Platform SDK Developer's Guide

Using the Cluster Protocol Application Block

12/12/2025

Using the Cluster Protocol Application Block

Java

This Application Block is designed to be used with applications where a large number of requests should be spread between a configured set of UCS servers - or other Genesys servers - in a cluster, providing a type of high-availability (HA) connection to that cluster. In addition to this application block, the Cluster Protocol also includes a set of configuration helpers in the Application Template Application Block.

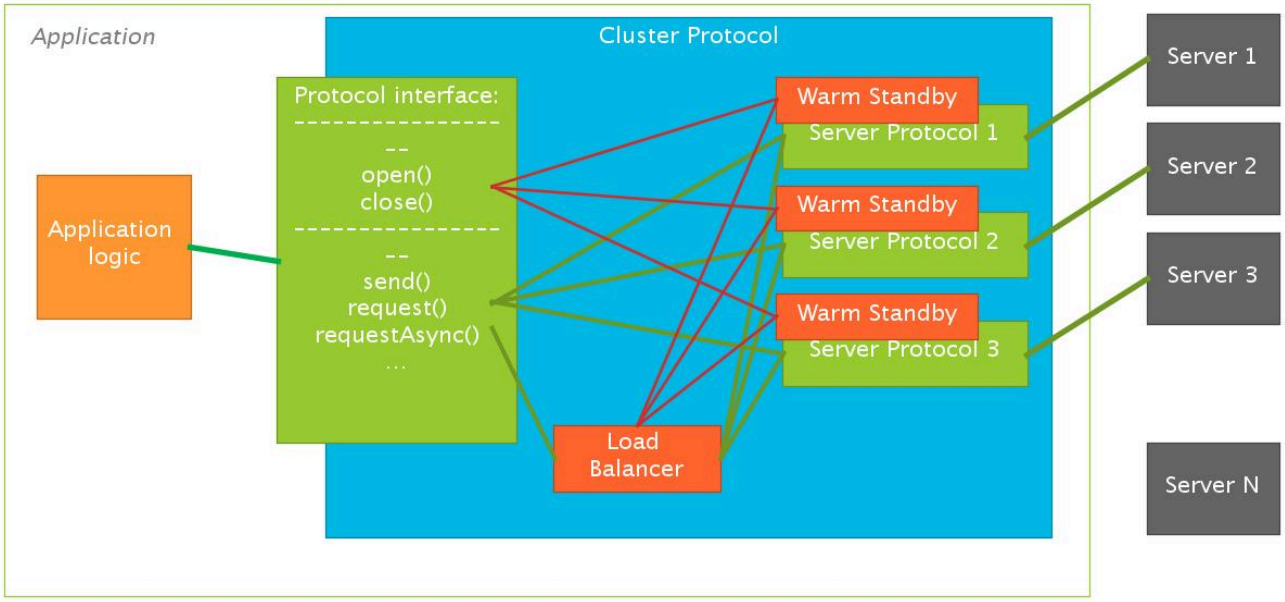
> **Tip**
>
> When the Cluster Protocol Application Block is connected to a cluster, it can be used in UCS9 N+1 server mode connected to all nodes, or in UCS9 N+1 client mode connected to one node. When the application block is connected to a single application that has a backup configured, it works in UCS8 mode Primary/Backup.

Architecture Overview

One of the simplest and most common uses of the Platform SDK interface is a protocol that interacts directly with a Genesys server using a set of standard protocol methods (such as Open, Close, Send, Request).

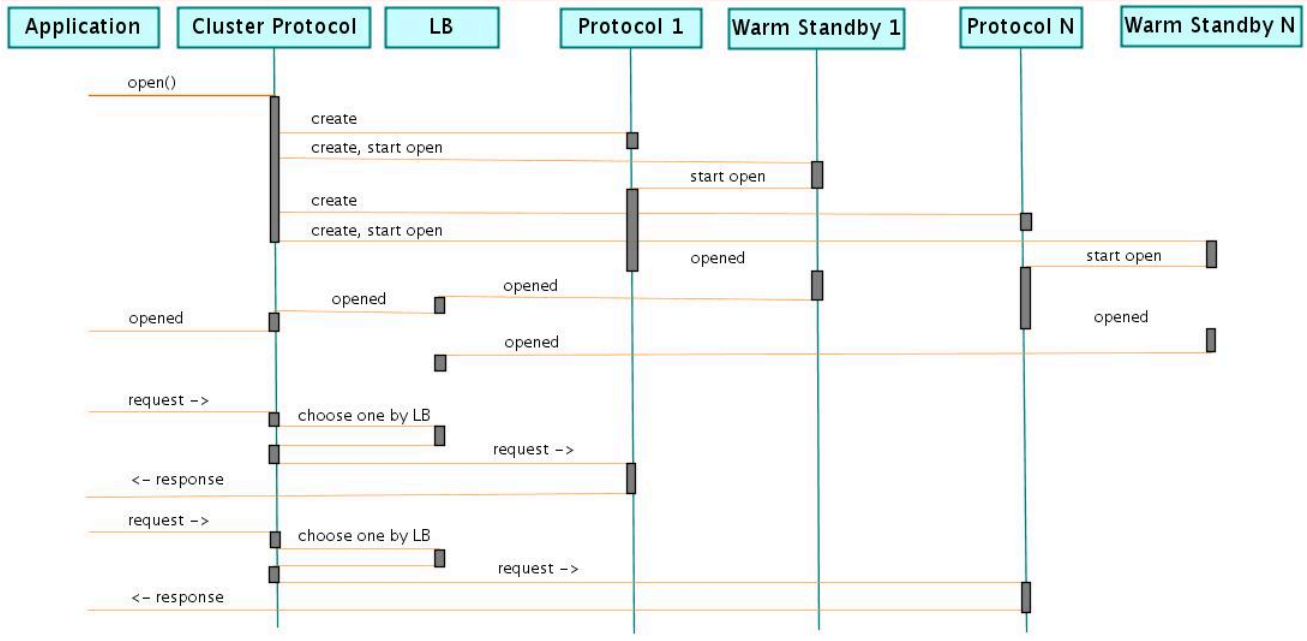
To provide a single working protocol with at least one backup, we use the Warm Standby Application Block. That application block intercepts control of the protocol interface and provides switch-over between multiple protocols, or protocol restoration, using a single Warm Standby endpoint.

The Cluster Protocol builds on this idea, allowing you to work simultaneously with a series of protocols and Warm Standby application blocks (each of which can represent one or many individual protocols) the same way that you would with a single, standard protocol. To configure the Cluster Protocol, you use a mixed list of protocol and Warm Standby endpoints gathered from Configuration Server.



The Cluster Protocol itself covers any scenarios where we need simultaneous connections through a load balancer to many servers, where each server may have one or many backups.

Sequence Diagram: Main Scenario When Using Cluster Protocol Application Block



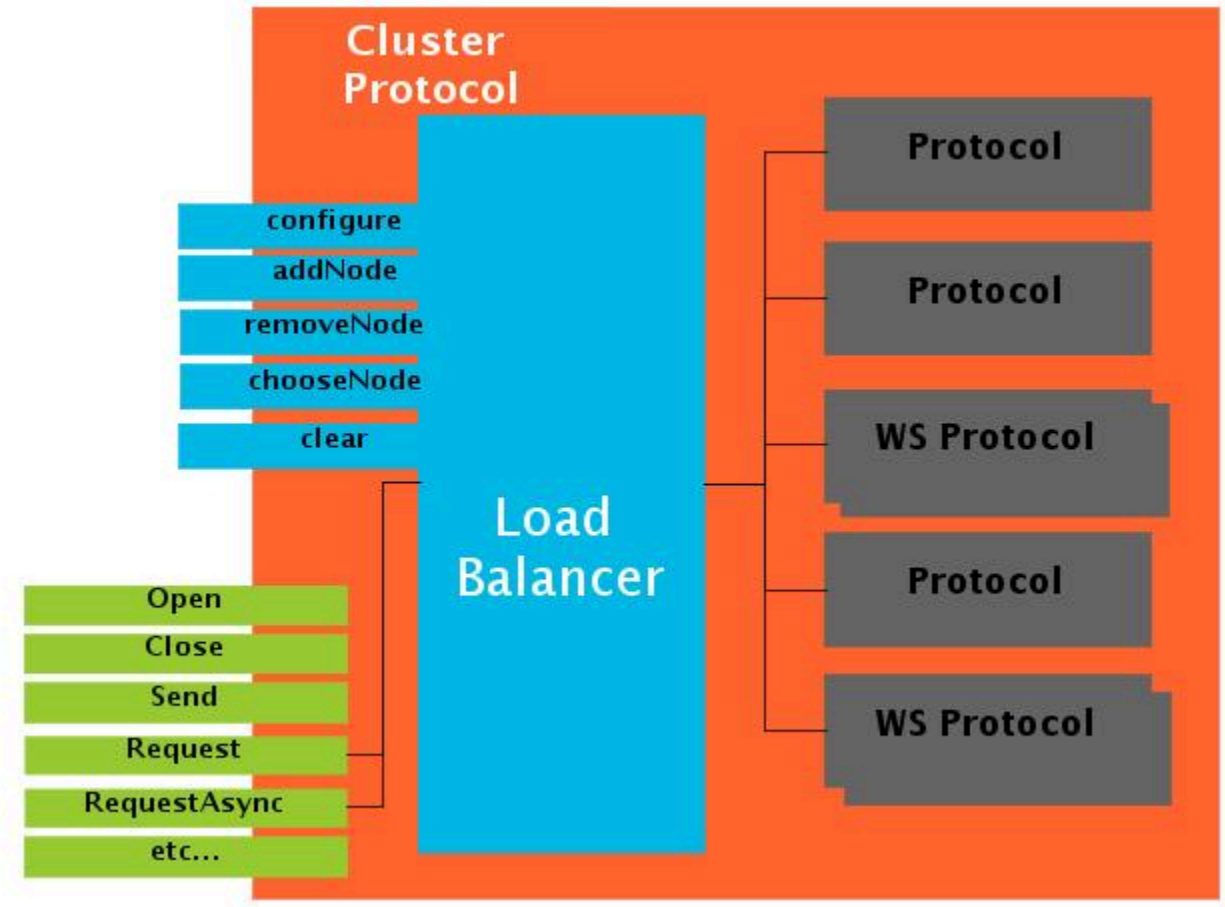
High Availability Options

The Cluster Protocol is able to control the state of protocol connections, and use the Warm Standby Application Block to keep them opened. Each cluster node may be initialized as a standard Platform SDK Endpoint, or using `WSConfig` (Warm Standby configuration) with one or more backup Endpoints added. A list of opened connections is tracked using a Load Balancer, which allows requests to be spread across connected protocols.

This gives your application the flexibility to configure one endpoint per connection ("server mode"), or to combine cluster endpoints in a single Warm Standby configuration and let the Cluster Protocol use a single connection to anyone of them ("client mode").

Load Balancing Options

The default Load Balancing strategy uses a "round robin" algorithm over connected servers for request forwarding. It is possible for you to create your own implementation of the Load Balancer interface, however, and provide it during creation of the Cluster Protocol instance.



Implementing a Custom LoadBalancer

```
final EspClusterProtocol haProtocol =
```

```
        new EspClusterProtocolBuilder()
                .withLoadBalancer(new MyLoadBalancer())
                .build();

protected class MyLoadBalancer implements ClusterProtocolLoadBalancer {
        @Override
        public void configure(final ConnectionConfiguration config) {
        }
        @Override
        public void addNode(final Protocol node) {
        }
        @Override
        public void removeNode(final Protocol node) {
        }
        @Override
        public Protocol chooseNode(final Message request) {
        }
        @Override
        public void clear() {
        }
}
```

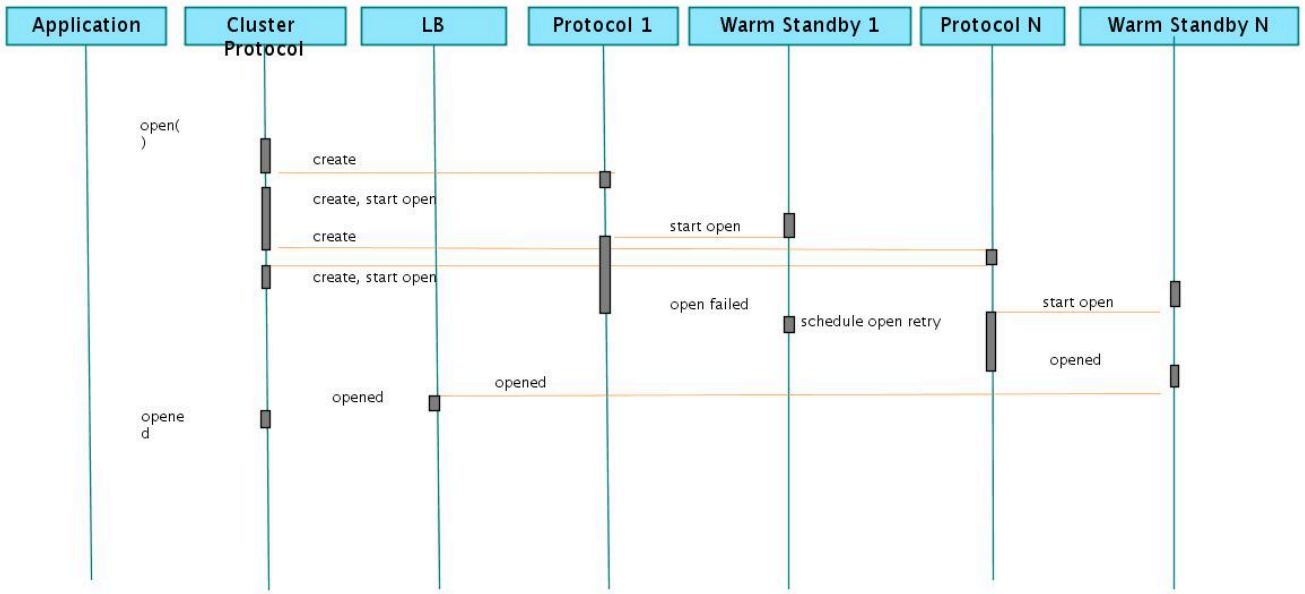
Disaster Recovery

Platform SDK does not inject any business functionality into connections, so the Cluster Protocol Application Block is able to provide disaster recovery by meeting the following requirements:

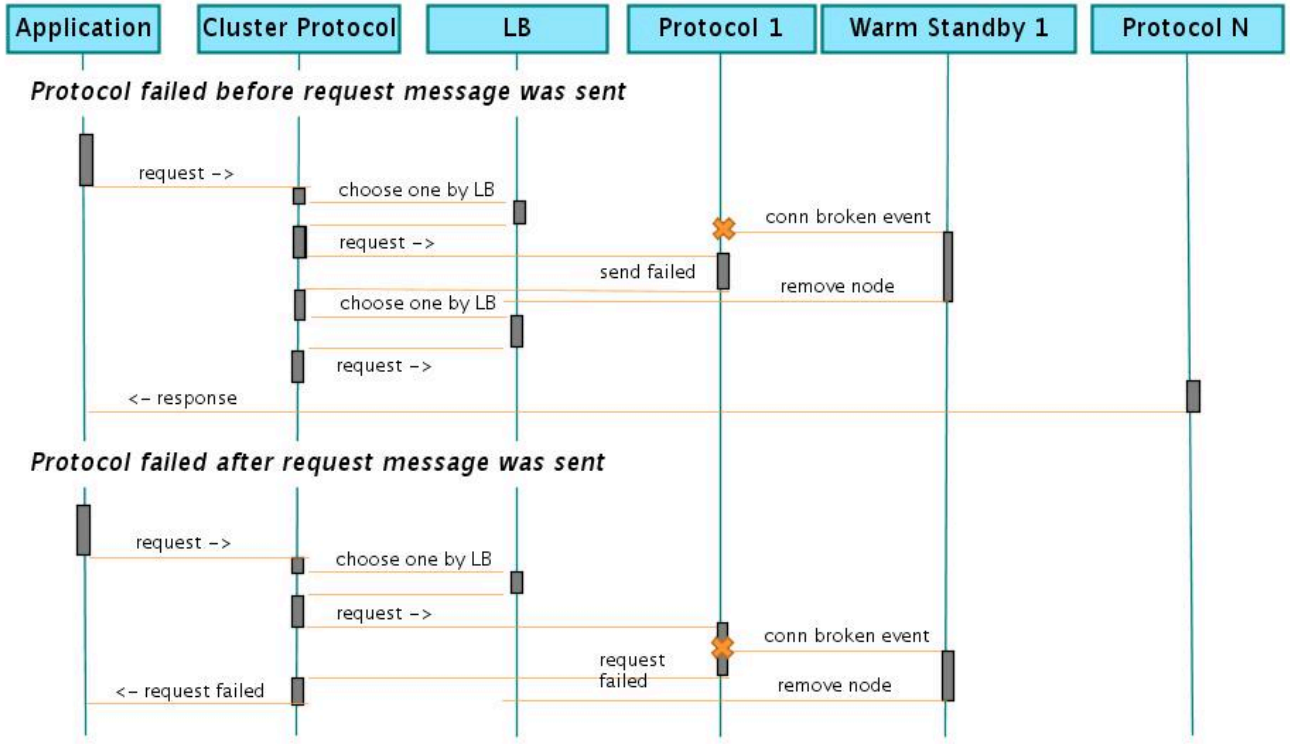
1. High availability maintains a list of active connections to ensure that requests are not sent to disconnected servers.
2. Any request that receives a `ChannelClosedOnSendException` or `ChannelClosedOnRequestException` response (because the connection was broken but not yet removed from the active list) is automatically resent to a different connection. Other exceptions are passed through to the client application to be handled manually.

There is no special configuration required to enable disaster recovery, but your application will need to include logic that handles generic IO exceptions or protocol timeout exceptions.

Sequence Diagram: Cluster Protocol with Node Connection Failure



Sequence Diagram: Cluster Protocol User Request Failure



How to Handle Lost Requests

It is possible for a communication error to occur where your application sends a request but the connection is broken before any response is received. When using the Cluster Protocol you may not

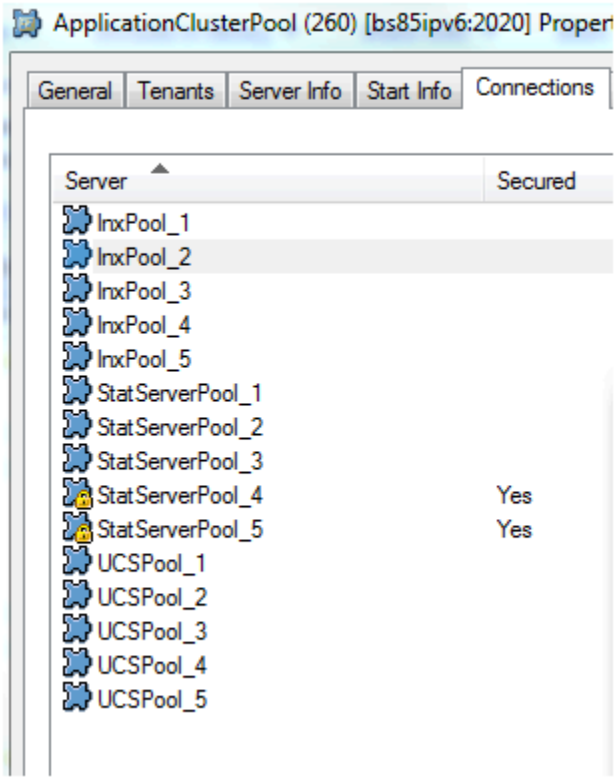
know if another node in the cluster handled the request. In this scenario your application won't know if the server was able to receive or process the request correctly.

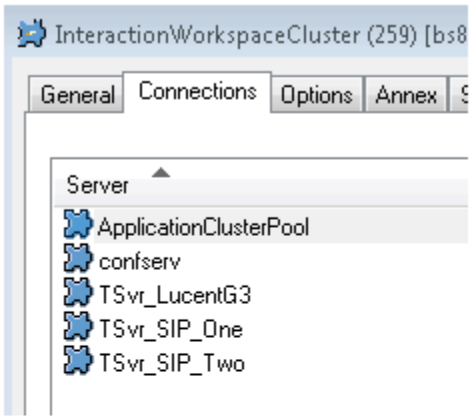
To address this, your application should include business logic that handles exceptions or null returns for a Cluster Protocol request and acts appropriately based on the type of request. For example, a request to read data can be sent again without impact, while a request that modifies data on the server may require your application to check the server state before retrying or providing notification that the request was successful.

Configuration Options

> **Important**
> For this release, only the UCS cluster type is supported.

The Cluster Protocol Application Block supports configuration in Configuration Manager for the client application and cluster.





Dynamic Configuration Options Updates

Cluster Protocol objects support graceful, dynamic updates of the cluster configuration, allowing you to add or remove nodes on an *opened* and *actively used* cluster protocol.

Adding a new node automatically creates the new node protocol connection in the background and attempts an asynchronous opening. The load balancer is notified about the new node connection immediately after it is connected.

Removing a connected node from the cluster protocol causes the protocol to exclude that node from the load balancer, and then disconnect the node after the protocol timeout delay. This delay allows for delivery of responses on any requests that were already started.

Code Examples

Cluster Protocol Usage Example

```
UcsClusterProtocol ucsNProtocol =
        new UcsClusterProtocolBuilder()
                .build();
ucsNProtocol.setClientName("MyClientName");
ucsNProtocol.setClientApplicationType("MyAppType");
ucsNProtocol.setNodesEndpoints(
        new Endpoint("ucs1", UCS_1_HOST, UCS_1_PORT),
        new Endpoint("ucs2", UCS_2_HOST, UCS_2_PORT),
        new Endpoint("ucs3", UCS_3_HOST, UCS_3_PORT));
ucsNProtocol.open();

EventGetVersion resp1 = (EventGetVersion) ucsNProtocol.request(RequestGetVersion.create());
EventGetVersion resp2 = (EventGetVersion) ucsNProtocol.request(RequestGetVersion.create());
```

Configuration Helper Example

The `ClusterClientConfigurationHelper` class is designed to make it easier for your application to make use of this Application Block by performing the following steps:

1. Checks if client application is connected to cluster application
2. If cluster application detected, then creates endpoints for all cluster connections of the specify type
3. If cluster application not detected or no connections in cluster application found, then creates endpoints for all client application connections of the specify type (compatibility mode)
4. If connected server has backup application, endpoint will have classical Primary\Backup configuration
5. Supports specifying shared ADDP options, Transport and Application parameters in connection to cluster application. Those parameters can be overridden in connection to particular cluster node.

For more information, including an overview of the new Cluster Connection Configuration Helpers, see Using the Application Template Application Block.

.NET

This Application Block is designed to be used with applications where a large number of requests should be spread between a configured set of UCS servers - or other Genesys servers - in a cluster, providing a type of high-availability (HA) connection to that cluster. In addition to this application block, the Cluster Protocol also includes a set of configuration helpers in the Application Template Application Block.

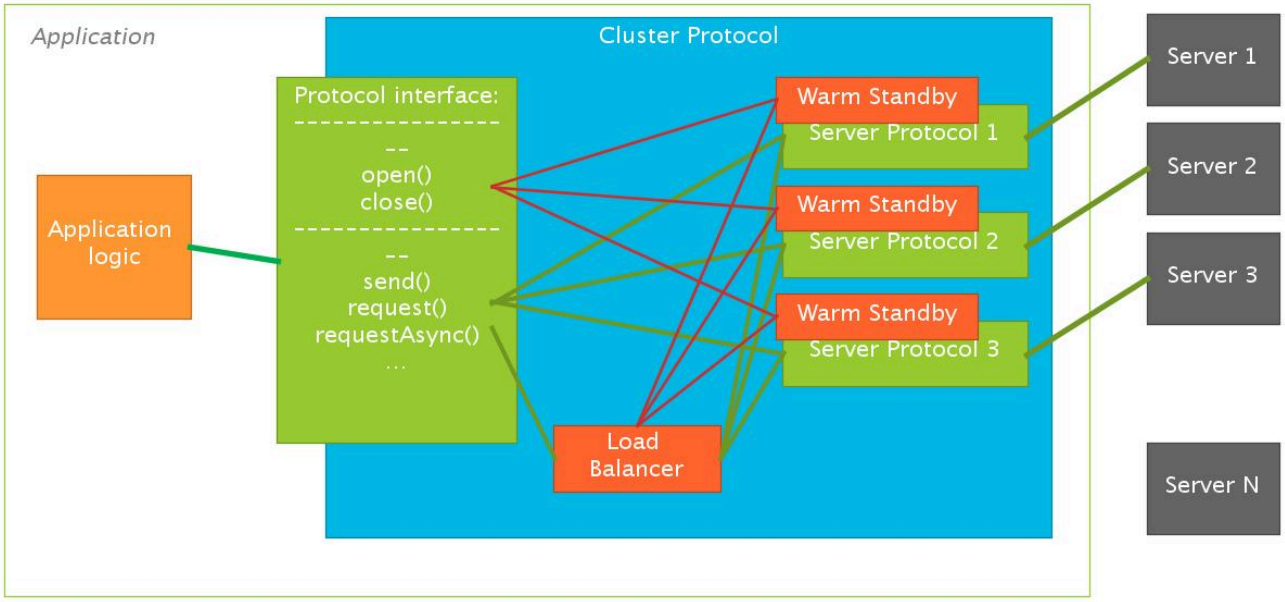
> **Tip**
> When the Cluster Protocol Application Block is connected to a cluster, it can be used in UCS9 N+1 server mode connected to all nodes, or in UCS9 N+1 client mode connected to one node. When the application block is connected to a single application that has a backup configured, it works in UCS8 mode Primary/Backup.

Architecture Overview

One of the simplest and most common uses of the Platform SDK interface is a protocol that interacts directly with a Genesys server using a set of standard protocol methods (such as Open, Close, Send, Request).

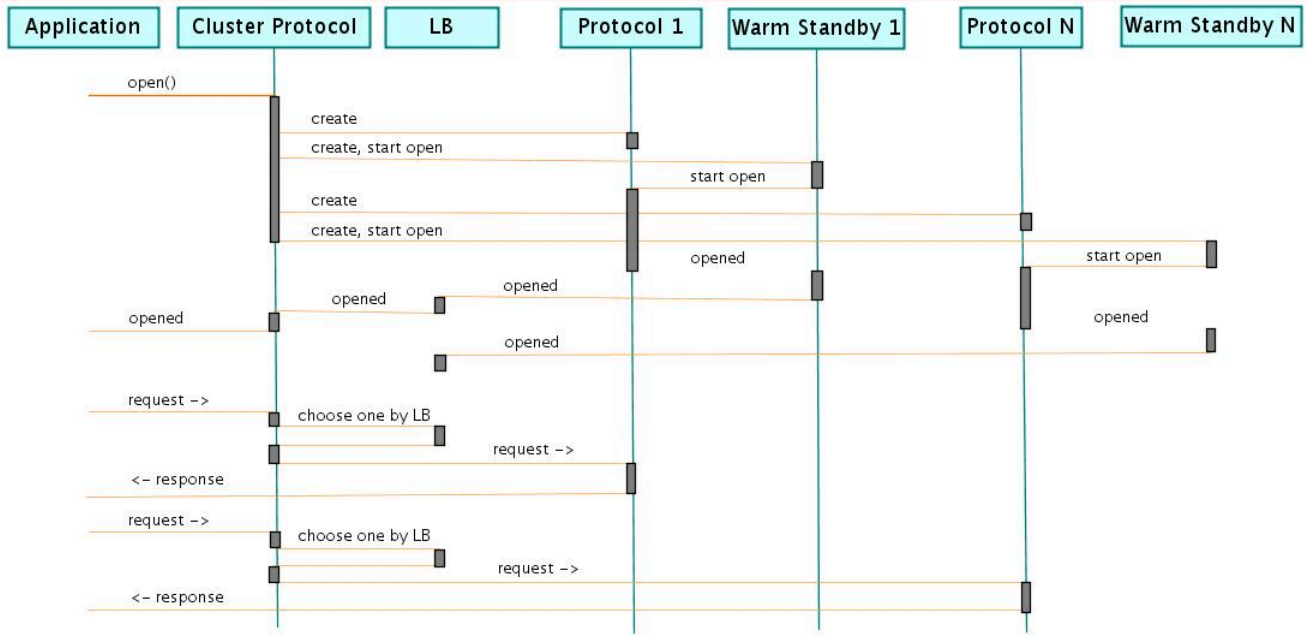
To provide a single working protocol with at least one backup, we use the Warm Standby Application Block. That application block intercepts control of the protocol interface and provides switch-over between multiple protocols, or protocol restoration, using a single Warm Standby endpoint.

The Cluster Protocol builds on this idea, allowing you to work simultaneously with a series of protocols and Warm Standby application blocks (each of which can represent one or many individual protocols) the same way that you would with a single, standard protocol. To configure the Cluster Protocol, you use a mixed list of protocol and Warm Standby endpoints gathered from Configuration Server.



The Cluster Protocol itself covers any scenarios where we need simultaneous connections through a load balancer to many servers, where each server may have one or many backups.

Sequence Diagram: Main Scenario When Using Cluster Protocol Application Block



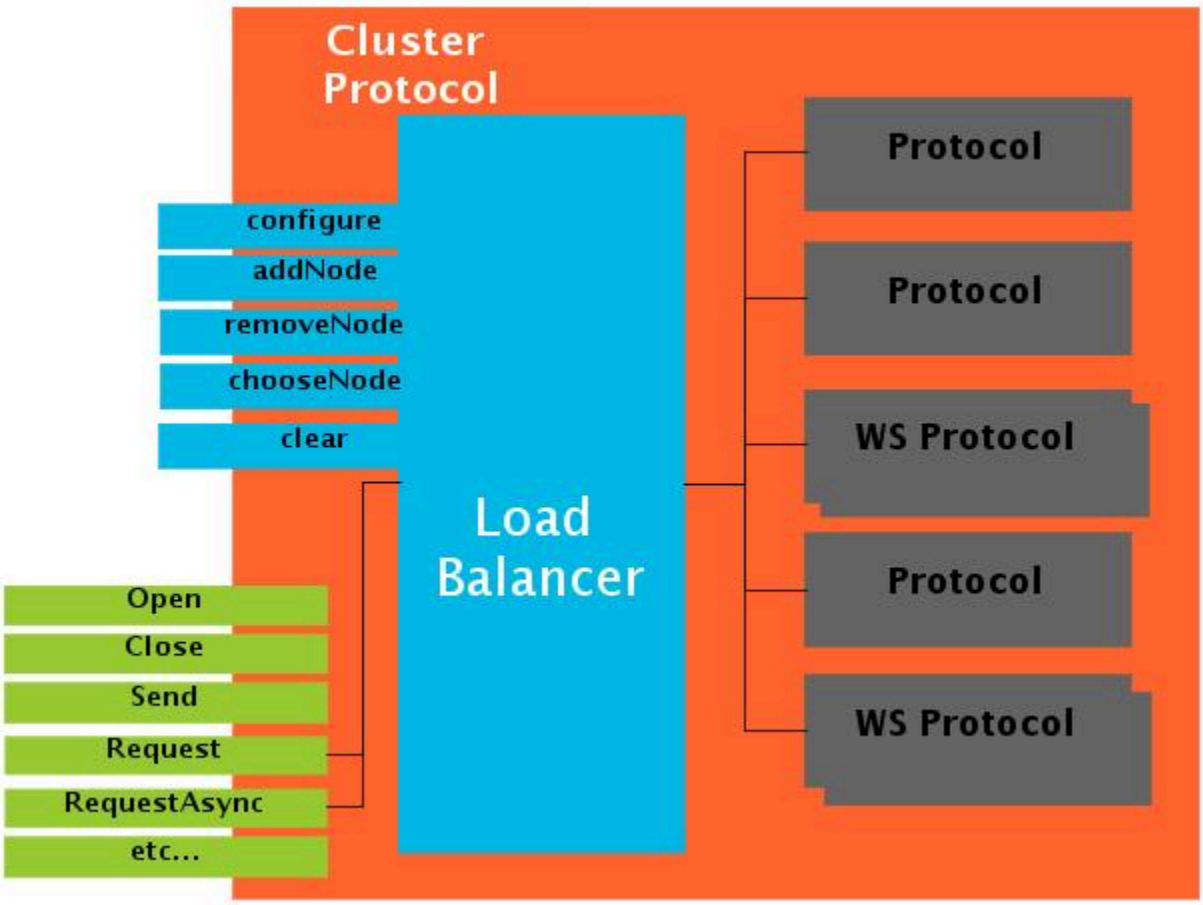
High Availability Options

The Cluster Protocol is able to control the state of protocol connections, and use the Warm Standby Application Block to keep them opened. Each cluster node may be initialized as a standard Platform SDK Endpoint, or using `WSConfig` (Warm Standby configuration) with one or more backup Endpoints added. A list of opened connections is tracked using a Load Balancer, which allows requests to be spread across connected protocols.

This gives your application the flexibility to configure one endpoint per connection ("server mode"), or to combine cluster endpoints in a single Warm Standby configuration and let the Cluster Protocol use a single connection to anyone of them ("client mode").

Load Balancing Options

The default Load Balancing strategy uses a "round robin" algorithm over connected servers for request forwarding. It is possible for you to create your own implementation of the Load Balancer interface, however, and provide it during creation of the Cluster Protocol instance.



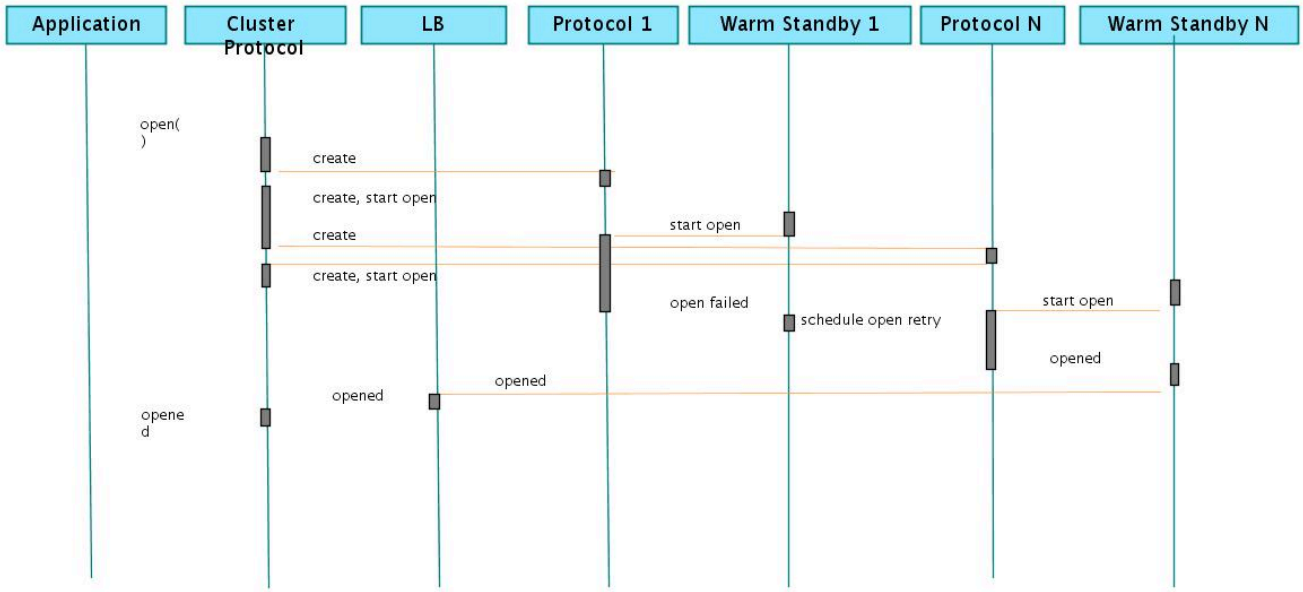
Disaster Recovery

Platform SDK does not inject any business functionality into connections, so the Cluster Protocol Application Block is able to provide disaster recovery by meeting the following requirements:

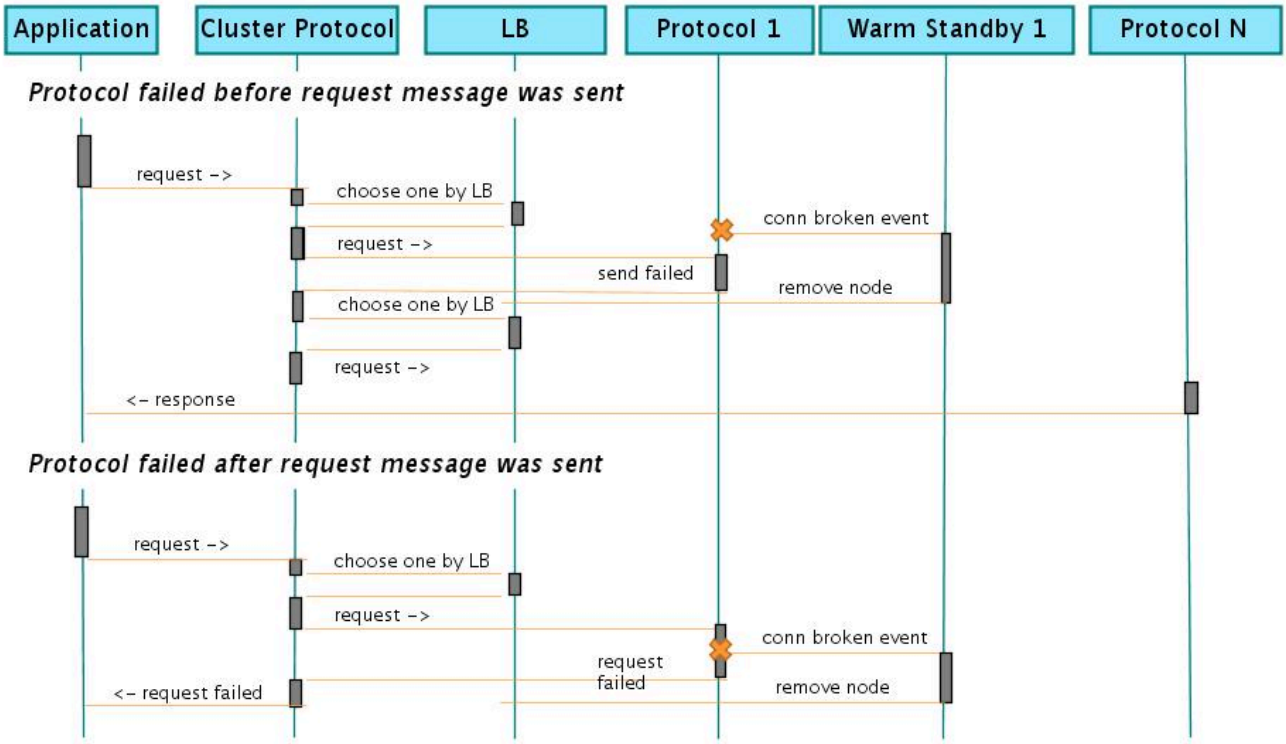
1. High availability maintains a list of active connections to ensure that requests are not sent to disconnected servers.
2. Any request that receives a `ChannelClosedOnSendException` or `ChannelClosedOnRequestException` response (because the connection was broken but not yet removed from the active list) is automatically resent to a different connection. Other exceptions are passed through to the client application to be handled manually.

There is no special configuration required to enable disaster recovery, but your application will need to include logic that handles generic IO exceptions or protocol timeout exceptions.

Sequence Diagram: Cluster Protocol with Node Connection Failure



Sequence Diagram: Cluster Protocol User Request Failure



How to Handle Lost Requests

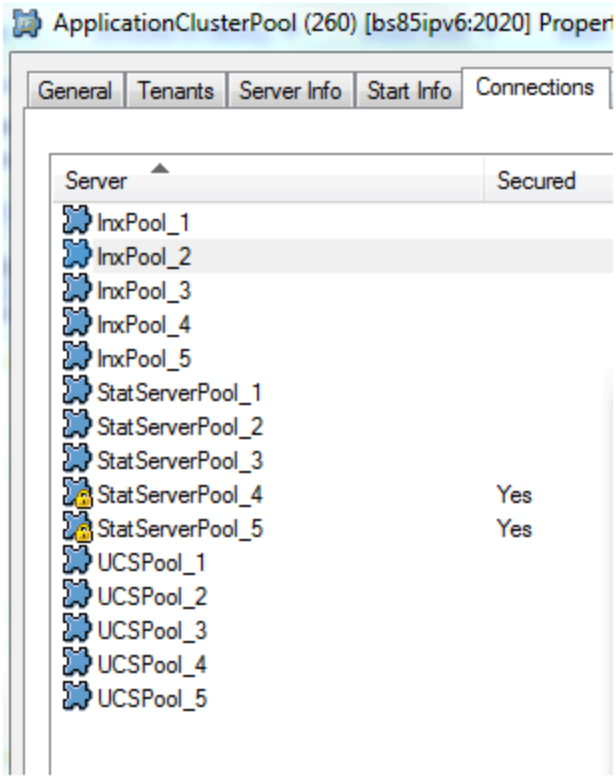
It is possible for a communication error to occur where your application sends a request but the connection is broken before any response is received. When using the Cluster Protocol you may not know if another node in the cluster handled the request. In this scenario your application won't know if the server was able to receive or process the request correctly.

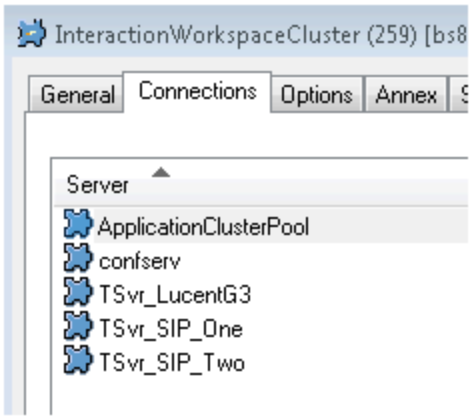
To address this, your application should include business logic that handles exceptions or null returns for a Cluster Protocol request and acts appropriately based on the type of request. For example, a request to read data can be sent again without impact, while a request that modifies data on the server may require your application to check the server state before retrying or providing notification that the request was successful.

Configuration Options

> **Important**
>
> For this release, only the UCS cluster type is supported.

The Cluster Protocol Application Block supports configuration in Configuration Manager for the client application and cluster.





Dynamic Configuration Options Updates

Cluster Protocol objects support graceful, dynamic updates of the cluster configuration, allowing you to add or remove nodes on an *opened* and *actively used* cluster protocol.

Adding a new node automatically creates the new node protocol connection in the background and attempts an asynchronous opening. The load balancer is notified about the new node connection immediately after it is connected.

Removing a connected node from the cluster protocol causes the protocol to exclude that node from the load balancer, and then disconnect the node after the protocol timeout delay. This delay allows for delivery of responses on any requests that were already started.

Code Examples

Cluster Protocol Usage Example

```
var ucsNProtocol = new UcsClusterProtocolBuilder().Build();
ucsNProtocol.ClientName = "MyClientName";
ucsNProtocol.ClientApplicationType = "MyAppType";
ucsNProtocol.SetNodesEndpoints(
        new Endpoint("ucs1", UCS_1_HOST, UCS_1_PORT),
        new Endpoint("ucs2", UCS_2_HOST, UCS_2_PORT),
```

```
        new Endpoint("ucs3", UCS_3_HOST, UCS_3_PORT));
ucsNProtocol.Open();

EventGetVersion resp1 = (EventGetVersion)ucsNProtocol.Request(RequestGetVersion.Create());
EventGetVersion resp2 = (EventGetVersion)ucsNProtocol.Request(RequestGetVersion.Create());
```

Configuration Helper Class

The `ClusterClientConfigurationHelper` class is designed to make it easier for your application to make use of this Application Block by performing the following steps:

1. Checks if client application is connected to cluster application
2. If cluster application detected, then creates endpoints for all cluster connections of the specify type
3. If cluster application not detected or no connections in cluster application found, then creates endpoints for all client application connections of the specify type (compatibility mode)
4. If connected server has backup application, endpoint will have classical Primary\Backup configuration
5. Supports specifying shared ADDP options, Transport and Application parameters in connection to cluster application. Those parameters can be overridden in connection to particular cluster node.

For more information, including an overview of the new Cluster Connection Configuration Helpers, see Using the Application Template Application Block.