Platform SDK Developer's Guide

Connecting to a Server

12/17/2025

Connecting to a Server

Java

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

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

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

Creating a Protocol Object

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

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

```
[Java]

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

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

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

Opening a Synchronous Connection

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

[Java]

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

Opening an Asynchronous Connection

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

> ### Important
> Be careful when using this method. Connecting asynchronously means that you need to be sure that the Opened event is received *before* you send any requests. Otherwise, you might be trying to use a connection that is not yet open.

[Java]

```
tserverProtocol.beginOpen();
// Watch for an Opened event before trying to send or recieve messages.
```

> ### Important
> When using the `BeginOpen()` method, make sure that your code waits for the Opened event to fire before attempting to send or receive messages.

Closing a Connection

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

[Java]

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

Or:

[Java]

```
// Asyncronous
tserverProtocol.beginClose();
```

You may want to set up a connection to more than one server. To do that, you will need to repeat the steps outlined above for every server you connect to.

Configuring ADDP

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

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

```
[Java]

PropertyConfiguration tserverConfig = new PropertyConfiguration();
tserverConfig.setUseAddp(true);
tserverConfig.setAddpServerTimeout(10);
tserverConfig.setAddpClientTimeout(10);
tserverConfig.setAddpTrace("both");

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

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

Configuring Warm Standby

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

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

If you are connecting to a single server, use the WarmStandby Application Block to retry the first connection or to reconnect after that server has been unavailable. In this case, configure the

WarmStandbyService to use the same Endpoint as primary and backup.

Activating the WarmStandby Application Block

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

```
[Java]

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

TServerProtocol tserverProtocol = new TServerProtocol(tserverEndpoint);

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

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

tserverProtocol.open();
```

Stopping the WarmStandby Application Block

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

```
[Java]

warmStandby.stop();
tserverProtocol.close();
```

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

AsyncInvokers

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

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

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

```
[Java]

public class SwingInvoker implements AsyncInvoker {

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

        @Override
        public void dispose() {}

}
```

Assigning a Protocol Invoker

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

```
[Java]

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

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

Enabling ADDP on Platform SDK Protocol Client Connections

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

The `ConnectionConfiguration` interface describes all connection properties (including details about ADDP, TLS, and the channel character set encoding) for a single instance. For example, if a connection has already configured TLS and later needs to add or change ADDP options then a new `ConnectionConfiguration` should be initialized with the previously set TLS options (along with and other values which are to be preserved) and then have new ADDP options added. The latest configuration applied with overwrite previously set properties.

> **Tip**
>
> Changing the configuration immediately after a connection is opened, or from the channel event handlers, is not recommended. Some connection configuration options (including ADDP) can be changed on the fly, however the channel configuration is not

> expected to change often or quickly - options are not treated as if they are dynamic values.

Platform SDK connections have the following ADDP configuration options available:

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

Here is an initialization code sample:

```
protocol = <SomePsdkProtocol>(<Endpoint>);
protocol.set<ProtoSpecificOptions>(<val>);
PropertyConfiguration conf = new PropertyConfiguration();
conf.setOption(AddpInterceptor.PROTOCOL_NAME_KEY, AddpInterceptor.NAME);
conf.setOption(AddpInterceptor.TIMEOUT_KEY, "10");
conf.setOption(AddpInterceptor.REMOTE_TIMEOUT_KEY, "11.5");
conf.setOption(AddpInterceptor.TRACE_KEY, "full");
protocol.configure(conf);
protocol.open();
```

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

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

Also note that in `conf.setOption(AddpInterceptor.TRACE_KEY, "full")`, the `conf.setOption(..)` method accepts string values for the option and it does not know `CfgTraceMode` enumeration - it is defined in configuration protocol which is out of the Platform SDK common libraries.

In release 8.1.0 of Platform SDK for Java, this property handling logic was improved with truncation of the "CFGTM" prefix. Platform SDK for .NET includes this feature starting from release 8.1.1.

So, if you use latest Platform SDK 8.1.0 version for Java, writing `CfgTraceMode.CFGTMBoth.toString()` is acceptable, but earlier versions of Platform SDK or Platform SDK for .NET require that you translate the enumeration values to the corresponding string values.

Possible values can be ["full", "local", "remote", "none"]:

- "local" means logging of ADDP activities locally on client side.
- "remote" for Platform SDK means sending of special initialization bit in ADDP initialization message to server side to ask server to write own ADDP tracing records to server side log.
- "full" means "local" + "remote".

Unknown values are treated as “none”.

Note that comparison is case-insensitive for option values, so "FULL" == "Full" == "full".

.NET

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

Creating a Protocol Object

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

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

[C#]

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

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

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

Opening a Synchronous Connection

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

[C#]

```
tserverProtocol.Open();
```

```
// You can start sending requests here.
```

Opening an Asynchronous Connection

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

Important: Be careful when using this method. Connecting asynchronously means that you need to be sure that the Opened event is received *before* you send any requests. Otherwise, you might be trying to use a connection that is not yet open.

```
[C#]

tserverProtocol.BeginOpen();
// Watch for an Opened event before trying to send or recieve messages.
```

> **Important**
>
> When using the `BeginOpen()` method, make sure that your code waits for the Opened event to fire before attempting to send or receive messages.

Closing a Connection

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

```
[C#]

tserverProtocol.Close();
```

Or:

```
[C#]

tserverProtocol.BeginClose();
```

Or:

```
[C#]

tserverProtocol.Dispose();
```

You may want to set up a connection to more than one server. To do that, you will need to repeat the steps outlined above for every server.

Configuring ADDP

The Advanced Disconnection Detection Protocol (ADDP) is a Genesys proprietary add-on to the TCP/IP

stack. It implements a periodic poll when no actual activity occurs over a given connection. If a configurable timeout expires without a response from the opposite process, the connection is considered lost.

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

```
[C#]

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

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

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

Configuring Warm Standby

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

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

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

Activating the WarmStandby Application Block

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

```
[C#]

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

```
tserverConfig);

var tserverProtocol = new TServerProtocol(tserverEndpoint);

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

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

tserverProtocol.Open();
```

Stopping the WarmStandby Application Block

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

```
[C#]

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

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

AsyncInvokers

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

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

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

```
[C#]

public class SyncContextInvoker : IAsyncInvoker
{
        private readonly SynchronizationContext syncContext;

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

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

```
        }

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

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

The Protocol Invoker

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

```
[C#]

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

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

Advanced: Multithreaded Message Parsing

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

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

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

By default, the Platform SDK uses a single thread for parsing all messages. This parsing can be time-consuming in certain cases, and some applications may face performance issues. For example, some applications may receive lots of large binary-format messages, such as some of the statistics

messages generated by Stat Server, while others might need to parse messages in non-binary formats, such as the XML format used to communicate with Genesys Multimedia (or e-Services) servers.

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

```
[C#]

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