Platform SDK Developer's Guide

Using the Application Template Application Block

4/3/2025

Using the Application Template Application Block

> **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.

Java

The Application Template Application Block provides a way to read configuration options for applications in Genesys Administrator and to configure Platform SDK protocols. It also allows standard connection settings (including ADDP or TLS details) to be retrieved from Configuration Server, and helps with common features like setting up WarmStandby or assigning message filters. Primary Application Template functionality includes:

- `ClientConfigurationHelper` sets up client connections and configures WarmStandby.
- `ServerConfigurationHelper` sets up server connections.
- `GConfigTlsPropertyReader` extracts TLS-related option values from configuration objects, and is intended to be used together with `com.genesyslab.platform.commons.connection.tls.TLSConfigurationParser`.
- `FilterConfigurationHelper` helps to bind message filters with protocol objects.
- `GFApplicationConfigurationManager` monitors the application configuration from Configuration Server and provides notification of any updates to options for your custom application, options of connected servers, or options of their host objects. If Log4j2 logging framework exists, then this component also enables Log4j2 configuration based on the application logging options in Configuration Manager. For more information, refer to the Additional Logging Features article.
- `ClusterClientConfigurationHelper` helps create and configure the Cluster Protocol Application Block.

Setting Up a Client Connection

Application Template helper creates `com.genesyslab.platform.Endpoint` instance with initialized configuration properties. Details about how to specify required options in Configuration server are available below. In order to retrieve specified options from Configuration Server, user should read `IGApplicationConfiguration` object where this properties are stored.

Sample:

```
//create Configuration Service
ConfServerProtocol confProtocol = new ConfServerProtocol(new Endpoint(host,port));
confProtocol.setUserName("...");
confProtocol.setClientName("...");
confProtocol.setClientApplicationType(CfgAppType.CFGSCE.ordinal());
IConfService confService = ConfServiceFactory.createConfService(confProtocol);
confProtocol.open();


//read your application options
String appName = "my-app-name";
CfgApplication cfgApplication = confService.retrieveObject(CfgApplication.class,
                new CfgApplicationQuery(appName));

GCOMApplicationConfiguration appConfiguration = new
GCOMApplicationConfiguration(cfgApplication);

//get particular connection definition
IGAppConnConfiguration connConfig = appConfiguration.getAppServer(CfgAppType.CFGStatServer);

//returns configured endpoint.
Endpoint epStatSrv = ClientConfigurationHelper.createEndpoint(appConfiguration,
    connConfig, connConfig.getTargetServerConfiguration());

//use protocol with configured endpoint
StatServerProtocol statProtocol = new StatServerProtocol(epStatSrv);
statProtocol.setClientName(clientName);
statProtocol.open();
```

> ### Tip
> Instructions on how to enable TLS using the Application Template Application Block are part of the TLS-specific documentation.

Configuring WarmStandby

Note: This section was updated for Platform SDK for Java release 8.5.102.02. For earlier releases, expand the "Legacy Content" text at the end of this section.

The Application Template helper `createWarmStandbyConfigEx()` allows you to create a configuration for the new implementation of the warm standby, as illustrated here:

```
String appName = "my-app-name"
CfgApplication cfgApplication = confService.retrieveObject(
           CfgApplication.class, new CfgApplicationQuery(appName));

GCOMApplicationConfiguration appConfiguration =
           new GCOMApplicationConfiguration(cfgApplication);

IGAppConnConfiguration connConfig = appConfiguration.getAppServer(CfgAppType.CFGStatServer);

//Helper method for new WarmStandby
```

```
WSConfig wsConfig = ClientConfigurationHelper.createWarmStandbyConfigEx(appConfiguration,
connConfig);

StatServerProtocol statProtocol = new StatServerProtocol();
statProtocol.setClientName(clientName);

WarmStandby warmStandby = new WarmStandby(statProtocol);
warmStandby.setConfig(wsConfig);
warmStandby.autoRestore();
```

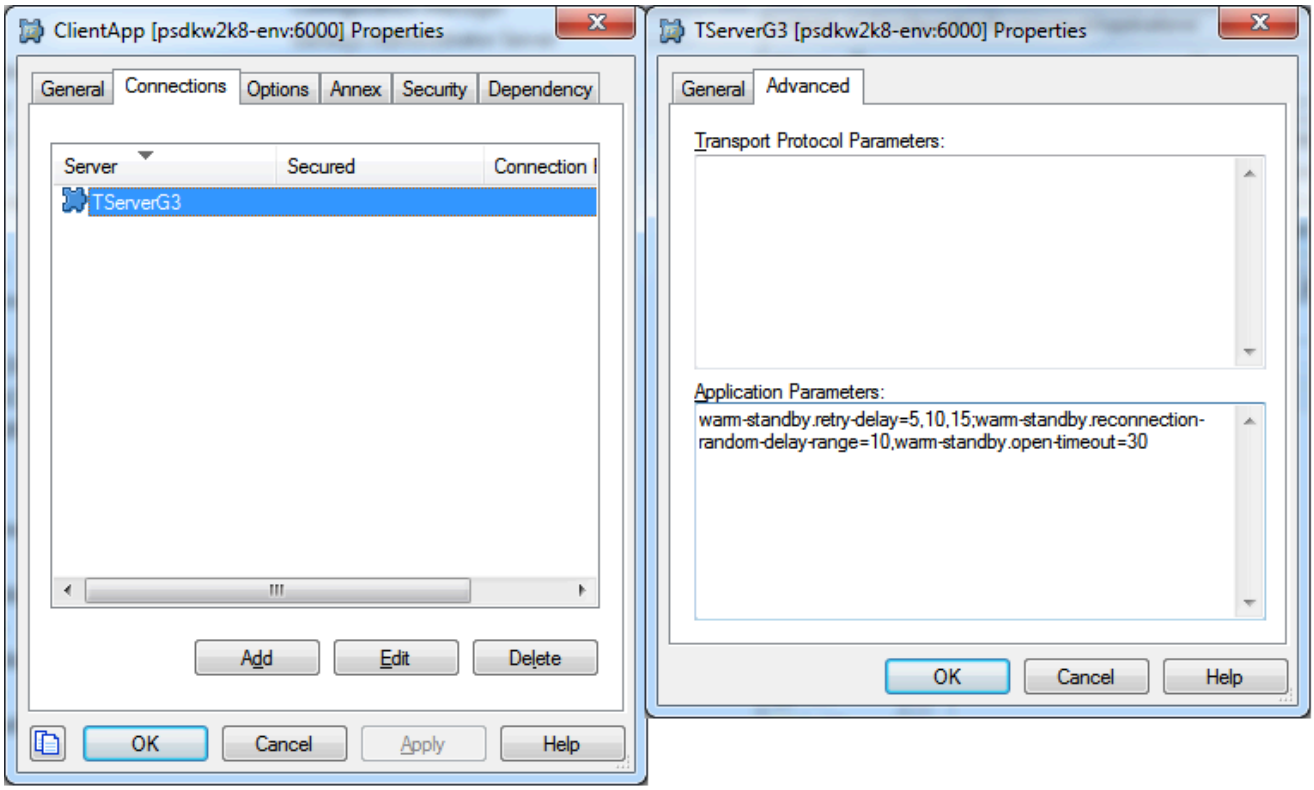
Configuration in Configuration Manager looks quite similar to the classic Warm Standby configuration with one difference: there are special timing parameters which are quite different than "Reconnection Timeout" and "Reconnection Attempts" and thus are specified apart from them. The "Reconnection Timeout" and "Reconnection Attempts" are not used in new Warm Standby Configuration.

Client Connection options:

Name	Values, in Seconds
warm-standby.retry-delay	5, 10, 15
warm-standby.reconnection-random-delay-range	10
warm-standby.open-timeout	30

See Warm Standby documentation and the API Reference guide for details about how these timing options customize Warm Standby behavior.

Options can be specified in Configuration Manager as shown below:



The last option is specified in Backup Server Applications on the Options tab. This characteristic of the backup server describes how much time is required for the server to step into primary mode.

Options Tab Section	Value, in Seconds
warm-standby	backup-delay=5

[+] Legacy Content

Configuring WarmStandby (Legacy Content)

Note: This section only applies to Platform SDK releases **prior** to:

- Java - 8.5.102.02
- .NET - 8.5.102.03

This section describes how to configure WarmStandby service with Application Template helpers. To find out how to use the WarmStandby service, see the corresponding Using the Warm Standby Application Block article.

Application Template helper method creates configuration for `com.genesyslab.platform.applicationblocks.warmstandby.WarmStandbyService`. The result includes parameters for the connection to primary and backup servers defined in the specified application configuration information.

Sample:

```
String appName = "my-app-name";
CfgApplication cfgApplication = confService.retrieveObject(
                  CfgApplication.class, new CfgApplicationQuery(appName));

GCOMApplicationConfiguration appConfiguration = new
GCOMApplicationConfiguration(cfgApplication);
IGAppConnConfiguration connConfig = appConfiguration.getAppServer(CfgAppType.CFGStatServer);

WarmStandbyConfiguration wsConfig =
ClientConfigurationHelper.createWarmStandbyConfig(appConfiguration, connConfig);

StatServerProtocol statProtocol = new StatServerProtocol(wsConfig.getActiveEndpoint());
statProtocol.setClientName(clientName);
WarmStandbyService wsService = new WarmStandbyService(statProtocol);

wsService.applyConfiguration(wsConfig);
wsService.start();
statProtocol.beginOpen();
```

Setting Up a Server Channel

Similar to the creation of the client connection, provide the `IGApplicationConfiguration` object to the helper class.

Sample:

```
Endpoint endpoint = ServerConfigurationHelper.createListeningEndpoint(appConfiguration,
appConfiguration.getPortInfo("default"));
ExternalServiceProtocolListener serverChannel = new ExternalServiceProtocolListener(endpoint);
```

This helper creates an Endpoint instance initialized with properties like the listening TCP port number, ADDP parameters, and so on.

Setting Up TLS

Instructions on how to enable TLS using the Application Template Application Block are part of the TLS-specific documentation. Please refer to that article for details about how to enable secure connections using the Application Template.

Also, see how to Configure TLS Parameters in Configuration Manager for a client or server channel.

Enabling Message Filtering

Using the Debug Log Level in Platform SDK protocol may affect Application performance due to the huge amount of log information output. It is possible to setup message filters for a protocol object, where the filter is configured in Genesys Administrator. This way, production applications will be able to provide appropriate log traces for troubleshooting without hurting performance with overly verbose logging.

See how to setup message filters for additional details.

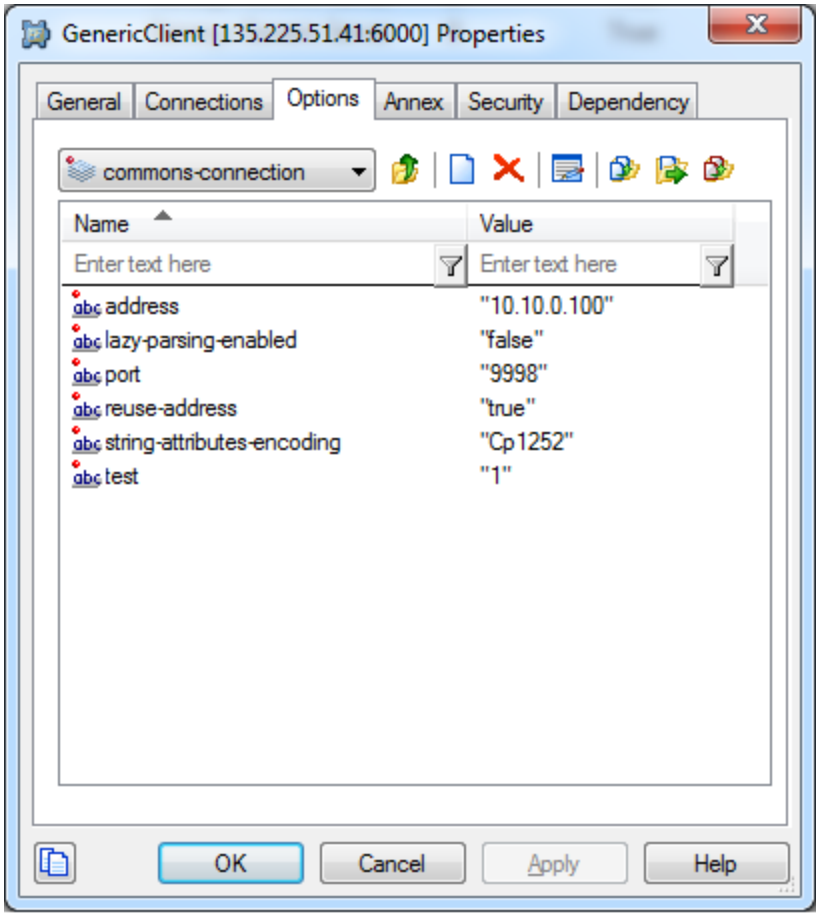
Defining Configuration Options in Genesys Administrator

Options can be specified in the `CfgApplication` object using Genesys Administrator. There are several possible option locations in the `CfgApplication` object:

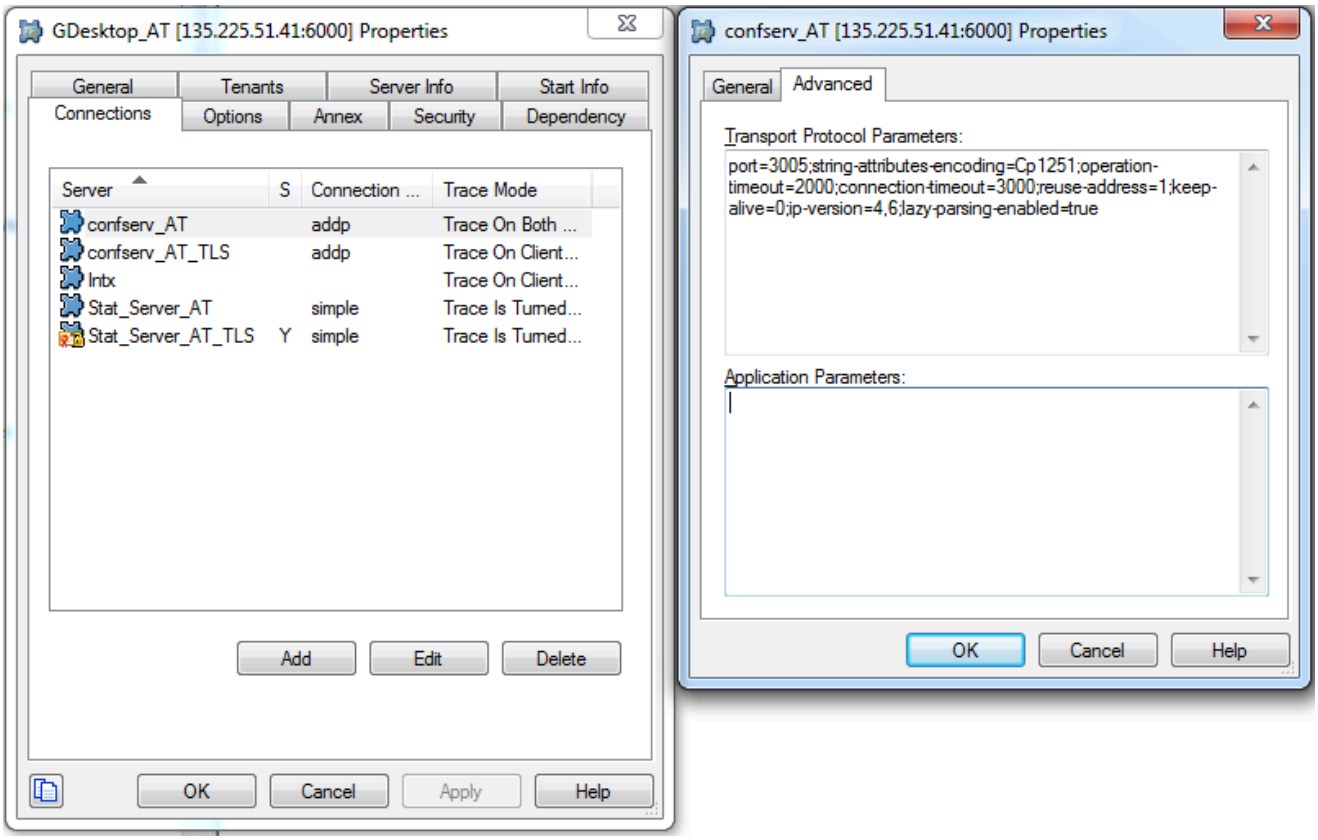
- *Options* tab
- *Annex* tab
- *Connections* parameters
- *Port* parameters
- *Host* annex

Common Options

Here is how options could be specified on the *Options* tab:



Here is how options could be specified for a particular connection (using the *Connection* parameters):



A complete options list is provided in the table below.

Section	Option	Description
commons-connection	string-attributes-encoding	Specifies encoding for string attributes.
	lazy-parsing-enabled	Boolean value. Enables or disables lazy parsing of properties, for which lazy parsing possibility is enabled in protocol. Currently used in Configuration Server protocol, enabled by default.
	address	Host bind option, specifies host from which connection should be made
	port	Port bind option, specifies port from which connection should be made
	backup-port	Port bind option, specifies port from which connection should be made (bind to) for backup server.
	operation-timeout	Integer value. Timeout for operations like stopReading and

Section	Option	Description
		resumeReading. Timeout is specified in milliseconds.
	connection-timeout	Integer value. Sets connection timeout option for the local socket to be opened by connection. Timeout is specified in milliseconds.
	reuse-address	Boolean value. Sets SO_REUSEADDR option for the local socket to be opened by connection
	keep-alive	Boolean value. Sets SO_KEEPALIVE option for the local socket to be opened by connection.
ucs-protocol	use-utf-for-responses	Boolean value. If set to false, UCSprotocol will add 'tkv.multibytes'='false' pair in Request KVlist of the message. It is false by default.
	use-utf-for-requests	Boolean value. If set to true, all string values of each KVlist will be packed as UtfStrings (in "UTF-16BE" encoding), instead of common strings. It is true by default.
webmedia-protocol	target-xml-version	Version of XML which is used to transport WebMedia protocol messages. See corresponding javax.xml.transform.OutputKeys.Version property for details.
	replace-illegal-unicode-chars	Boolean value. Enables or disables replacing of the illegal unicode chars in Webmedia XML messages.
	illegal-unicode-chars-replacement	String to replace illegal unicode chars

IPV6 Options

The IPV6 usage can be enabled with "enable-ipv6" option.

Section	Option	Description
common	enable-ipv6	Turns IPv6 support on/off.

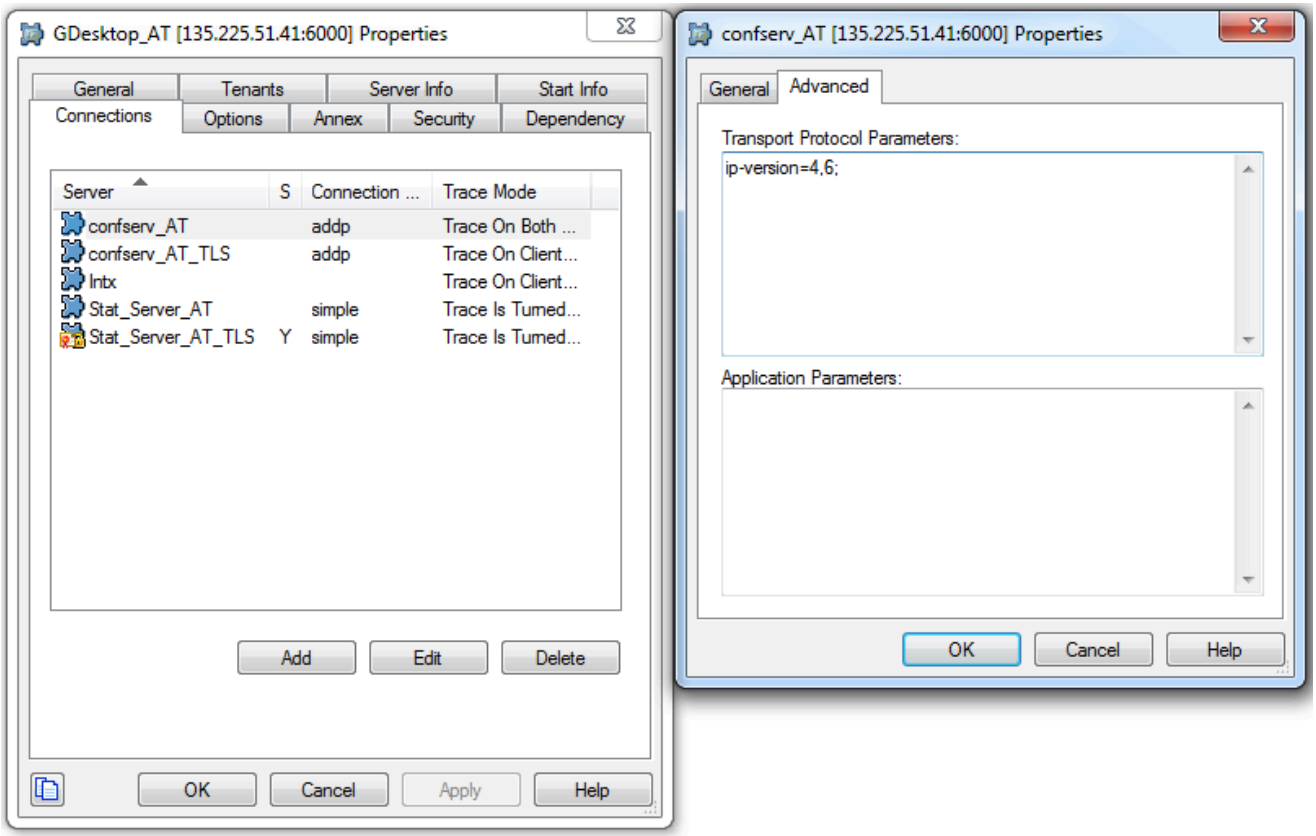
Section	Option	Description
		Possible values: 0 (default, implied) and 1. If set to 0, IPv6 support would be disabled, even if supported by OS/ platform.

The `ip-version` constant specifies the order in which connection attempts will be made to IPv6 and IPv4 addresses. Possible values include the following strings:

- "4,6" (default)
- "6,4"

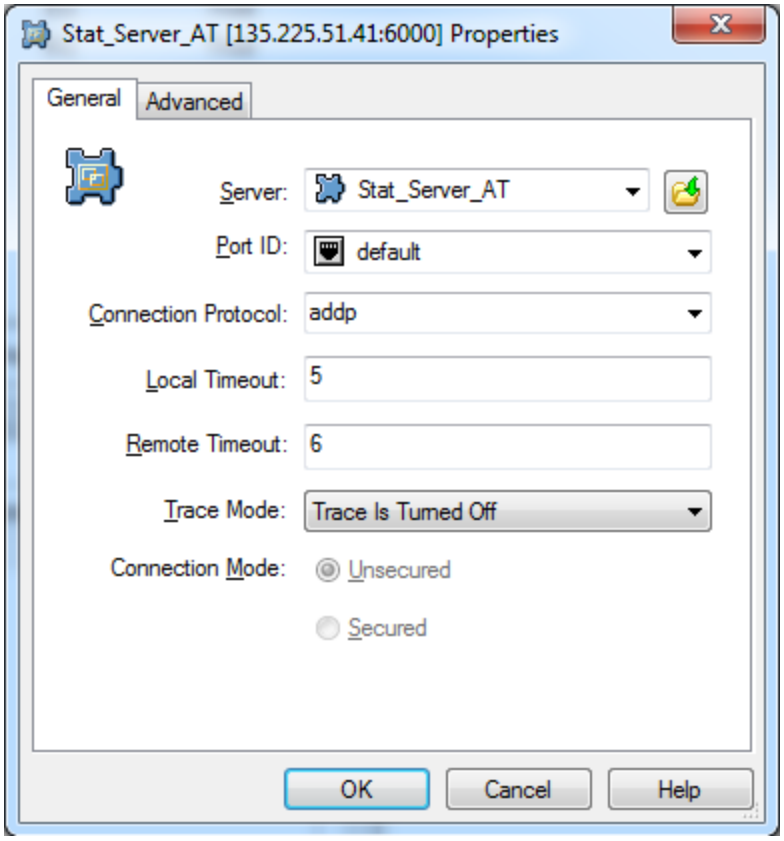
This setting has no effect on the connection if `enable-ipv6` is set to 0; warning would be logged. Has no effect on server side; warning would be logged.

Option values should be specified within the connection transport properties:



ADDP Options

ADDP options, which can be handled with the Application Template helper, should be specified on the *Connections* tab.



TLS Options

See the article on Configuring TLS Parameters in Configuration Manager for more information about TLS options.

Cluster Connection Configuration Helpers

This section describes Application Template helper methods that support the configuration of cluster connections to Genesys server, including use cases and samples.

Cluster Connection Configuration Types

There are two types of Genesys server cluster configuration available in Configuration Server: client-aligned configuration, and node-aligned configuration.

Both types provide information about addresses of the target servers in a cluster, and connection options used to communicate with them. You must choose the proper helper to use for each type of cluster configuration.

Client-aligned configuration methods:

- `ClusterClientConfigurationHelper.createClusterProtocolEndpoints(IGApplicationConfiguration appConfig, CfgAppType serverType)`
- `ClusterClientConfigurationHelper.createClusterProtocolEndpoints(IGApplicationConfiguration appConfig, IGAppConnConfiguration clusterConn, CfgAppType serverType)`

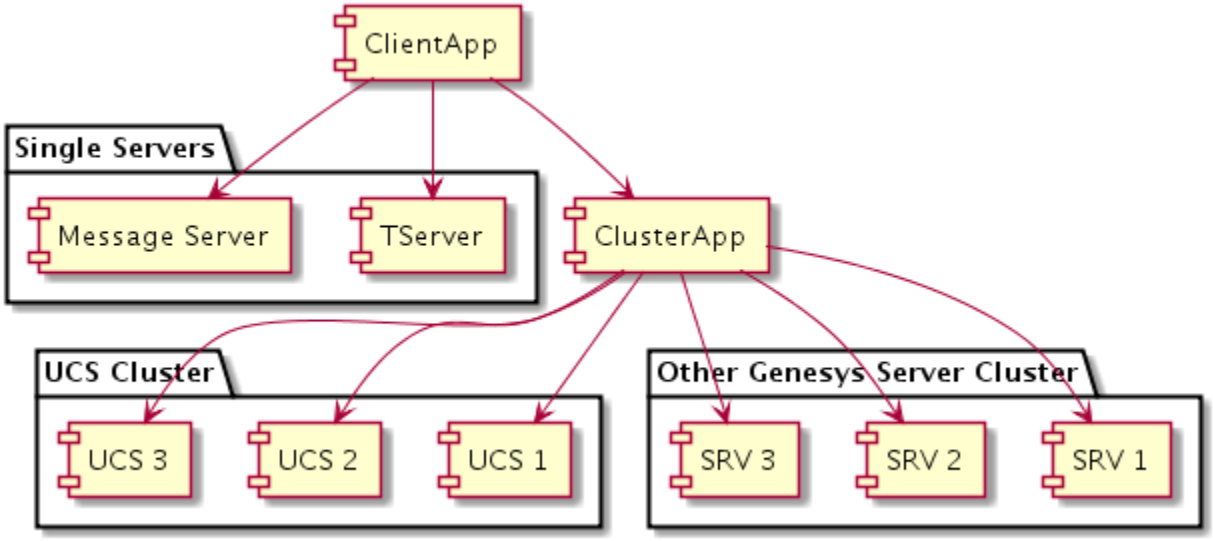
Node-aligned configuration methods:

- `ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(IConfService confService, IGApplicationConfiguration appConfig, CfgAppType serverType)`
- `ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(IConfService confService, IGApplicationConfiguration appConfig, IGAppConnConfiguration clusterConn, CfgAppType serverType)`

Client-Aligned Configuration Samples

Sample 1

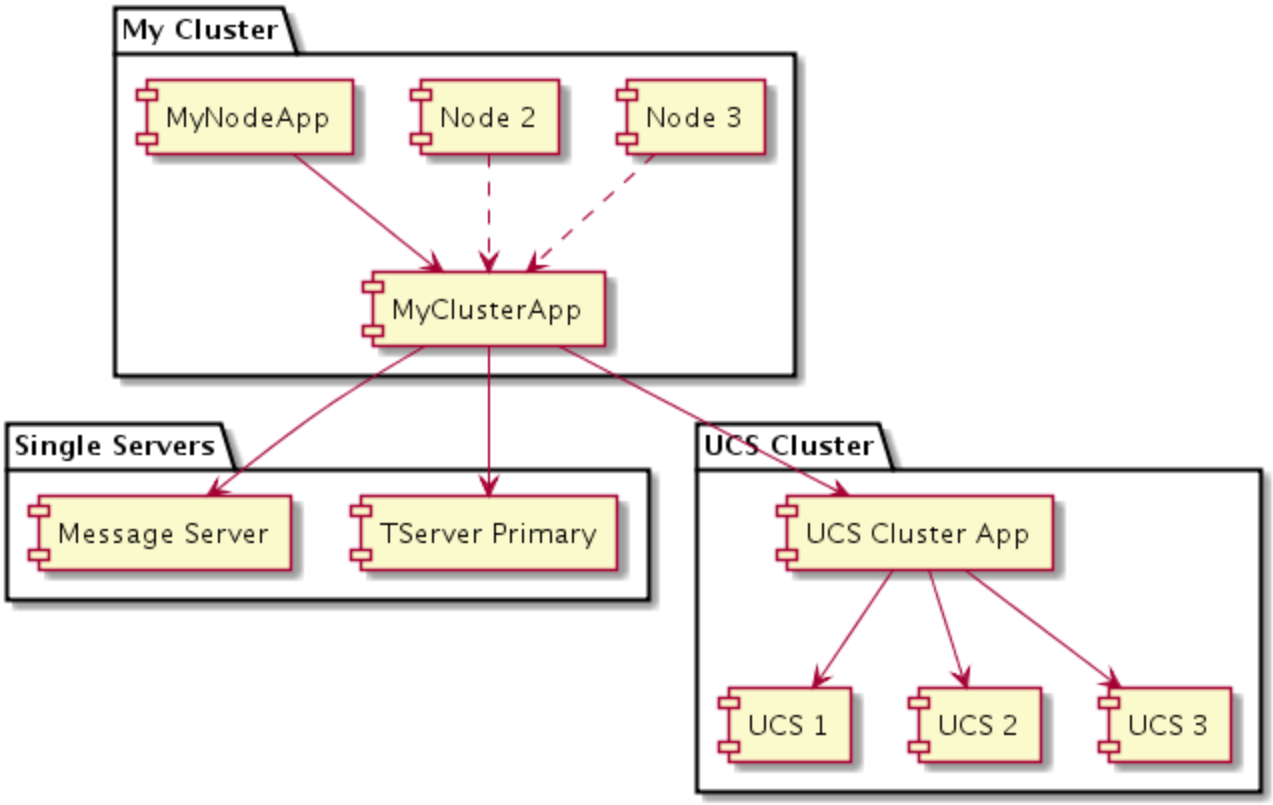
In the first sample scenario, a "ClientApp" Application is connected to the "ClusterApp'" virtual application (type=CFGApplicationCluster) that has connections to cluster nodes of one ore more clusters.




```
ClusterProtocol protocol ...
List<WSConfig> nodesList =
ClusterClientConfigurationHelper.createClusterProtocolEndpoints(ClientApp,
CfgAppType.CFGContactServer);
protocol.setNodes(nodesList);
```

Sample 2

In the second sample scenario, the "MyNodeApp" Application is a cluster node that is connected to the "MyClusterApp" virtual application. In this case, "MyClusterApp" is a shared store of connection configurations for all cluster nodes, which can have connections to other clusters like "UCS" as well as stand-alone servers.




```
ClusterProtocol protocol ...
List<WSConfig> nodesList =
ClusterClientConfigurationHelper.createClusterProtocolEndpoints(MyClusterApp,
clusterAppConnection, CfgAppType.CFGContactServer);
protocol.setNodes(nodesList);
```

Connection Options

You can use Genesys Administrator to define connection parameter options (such as ADDP, ip-version, strings encoding, or TCP socket options) that are used for all nodes in a cluster, or to define connection parameter for a particular node that will overriding common parameters.

- Common Options: Specified in the connection from ClientApp to ClusterApp (or from OwnClusterApp if application is a node of some type of cluster).
- Node-Specific Options: can be specified in Configuration Manager in connection from ClusterApp to NodeApp.

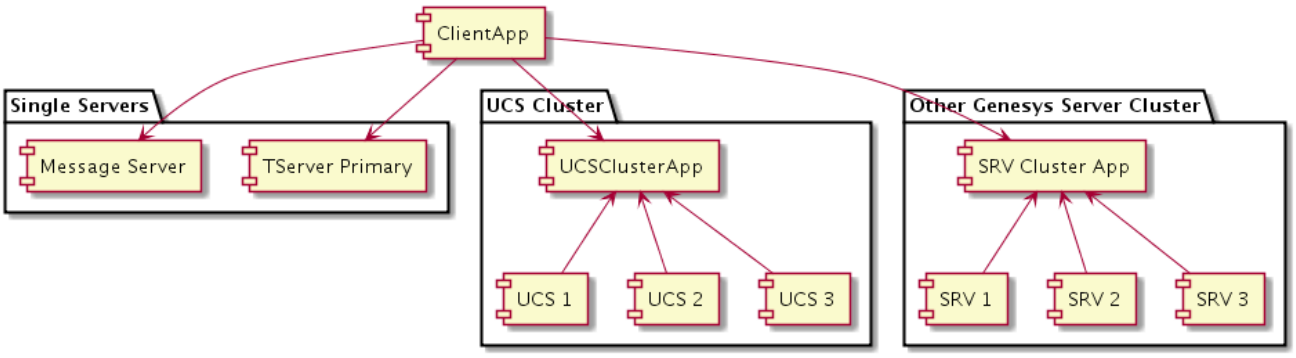
See eServices Load Balancing Business Continuity for examples.

Node-Aligned Configuration

Sample 3

In the third sample scenario, the "ClientApp" Application is connected to the "UCSClusterApp" virtual

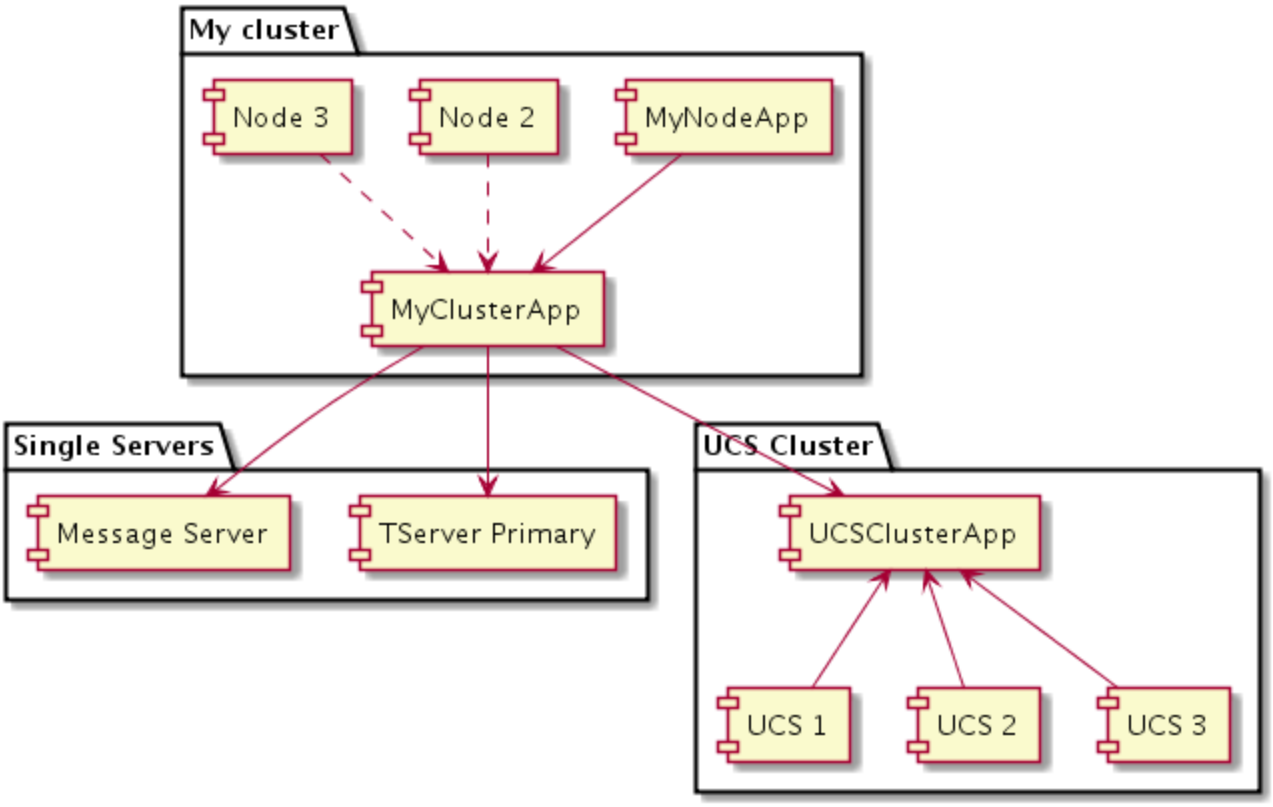
application that groups UCS nodes. Unlike previous configurations, the cluster nodes in this scenario are also connected to the "UCSClusterApp" virtual application.




```
ClusterProtocol protocol ...
List<WSConfig> nodesList =
ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(confService, ClientApp,
CfgAppType.CFGContactServer);
protocol.setNodes(nodesList);
```

Sample 4

In the fourth sample scenario, the "MyNodeApp" Application is a cluster node. The "MyClusterApp" application has a connection to the UCS cluster ("UCSClusterApp" application). UCS nodes also have connections to "UCSClusterApp".




```
ClusterProtocol protocol ...
List<WSConfig> nodesList =
ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(confService, MyNodeApp,
clusterAppConnection, CfgAppType.CFGContactServer);
protocol.setNodes(nodesList);
```

Connection Options

You can use Genesys Administrator to define connection parameter options (such as ADDP, ip-version, strings encoding, or TCP socket options), but in this scenario this configuration is applied for all nodes in a cluster.

- Common Options: Specified in the connection from ClientApp to ClusterApp.

Client Connection Active Nodes Randomizer

> **Important**
>
> Dynamic updates to the cluster configuration are still possible with this approach. Your application should use the `shuffler.setNodes(nodes)` function to change the list of cluster nodes, instead of using Cluster Protocol methods directly.

The randomizer helper component supports cluster load balancing from client applications, by allowing your applications to work with a server cluster without creating live connections to all of its nodes. This also prevents many clients from being stuck on a single connection, to resolve issues such as a single server overloading on startup, and can be of critical importance in use cases where the number of clients could be in the tens of thousands.

This helper takes a list of cluster nodes, and performs periodical endpoint rotation. The existing Cluster Protocol functionality to update its nodes configuration dynamically..

For each new node that is added:

1. a new instance of the related PSDK protocol class is created
2. asynchronous open is started
3. after the node has connected to its server, the node is added to load balancer

For each node that is removed:

1. the removed node is immediately excluded from the load balancer
2. the node is scheduled to be closed after its protocol timeout delay (which allows responses to be delivered for in-progress requests)

The randomizer uses the `Collections.shuffle(nodes)` method to randomize the sequence of given nodes.When updating the protocol node configuration, the randomizer uses a round-robin rotation over a whole list of randomly-sorted cluster nodes, choosing a subset of given number of elements. This allows the protocol to avoid breaking all connections at a single moment, and to stay online during switchover.

The randomizer component also has an optional ability to attach truncated nodes' endpoints as backup endpoints for selected ones. This allows the internal WarmStandby service to quickly switchover from an unresponsive node.

Sample usage of this component could look like the following:

```
 final UcsClusterProtocol ucsNProtocol = new UcsClusterProtocolBuilder().build();
 List<WSConfig> nodes = ...;
 ClusterNodesShuffler shuffler = new ClusterNodesShuffler(ucsNProtocol, 2); // creates nodes
shuffler for 2 active connections
 shuffler.setUseBackups(true); // - lets the shuffler to use truncated nodes endpoints as
backup endpoints for the selected ones
 shuffler.setNodes(nodes); // - initializes shuffler with the whole list of cluster nodes

 ucsNProtocol.open();

 TimerActionTicket shufflerTimer = TimerFactory.getTimer().schedule(3000, 3000, shuffler); //
- schedules shuffling operation with 3 secs delay and 3 secs period

 // Do the business logic on the cluster protocol...
 // In case of update in the cluster configuration, application should use
'shuffler.setNodes(newNodes)'
 //     instead of ClusterProtocol's methods related to nodes configuration.

 // Shutdown:
 shufflerTimer.cancel();
 ucsNProtocol.close();
```

Code Samples

Simple Client Application Connecting to Any UCS Cluster

This sample checks the connection configuration for "WCC mode" UCS 9.0 cluster, then for "WDE mode" UCS cluster, and then for "legacy mode" connection to UCS (pri/bck) server.

```
// Take "my application configuration" from context, or read it in a way like this:
IGApplicationConfiguration myApp = new GCOMApplicationConfiguration(
        confService.retrieveObject(CfgApplication.class,
                new CfgApplicationQuery(myAppName)));

// For the first, try UCS 9 connection cluster:
List<WSConfig> conns = ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(
        confService, myApp, CfgAppType.CFGContactServer);
if (conns == null || conns.isEmpty()) {
    // If there is no UCS 9 cluster connected, then we try older UCS cluster, or simple UCS
connection(s):
    conns = ClusterClientConfigurationHelper.createClusterProtocolEndpoints(
            myApp, CfgAppType.CFGContactServer);
}

System.out.println("Connections: " + conns);
```

WCC-Based Cluster Node Application Connecting to Any UCS Cluster

This sample works in context of WCC.

```
// Take "my application configuration" from context, or read it in a way like this:
final IGApplicationConfiguration myApp = new GCOMApplicationConfiguration(
        confService.retrieveObject(CfgApplication.class,
                new CfgApplicationQuery(myAppName)));

IGApplicationConfiguration myClusterApp = null;
// if we do not have 'myClusterApp' from WCC context, we may take it by this way:
final List<IGAppConnConfiguration> clusters = GApplicationConfiguration
        .getAppServers(myApp.getAppServers(), CfgAppType.CFGApplicationCluster);
if (clusters != null) {
    if (clusters.size() == 1) {
        myClusterApp = clusters.get(0).getTargetServerConfiguration();
        log.infoFormat(
                "Application is recognized as a node of cluster ''{0}''",
                myClusterApp.getApplicationName());
    } else if (clusters.size() > 1) {
        log.error("Application has more than one application cluster connected"
                + " - its treated as a standalone app");
    }
}

// Select application cluster connection start point:
final IGApplicationConfiguration connSrc = (myClusterApp != null) ? myClusterApp : myApp;

// For the first, try UCS 9 connection cluster:
List<WSConfig> conns = ClusterClientConfigurationHelper.createRefClusterProtocolEndpoints(
        confService, connSrc, CfgAppType.CFGContactServer);
if (conns == null || conns.isEmpty()) {
    // If there is no UCS 9 cluster connected, then we try older UCS cluster, or simple UCS
connection(s):
    conns = ClusterClientConfigurationHelper.createClusterProtocolEndpoints(
            connSrc, CfgAppType.CFGContactServer);
```

```
}

System.out.println("Connections: " + conns);
```

Client Nodes Randomizer Usage Sample

Application code using the randomizer component may look like the following sample:

```
 final List<WSConfig> nodes = ...;
 final UcsClusterProtocol ucsNProtocol =
         new UcsClusterProtocolBuilder()
                 .build();
 ucsNProtocol.setTimeout(5000); // sets protocol timeout to 5 secs
 ucsNProtocol.setClientName("MyClientName");
 ucsNProtocol.setClientApplicationType("MyAppType");

 ClusterNodesShuffler shuffler = new ClusterNodesShuffler(ucsNProtocol, 2); // creates nodes
shuffler for 2 active connections
 TimerActionTicket shufflerTimer = null;

 try {
     shuffler.setNodes(nodes); // - initializes shuffler with the whole list of cluster nodes.
     ucsNProtocol.open();
     shufflerTimer = TimerFactory.getTimer().schedule(3000, 3000, shuffler); // - schedules
shuffling operation with 3 secs delay and 3 secs period

     // do the business logic on the cluster protocol...
     for (int i = 0; i < 200; i++) {
         EventGetVersion resp = (EventGetVersion)
ucsNProtocol.request(RequestGetVersion.create());
         System.err.println("Resp from: " + resp.getEndpoint());
         Thread.sleep(300);
     }
 } finally {
     if (shufflerTimer != null) {
         shufflerTimer.cancel();
         shufflerTimer = null;
     }
     ucsNProtocol.close();
 }
```

Handling Updates From Config Server

The `GFApplicationConfigurationManager` component monitors Config Server for updates and provides notifications about changes in applications.

You should register for updates at `GFApplicationConfigurationManager`.

```
GFApplicationConfigurationManager appManager = ...
appManager.register(new ClientConnEventListener());
appManager.init();
```

In the handle (`GFAppCfgEvent` event) method implementation, create a new connection configuration using one of the helpers mentioned above:

```
import com.genesyslab.platform.apptemplate.application;

public class ClientConnEventListener extends GFAppCfgEventListener {
```

```
    private UcsClusterProtocol ucsProtocol;
    @Override
    public void handle(GFAppCfgEvent event) {
        //get new application configuration
        IGApplicationConfiguration appconfig = event.getAppConfig();

        //create new connection configuration
        List<WSConfig> conns =
ClusterClientConfigurationHelper.createClusterProtocolEndpoints(appconfig,
CfgAppType.CFGContactServer);

        //apply connection configuration
        ucsProtocol.setNodes(conns);
    }
}
```

> **Important**
>
> WCC-based cluster configuration does not support an update handler at this time. Subscribing to updates in this case will lead to Config Server overloading, so customers are encouraged to make direct requests to Config Server to actualize the cluster configuration before opening `ClusterProtocol`.

.NET

The Application Template Application Block provides a way to read configuration options for applications in Genesys Administrator and to configure Platform SDK protocols. It also allows standard connection settings (including ADDP or TLS details) to be retrieved from Configuration Server, and helps with common features like setting up WarmStandby or assigning message filters. Primary Application Template functionality includes:

- `ClientConfigurationHelper` sets up client connections and configures WarmStandby.
- `ServerConfigurationHelper` sets up server connections.

Setting Up a Client Connection

Application Template helper creates an `Endpoint` instance with initialized configuration properties. Details about how to specify required options in Configuration server are available below. In order to retrieve specified options from Configuration Server, user should read `IGApplicationConfiguration` object where this properties are stored.

Sample:

```
//create Configuration Service
ConfServerProtocol confProtocol = new ConfServerProtocol(new Endpoint(host,port)){
  UserName = "...",
  ClientName = "...",
  ClientApplicationType= (int)cfgAppType,
```

```
  UserPassword = "..."
};
IConfService confService = ConfServiceFactory.CreateConfService(confProtocol);
confProtocol.Open();

//read your application options
var myApplicationName = "...";
var applicationConfiguration = new GCOMApplicationConfiguration(
      new CfgApplicationQuery(confService){ Name = myApplicationName }.ExecuteSingleResult()
);

CfgAppType myApplicationType = default(CfgAppType);
var applicationEndPoint = ClientConfigurationHelper.CreateEndpoint(applicationConfiguration,
      applicationConfiguration.GetAppServer(myApplicationType),
      applicationConfiguration.GetAppServer(myApplicationType).TargetServerConfiguration);

//use protocol with configured endpoint
StatServerProtocol statProtocol = new StatServerProtocol(applicationEndPoint);
statProtocol.ClientName = clientName;
statProtocol.Open();
```

Configuring WarmStandby

The Application Template helper `CreateWarmStandbyConfigEx()` allows you to create a configuration for the new implementation of the warm standby, as illustrated here:

```
var myApplicationName = "…";
CfgAppType myApplicationType = default(CfgAppType);
var applicationConfiguration = new GCOMApplicationConfiguration(
      new CfgApplicationQuery(confService){ Name = myApplicationName }.ExecuteSingleResult()
);
WSConfig warmStandbyConfig = ClientConfigurationHelper.CreateWarmStandbyConfigEx(
      applicationConfiguration,applicationConfiguration.GetAppServer(myApplicationType));
```

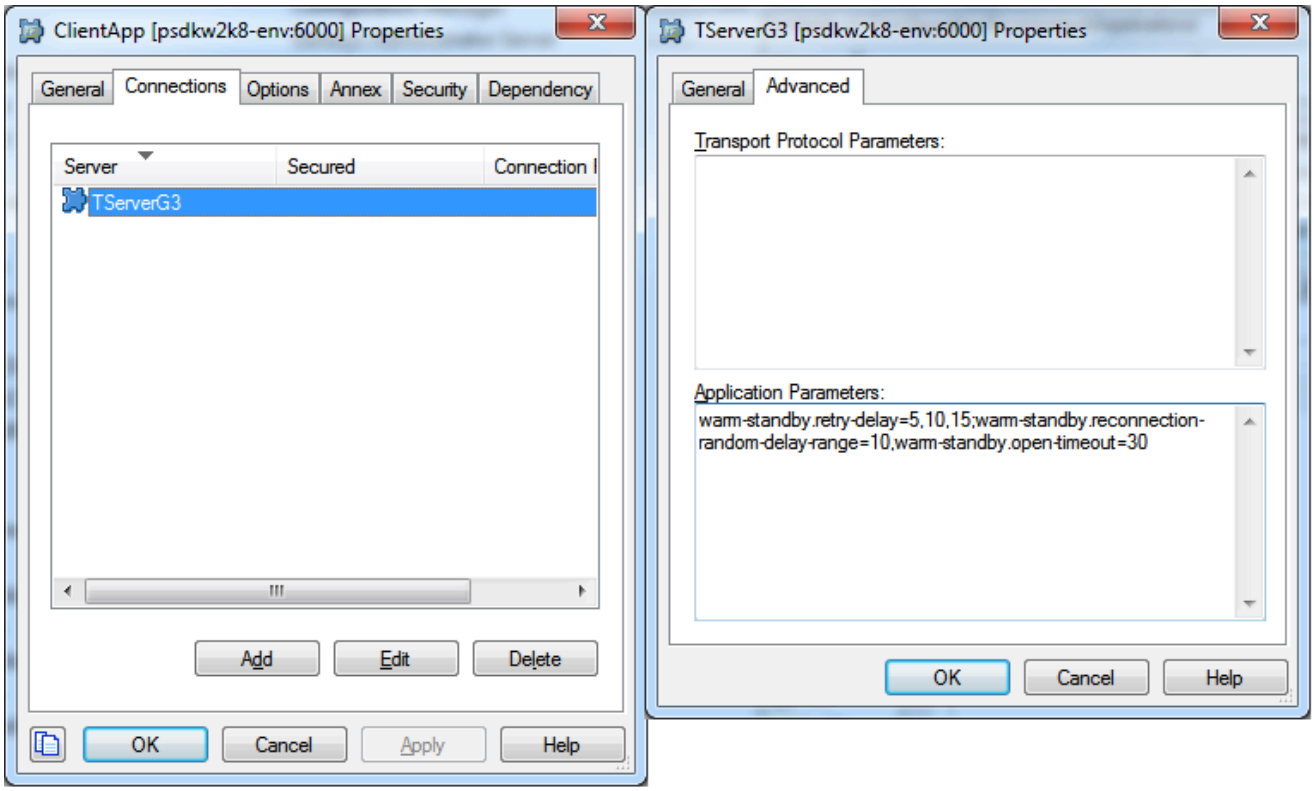
Configuration in Configuration Manager looks quite similar to the classic Warm Standby configuration with one difference: there are special timing parameters which are quite different than "Reconnection Timeout" and "Reconnection Attempts" and thus are specified apart from them. The "Reconnection Timeout" and "Reconnection Attempts" are not used in new Warm Standby Configuration.

Client Connection options:

Name	Values, in Seconds
warm-standby.retry-delay	5, 10, 15
warm-standby.reconnection-random-delay-range	10
warm-standby.open-timeout	30

See Warm Standby documentation and the API Reference guide for details about how these timing options customize Warm Standby behavior.

Options can be specified in Configuration Manager as shown below:



The last option is specified in Backup Server Applications on the Options tab. This characteristic of the backup server describes how much time is required for the server to step into primary mode.

Options Tab Section	Value, in Seconds
warm-standby	backup-delay=5

[+] Legacy Content

Configuring WarmStandby (Legacy Content)

This section describes how to configure WarmStandby service with Application Template helpers. To find out how to use the WarmStandby service, see the corresponding Using the Warm Standby Application Block article.

Application Template helper method creates configuration for `WarmStandbyService`. The result includes parameters for the connection to primary and backup servers defined in the specified application configuration information.

Sample:

```
var myApplicationName = "…";
CfgAppType myApplicationType = default(CfgAppType);
var applicationConfiguration = new GCOMApplicationConfiguration(
      new CfgApplicationQuery(confService){ Name = myApplicationName }.ExecuteSingleResult()
```

```
);
WarmStandbyConfiguration warmStandbyConfig =
ClientConfigurationHelper.CreateWarmStandbyConfig(
      applicationConfiguration,applicationConfiguration.GetAppServer(myApplicationType));
```

Setting Up a Server Channel

Similar to the creation of the client connection, provide the `IGApplicationConfiguration` object to the helper class.

Sample:

```
Endpoint endpoint = ServerConfigurationHelper.CreateListeningEndpoint(appConfiguration,
appConfiguration.PortInfo("default"));
ExternalServiceProtocolListener serverChannel = new ExternalServiceProtocolListener(endpoint);
```

This helper creates an Endpoint instance initialized with properties like the listening TCP port number, ADDP parameters, and so on.

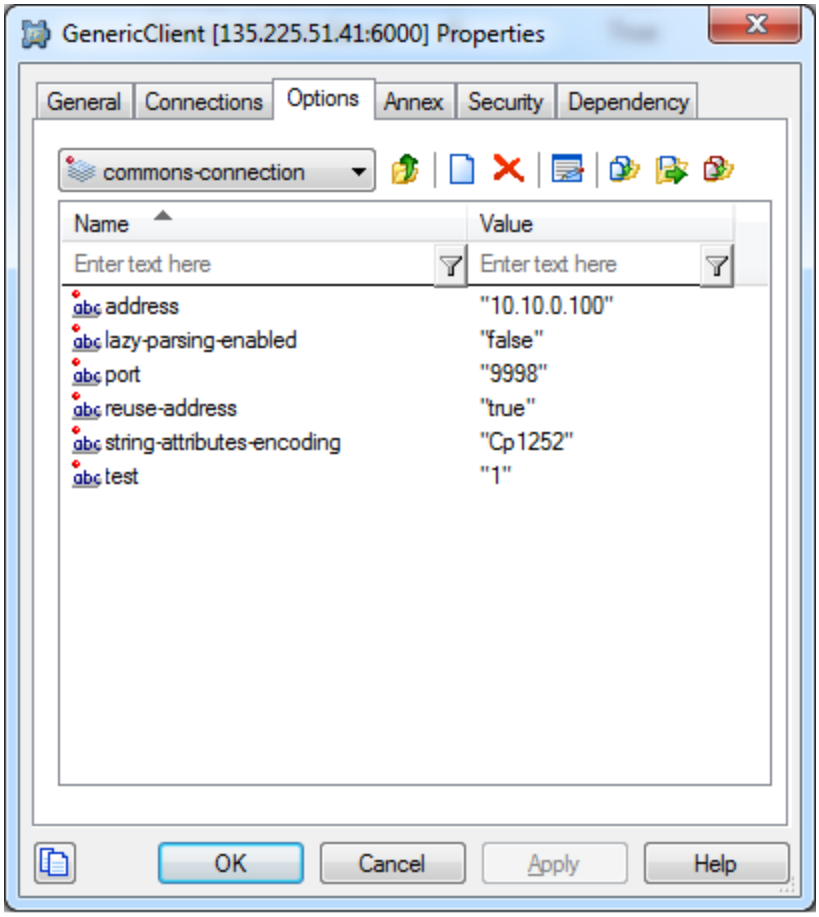
Defining Configuration Options in Genesys Administrator

Options can be specified in the `CfgApplication` object using Genesys Administrator. There are several possible option locations in the `CfgApplication` object:

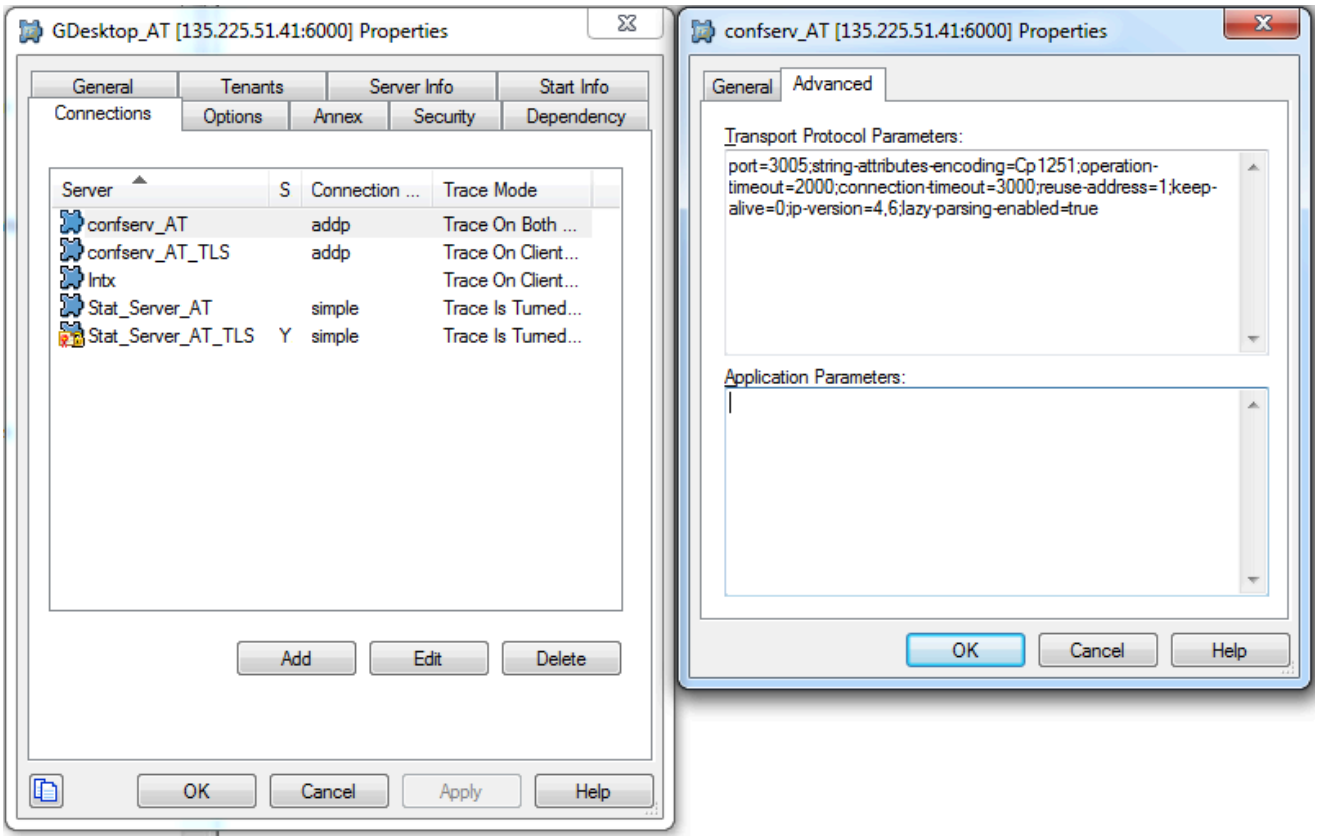
- *Application* options tab
- *Connection* parameters
- *Port* parameters

Common Options

Here is how options could be specified on the *Options* tab:



Here is how options could be specified for a particular connection (using the *Connection* parameters):



A complete options list is provided in the table below.

Section	Option	Description
commons-connection	string-attributes-encoding	Specifies encoding for string attributes.
	lazy-parsing-enabled	Boolean value. Enables or disables lazy parsing of properties, for which lazy parsing possibility is enabled in protocol. Currently used in Configuration Server protocol, enabled by default.
	address	Host bind option, specifies host from which connection should be made
	port	Port bind option, specifies port from which connection should be made
	backup-address	Host bind option, specifies host from which connection should be made for backup server.
	backup-port	Port bind option, specifies port from which connection should be

Section	Option	Description
		made (bind to) for backup server.
ucs-protocol	use-utf-for-responses	Boolean value. If set to false, UCSprotocol will add 'tkv.multibytes'='false' pair in Request KVlist of the message. It is false by default.
	use-utf-for-requests	Boolean value. If set to true, all string values of each KVlist will be packed as UtfStrings (in "UTF-16BE" encoding), instead of common strings. It is true by default.
webmedia-protocol	replace-illegal-unicode-chars	Boolean value. Enables or disables replacing of the illegal unicode chars in Webmedia XML messages.
	illegal-unicode-chars-replacement	String to replace illegal unicode chars

IPV6 Options

The IPV6 usage can be enabled with "enable-ipv6" option.

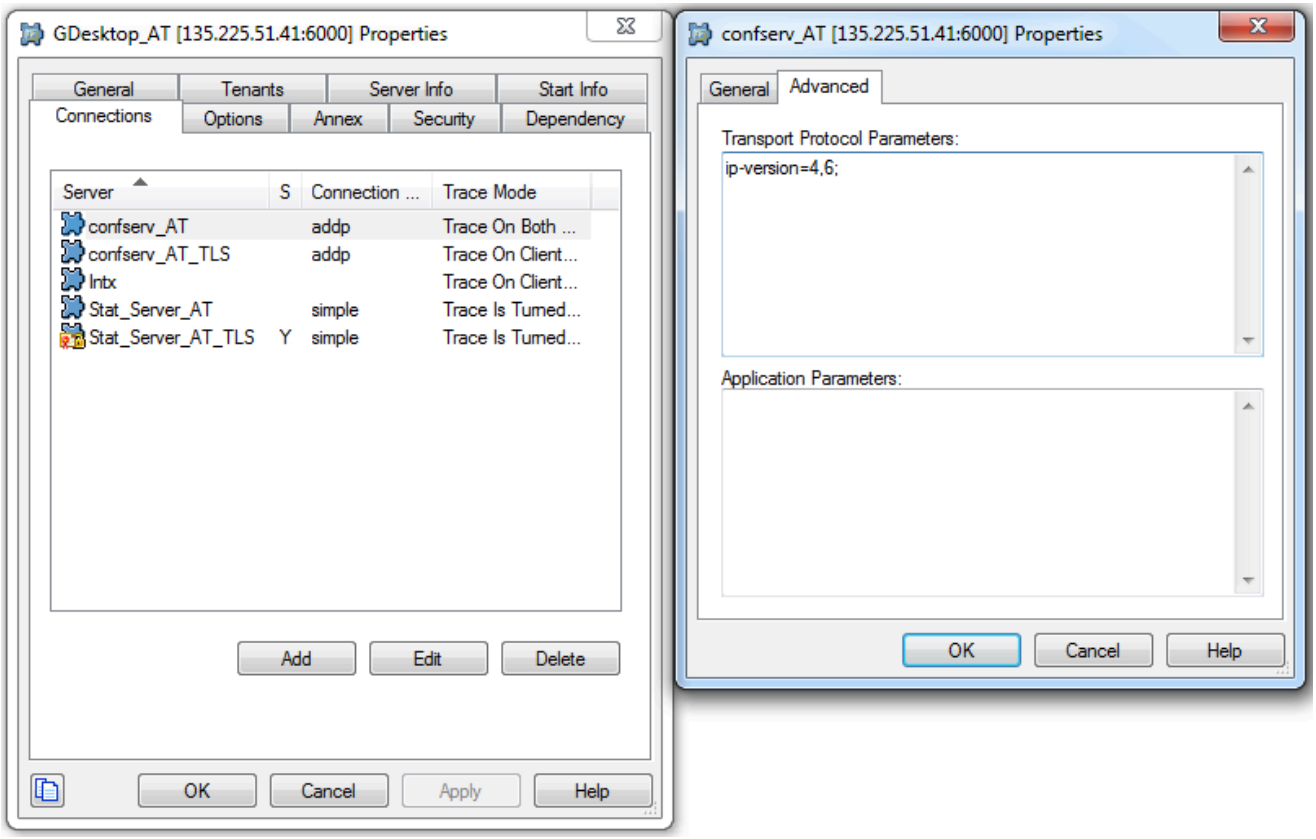
Section	Option	Description
common	enable-ipv6	Turns IPv6 support on/off. Possible values: 0 (default, implied) and 1. If set to 0, IPv6 support would be disabled, even if supported by OS/ platform.

The `ip-version` constant specifies the order in which connection attempts will be made to IPv6 and IPv4 addresses. Possible values include the following strings:

- "4,6" (default)
- "6,4"

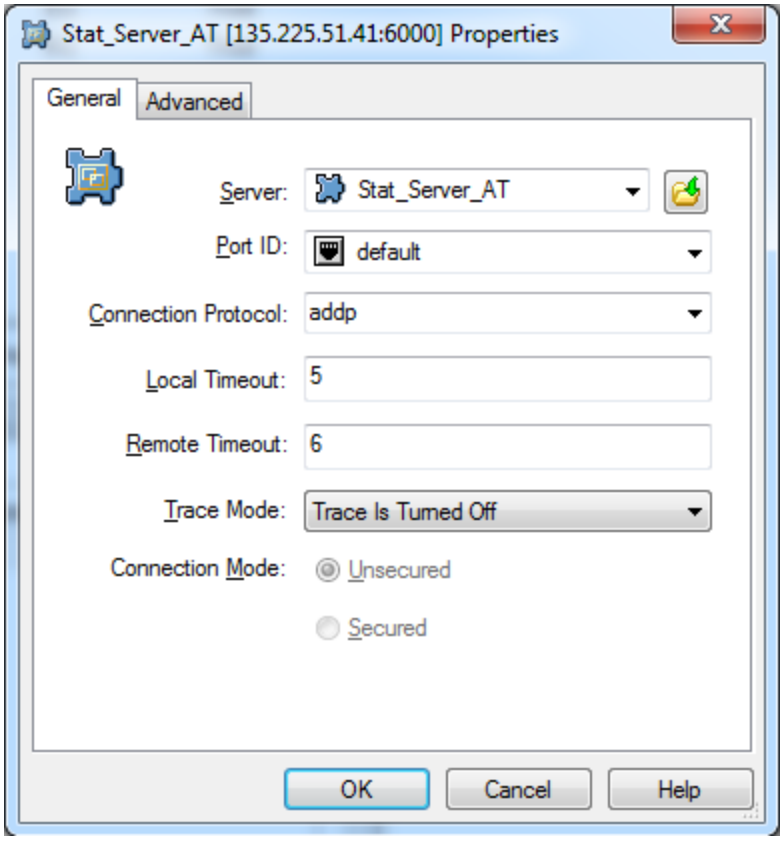
This setting has no effect on the connection if `enable-ipv6` is set to 0; warning would be logged. Has no effect on server side; warning would be logged.

Option values should be specified within the connection transport properties:



ADDP Options

ADDP options, which can be handled with the Application Template helper, should be specified on the *Connections* tab.



TLS Options

See the article on Configuring TLS Parameters in Configuration Manager for more information about TLS options.

Cluster Connection Configuration Helpers

This section describes Application Template helper methods that support the configuration of cluster connections to Genesys server, including use cases and samples.

Cluster Connection Configuration Types

There are two types of Genesys server cluster configuration available in Configuration Server: client-aligned configuration, and node-aligned configuration.

Both types provide information about addresses of the target servers in a cluster, and connection options used to communicate with them. You must choose the proper helper to use for each type of cluster configuration.

Client-aligned configuration methods:

- `ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(IGApplicationConfiguration appConfig, CfgAppType serverType)`
- `ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(IGApplicationConfiguration appConfig, IGAppConnConfiguration clusterConn, CfgAppType serverType)`

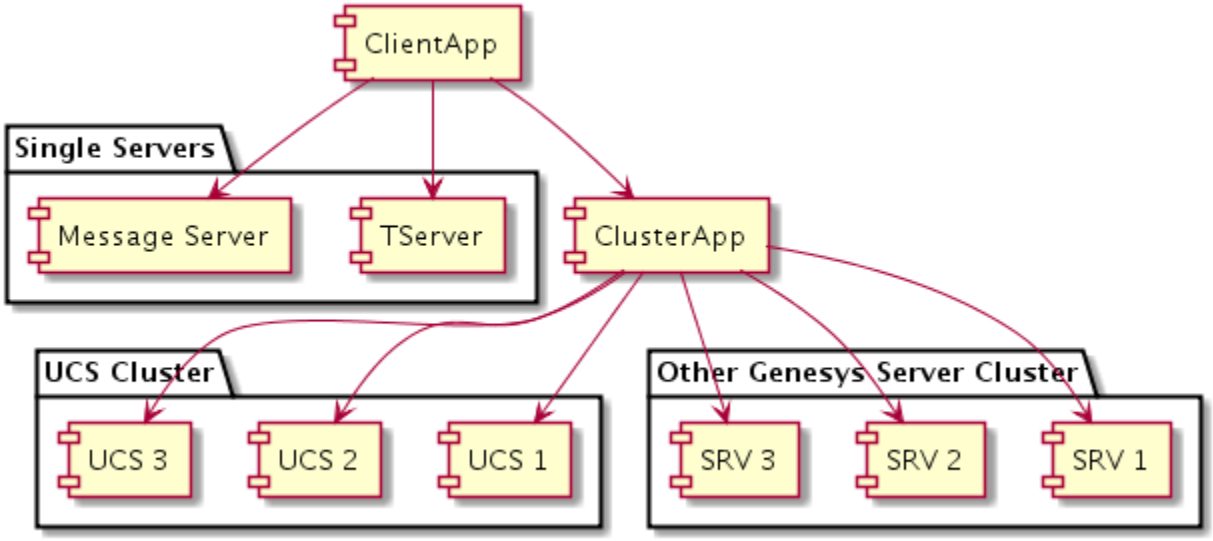
Node-aligned configuration methods:

- `ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(IConfService confService, IGApplicationConfiguration appConfig, CfgAppType serverType)`
- `ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(IConfService confService, IGApplicationConfiguration appConfig, IGAppConnConfiguration clusterConn, CfgAppType serverType)`

Client-Aligned Configuration Samples

Sample 1

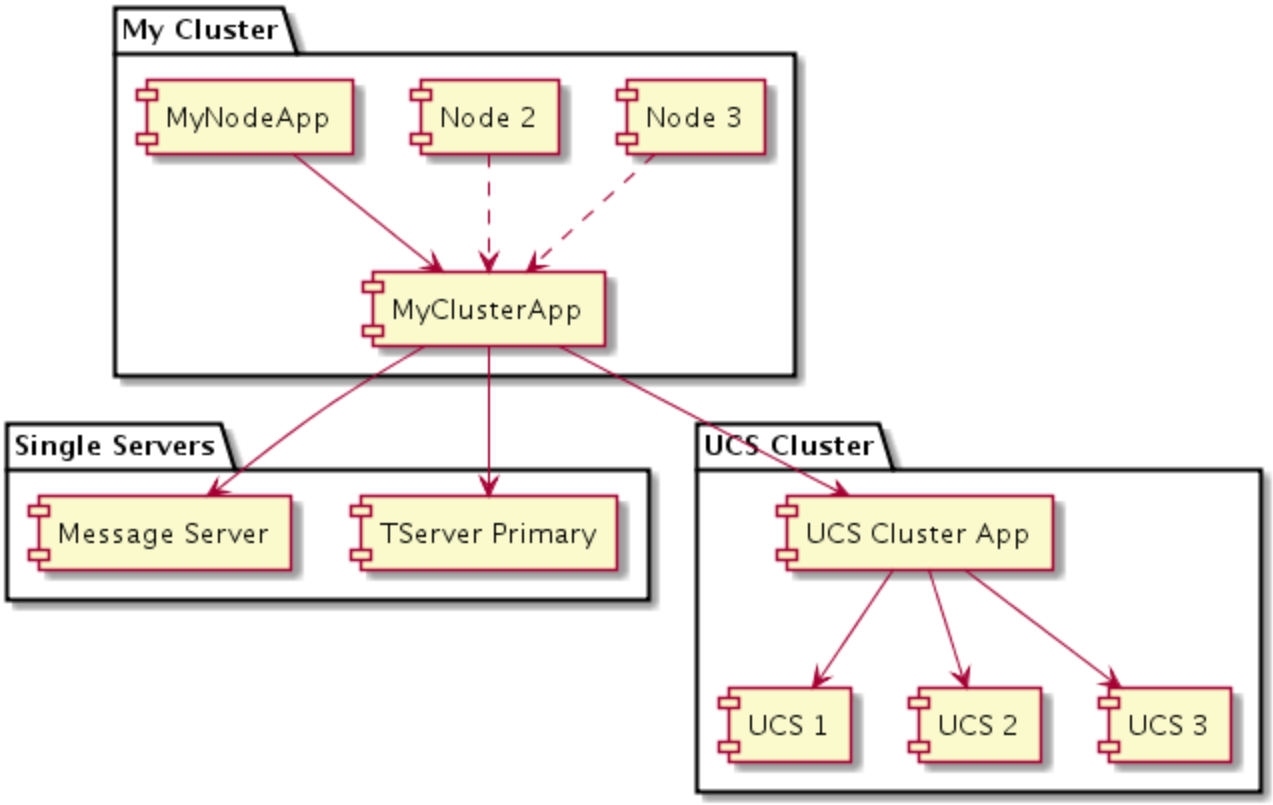
In the first sample scenario, a "ClientApp" Application is connected to the "ClusterApp'" virtual application (type=CFGApplicationCluster) that has connections to cluster nodes of one ore more clusters.




```
IClusterProtocol protocol ...
IList<WSConfig> nodesList =
ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(ClientApp,
CfgAppType.CFGContactServer);
protocol.SetNodes(nodesList);
```

Sample 2

In the second sample scenario, the "MyNodeApp" Application is a cluster node that is connected to the "MyClusterApp" virtual application. In this case, "MyClusterApp" is a shared store of connection configurations for all cluster nodes, which can have connections to other clusters like "UCS" as well as stand-alone servers.




```
IClusterProtocol protocol ...
IList<WSConfig> nodesList =
ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(MyClusterApp,
clusterAppConnection, CfgAppType.CFGContactServer);
protocol.SetNodes(nodesList);
```

Connection Options

You can use Genesys Administrator to define connection parameter options (such as ADDP, ip-version, strings encoding, or TCP socket options) that are used for all nodes in a cluster, or to define connection parameter for a particular node that will overriding common parameters.

- Common Options: Specified in the connection from ClientApp to ClusterApp (or from OwnClusterApp if application is a node of some type of cluster).
- Node-Specific Options: can be specified in Configuration Manager in connection from ClusterApp to NodeApp.

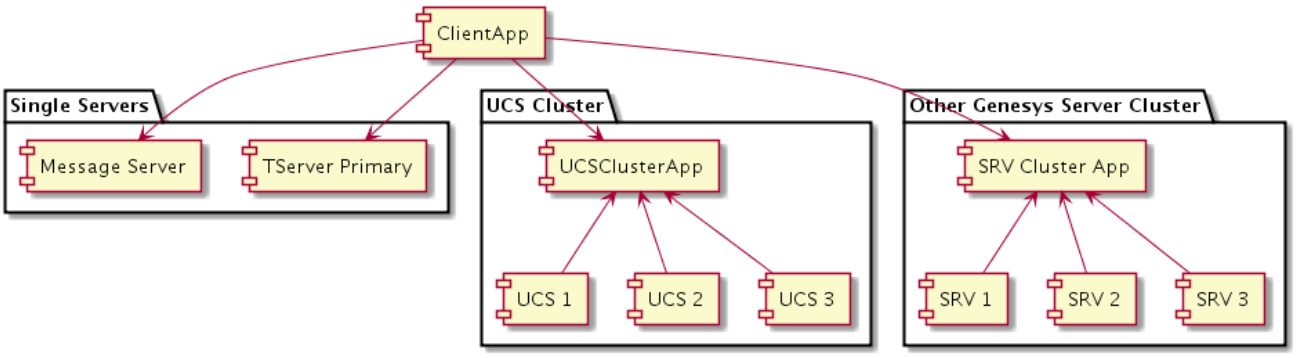
See eServices Load Balancing Business Continuity for examples.

Node-Aligned Configuration

Sample 3

In the third sample scenario, the "ClientApp" Application is connected to the "UCSClusterApp" virtual

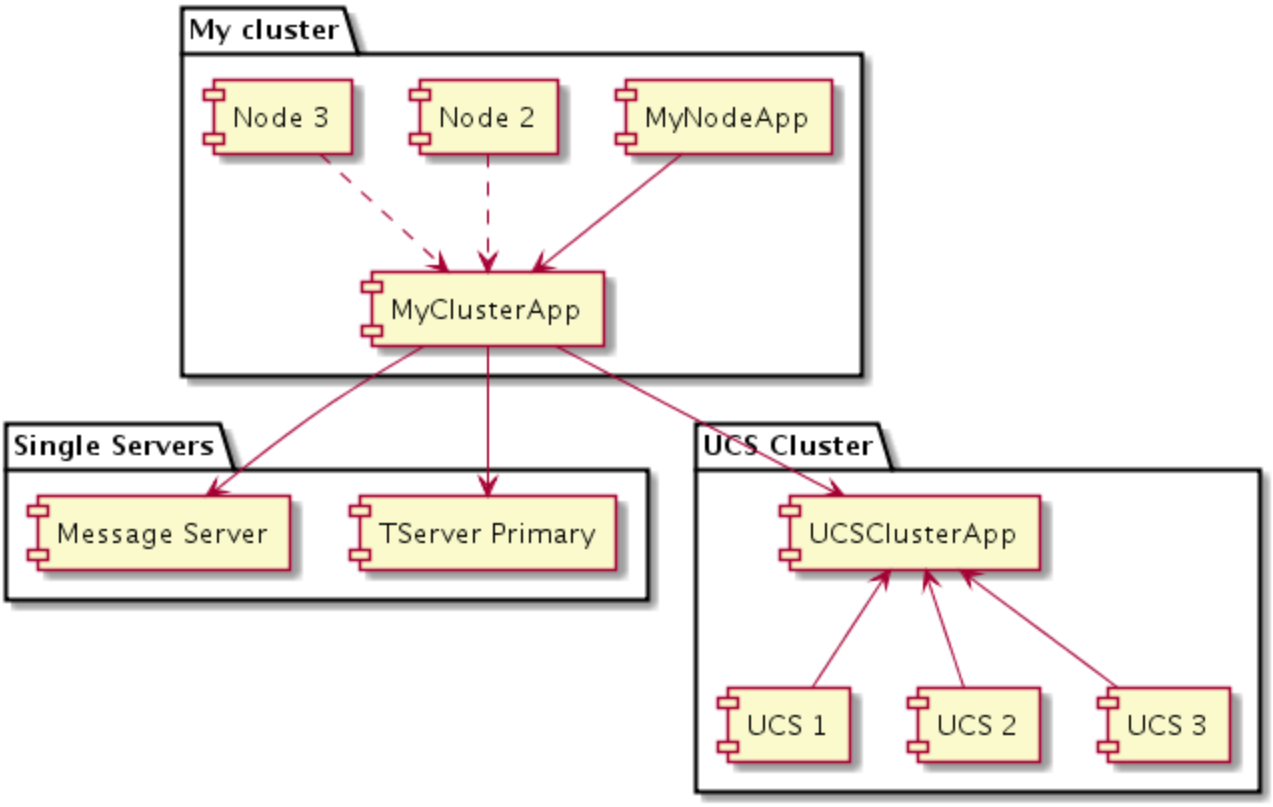
application that groups UCS nodes. Unlike previous configurations, the cluster nodes in this scenario are also connected to the "UCSClusterApp" virtual application.




```
IClusterProtocol protocol ...
IList<WSConfig> nodesList =
ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(confService, ClientApp,
CfgAppType.CFGContactServer);
protocol.SetNodes(nodesList);
```

Sample 4

In the fourth sample scenario, the "MyNodeApp" Application is a cluster node. The "MyClusterApp" application has a connection to the UCS cluster ("UCSClusterApp" application). UCS nodes also have connections to "UCSClusterApp".




```
IClusterProtocol protocol ...
IList<WSConfig> nodesList =
ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(confService, MyNodeApp,
clusterAppConnection, CfgAppType.CFGContactServer);
protocol.SetNodes(nodesList);
```

Connection Options

You can use Genesys Administrator to define connection parameter options (such as ADDP, ip-version, strings encoding, or TCP socket options), but in this scenario this configuration is applied for all nodes in a cluster.

- Common Options: Specified in the connection from ClientApp to ClusterApp.

Client Connection Active Nodes Randomizer

> **Important**
>
> Dynamic updates to the cluster configuration are still possible with this approach. Your application should use the `shuffler.SetNodes(nodes);` function to change the list of cluster nodes, instead of using Cluster Protocol methods directly.

The randomizer helper component supports cluster load balancing from client applications, by allowing your applications to work with a server cluster without creating live connections to all of its nodes. This also prevents many clients from being stuck on a single connection, to resolve issues such as a single server overloading on startup, and can be of critical importance in use cases where the number of clients could be in the tens of thousands.

This helper takes a list of cluster nodes, and performs periodical endpoint rotation. The existing Cluster Protocol functionality to update its nodes configuration dynamically..

For each new node that is added:

1. a new instance of the related PSDK protocol class is created
2. asynchronous open is started
3. after the node has connected to its server, the node is added to load balancer

For each node that is removed:

1. the removed node is immediately excluded from the load balancer
2. the node is scheduled to be closed after its protocol timeout delay (which allows responses to be delivered for in-progress requests)

The randomizer uses the `Collections.shuffle(nodes)` method to randomize the sequence of given nodes.When updating the protocol node configuration, the randomizer uses a round-robin rotation over a whole list of randomly-sorted cluster nodes, choosing a subset of given number of elements. This allows the protocol to avoid breaking all connections at a single moment, and to stay online during switchover.

The randomizer component also has an optional ability to attach truncated nodes' endpoints as backup endpoints for selected ones. This allows the internal WarmStandby service to quickly switchover from an unresponsive node.

Sample usage of this component could look like the following:

```
IList<WSConfig> nodes = ...;
UcsClusterProtocol ucsNProtocol = new UcsClusterProtocolBuilder().Build();

var shuffler = new ClusterNodesShuffler(ucsNProtocol, 2); // creates nodes shuffler for 2
active connections
shuffler.UseBackups = true;// - lets the shuffler to use truncated nodes endpoints as backup
endpoints for the selected ones
try {
    shuffler.SetNodes(nodes); // - initializes shuffler with the whole list of cluster nodes.
    ucsNProtocol.Open();
    shuffler.StartTimer(3000, 3000); // - schedules shuffling operation with 3 secs delay and
3 secs period

    // Do the business logic on the cluster protocol...
    // In case of update in the cluster configuration, application should use
'shuffler.SetNodes(newNodes)'
    //     instead of ClusterProtocol's methods related to nodes configuration.
} finally {
    shuffler.StopTimer();
    ucsNProtocol.Close();
}
```

Code Samples

Simple Client Application Connecting to Any UCS Cluster

This sample checks the connection configuration for "WCC mode" UCS 9.0 cluster, then for "WDE mode" UCS cluster, and then for "legacy mode" connection to UCS (pri/bck) server.

```
 // Take "my application configuration" from context, or read it in a way like this:
ConfServerProtocol cfgProtocol=new ConfServerProtocol(new Endpoint(host, port));
IConfService confService=ConfServiceFactory.CreateConfService(cfgProtocol);
IGApplicationConfiguration myApp = new GCOMApplicationConfiguration(
  new CfgApplicationQuery(confService){Name = "myAppName"}.ExecuteSingleResult());

// For the first, try UCS 9 connection cluster:
IList<WSConfig> conns = ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(
        confService, myApp, CfgAppType.CFGContactServer);
if ((conns == null) || (conns.Count==0)) {
    // If there is no UCS 9 cluster connected, then we try older UCS cluster, or simple UCS
connection(s):
    conns = ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(
            myApp, CfgAppType.CFGContactServer);
}

Console.WriteLine("Connections: " + conns);
```

WCC-Based Cluster Node Application Connecting to Any UCS Cluster

This sample works in context of WCC.

```
var cfgProtocol=new ConfServerProtocol(new Endpoint(host, port));
var confService=ConfServiceFactory.CreateConfService(cfgProtocol);
// Take "my application configuration" from context, or read it in a way like this:
IGApplicationConfiguration myApp = new GCOMApplicationConfiguration(
        new CfgApplicationQuery(confService){Name = "myAppName"}.ExecuteSingleResult());

IGApplicationConfiguration myClusterApp = null;
// if we do not have 'myClusterApp' from WCC context, we may take it by this way:
IList<IGAppConnConfiguration> clusters =
GApplicationConfiguration.GetAppServers(myApp.AppServers, CfgAppType.CFGApplicationCluster);
if (clusters != null) {
    if (clusters.Count == 1) {
        myClusterApp = clusters[0].TargetServerConfiguration;
        log.InfoFormat("Application is recognized as a node of cluster
''{0}''",myClusterApp.ApplicationName);
    } else if (clusters.Count > 1) {
        log.Error("Application has more than one application cluster connected - its treated
as a standalone app");
    }
}

// Select application cluster connection start point:
IGApplicationConfiguration connSrc =  myClusterApp ?? myApp;

// For the first, try UCS 9 connection cluster:
IList<WSConfig> conns = ClusterClientConfigurationHelper.CreateRefClusterProtocolEndpoints(
        confService, connSrc, CfgAppType.CFGContactServer);
if (conns == null || conns.Count==0) {
    // If there is no UCS 9 cluster connected, then we try older UCS cluster, or simple UCS
connection(s):
    conns = ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(
```

```
            connSrc, CfgAppType.CFGContactServer);
}

Console.WriteLine("Connections: " + conns);
```

Client Nodes Randomizer Usage Sample

Application code using the randomizer component may look like the following sample:

```
 IList<WSConfig> nodes = ...;
 UcsClusterProtocol ucsNProtocol = new UcsClusterProtocolBuilder().Build();
 ucsNProtocol.Timeout = TimeSpan.FromSeconds(5); // sets protocol timeout to 5 secs
 ucsNProtocol.ClientName = "MyClientName";
 ucsNProtocol.ClientApplicationType = "MyAppType";

 var shuffler = new ClusterNodesShuffler(ucsNProtocol, 2); // creates nodes shuffler for 2
active connections

 try {
     shuffler.SetNodes(nodes); // - initializes shuffler with the whole list of cluster nodes.
     ucsNProtocol.Open();
     shuffler.StartTimer(3000, 3000); // - schedules shuffling operation with 3 secs delay
and 3 secs period

     // do the business logic on the cluster protocol...
     for (int i = 0; i < 200; i++)
     {
       var resp = ucsNProtocol.Request(RequestGetVersion.Create()) as EventGetVersion;
       if (resp!=null) Console.WriteLine("Resp from: " + resp.Endpoint);
       Thread.Sleep(300);
     }
 } finally {
     shuffler.StopTimer();
     ucsNProtocol.Close();
 }
```

Handling Updates From Config Server

The GFApplicationConfigurationManager component monitors Config Server for updates and provides notifications about changes in applications.

You should register for updates at GFApplicationConfigurationManager.

In the handle (GFAppCfgEvent event) method implementation, create a new connection configuration using one of the helpers mentioned above:

```
using Genesyslab.Platform.AppTemplate.Application;

UcsClusterProtocol ucsProtocol = ...;
GFApplicationConfigurationManager appManager = ...;
appManager.Register(@event =>
{
  //get new application configuration
  IGApplicationConfiguration appconfig = @event.AppConfig;

  //create protocol config
  var wsconfig = ClusterClientConfigurationHelper.CreateClusterProtocolEndpoints(appconfig,
CfgAppType.CFGContactServer);
```

```
  //apply new set of protocol endpoints
  ucsProtocol.SetNodes(wsconfig );

});
appManager.Init();
```

> **Important**
>
> WCC-based cluster configuration does not support an update handler at this time. Subscribing to updates in this case will lead to Config Server overloading, so customers are encouraged to make direct requests to Config Server to actualize the cluster configuration before opening `ClusterProtocol`.