Platform SDK Developer's Guide

Connecting to a Server Using the Protocol Manager Application Block

4/9/2025

Connecting to a Server Using the Protocol Manager Application Block

> **Important**
>
> The Protocol Manager Application Block is considered a legacy product as of release 8.1.1 due to improvements in the configuration of core protocol classes. Documentation related to this application block is retained for backwards compatibility. For information about connecting to Genesys servers *without* use of the Protocol Manager Application Block, refer to the Connecting to a Server article.

The applications you write with the Platform SDK will need to communicate with one or more Genesys servers. So the first thing you need to do is create a connection with these servers. Genesys recommends that you use the **Protocol Manager Application Block** to do this. Protocol Manager is designed for high-performance communication with Genesys servers. It also includes built-in support for warm standby.

Once you have connected to a server, you will be sending and receiving messages to and from this server. The next article shows how to use the Message Broker Application Block for efficient event handling using the Message Broker Application Block.

> **Tip**
>
> Protocol Manager may not support all of the servers you need to use in your application. For information about how to update Protocol Manager to communicate with these servers, see the Using the Protocol Manager Application Block article.

Java

To use the Protocol Manager Application Block, add the following file to your classpath:

- `protocolmanagerappblock.jar`

This jar file was precompiled using the default Application Block code, and can be located at: `<Platform SDK Folder>\lib`.

> **Tip**

> You can also view or modify the Protocol Manager Application Block source code. To do this, open the Protocol Manager Java source files that were installed with the Platform SDK. The Java source files for this project are located at: <Platform SDK Folder>\applicationblocks\protocolmanager\src\java. If you make any changes to the project, you will have to run Ant (or use the build.bat file for this Application Block) to rebuild the jar archive listed above. After you run Ant, add the resulting jar to your classpath.

Now you can add import statements to your source code. For example:

```
[Java]

import com.genesyslab.platform.applicationblocks.commons.protocols.*;
import com.genesyslab.platform.applicationblocks.warmstandby.*;
```

You will also have to add additional JAR archives to your classpath and add import statements to your project for each specific protocol you are working with. The steps are not explicitly described here because the archives and classes required will vary depending on which SDKs and protocols you plan to use.

In order to use the Protocol Manager, you need to create a ProtocolManagementServiceImpl object. This object manages all of your server connections. Declare it with your other fields:

```
[Java]

ProtocolManagementServiceImpl protocolManagementServiceImpl;
```

Then you can initialize the service object inside the appropriate method body:

```
[Java]

protocolManagementServiceImpl =
        new ProtocolManagementServiceImpl();
```

You are now ready to create an object that will be used to specify how to communicate with the server. For example, if you are working with Configuration Server, you will set up a ConfServerConfiguration object:

[Java]

ConfServerConfiguration confServerConfiguration = new
ConfServerConfiguration("Config_Server_App");

Note that you have to provide a string when you create the ConfServerConfiguration object. This string should be unique for each protocol used in your application. It might be a good idea to use the name of the server's application object from the configuration layer, which guarantees uniqueness as well as clearly identifying which server you are communicating with.

After setting up the ConfServerConfiguration object, you need to specify the URI of the Configuration Server you want to communicate with, as well as a few other necessary pieces of information:

```
[Java]
```

```
try {
        confServerConfiguration.setUri(
                new URI("tcp://" + confServerHost + ":" + confServerPort));
} catch (URISyntaxException e) {
        e.printStackTrace();
}
confServerConfiguration.setClientApplicationType(CfgAppType.CFGSCE);
confServerConfiguration.setClientName(clientName);
confServerConfiguration.setUserName(userName);
confServerConfiguration.setUserPassword(password);
```

At this point, you can register your `ConfServerConfiguration` object with Protocol Manager:

```
[Java]

protocolManagementServiceImpl.register(confServerConfiguration);
```

Now you can tell Protocol Manager to open the connection to your server:

```
[Java]

try {
        protocolManagementServiceImpl.getProtocol("Config_Server_App").open();
} catch (ProtocolException e) {
        e.printStackTrace();
} catch (IllegalStateException e) {
        e.printStackTrace();
} catch (InterruptedException e) {
        e.printStackTrace();
}
```

You may want to set up a connection to more than one server. To do that, you could repeat the steps outlined above. Here is an example of how you might do that in order to add a connection to Stat Server:

```
[Java]

StatServerConfiguration statServerConfiguration = new StatServerConfiguration(
                "Stat_Server_App");
try {
        statServerConfiguration.setUri(new URI(statServerUri));
} catch (URISyntaxException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
}
protocolManagementServiceImpl.register(statServerConfiguration);
.
.
.
// Add this line to the try block for the Configuration Server open()
protocolManagementServiceImpl.getProtocol("Stat_Server_App").open();
```

In some cases, you may want to use the `beginOpen()` method instead of using the `open()` method. `beginOpen()` will open all of your connections with a single method call. However, unlike `open()`, `beginOpen()` is asynchronous. This means you will need to make sure you have received the `onChannelOpened` event before you send any messages. Otherwise, you might be trying to use a connection that does not yet exist.

In order to use `beginOpen()`, you need to implement the `ChannelListener` interface:

```
[Java]

import com.genesyslab.platform.commons.protocol.ChannelListener;
.
.
.
public class YourApplication
        implements ChannelListener, ...
```

You will also need to add a channel listener after you register your `ServerConfiguration` objects:

```
[Java]

protocolManagementServiceImpl.register(confServerConfiguration);
protocolManagementServiceImpl.register(statServerConfiguration);
.
.
.
protocolManagementServiceImpl.addChannelListener(this);
```

Now you can add a method to handle the `OnChannelOpened` event:

```
[Java]

public void onChannelOpened(EventObject event) {
        if ( event.getSource() instanceof ClientChannel )        {
                ClientChannel channel = (ClientChannel)event.getSource();

                if ( channel instanceof ConfServerProtocol ) {
                        // Work with Configuration Server messages...
                }
                else if ( channel instanceof StatServerProtocol ) {
                        // Work with Stat Server messages...
                }
        }
}
```

Having done that, you can remove these lines from the `try` block:

```
[Java]

protocolManagementServiceImpl.getProtocol("Config_Server_App").open();
protocolManagementServiceImpl.getProtocol("Stat_Server_App").open();
```

And replace them with this one:

```
[Java]

protocolManagementServiceImpl.beginOpen();
```

However, if you want to issue an asynchronous open for a specific protocol, you can invoke `beginOpen` for that protocol, like this:

```
[Java]

protocolManagementServiceImpl.getProtocol("Config_Server_App").beginOpen();
protocolManagementServiceImpl.getProtocol("Stat_Server_App").beginOpen();
```

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

Once you have opened your connection, you can send and receive messages, as shown in the article on Event Handling. But before getting to that, please note that when you have finished communicating with your servers, you should close the connection, like this:

```
[Java]

protocolManagementServiceImpl.beginClose();
```

Or like this:

```
[Java]

protocolManagementServiceImpl.getProtocol("Config_Server_App")
        .close();
protocolManagementServiceImpl.getProtocol("Stat_Server_App")
        .close();
```

Or like this:

```
[Java]

protocolManagementServiceImpl.getProtocol("Config_Server_App")
                .beginClose();
protocolManagementServiceImpl.getProtocol("Stat_Server_App")
                .beginClose();
```

This introduction has only covered the most basic features of the Protocol Manager Application Block. Consult the Protocol Manager Application Block Guide for more information on how to use Protocol Manager, including the following topics:

- Configuring ADDP
- Configuring Warm Standby
- High-Performance Message Parsing
- Supporting New Protocols

To learn how to send and receive messages, go to the article on Event Handling Using the Message Broker Application Block.

.NET

To use the Protocol Manager Application Block, open the Solution Explorer for your application project and add references to the following files:

- `Genesyslab.Platform.ApplicationBlocks.Commons.Protocols.dll`

- Genesyslab.Platform.ApplicationBlocks.WarmStandby.dll

These dll files are precompiled using the default Application Block code, and can be located at: <Platform SDK Folder>\Bin.

> **Tip**
>
> You can also view or modify the Protocol Manager Application Block source code. To do this, open the Protocol Manager Visual Studio project that was installed with the Platform SDK. The solution file for this project is located at: <Platform SDK Folder>\ApplicationBlocks\ProtocolManager. If you make any changes to the project, you will have to rebuild the two .dll files listed above.

Once you have added the references, you can add using statements to your source code:

```
[C#]

using Genesyslab.Platform.ApplicationBlocks.Commons.Protocols;
using Genesyslab.Platform.ApplicationBlocks.WarmStandby;
```

You will also have to reference additional libraries and add using statements to your project for each specific protocol you are working with. The steps are not explicitly described here because the files and namespaces required will vary depending on which SDKs and protocols you plan to use.

In order to use the Protocol Manager, you now need to create a ProtocolManagementService object. This object manages all of your server connections. Declare it with your other fields:

```
[C#]

ProtocolManagementService protocolManagementService;
```

Then you can initialize the service object inside the appropriate method body:

```
[C#]

protocolManagementService =
        new ProtocolManagementService();
```

You are now ready to create an object that will be used to specify how to communicate with the server. For example, if you are working with Configuration Server, you will set up a ConfServerConfiguration object:

```
[C#]

ConfServerConfiguration confServerConfiguration =
        new ConfServerConfiguration("Config_Server_App");
```

Note that you have to provide a string when you create the ConfServerConfiguration object. This string should be unique for each protocol used in your application. It might be a good idea to use the name of the server's application object from the configuration layer, which guarantees uniqueness as well as clearly identifying which server you are communicating with.

After setting up the ConfServerConfiguration object, you need to specify the URI of the

Configuration Server you want to communicate with, as well as a few other necessary pieces of information:

```
[C#]

confServerConfiguration.Uri =
        new Uri("tcp://" + confServerHost + ":" + confServerPort);
confServerConfiguration.ClientApplicationType = CfgAppType.CFGSCE;
confServerConfiguration.ClientName = clientName;
confServerConfiguration.UserName = userName;
confServerConfiguration.UserPassword = password;
```

At this point, you can register your `ConfServerConfiguration` object with Protocol Manager:

```
[C#]

protocolManagementService.Register(confServerConfiguration);
```

Now you can tell Protocol Manager to open the connection to your server:

```
[C#]

protocolManagementService["Config_Server_App"].Open();
```

You may want to set up a connection to more than one server. To do that, you could repeat the steps outlined above. Here is an example of how you might do that in order to add a connection to Stat Server:

```
[C#]

StatServerConfiguration statServerConfiguration = new
StatServerConfiguration("Stat_Server_App");
statServerConfiguration.Uri = statServerUri;
protocolManagementService.Register(statServerConfiguration);
.
.
.
protocolManagementService["Stat_Server_App"].Open();
```

In some cases, you may want to use the `BeginOpen()` method instead of using the `Open()` method. `BeginOpen()` will open all of your connections with a single method call. However, unlike `Open()`, `BeginOpen()` is asynchronous. This means you will need to make sure you have received the `Opened` event before you send any messages. Otherwise, you might be trying to use a connection that does not yet exist.

Once you have set up an event handler for the `Opened` event, you can remove these lines from your code:

```
[C#]

protocolManagementService["Config_Server_App"].Open();
protocolManagementService["Stat_Server_App"].Open();
```

And replace them with this one:

```
[C#]

protocolManagementService.BeginOpen();
```

However, if you want to issue an asynchronous open for a specific protocol, you can invoke BeginOpen for that protocol, like this:

```
[C#]

protocolManagementService["Config_Server_App"].BeginOpen();
protocolManagementService["Stat_Server_App"].BeginOpen();
```

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

Once you have opened your connection, you can send and receive messages, as shown in the article on Event Handling Using the Message Broker Application Block. But before getting to that, please note that when you have finished communicating with your servers, you should close the connection, like this:

```
[C#]

protocolManagementService.BeginClose();
```

Or like this:

```
[C#]

protocolManagementService["Config_Server_App"].Close();
protocolManagementService["Stat_Server_App"].Close();
```

Or like this:

```
[C#]

protocolManagementService["Config_Server_App"].BeginClose();
protocolManagementService["Stat_Server_App"].BeginClose();
```

This introduction has only covered the most basic features of the Protocol Manager Application Block. Consult Using the Protocol Manager Application Block for more information on how to use Protocol Manager, including the following topics:

- Configuring ADDP
- Configuring Warm Standby
- High-Performance Message Parsing
- Supporting New Protocols

To learn how to send and receive messages, go to the article on Event Handling Using the Message Broker Application Block.

Transport Layer Security (TLS) Support

Platform SDK now supports Transport Layer Security (TLS). This section contains two sample configurations, but it is important to understand your environment and its unique requirements before using this new support. You should refer to the appropriate server manual to configure TLS on your server. You should also refer to Part 3 of the Genesys 8.0 Security Deployment Guide, "Communications Integrity—Transport Layer Security".

The first sample configuration shows a situation where the client application specifies the name of a server-based certificate:

```
[C#]

SolutionControlServerProtocol scsProtocol
        = new SolutionControlServerProtocol(myEndpoint);
KeyValueCollection kvCollection = new KeyValueCollection();
kvCollection[CommonConnection.TlsKey] = 1;
kvCollection[CommonConnection.CertificateNameKey] = "name";
KeyValueConfiguration kvConfig = new KeyValueConfiguration(kvCollection);
scsProtocol.Configure(kvConfig);
```

In this sample configuration, "name" is the name of the certificate, which is located in the certificate store on the server and used in the TLS configuration of the port/application/server in the Genesys Configuration Layer.

The second sample configuration shows a client application using its own client certificate to authenticate on the server:

```
[C#]

SolutionControlServerProtocol scsProtocol
        = new SolutionControlServerProtocol(myEndpoint);
KeyValueCollection kvCollection = new KeyValueCollection();
kvCollection[CommonConnection.TlsKey] = 1;
kvCollection[CommonConnection.CertificateKey] = @"c:\directory\certificate.p12";
kvCollection[CommonConnection.CertificatePwdKey] = "password";
KeyValueConfiguration kvConfig = new KeyValueConfiguration(kvCollection);
scsProtocol.Configure(kvConfig);
```

In this sample configuration, `CommonConnection.CertificateKey` is the path to the certificate file located on the client machine, while `CommonConnection.CertificatePwdKey` is the password which will be used to open the certificate file, if it is password protected.