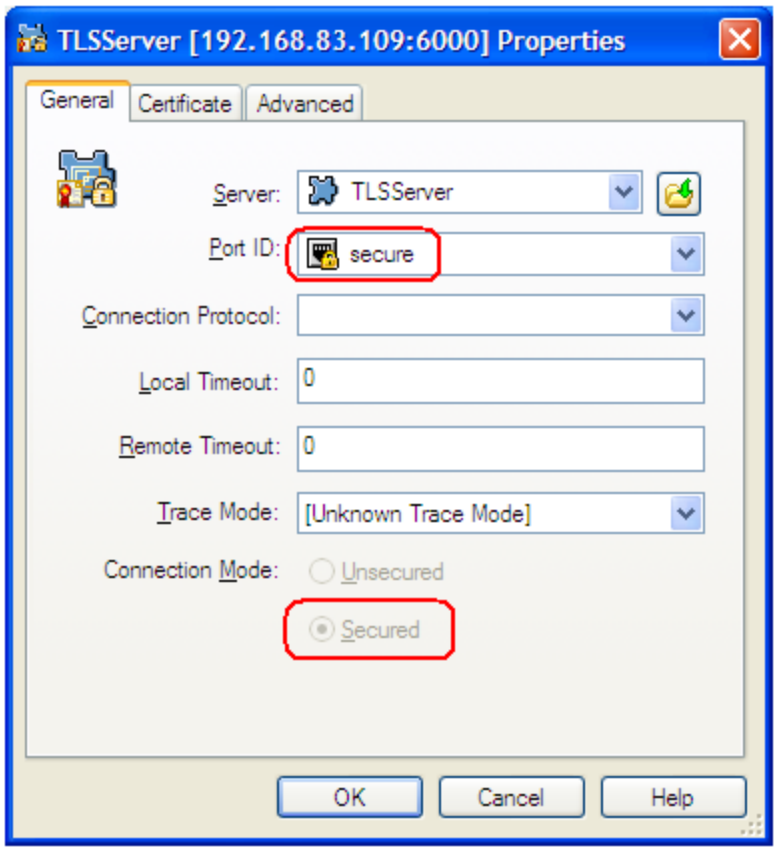


When processing a client Application object, Platform SDK looks at parameters from both sections. If any parameters are specified in both places, then the values from the *Options* tab take precedence.

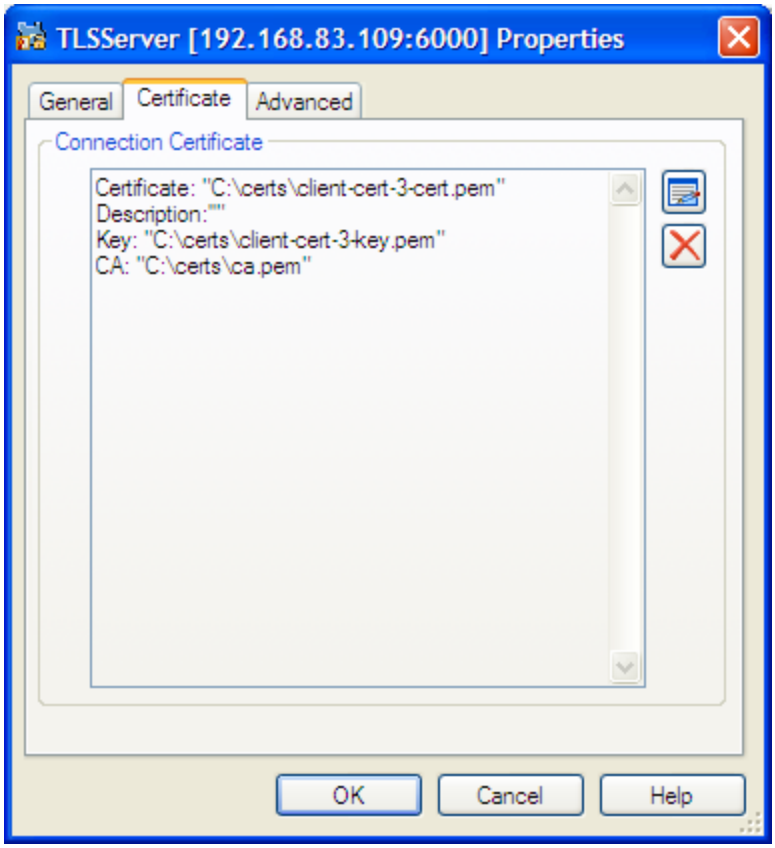
Connection Object

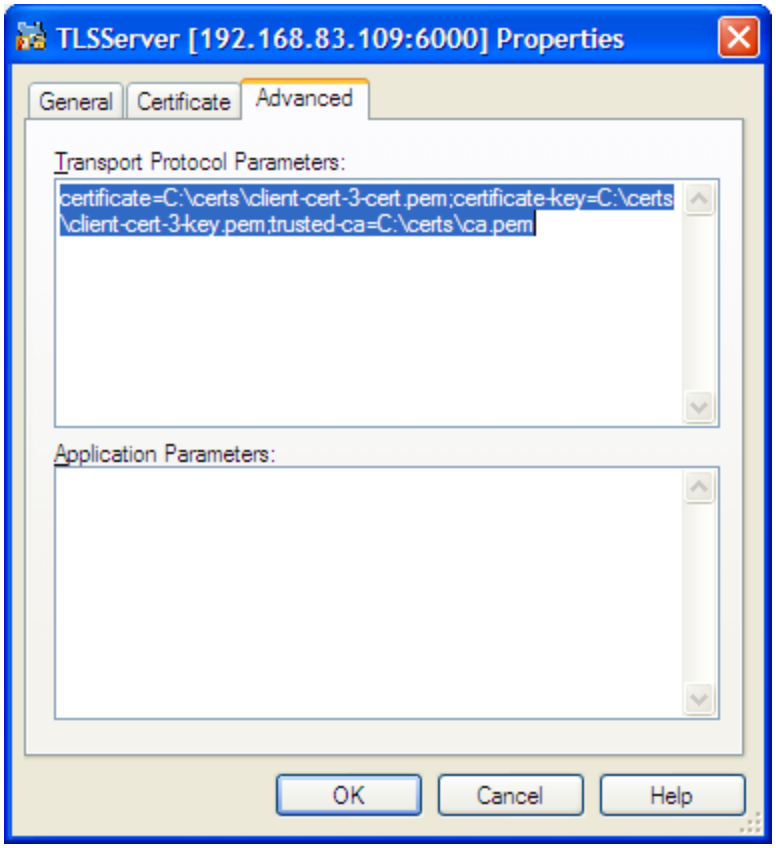
The properties window for all Application objects includes a *Connection* tab where connections to servers can be added or edited. Each connection determines if TLS mode should be enabled based on port settings for the target server.


TLSClient [192.168.83.109:6000] Properties
General
Connections
Options
Annex
Security
Dependency
Server
Sec...
Connectio...
confserv
T...
TLSServer
Yes
U...
Add
Edit
Delete
OK
Cancel
Apply
Help



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.


TLSServer [192.168.83.109:6000] Properties
General
Certificate
Advanced
Connection Certificate
Certificate: "C:\certs\client-cert-3-cert.pem"
Description:""
Key: "C:\certs\client-cert-3-key.pem"
CA: "C:\certs\ca.pem"
OK
Cancel
Help



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
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.
provider	"PEM", "MSCAPI", "PKCS11" Not case-sensitive. Example: • "provider=MSCAPI"	Explicit selection of security provider to be used. For example, MSCAPI and PKCS11 providers can contain all other parameters in their internal database. This parameter allow configuration of TLS through security provider tools.

Parameter Name	Acceptable Values	Purpose
certificate	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: • "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"	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.
certificate-key	PEM provider: path to a PKCS#8 private key file without password protection in PEM format. Path can use both forward and backward slash characters. • MSCAPI provider: this parameter is ignored; key is taken from the entry identified by "certificate" field. • PKCS11 provider: this parameter is ignored. Examples: • "certificate-key= C:\certs\client-cert-3-key.pem"	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 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	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.

Parameter Name	Acceptable Values	Purpose
	45 6A A6 2F 15 BE 89 FD 46 F0 EE 82 1A 58 B9"	
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.
tls-crl	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"	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.
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.
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.
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 provider type, PSDK will throw exception.

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

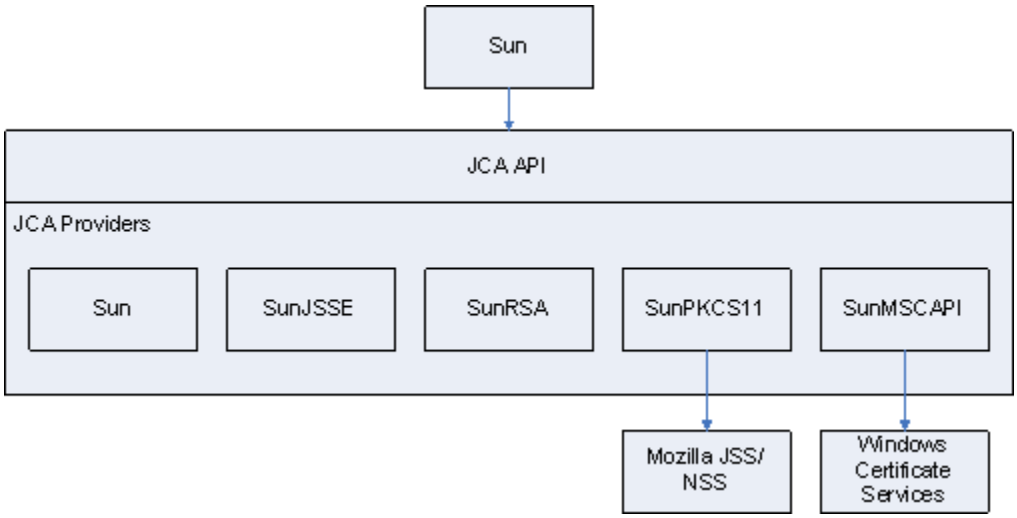
Introduction

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 e-mail 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
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.

File or Folder	Generated by OpenSSL?	Description
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	1. `openssl genrsa -des3 -out ca\private\cakey.pem 1024 -passin file:ca-password.txt`

Task	Description	Command
	3. Export CA Certificate	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`

Revoke a Certificate		`openssl ca -revoke <certificate-pem-file> -config openssl.cfg -passin file:ca-password.txt`
Export the CRL		`openssl ca -gencrl -crldays`

Task	Description	Command
		`<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. The minimum required version for correct MSCAPI support is Java 6 update 38, with additional compatibility details outlined below:
  - Java 5 and lower versions—MSCAPI is not supported.
  - Java 6 32-bit version—MSCAPI provider is only supported since update 27: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6931562.
  - Java 6 64-bit version—MSCAPI provider is only supported since update 38: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=2215540.
  - Java 7—MSCAPI is supported in all versions.
- 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.

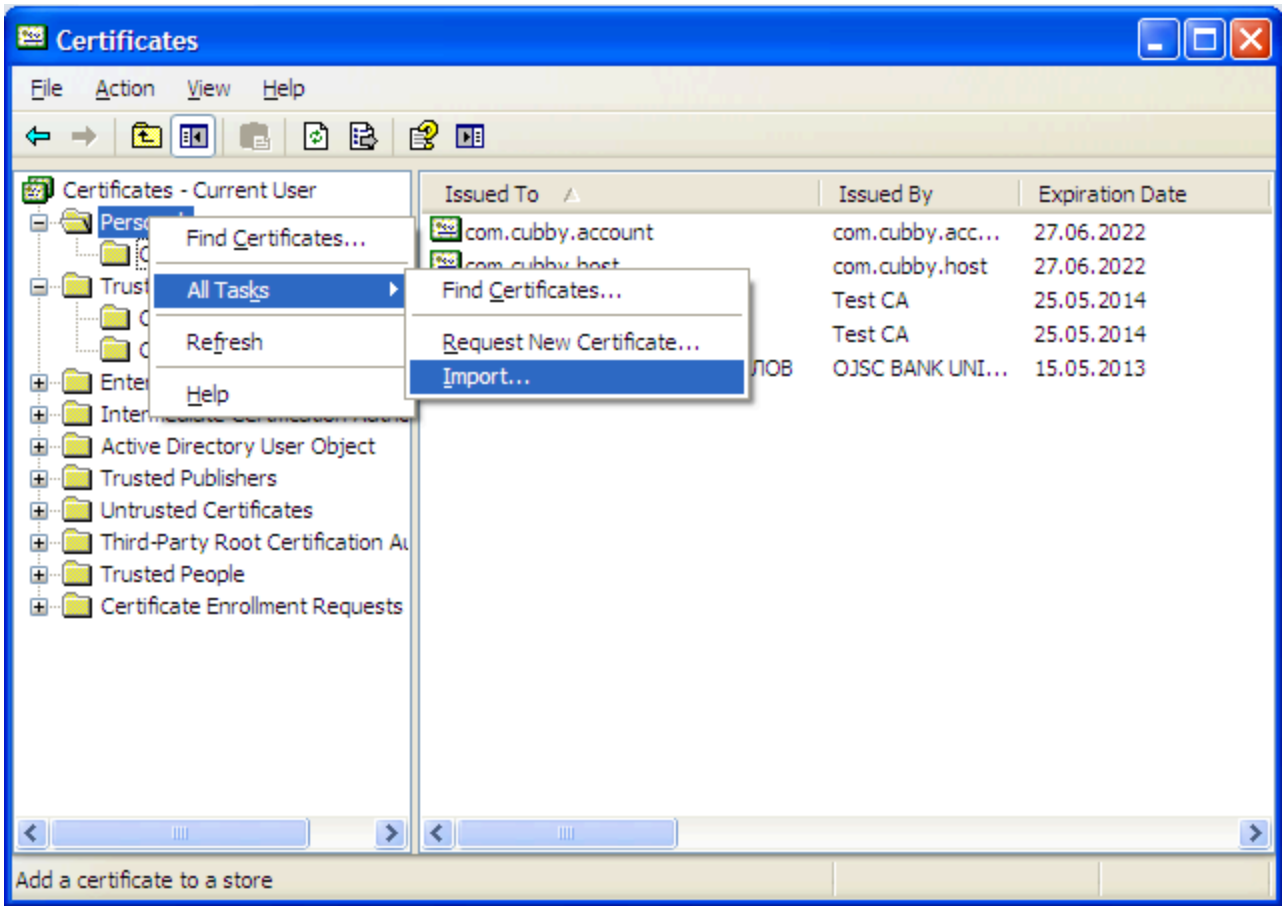
Note: Configuration Manager is also a regular application in this aspect and can access WCS only for the Local Computer (System) account on the local machine. It will not show certificates configured for different accounts or on remote machines. Please consult your system and/or security administrator 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 on 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.



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 is available from 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 references for more information:

- https://www.mozilla.org/projects/security/pki/nss/tools/certutil.html
- https://www.mozilla.org/projects/security/pki/nss/tools/crlutil.html
- https://www.mozilla.org/projects/security/pki/nss/tools/pk12util.html

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 reference for more information:

- http://docs.oracle.com/javase/1.5.0/docs/tooldocs/solaris/keytool.html

Task	Command
Creating and Importing - These commands allow you to generate a new Java Keytool keystore file, create a Certificate Signing Request (CSR), and import certificates. Any root or intermediate certificates	

Task	Command
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        = default                 # use public key default MD
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 (e.g. 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.

Setting up logging in Platform SDK

Logging for Java

Setting up log4j logging

The easiest way to set up Platform SDK logging in Java is to use the built-in integration with log4j. There are two possible ways to do this:

- Using code, by creating a `Log4JLoggerFactoryImpl` instance and setting it as the global logger factory for Platform SDK at the beginning of your program, like this:

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

Or:

- 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=com.genesyslab.platform.commons.log.Log4JLoggerFactoryImpl
<MyMainClass>
```

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.

Logging for .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.

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.

Note: 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.

System Requirements

Genesys Management Layer:

- Release 8.0.1 or later

Platform SDK for .NET:

- Management SDK protocol release 8.1 or later
- .NET Framework 3.5
- Visual Studio 2008 (required for .NET project files)

Platform SDK for Java:

- Management SDK protocol release 8.1 or later

- J2SE 5.0 or Java 6 SE runtime

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.

> **NOTE:** 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

Note: 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<ref>MSDN PerformanceCounter Class (http://msdn.microsoft.com/en-us/library/system.diagnostics.performancecounter.aspx)</ref> for details about performance counters.

Note: 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<ref>JMX: Java Management Extensions (http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/)</ref> 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();
```

Note: 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.

References

<references />

Using the Switch Policy Library

This document shows how to add simple T-Server functionality to your applications by using the Switch Policy Library.

The Platform SDK Switch Policy Library (SPL) can be used in applications that need to perform agent-related switch activity with a variety of T-Servers, without knowing beforehand what kinds of T-Servers will be used. It simplifies these applications by indicating which switch functions are available at any given time and also by showing how you can use certain switch features in your applications. However, if your application works with only one kind of T-Server, you may want to have your application communicate directly with the T-Server, rather than using SPL.

Switch Policy Library Overview

Some telephony applications need to work with more than one type of switch. Unfortunately, however, one switch may not perform a particular telephony function in the same way as another switch. This means that it can be useful to have an abstraction layer of some kind when working with multiple switches, so that you do not need custom code for each switch that is used by the application. The Switch Policy Library is designed with just this kind of abstraction in mind.

Setting Up Switch Policy Library

SPL should be used by your agent desktop applications as a library, which means that it would be located within the agent desktop application shown above. The application can call SPL for guidance on how to send requests to or process events from your T-Server, as shown in the Code Samples section.

SPL is driven by an XML-based configuration file that supports many commonly-used switches in performing agent-related functions. Your application can query SPL to determine whether a particular feature is supported for the switch you want to work with. If a feature you need is not supported for the switches you need to work with, you can make a copy of the default configuration file and modify it as needed.

Note: Genesys does not support modifications to the SPL configuration file. Any modifications you make are performed at your own risk.

A copy of the default configuration file is included inside the Switch Policy Library DLL. There is also a copy in the Bin directory of the Platform SDK installation package. If you need to modify the configuration file, you can use the app.config file for SPL to point to your copy.

Code Samples

This section contains examples of how to perform useful functions with SPL.

The functions discussed here are all contained in a compilable and runnable sample application that is available on the Downloads page of the Genesys Documentation Wiki. This site also hosts the SPL IsPossible Feature Demo application. This sample application lets you specify a switch and certain characteristics of the main and secondary parties to a call, as well DN state information. Once you have done this, it will show you which functions are available for that switch, based on the characteristics you have specified. This application can be very helpful in understanding the kinds of things that are available to your application when you use SPL.

These samples each require a valid instance of the ISwitchPolicyService, which can be created as shown here:

```
ISwitchPolicyService policyService =
        SwitchPolicyFactory.CreateSwitchPolicyService();
```

Note: The DN classes specified below implement the IDNContext interface, while the Party classes implement the IPartyContext interface, and the Call classes implement the ICallContext interface.

Get A Phone Set Configuration

On some switches, phone sets are presented as more than one Directory Number (DN). These DNs may also have different types, such as Position and Extension. Because these configurations vary by switch type, an application needs to know how the phone set configuration for a particular switch is structured. For example, it needs to know how many DNs are used to represent a phone set, and what their types are. To retrieve this phone set configuration information, perform the following steps:

1. Create an instance of PhoneSetConfigurationContext, specifying the switch type.
2. Call ISwitchPolicyService.GetPolicy, using this PhoneSetConfigurationContext.
3. Analyze the returned PhoneSetConfigurationPolicy. The PhoneSetConfigurationPolicy.Configurations property will contain all possible phone set configurations for the specified switch.

The following code snippet shows how to do this:

```
PhoneSetConfigurationContext context =
        new PhoneSetConfigurationContext("SomeSwitch");
PhoneSetConfigurationPolicy policy =
switchPolicyService.GetPolicy<PhoneSetConfigurationPolicy>(context);
foreach (PhoneSetConfiguration configuration in policy.Configurations)
{
        Console.WriteLine(configuration);
}
```

Get Phone Set Availability Information

When working with a phone set, additional information about the included DNs may be required. This could include information about which of the DNs should be available to the end user (for example, which ones should be visible in the user interface), which of them is callable, and which number (the Callable Number) the application should use to reach the agent who is logged into the phone set. To

retrieve this phone set availability information, perform the following steps:

1. Create an instance of DNAvailabilityContext and populate it with the following required information:
   - Specify the switch type
   - Specify the Agent ID
   - Fill the DN collection with valid implementations of IDNContext
2. Call ISwitchPolicyService.GetPolicy, using this DNAvailabilityContext.
3. Analyze the returned DNAvailabilityPolicy. The DNAvailabilityPolicy.DNStatuses property will contain availability information for each DN in the request.

The following code snippet shows how to do this:

```
private static void DemonstrateDNAvailability(ISwitchPolicyService service)
{
        DNAvailabilityContext dnacontext =
                new DNAvailabilityContext("SomeSwitch");
        dnacontext.AgentId = "AgentLogin1000";
        dnacontext.DNs.Add(new Dn
        {
                AgentStatus = AgentStatus.Ready,
                        Identifier = "1000",
                        ServiceStatus = ServiceStatus.InService,
                        Type = AddressType.DN
        });
        dnacontext.DNs.Add(new Dn
        {
                AgentStatus = AgentStatus.Ready,
                        Identifier = "2000",
                        ServiceStatus = ServiceStatus.InService,
                        Type = AddressType.Position
        });

        DNAvailabilityPolicy dnpolicy =
                service.GetPolicy<DNAvailabilityPolicy>(dnacontext);
        DisplayInColor(dnpolicy, ConsoleColor.Red);
}
```

Get Function Availability Information for the Current Context

Some switches differ in when they allow certain functions to be performed. Also, some functions can always be performed on certain switches, while others may be impossible to perform. For example, RequestMergeCalls can never be performed on some switches. For other functions, whether or not the function can be performed varies depending on context. For example, on some switches RequestReleaseCall can only be used when a call is in a Held, Dialing, or Established state, while on other switches it is also possible to release a call when it is in a Ringing state. In addition to this, on some switches the phone set is presented as more than one Directory Number (DN) and each DN can have a different type, such as Position and Extension. Some functions are allowed for both types, while some other functions may be restricted to a certain DN type. To retrieve this kind of function availability information for the current context, perform the following steps:

1. Create an instance of FunctionHandlingContext and populate it with the following required information:
   - Specify the switch type

- Specify the request by setting the Message property
- Describe the context as fully as possible

2. Call ISwitchPolicyService.GetPolicy, using this FunctionHandlingContext.
3. Analyze the returned FunctionAvailabilityPolicy. If the specified request is possible in the given context, the IsFunctionAvailable property will be true. However, if the request is not supported, SPL will return null.

The following code snippet shows how to do this:

```
foreach (string switchType in new[] { swTypeA4400Classic, swTypeA4400emul, swTypeA4400Subs })
{
        DNContext dn = new DNContext //implements IDNContext
        {
                Identifier = "1001",
                Type = AddressType.DN,
                AgentStatus = AgentStatus.Ready,
                ServiceStatus = ServiceStatus.InService,
                DndStatus = FunctionStatus.Off,
                ForwardStatus = FunctionStatus.Off
        };

        DNContext otherDN = new DNContext
        {
                Identifier = "2001",
                Type = AddressType.DN,
                AgentStatus = AgentStatus.Ready,
                ServiceStatus = ServiceStatus.InService,
                DndStatus = FunctionStatus.Off,
                ForwardStatus = FunctionStatus.Off
        };

        foreach (CallType callType in Enum.GetValues(typeof(CallType)))
        {
                PartyContext mainParty = new PartyContext //implements IPartyContext
                {
                        Identifier = "1002",
                        Status = PartyStatus.Established,
                        CallType = callType,
                        IsConferencing = true,
                        IsTransferring = true,
                        DN = dn
                };

                PartyContext otherParty = new PartyContext
                {
                        Identifier = "1002",
                        CallType = callType,
                        DN = otherDN,
                        IsConferencing = true,
                        IsTransferring = true,
                        Status = PartyStatus.Established
                };
                CallContextStub ccontext = new CallContextStub //implements ICallContext
                {
                        CallType = callType,
                        Destination = mainParty,
                        Origination = otherParty,
                        Identifier = "1002",
                        IsConferencing = true,
                        IsTransferring = true,
```

```
                        Parties = new List<IPartyContext>{mainParty,otherParty},
                        Parent = null//no parentCall - our call is solitary call.
                };

                FunctionHandlingContext context = new FunctionHandlingContext(switchType)
                {
                        Message = RequestHoldCall.Create(),
                        DN = dn,
                        Party = mainParty,
                        Call = ccontext
                };
                FunctionAvailabilityPolicy policy =
service.GetPolicy<FunctionAvailabilityPolicy>(context);

                Console.WriteLine(policy);
        }
}
```

Get Instructions On How To Implement a Feature

Some switches differ in how certain features can be accessed. The majority of their features may map directly to individual switch functions, but this is not always so. For example, for some switches it is not possible to log the agent out while the agent is in the ready state. So, the feature which implements agent logout for these switches would require two steps:

1. Make sure the agent is in a NotReady state
2. Log the agent out

SPL implements a feature handler for each feature that it supports. To create and run a feature handler, perform the following steps:

1. Create a new instance of FunctionHandlingContext and populate it with the following required information:
   - Specify the switch type.
   - Specify the request by setting the Message property. This step can be omitted if the feature handler is created by using the featureName parameter in the ISwitchPolicyService.CreateFeatureHandler(String featureName, FunctionHandlingContext context) method.
   - Provide a valid IProtocol instance as the value of the Protocol property.
   - Describe the context as fully as possible.
2. Call the ISwitchPolicyService.CreateFeatureHandler and pass this FunctionHandlingContext, either alone or with the name of the feature.
3. Call the BeginExecute method on the returned handler, passing the same instance of FunctionHandlingContext.
4. The remainder of the processing depends on the implementation, but the general approach is to perform the following actions while the status of the handler is Executing:
   1. Receive event from TServer
   2. Update FunctionHandlingContext based on the received event
   3. Assign the received event to the Message property of your FunctionHandlingContext instance

4. Call the Handle method of IFeatureHandler passing with it the updated FunctionHandlingContext

The following code snippet shows how to do this:

```
private static void LoginReadyAgent(IProtocol protocol,
        ISwitchPolicyService service, string thisdn, string agentID)
{
        FunctionHandlingContext context = new FunctionHandlingContext("SomeSwitch");
        RequestAgentLogin requestAgentLogin = RequestAgentLogin.Create();
        requestAgentLogin.ThisDN = thisdn;
        requestAgentLogin.AgentID = agentID;
        requestAgentLogin.AgentWorkMode = AgentWorkMode.AutoIn;
        context.Message = requestAgentLogin;
        context.Protocol = protocol;

        IFeatureHandler loginHandler = service.CreateFeatureHandler(context);

        if(loginHandler == null)
        {
                protocol.Send(requestAgentLogin);
                // Process the incoming events for the scenario
                return;
        }

        // Processing feature handler
        loginHandler.BeginExecute(context);
        while (loginHandler.Status == FeatureStatus.Executing)
        {
                context.Message = context.Protocol.Receive();
                // Update the context based on the received T-Server event
                loginHandler.Handle(context);
        }
}
```

Get Instructions On How To Accomplish Complex Functionality

Your application may sometimes need access to functionality that depends on the switch type. For example, when an application receives events from the T-Server, the way a given event's fields are used can depend on both the call scenario and the switch type. To retrieve this information, perform the following steps:

1. Create a MessageHandlingContext and populate it with the following required information:
   - Name of switch
   - Name of handler
2. Call ISwitchPolicyService.CreateMessageHandler, pass this context into it, and receive the resulting IMessageHandler.
3. Call the IMessageHandler.Handle method on the received handler.

The following code snippet shows how to do this:

```
private static void DemonstrateMessageHandler(ISwitchPolicyService service)
{
        EventRinging message = EventRinging.Create();
        message.ThirdPartyDN = "12345";
        message.DNIS = "18009870987";
        message.CallType = CallType.Internal;
```

```
        message.OtherDN = "9875";
        MessageHandlingContext context35 =
                new MessageHandlingContext("AlcatelA4400DHS3::Classic")
                { HandlerName = "OtherDN" };
        IMessageHandler handler = service.CreateMessageHandler(context35);
        string res = (string)handler.Handle(message);
        DisplayInColor(res, ConsoleColor.Yellow);
}
```

Add Logging Support

To add logging support, carry out the following steps:

1. Create an instance of IUnityContainer and register an anonymous instance or type mapping for the ILogger interface.
2. Pass the IUnityContainer created during the previous step to the factory method, which creates an instance of ISwitchPolicyService.

The following code snippet shows how to do this:

```
IUnityContainer root = new UnityContainer();
root.RegisterInstance(new ConsoleLogger());
ISwitchPolicyService service =
        SwitchPolicyFactory.CreateSwitchPolicyService(root);
```

SPL also provides the following options:

- Your application can log the topmost messages into a distinct log. To use this option, call the CreateSwitchPolicyService(IUnityContainer container, ILogger logger) method of the SwitchPolicyServiceFactory class. The passed logger (if it is not null) will be used for logging the topmost messages.
- You can configure any switch container to use a specific logger. Objects created by the Unity container (feature handlers, policy providers and so on) can use the container to resolve the ILogger for further logging.

Note: the classes provided by SPL resolve the ILogger (if there is one) at creation time. So, if your application changes the ILogger resolution rule for the root container that was previously passed into the SwitchPolicyService constructor after the corresponding method call, this will not affect:

- Existing instances
- Objects which are created in the container(s), for which special ILogger mapping rule is configured

Supported Functions

As mentioned above, SPL is driven by a configuration file that makes it possible to support a wide variety of switch functions. Table 1 shows the functions that are supported by SPL at installation time, using the default configuration file.

Switch Functions Supported by SPL At Installation Time

Switch Function	Description
DN and Agent Functions	
RequestAgentLogin	Logs in the agent specified by the AgentId parameter to the ACD group specified by the parameter.
RequestAgentLogout	Logs the agent out of the ACD group specified by the Queue parameter.
RequestAgentNotReady	Sets a state in which the agent is not ready to receive calls. The agents telephone set is specified by the DN parameter; the ACD group into which the agent is logged is specified by the Queue parameter.
RequestAgentReady	Sets a state in which the agent is ready to receive calls. The agents phone set is specified by the DN parameter; the ACD group into which the agent is logged is specified by the Queue parameter.
RequestCallForwardCancel	Sets the Forwarding feature to Off for the telephony object that is specified by the DN parameter.
RequestCallForwardSet	Sets the Forwarding feature to On for the telephony object that is specified by the DN parameter.
RequestCancelMonitoring	A request by a supervisor to cancel monitoring the calls delivered to the agent. If this request is successful, T-Server distributes EventMonitoringCancelled to all clients registered on the supervisor's and agent's DNs.
RequestMonitorNextCall	A request by a supervisor to monitor (be automatically conferenced in as a party on) the next call delivered to an agent. Supervisors can request to monitor one subsequent call or all calls until the request is explicitly canceled. If a request is successful, EventMonitoringNextCall is distributed to all clients registered on the supervisor's and agent's DNs. Supervisors start monitoring each call in Mute mode. To speak, they must execute the function
RequestSetDNDOff	Sets the Do-Not-Disturb (DND) feature to Off for the telephony object specified by the DN parameter.
RequestSetDNDOn	Sets the Do-Not-Disturb (DND) feature to On for the telephony object specified by the DN parameter.
RequestSetMuteOff	On an existing conference call, cancels the Mute mode for the party specified by the DN parameter.
RequestSetMuteOn	On an existing conference call, sets Mute mode for the party specified by the DN parameter.
Call Handling Functions	
RequestAlternateCall	On behalf of the telephony object specified by the DN parameter, places the active call specified by thecurrent_conn_id parameter on hold and connects the call specified by the held_conn_id parameter.

Switch Function	Description
RequestAnswerCall	Answers the alerting call specified by the conn_id parameter.
RequestAttachUserData	On behalf of the telephony object specified by the DN parameter, attaches the user data structure specified by the user_data parameter to the T-Server information that is related to the call specified by the conn_id parameter.
RequestClearCall	Deletes all parties, that is, all telephony objects, from the call specified by conn_id and disconnects the call.
RequestCompleteConference	Completes a previously-initiated conference by merging the held call specified by the held_conn_id parameter with the active consultation call specified by the current_conn_id parameter on behalf of the telephony object specified by the DN. Assigns the held_conn_id to the resulting conference call. Clears the consultation call specified by the current_conn_id parameter.
RequestCompleteTransfer	On behalf of the telephony object specified by the DN parameter, completes a previously initiated two-step transfer by merging the held call specified by the conn_id parameter with the active consultation call specified by the current_conn_id parameter. Assigns held_conn_id to the resulting call. Releases the telephony object specified by the DN parameter from both calls and clears the consultation call specified by the current_conn_id parameter.
RequestDeleteFromConference	A telephony object specified by DN deletes the telephony object specified by dn_to_drop from the conference call specified by conn_id. The client that invokes this service must be a party on the call in question.
RequestDeletePair	On behalf of the telephony object specified by the DN parameter, deletes the key-value pair specified by the key parameter from the user data attached to the call specified by the conn_id parameter.
RequestDeleteUserData	On behalf of the telephony object specified by the DN parameter, deletes all of the user data attached to the call specified by the conn_id parameter.
RequestHoldCall	On behalf of the telephony object specified by the DN parameter, places the call specified by the conn_id parameter on hold.
RequestInitiateConference	On behalf of the telephony object specified by the DN parameter, places the existing call specified by the conn_id parameter on hold and originates a consultation call from the same telephony object to the called party, which is specified by the destination parameter with the purpose of a conference call.
RequestInitiateTransfer	On behalf of the telephony object specified by the

Switch Function	Description
	DN parameter, places the existing call specified by the conn_id parameter on hold and originates a consultation call from the same telephony object to the called party, which is specified by the destination parameter for the purpose of a two-step transfer.
RequestListenDisconnect	On an existing conference call, sets Deaf mode for the party specified by the listener_dn parameter. For example, if two agents wish to consult privately, the subscriber may temporarily be placed in Deaf mode.
RequestListenReconnect	On an existing conference call, cancels Deaf mode for the party defined by the listener_dn parameter.
RequestMakeCall	Originates a regular call from the telephony object specified by the DN parameter to the called party specified by the Destination parameter.
RequestMakePredictiveCall	Makes a predictive call from the thisDN DN to the otherDN called party. A predictive call occurs before any agent-subscriber interaction is created. For example, if a fax machine answers the call, no agent connection occurs. The agent connection occurs only if there is an actual subscriber available on line.
RequestMergeCalls	On behalf of the telephony object specified by the DN parameter, merges the held call specified by the held_conn_id parameter with the active call specified by the current_conn_id parameter in a manner specified by the merge_type parameter. The resulting call will have the same conn_id as the held call.
RequestMuteTransfer	Initiates a transfer of the call specified by the conn_id parameter from the telephony object specified by the DN parameter to the party specified by the destination parameter; completes the transfer without waiting for the destination party to pick it up. Releases the telephony object specified by the DN parameter from the call.
RequestQueryCall	Requests the information specified by info_type about the telephony object specified by conn_id. If the query type is supported, the requested information will be returned in EventPartyInfo.
RequestReconnectCall	Releases the telephony object specified by the DN parameter from the active call specified by the current_conn_id parameter and retrieves the previously held call, specified by the held_conn_id parameter, to the same object. This function is commonly used to clear an active call and to return to a held call, or to cancel a consult call (due to lack of an answer, because the device is busy, and so on) and then to return to a held call.
RequestRedirectCall	Requests that the call be redirected, without an

Switch Function	Description
	answer, from the party specified by the DN parameter to the party specified by the dest_dn parameter.
RequestRegisterAddress	Registers for a DN. Your application must register the DN before sending the RequestAgentLogin.
RequestReleaseCall	Releases the telephony object specified by the DN parameter from the call specified by the conn_id parameter.
RequestRetrieveCall	Connects the held call specified by the conn_id parameter to the telephony object specified by the DN parameter.
RequestSendDtmf	On behalf of the telephony object specified by the DN parameter, sends the digits that are expected by an interactive voice response system.
RequestSetCallInfo	Changes the call attributes. Warning: Improper use of this function may result in unpredictable behavior on the part of the T-Server and the Genesys Framework. If you have any doubt on how to use it, please consult with Genesys.
RequestSetMessageWaitingOff	Sets the Message Waiting indication to off for the telephony object specified by the DN parameter.
RequestSetMessageWaitingOn	Sets the Message Waiting indication to on for the telephony object specified by the DN parameter.
RequestSetMuteOff	On an existing conference call, cancels the Mute mode for the party specified by the DN parameter.
RequestSetMuteOn	On an existing conference call, sets Mute mode for the party specified by the DN parameter.
RequestSingleStepConference	Adds a new party to an existing call and creates a conference.
RequestSingleStepTransfer	Transfers the call from a specified directory number DN that is currently engaged in the call specified by the conn_id parameter to a destination DN that is specified by the destination parameter.
RequestUnregisterAddress	Unregisters a DN.
RequestUpdateUserData	On behalf of the telephony object specified by the DN parameter, updates the user data that is attached to the call specified by the conn_id parameter with the data specified by the user_data parameter.

Using the Warm Standby Application Block

The Warm Standby Application Block is a reusable production-quality component that enables developers to switch to a backup server in case their primary server fails, without needing to guarantee the integrity of existing interactions. 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, as well as giving details about how to setup the QuickStart application that ships with this application block.

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.

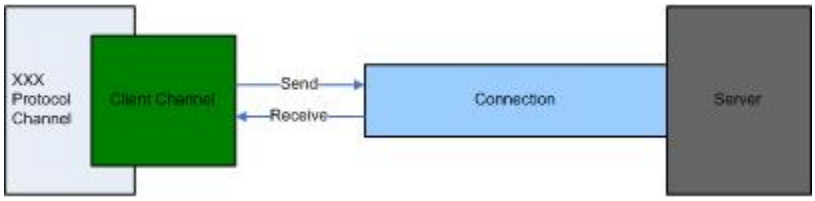
The Client Channel Architecture

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

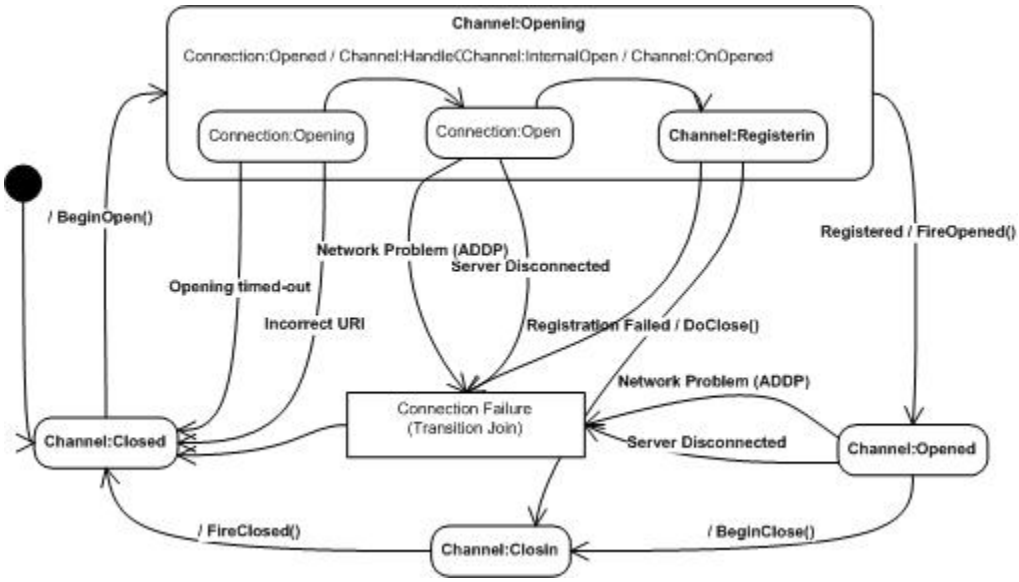
the protocol- and server-independent channel functionality that is common for a protocol channel. Finally, the *protocol channel* controls all of the client channel activities that are dependent on the protocol and the server.

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

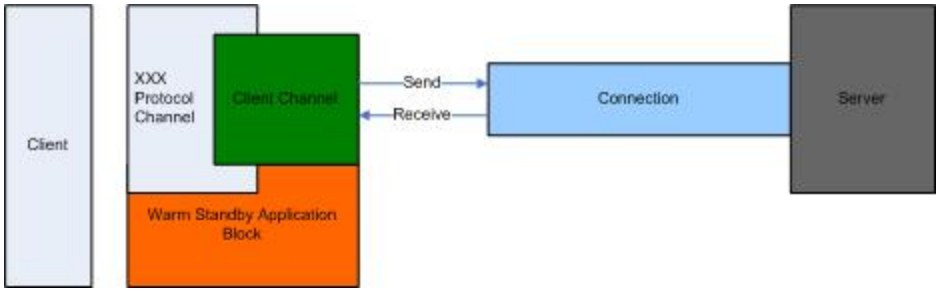
Scenario	Description	Source States	Condition	Target State	Protocol-Dependent
Opening Timed Out	Channel tries to open connection to non-existing URI	Opening	Connection opening timeout	Closed	No
Wrong URI	Channel tries to open connection to non-existing URI	Opening	Incorrect URI exception	Closed	No
Connection Problem	Channel connection detects a connection problem	Opened Opening	Server disconnected	Closed	No
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:

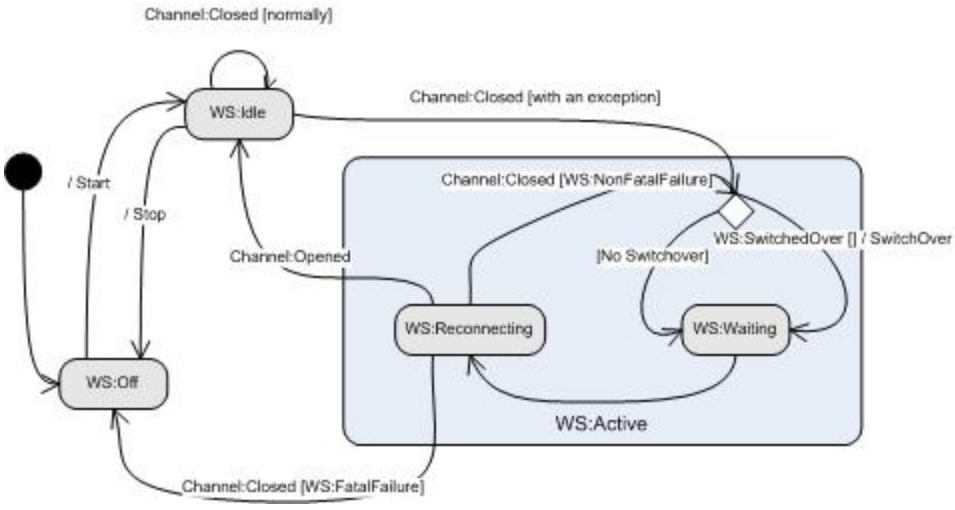
- Do nothing (close the channel by request of the user application)
- Attempt to open the channel without switching over its connectivity configuration from primary to backup
- Attempt to open the channel, switching its connectivity configuration from primary to backup
- Deactivate, in case of a fatal 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

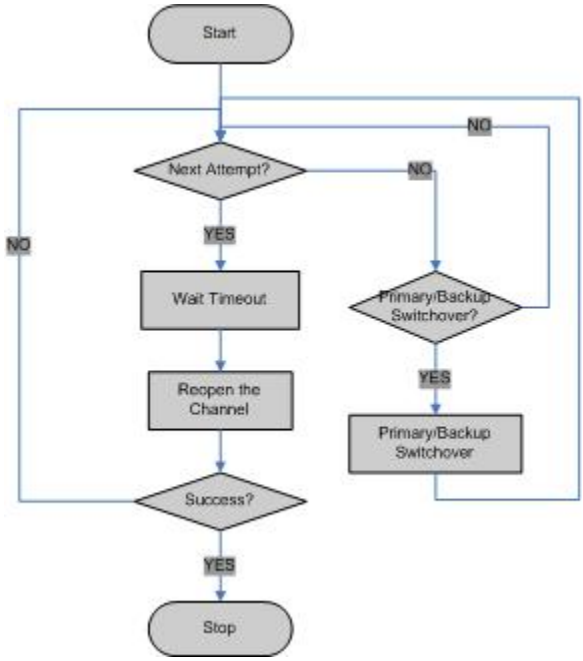
To control channel connectivity with a warm standby mechanism, the user application should activate the Warm Standby Application Block instance that is responsible for handling the particular channel's connectivity failure and recovery.

Warm Standby Application Block Algorithm

The Warm Standby Application Block has 4 states, as shown below.



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:

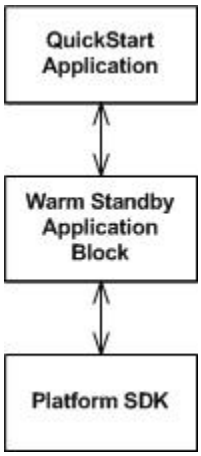
- The user should activate the Warm Standby Application Block for every channel he or she intends to work with.
- In the active state, the application block waits for the channel's `Closed` event.
- On receiving the channel's `Closed` event, the application block activates the channel connectivity recovery 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.

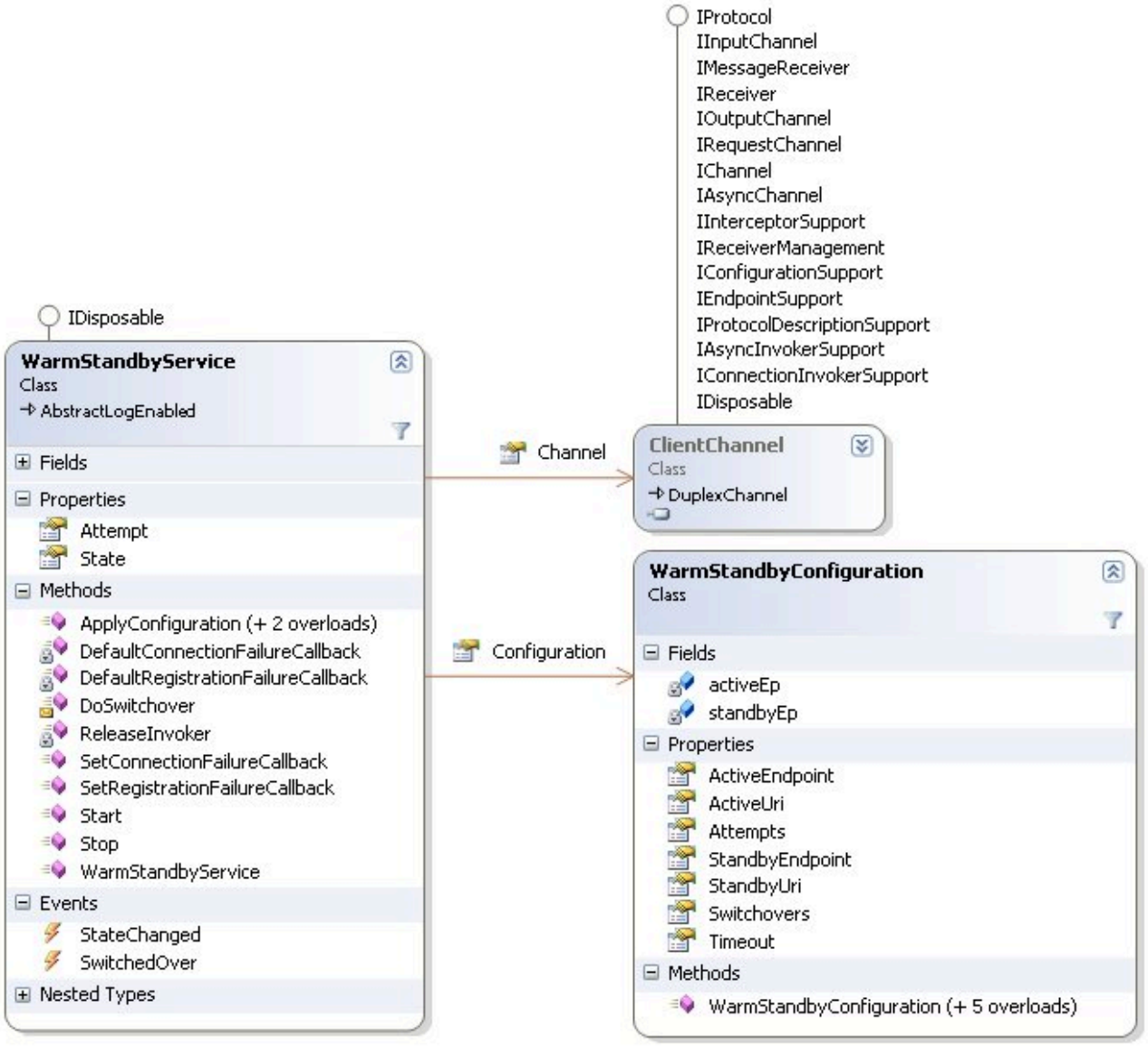


The Warm Standby Application Block Interface

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.

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.

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 Application Block for .NET

Installing 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.

Software Requirements

To work with the Warm Standby Application Block, you must ensure that your system meets the software requirements established in the Genesys Supported Operating Environment Reference Manual.

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.

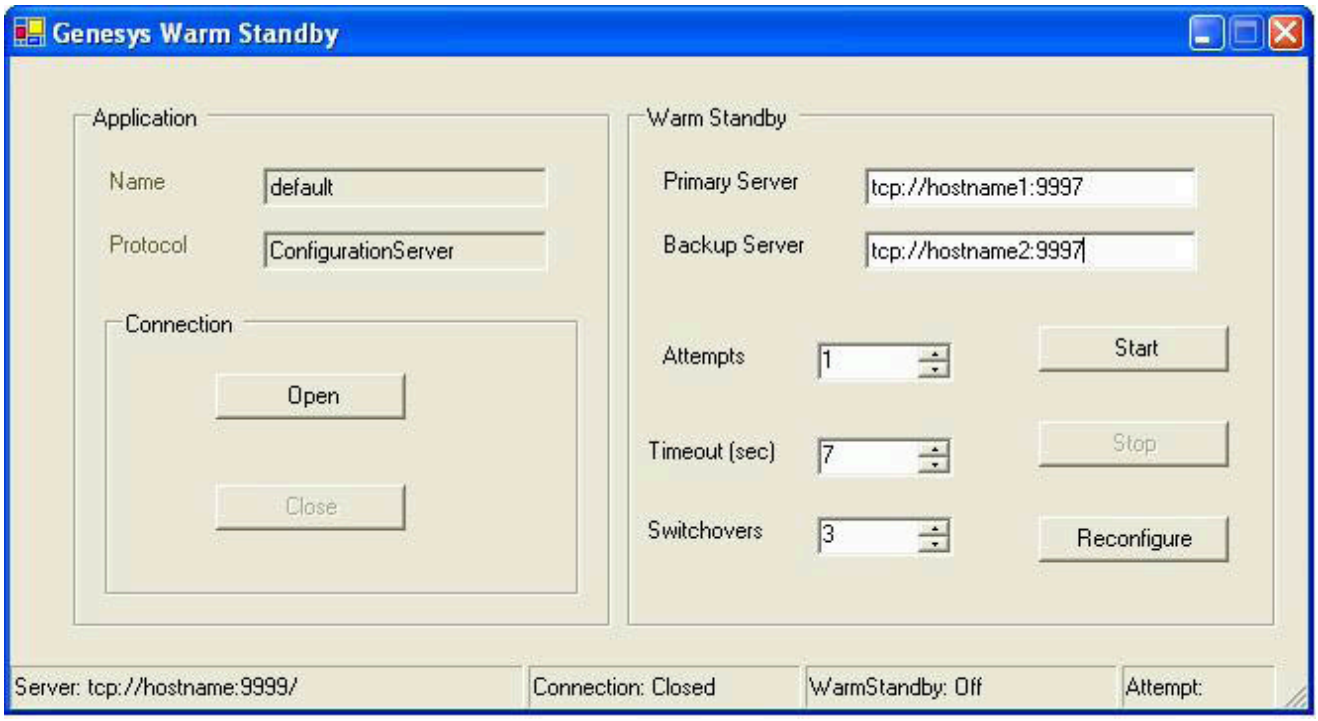
Using the QuickStart Application

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.



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.

Using the Application Block for Java

Installing 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.

Software Requirements

To work with the Warm Standby Application Block, you must ensure that your system meets the software requirements established in the Genesys Supported Operating Environment Reference Manual, as well as meeting the following minimum software requirements:

- JDK 1.6 or higher

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.

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

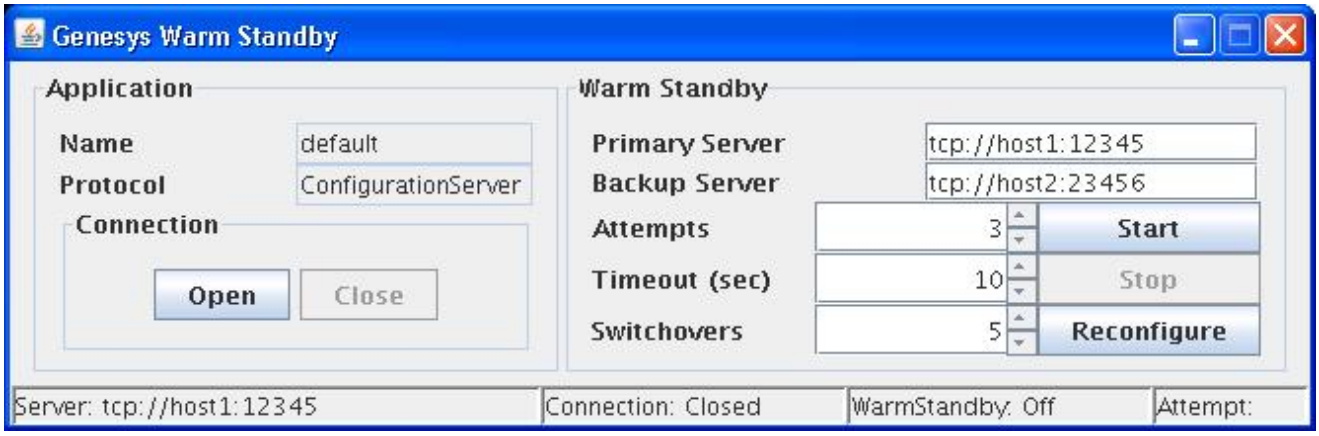
The easiest way to start using the Warm Standby Application Block is to use the bundled QuickStart application. This application ships in the same folder as the application block.

To run the QuickStart application:

1. Open the *\ApplicationBlocks\WarmStandby\quickstart* folder.
2. Run either *quickstart.bat* or *quickstart.sh*, depending on your platform.

You may need to edit the path specified in the quickstart file by adding quotation marks if your ANT_HOME environment variable contains spaces.

After you start the application, you will see the user interface shown below.



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.

Platform SDK Resources

Purpose: Describes additional resources located on this site and the Genesys Support site.

Related Documentation

Depending on what type of development you are doing with the Platform SDKs, the following resources may be useful for providing background information about your Genesys environment.

Genesys Events and Models Reference Manual

Use with: *T-Server, Interaction Server*

Download: Genesys Events and Models Reference Manual

If you are working with T-Server or Interaction Server, you should download and start reading the *Genesys Events and Models Reference Manual* right away. This document provides you with a large collection of two different types of important information, organized into two separate sections.

- **Part 1: Genesys Events** is the events portion of this document. The information in this part is wide-ranging, and includes everything from the names and descriptions of events, to the attributes that go with these events, to the definitions of event sub-states.
- **Part 2: Genesys Interaction Models** is the models portion of this document. It contains a selected list of call and interaction models. This information is also wide ranging. Based on the history of how this information has been presented in the past in various documents, model details may differ from chapter to chapter.

In both parts of this document, chapters are organized according to the type of event or model being described. So, for example, both parts one and two have specific chapters on voice-based issues that center on T-Library's generation of events and how calls are routed in a contact center.

Framework Stat Server User's Guide

Use with: *Stat Server*

Download:

- Framework 8.1 Stat Server User's Guide
- Framework 8.0 Stat Server User's Guide

Reporting Technical Reference

Use with: *Stat Server*

Download: Reporting Technical Reference 8.0 Overview

Download: Reporting Technical Reference Guide for the Genesys 7.2 Release

Code Samples

The documentation for the Platform SDK includes a number of code samples. These samples are for illustrative purposes only:

- Complex Platform SDK 7.6 .NET Code Sample
- Configuration Platform SDK 7.6 Java (with Message Broker) Code Sample
- Configuration Platform SDK 7.6 .NET Code Sample
- Open Media Platform SDK 7.6 Java (Client) Code Sample
- Open Media Platform SDK 7.6 .NET (Client) Code Sample
- Open Media Platform SDK 7.6 Java (Server) Code Sample
- Open Media Platform SDK 7.6 .NET (Server) Code Sample
- Statistics Platform SDK 7.6 Java Code Sample
- Statistics Platform SDK 7.6 .NET Code Sample
- Voice Platform SDK 7.6 Java Code Sample
- Voice Platform SDK 7.6 .NET Code Sample