Platform SDK Developer's Guide

Using the Protocol Manager Application Block

4/9/2025

Using the Protocol Manager Application Block

Deprecation Notice: This application block is considered a legacy product staring with release 8.1.1. Documentation is provided for backwards compatibility, but new development should consider using the improved method of connecting to servers.

> **Important**
>
> This application block is a reusable production-quality component. It has been designed using industry best practices and provided with source code so it can be used "as is," extended, or tailored if you need to.
>
> Please see the License Agreement for details.

One of the two main functions of the Platform SDK is to enable your applications to establish and maintain connections with Genesys servers. The Protocol Manager Application Block provides unified management of server protocol objects. It takes care of opening and closing connections to many different servers, as well as reconfiguration of high availability connections.

Java

Installing the Protocol Manager Application Block

Before you install the Protocol Manager Application Block, it is important to review the software requirements and the structure of the software distribution.

Building the Protocol Manager Application Block

To build the Protocol Manager Application Block:

1. Open the `<Platform SDK Folder>\applicationblocks\protocolmanager` folder.
2. Run either `build.bat` or `build.sh`, depending on your platform.

This will create the `protocolmanagerappblock.jar` file, located within the `<Platform SDK Folder>\applicationblocks\protocolmanager\dist\lib` directory.

Working with the Protocol Manager Application Block

You can find basic information on how to use the Protocol Manager Application Block in the article on Connecting to a Server Using the Protocol Manager Application Block.

Configuring ADDP

To enable ADDP, set the `UseAddp` property of your Configuration object to `true`. You can also set server and client timeout intervals, as shown here:

```
[Java]

statServerConfiguration.setUseAddp(true);
statServerConfiguration.setAddpServerTimeout(10);
statServerConfiguration.setAddpClientTimeout(10);
```

> **Tip**
>
> To avoid connection exceptions in the scenario where a client has configured ADDP but the server has not, "ADDP" is included as a default value for the "protocol" key in the `configure()` method of the `ServerChannel` class.

Configuring Warm Standby

Enable warm standby in your application by setting your Configuration object's `FaultTolerance` property to `FaultToleranceMode.WarmStandby`, as shown here. You can also configure the backup server's URI, the timeout interval, and the number of times your application will attempt to contact the primary server before switching to the backup:

```
[Java]

statServerConfiguration
                .setFaultTolerance(FaultToleranceMode.WarmStandby);
statServerConfiguration.setWarmStandbyTimeout(10);
statServerConfiguration.setWarmStandbyAttempts((short) 5);
try {
        statServerConfiguration.setWarmStandbyUri(new URI("tcp://"
                + statServerBackupHost
                + ":"
                + statServerBackupPort));
} catch (URISyntaxException e) {
        e.printStackTrace();
}
```

High-Performance Message Parsing

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

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

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

This section gives an example of how you can modify Protocol Manager to selectively enable multi-threaded parsing of incoming messages, in order to work around these kinds of performance issues. It is important to stress that you must take a careful look at which kind of multi-threading options to pursue in your applications, since your needs are specific to your situation.

> ### Tip
> Your application may also face other performance bottlenecks. For example, you may need more than one instance of the Message Broker Application Block if you handle large numbers of messages. For more information on how to configure Message Broker for high-performance situations, see the Message Broker Application Block Guide.

This example shows how to call `com.genesyslab.platform.commons.threading.DefaultInvoker`, which uses `SingleThreadInvoker` behind the scenes. As mentioned, you need to determine whether this is the right solution for your application.

The main thing to take from this example is how to set up an invoker interface, so that you can use another invoker if `DefaultInvoker` doesn't meet your needs. For example, Genesys also supplies `com.genesyslab.platform.commons.threading.SingleThreadInvoker`, which assigns a single dedicated thread to each protocol that enables it in your application. This may be useful in some cases where you have to parse XML messages.

The enhancement shown here will only require small changes to two of the classes in Protocol Manager, namely `ProtocolConfiguration` and `ProtocolFacility`.

To get started, let's declare a new multi-threaded parsing property in the `ProtocolConfiguration` class. In this example, the property is called `useMultiThreadedMessageParsing`. It is declared right after some ADDP and Warm Standby declarations:

```
[Java]

private boolean useAddp;
private FaultToleranceMode faultTolerance;
```

```
private Boolean useMultiThreadedMessageParsing;
```

Now you can code the getter and setter methods for the property itself, as shown here:

```
[Java]

public Boolean getUseMultiThreadedMessageParsing()
{
        return useMultiThreadedMessageParsing;
}

public void setUseMultiThreadedMessageParsing(Boolean value)
{
        useMultiThreadedMessageParsing = value;
}
```

Once you have made these changes, add an if statement to the `ApplyChannelConfiguration` method of the `ProtocolFacility` class so that your applications can selectively enable this property:

```
[Java]

private void applyChannelConfiguration(
        ProtocolConfiguration conf, ProtocolInstance instance)
{
        if (conf.getUri() != null)
        {
                instance.getProtocol().setEndpoint(
                        new Endpoint(conf.getName(), conf.getUri()));
        }

        if (conf.getUseMultiThreadedMessageParsing() != null &&
                        conf.getUseMultiThreadedMessageParsing().booleanValue())
        {
                instance.getProtocol().
                        setConnectionInvoker(DefaultInvoker.getSingletonInstance());
        }
...
```

Enabling `UseMultiThreadedMessageParsing` now calls `DefaultInvoker`.

To enable multi-threaded parsing, set the `useMultiThreadedMessageParsing` property of your Configuration object to true. Here is how to enable the new property for Stat Server messages:

```
[Java]

statServerConfiguration.setUseMultiThreadedMessageParsing(true);
```

Receiving Copies of Synchronous Server Messages

Most of the time, when you send a synchronous message to a server, you are satisfied to receive the response synchronously. But there can be situations where you want to receive a copy of the response asynchronously, as well. This section shows how to do that.

As in the previous section, this enhancement will only require small changes to the `ProtocolConfiguration` and `ProtocolFacility` classes.

To get started, let's declare a new `copyResponse` property in the `ProtocolConfiguration` class. You can put this declaration right after the `useMultiThreadedMessageParsing` declaration we created in the previous section:

```
[Java]

private boolean useAddp;
private FaultToleranceMode faultTolerance;
private Boolean useMultiThreadedMessageParsing;
private Boolean copyResponse;
```

Now you can code the getter and setter methods for the property itself, as shown here:

```
[Java]

public Boolean getCopyResponse()
{
        return copyResponse;
}

public void setCopyResponse(Boolean value)
{
        copyResponse = value;
}
```

It might be a good idea to let anyone using Protocol Manager know whether this property is enabled. One way to do this is to add it to the `toString` method in this class:

```
[Java]

public String toString()
{
        StringBuilder sb = new StringBuilder();
        .
        .
        .
        sb.append(MessageFormat.format(
                    "AddpClientTimeout: {0}\n", addpClientTimeout));
        sb.append(MessageFormat.format(
                    "AddpServerTimeout: {0}\n", addpServerTimeout));
        sb.append(MessageFormat.format(
                "CopyResponse: {0}\n", copyResponse));
        ...
```

Once you have made these changes, add an if statement to the `applyChannelConfiguration` method of the `ProtocolFacility` class so that your applications can selectively enable this property:

```
[Java]

private void applyChannelConfiguration(
        ProtocolConfiguration conf, ProtocolInstance instance)
{
        if (conf.getUri() != null)
        {
                instance.getProtocol().setEndpoint(
                        new Endpoint(conf.getName(), conf.getUri()));
        }

        if (conf.getCopyResponse() != null)
        {
                instance.getProtocol().setCopyResponse(
```

```
                        conf.getCopyResponse());
        }
...
```

To receive a copy of synchronous server messages, set the `CopyResponse` property of your Configuration object to true. Here is how to enable the new property for Stat Server messages:

```
[Java]

statServerConfiguration.setCopyResponse(true);
```

Supporting New Protocols

When the Platform SDK was first developed, it supported many, but not all, of the servers in the Genesys environment. As the SDK has matured, support has been added for more servers. As you might expect, a given version of the Protocol Manager Application Block only supports those servers that were supported by the Platform SDK at the time of its release. Since you may want to work with a server that is not currently supported by Protocol Manager, it can be helpful to know how add support for that server.

This section shows how the Protocol Manager Application Block supports the Stat Server Protocol. You can use it as a guide if you need to add support for other servers or protocols.

Adding support for the Stat Server Protocol involved three basic steps:

1. Create a new subclass of `ProtocolConfiguration` called `StatServerConfiguration`.
2. Create a new subclass of `ProtocolFacility` called `StatServerFacility`.
3. Add a statement to the `initialize` method of `ProtocolManagementServiceImpl` that associates `StatServerFacility` with `StatServerProtocol`.

The StatServerConfiguration Class

Here is the code for `StatServerConfiguration`:

```
[Java]

package com.genesyslab.platform.applicationblocks.commons.protocols;
import com.genesyslab.platform.reporting.protocol.StatServerProtocol;
import java.text.MessageFormat;

public final class StatServerConfiguration extends ProtocolConfiguration
{

    private String clientName;
    private Integer clientId;

    public StatServerConfiguration(String name)
    {
        super(name, StatServerProtocol.class);
    }

     public Integer getClientId()
    {
```

```
        return clientId;
    }

    public void setClientId(Integer clientId)
    {
        this.clientId = clientId;
    }

    public String getClientName()
    {
        return clientName;
    }

    public void setClientName(String clientName)
    {
        this.clientName = clientName;
    }

    public String toString()
    {
        StringBuilder sb = new StringBuilder(super.toString());

        sb.append(MessageFormat.format("ClientName: {0}\n", clientName));
        sb.append(MessageFormat.format("ClientId: {0}\n", this.clientId));

        return sb.toString();
    }
}
```

As you can see, this class imports the protocol object, but you will also need to use `MessageFormat` when we create the `toString()` method, so there must be an import statement for that class, as well:

```
[Java]

import com.genesyslab.platform.reporting.protocol.StatServerProtocol;
import java.text.MessageFormat;
```

Here are the class declaration and the field and constructor declarations. Stat Server requires client name and ID, so these must both be present in `StatServerConfiguration`:

```
[Java]

public final class StatServerConfiguration extends ProtocolConfiguration
{

    private String clientName;
    private Integer clientId;

    public StatServerConfiguration(String name)
    {
        super(name, StatServerProtocol.class);
    }
```

Here are the getter and setter methods for the client name and ID:

```
[Java]

public Integer getClientId()
{
        return clientId;
}
```

```
public void setClientId(Integer clientId)
{
        this.clientId = clientId;
}

public String getClientName()
{
        return clientName;
}

public void setClientName(String clientName)
{
        this.clientName = clientName;
}
```

And finally, the `toString()` method:

```
[Java]

public String toString()
{
        StringBuilder sb = new StringBuilder(super.toString());

        sb.append(MessageFormat.format("ClientName: {0}\n", clientName));
        sb.append(MessageFormat.format("ClientId: {0}\n", this.clientId));

        return sb.toString();
}
```

The StatServerFacility Class

Now we can take a look at the `StatServerFacility` class. Once again, we will start with the code for the entire class:

```
[Java]

package com.genesyslab.platform.applicationblocks.commons.protocols;

import com.genesyslab.platform.commons.protocol.Endpoint;
import com.genesyslab.platform.commons.protocol.Protocol;
import com.genesyslab.platform.reporting.protocol.StatServerProtocol;
import java.net.URI;

public final class StatServerFacility extends ProtocolFacility
{

    public void applyConfiguration(
                ProtocolInstance instance, ProtocolConfiguration conf)
    {
        super.applyConfiguration(instance, conf);
        StatServerConfiguration statConf = (StatServerConfiguration)conf;
        StatServerProtocol statProtocol =
                    (StatServerProtocol) instance.getProtocol();

/*
        if (statConf.getClientName() != null)
        {
            statProtocol.setClientName(statConf.getClientName());
        }
*/
```

```
        if (statConf.getClientId() != null)
        {
            statProtocol.setClientId(statConf.getClientId());
        }
    }

    public Protocol createProtocol(String name, URI uri)
    {
        return new StatServerProtocol(new Endpoint(name, uri));
    }
}
```

This class needs the following import statements:

```
[Java]

import com.genesyslab.platform.commons.protocol.Endpoint;
import com.genesyslab.platform.commons.protocol.Protocol;
import com.genesyslab.platform.reporting.protocol.StatServerProtocol;
import java.net.URI;
```

Here is how to declare the class:

```
[Java]

public final class StatServerFacility extends ProtocolFacility
```

There are two methods in this class. The first is `applyConfiguration`:

```
[Java]

public void applyConfiguration(
                        ProtocolInstance instance, ProtocolConfiguration conf)
{
        super.applyConfiguration(instance, conf);
        StatServerConfiguration statConf = (StatServerConfiguration)conf;
        StatServerProtocol statProtocol =
                                (StatServerProtocol) instance.getProtocol();

/*
        if (statConf.getClientName() != null)
        {
                statProtocol.setClientName(statConf.getClientName());
        }
*/
        if (statConf.getClientId() != null)
        {
                statProtocol.setClientId(statConf.getClientId());
        }
}
```

The second method is `createProtocol`:

```
[Java]

public Protocol createProtocol(String name, URI uri)
{
        return new StatServerProtocol(new Endpoint(name, uri));
}
```

Updating ProtocolManagementServiceImpl

To complete this enhancement, a single line of code was added to the initialize method of `ProtocolManagementServiceImpl`:

```
[Java]

private void Initialize()
{
        this.facilities.Add(typeof(ConfServerProtocol), new ConfServerFacility());
        this.facilities.Add(typeof(TServerProtocol), new TServerFacility());
        this.facilities.Add(typeof(InteractionServerProtocol), new
InteractionServerFacility());
        this.facilities.Add(typeof(StatServerProtocol), new StatServerFacility());
        this.facilities.Add(typeof(OutboundServerProtocol), new OutboundServerFacility());
        this.facilities.Add(typeof(LocalControlAgentProtocol), new LcaFacility());
        this.facilities.Add(typeof(SolutionControlServerProtocol), new ScsFacility());
        this.facilities.Add(typeof(MessageServerProtocol), new MessageServerFacility());
}
```

Architecture and Design

The Protocol Manager Application Block uses a service-based API. You can use this API to open and close your connection with Genesys servers and to dynamically reconfigure the parameters for a given protocol. Protocol Manager also includes built-in warm standby capabilities.

Protocol Manager uses a `ServerConfiguration` object to describe each server it manages.

.NET

Installing the Protocol Manager Application Block

Before you install the Protocol Manager Application Block, it is important to review the software requirements and the structure of the software distribution.

Building the Protocol Manager Application Block

The Platform SDK distribution includes a Genesyslab.Platform.ApplicationBlocks.Commons.Protocols.dll file that you can use as is. This file is located in the bin directory at the root level of the Platform SDK directory. To build your own copy of this application block, follow the instructions below:

1. Open the <Platform SDK Folder>\ApplicationBlocks\ProtocolManager folder.
2. Double-click ProtocolManager.sln.
3. Build the solution.

Working with the Protocol Manager Application Block

You can find basic information on how to use the Protocol Manager Application Block in the article on Connecting to a Server Using the Protocol Manager Application Block at the beginning of this guide.

Configuring ADDP

To enable ADDP, set the UseAddp property of your Configuration object to true. You can also set server and client timeout intervals, as shown here:

```
[C#]

statServerConfiguration.UseAddp = true;
statServerConfiguration.AddpServerTimeout = 10;
statServerConfiguration.AddpClientTimeout = 10;
```

Configuring Warm Standby

Hot standby is not designed to handle situations where both the primary and backup servers are down. It is also not designed to connect to your backup server if the primary server was down when you initiated your connection. However, in cases like these, warm standby will attempt to connect. In fact, warm standby will keep trying one server and then the other, until it does connect. Because of this, you will probably want to enable warm standby in your applications, even if you are already using hot standby.

You can enable warm standby in your application by setting your Configuration object's FaultTolerance property to FaultToleranceMode.WarmStandby, as shown here. You can also configure the backup server's URI, the timeout interval, and the number of times your application will attempt to contact the primary server before switching to the backup:

```
[C#]

statServerConfiguration.FaultTolerance = FaultToleranceMode.WarmStandby;
statServerConfiguration.WarmStandbyTimeout = 5000;
statServerConfiguration.WarmStandbyAttempts = 5;
statServerConfiguration.WarmStandbyUri = statServerBackupUri;
```

High-Performance Message Parsing

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

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

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

This section gives an example of how you can modify Protocol Manager to selectively enable multi-threaded parsing of incoming messages, in order to work around these kinds of performance issues. It is important to stress that you must take a careful look at which kind of multi-threading options to pursue in your applications, since your needs are specific to your situation.

> **Tip**
>
> Your application may also face other performance bottlenecks. For example, you may need more than one instance of the Message Broker Application Block if you handle large numbers of messages. For more information on how to configure Message Broker for high-performance situations, see the Using the Message Broker Application Block.

This example shows how to call `Genesyslab.Platform.Commons.Threading.DefaultInvoker`, which uses the .NET thread pool for your message parsing needs. As mentioned, you need to determine whether this is the right solution for your application, since, for example, the .NET thread pool may be heavily used for other tasks.

The main thing to take from this example is how to set up an invoker interface, so that you can use another invoker if `DefaultInvoker` doesn't meet your needs. For example, Genesys also supplies `Genesyslab.Platform.Commons.Threading.SingleThreadInvoker`, which assigns a single dedicated thread to each protocol that enables it in your application. This may be useful in some cases where you have to parse XML messages.

The enhancement shown here will only require small changes to two of the classes in Protocol Manager, namely `ProtocolConfiguration` and `ProtocolFacility`.

To get started, let's declare a new multi-threaded parsing property in the `ProtocolConfiguration` class. In this example, the property is called `useMultiThreadedMessageParsing`. It is nullable and is declared right after some ADDP and Warm Standby declarations:

```
[C#]

private bool? useAddp;
private FaultToleranceMode? faultTolerance;
private string addpTrace;
private bool? useMultiThreadedMessageParsing;
```

Now you can code the property itself, as shown here:

```
[C#]

public bool? UseMultiThreadedMessageParsing
{
        get { return this.useMultiThreadedMessageParsing; }
        set { this.useMultiThreadedMessageParsing = value; }
```

```
}
```

Once you have made these changes, add an if statement to the `ApplyChannelConfiguration` method of the `ProtocolFacility` class so that your applications can selectively enable this property:

```
[C#]

private void ApplyChannelConfiguration(ProtocolInstance entry, ProtocolConfiguration conf)
{
        if( conf.Uri != null )
        {
                entry.Protocol.Endpoint = new Endpoint(conf.Name, conf.Uri);
        }

        if (conf.UseMultiThreadedMessageParsing != null &&
conf.UseMultiThreadedMessageParsing.Value)
                {
                        entry.Protocol.SetConnectionInvoker(DefaultInvoker.InvokerSingleton);
        }
...
```

Enabling `UseMultiThreadedMessageParsing` now calls `DefaultInvoker`, which uses the .NET thread pool, as mentioned above.

To enable multi-threaded parsing, set the `UseMultiThreadedMessageParsing` property of your Configuration object to true. Here is how to enable the new property for Stat Server messages:

```
[C#]

statServerConfiguration.UseMultiThreadedMessageParsing = true;
```

Receiving Copies of Synchronous Server Messages

Most of the time, when you send a synchronous message to a server, you are satisfied to receive the response synchronously. But there can be situations where you want to receive a copy of the response asynchronously, as well. This section shows how to do that.

As in the previous section, this enhancement will only require small changes to the `ProtocolConfiguration` and `ProtocolFacility` classes.

To get started, let's declare a new `copyResponse` property in the `ProtocolConfiguration` class. You can put this declaration right after the `useMultiThreadedMessageParsing` declaration we created in the previous section:

```
[C#]

private bool? useAddp;
private FaultToleranceMode? faultTolerance;
private string addpTrace;
private bool? useMultiThreadedMessageParsing;
private bool? copyResponse;
```

Now you can code the property itself, as shown here:

```
[C#]
```

```
public bool? CopyResponse
{
        get { return this.copyResponse; }
        set { this.copyResponse = value; }
}
```

It might be a good idea to let anyone using Protocol Manager know whether this property is enabled. One way to do this is to add it to the `ToString` method overrides in this class:

```
[C#]

public override string ToString()
{
        StringBuilder sb = new StringBuilder();
        .
        .
        .
        sb.AppendFormat("AddpClientTimeout: {0}\n", this.addpClientTimeout.ToString());
        sb.AppendFormat("AddpServerTimeout: {0}\n", this.addpServerTimeout.ToString());
        sb.AppendFormat("CopyResponse: {0}\n", this.copyResponse.ToString());
        ...
```

Once you have made these changes, add an if statement to the `ApplyChannelConfiguration` method of the `ProtocolFacility` class so that your applications can selectively enable this property:

```
[C#]

private void ApplyChannelConfiguration(ProtocolInstance entry, ProtocolConfiguration conf)
{
        if( conf.Uri != null )
        {
                entry.Protocol.Endpoint = new Endpoint(conf.Name, conf.Uri);
        }

        if (conf.CopyResponse != null)
{
        entry.Protocol.CopyResponse = conf.CopyResponse.Value;
}
...
```

To receive a copy of synchronous server messages, set the `CopyResponse` property of your Configuration object to true. Here is how to enable the new property for Stat Server messages:

```
[C#]

statServerConfiguration.CopyResponse = true;
```

Supporting New Protocols

When the Platform SDK was first developed, it supported many, but not all, of the servers in the Genesys environment. As the SDK has matured, support has been added for more servers. As you might expect, a given version of the Protocol Manager Application Block only supports those servers that were supported by the Platform SDK at the time of its release. Since you may want to work with a server that is not currently supported by Protocol Manager, it can be helpful to know how add support for that server.

For example, early versions of Protocol Manager were developed before the Platform SDK supported

Universal Contact Server (UCS). This section shows how to add UCS support to the Protocol Manager Application Block. You can also use these instructions as a guide if you need to add support for other servers.

This enhancement involves three basic steps:

- Create a new subclass of `ProtocolConfiguration`. We will call this class `ContactServerConfiguration`.
- Create a new subclass of `ProtocolFacility` called `ContactServerFacility`.
- Add a statement to the `Initialize` method of `ProtocolManagementService` that associates the new `ContactServerFacility` class with `UniversalContactServerProtocol`.

Creating a ContactServerConfiguration Class

We will use the `StatServerConfiguration` class as a template for the new `ContactServerConfiguration` class. Here is the code for `StatServerConfiguration`:

```
[C#]

using System;
using System.Text;

using Genesyslab.Platform.Reporting.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    public sealed class StatServerConfiguration : ProtocolConfiguration
    {
        #region Fields

        private string clientName;
        private int? clientId;

        #endregion Fields

        public StatServerConfiguration(string name)
            : base(name, typeof(StatServerProtocol))
        {
        }

        #region Properties

                public string ClientName
                {
                        get { return this.clientName; }
                        set { this.clientName = value; }
                }

                public int? ClientId
                {
                        get { return this.clientId; }
                        set { this.clientId = value; }
                }

                #endregion Properties

        public override string ToString()
        {
```

```
            StringBuilder sb = new StringBuilder();
            sb.Append(base.ToString());

            sb.AppendFormat("ClientName: {0}\n", this.clientName);
            sb.AppendFormat("ClientId: {0}\n", this.clientId.ToString());

            return sb.ToString();
        }
    }
}
```

To get started, make a copy of `StatServerConfiguration.cs` and call it `ContactServerConfiguration.cs`. Rename the Platform SDK using statement and the class name, as shown here:

```
[C#]

using System;
using System.Text;
using Genesyslab.Platform.Contacts.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    public sealed class ContactServerConfiguration : ProtocolConfiguration
    {
    ...
```

The connection parameters required by Stat Server are different from those used by UCS. Instead of `clientName` and `clientId`, UCS requires `applicationName`. Like `clientName`, `applicationName` is of type string. One fairly simple way to modify this class is to delete all references to `clientId` and rename the references to `clientName` to `applicationName`. Make sure to retain the capitalization in the property name, which should become `ApplicationName`.

```
[C#]

using System;
using System.Text;
using Genesyslab.Platform.Contacts.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    public sealed class ContactServerConfiguration : ProtocolConfiguration
    {
        #region Fields

        private string applicationName;
        private int? clientId;

        #endregion Fields

        ...

        #region Properties

                public string ApplicationName
                {
                        get { return this.applicationName; }
                        set { this.applicationName = value; }
                }

                public int? ClientId
```

```
                {
                        get { return this.clientId; }
                        set { this.clientId = value; }
                }

                #endregion Properties

        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();
            sb.Append(base.ToString());

            sb.AppendFormat("applicationName: {0}\n", this.applicationName);
            sb.AppendFormat("ClientId: {0}\n", this.clientId.ToString());

            return sb.ToString();
        }
    }
}
```

The constructor also needs to be renamed. This code:

```
[C#]

public StatServerConfiguration(string name)
        : base(name, typeof(StatServerProtocol))
{
}
```

should be replaced with this:

```
[C#]

public ContactServerConfiguration(string name)
        : base(name, typeof(UniversalContactServerProtocol))
{
}
```

When you have made all of these changes, your new class should look like this:

```
[C#]

using System;
using System.Text;
using Genesyslab.Platform.Contacts.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    public sealed class ContactServerConfiguration : ProtocolConfiguration
    {
        #region Fields

        private string applicationName;

        #endregion Fields

        public ContactServerConfiguration(string name)
            : base(name, typeof(UniversalContactServerProtocol))
        {
        }
        #region Properties
```

```
        public string ApplicationName
        {
            get { return this.applicationName; }
            set { this.applicationName = value; }
        }

        #endregion Properties

        public override string ToString()
        {
            StringBuilder sb = new StringBuilder();
            sb.Append(base.ToString());

            sb.AppendFormat("ApplicationName: {0}\n", this.applicationName);

            return sb.ToString();
        }

    }
}
```

Creating a ContactServerFacility Class

The next step is to create a copy of `StatServerFacility.cs` and name it `ContactServerFacility.cs`. Here is what the `StatServerFacility` class looks like:

```
[C#]

using System;
using System.Text;

using Genesyslab.Platform.Commons.Collections;
using Genesyslab.Platform.Commons.Protocols;
using Genesyslab.Platform.Reporting.Protocols;
using Genesyslab.Platform.Commons.Logging;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    internal sealed class StatServerFacility : ProtocolFacility
    {
        public override void ApplyConfiguration(ProtocolInstance entry, ProtocolConfiguration
conf, ILogger logger)
        {
            base.ApplyConfiguration(entry, conf, logger);

            StatServerConfiguration statConf = (StatServerConfiguration)conf;
            StatServerProtocol statProtocol = (StatServerProtocol)entry.Protocol;

            if (statConf.ClientName != null)
            {
                statProtocol.ClientName = statProtocol.ClientName;
            }
            if (statConf.ClientId != null)
            {
                statProtocol.ClientId = statConf.ClientId.Value;
            }
        }

        public override ClientChannel CreateProtocol(string name, Uri uri)
        {
            return new StatServerProtocol(new Endpoint(name, uri));
```

```
        }
    }
}
```

Start by renaming the `using` statement and the class name:

```
[C#]

using System;
using Genesyslab.Platform.Commons.Logging;
using Genesyslab.Platform.Commons.Protocols;
using Genesyslab.Platform.Contacts.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    internal sealed class ContactServerFacility : ProtocolFacility
    ...
```

Rename `statConf` and `statProtocol`, giving them the correct configuration and protocol types:

```
[C#]

ContactServerConfiguration ucsConf = (ContactServerConfiguration)conf;
UniversalContactServerProtocol ucsProtocol =
        (UniversalContactServerProtocol)entry.Protocol;
```

And delete the references to `ClientId`:

```
[C#]

if (statConf.ClientId != null)
{
        statProtocol.ClientId = statConf.ClientId.Value;
```

Now you can rename `ClientName` to `ApplicationName`:

```
[C#]

if (ucsConf.ApplicationName != null)
{
        ucsProtocol.ApplicationName = ucsConf.ApplicationName;
}
```

When you are finished, you will have a new class that looks like this:

```
[C#]

using System;
using Genesyslab.Platform.Commons.Logging;
using Genesyslab.Platform.Commons.Protocols;
using Genesyslab.Platform.Contacts.Protocols;

namespace Genesyslab.Platform.ApplicationBlocks.Commons.Protocols
{
    internal sealed class ContactServerFacility : ProtocolFacility
    {
        public override void ApplyConfiguration(ProtocolInstance entry, ProtocolConfiguration
conf, ILogger logger)
        {
            base.ApplyConfiguration(entry, conf, logger);
```

```
            ContactServerConfiguration ucsConf = (ContactServerConfiguration)conf;
            UniversalContactServerProtocol ucsProtocol =
(UniversalContactServerProtocol)entry.Protocol;

            if (ucsConf.ApplicationName != null)
            {
                ucsProtocol.ApplicationName = ucsConf.ApplicationName;
            }
        }

        public override ClientChannel CreateProtocol(string name, Uri uri)
        {
            return new UniversalContactServerProtocol(new Endpoint(name, uri));
        }
    }
}
```

Updating ProtocolManagementService

To complete this enhancement, add a single line of code to the `Initialize` method of `ProtocolManagementService`:

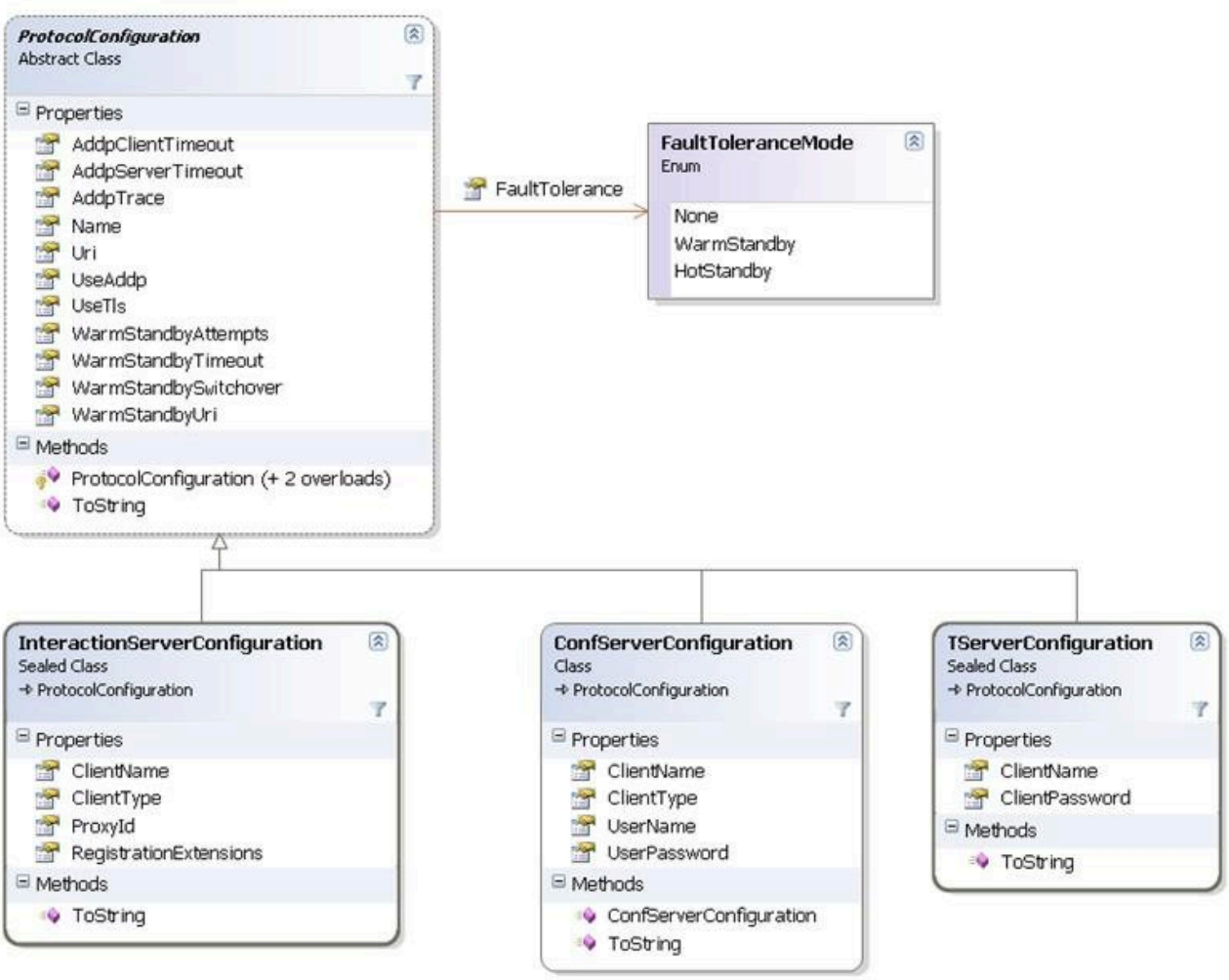
```
[C#]

private void Initialize()
{
        this.facilities.Add(typeof(ConfServerProtocol), new ConfServerFacility());
        this.facilities.Add(typeof(TServerProtocol), new TServerFacility());
        this.facilities.Add(typeof(InteractionServerProtocol), new
InteractionServerFacility());
        this.facilities.Add(typeof(StatServerProtocol), new StatServerFacility());
        this.facilities.Add(typeof(OutboundServerProtocol), new OutboundServerFacility());
        this.facilities.Add(typeof(LocalControlAgentProtocol), new LcaFacility());
        this.facilities.Add(typeof(SolutionControlServerProtocol), new ScsFacility());
        this.facilities.Add(typeof(MessageServerProtocol), new MessageServerFacility());
        this.facilities.Add(typeof(UniversalContactServerProtocol), new
ContactServerFacility());
}
```

Your copy of Protocol Manager now works with Universal Contact Server!

Architecture and Design

The Protocol Manager Application Block uses a service-based API. You can use this API to open and close your connection with Genesys servers and to dynamically reconfigure the parameters for a given protocol. Protocol Manager also includes built-in warm standby capabilities.

Protocol Manager uses a `ServerConfiguration` object to describe each server it manages. The figure below gives examples of the structure of some of these objects.



> **Tip**
>
> Any protocol can be reconfigured dynamically.