# SAS® Event Stream Processing 6.2: Implementing Failover

# Failover Quick Start

## Overview

An event triggers failover under one of the following circumstances:

- There is a hardware failure.
- A process on the server fails and causes the server to stop running.

There are two ways to update models during failover: dynamic model replacement and complete model replacement.

**Note:** Ensure that you meet the software and hardware requirements before proceeding.

## Dynamic Model Replacement

There are a few restrictions on updating a model dynamically. In general, model updates that do not change schemas, update windows with connectors, or require Pattern windows to retain open pattern state can be performed dynamically. For more information, see "Dynamic Model Changes" in *SAS Event Stream Processing: Using the ESP Server*.

To update a model dynamically, use the RESTful API to update the models on each ESP server. Use the RESTful API projects request with a state value of modified, to update the model. Include the XML model definition on the request as either a URI or data.

```
http://server_name:http_port/eventStreamProcessing/v1/projects/project_name/state?
value=modified
```

For example, suppose that you have a project named sample_project on the ESP server on localhost at port 8181. You can use the ESP client to update the project with an XML model that is defined in the file model-updated.xml with one of the following commands:

```
dfesp_xml_client -url "http://localhost:8181/eventStreamProcessing/v1/projects/
sample_project/state?value=modified&projectUrl=file://model-updated.xml" -put
```

```
dfesp_xml_client -url "http://localhost:8181/eventStreamProcessing/v1/projects/
sample_project/state?value=modified" -put file://model-updated.xml
```

# Complete Model Replacement

If you cannot update the model dynamically, then you must replace the entire model. This requires stopping the previous model and starting the updated model.

**Note:** Because the model is changing, you cannot use the persist-and-restore process with the updated model.

If the schema of any of the windows with active subscribers or publishers changes, then all instances of the model must stop and all publishers and subscribers must stop.

You can restart the failover cluster after each server is updated with the new model. If the failover cluster can recover the state of the model from the message bus, then each server can be stopped and started with the new model. See “Restoring Failed Active Engine State after Restart” for details. If the failover cluster cannot recover the engine state, then you must stop all instances of the model, publishers, and subscribers. You can restart the failover cluster after each server is updated with the new model.

# Overview

SAS Event Stream Processing can use message buses to provide failover. You can configure a message bus connector or adapter to exchange CSV, JSON, or data in other text formats across the message bus. However, when enabled for failover, messaging subscribers must exchange binary event blocks. This is because only binary event blocks contain the required message IDs.

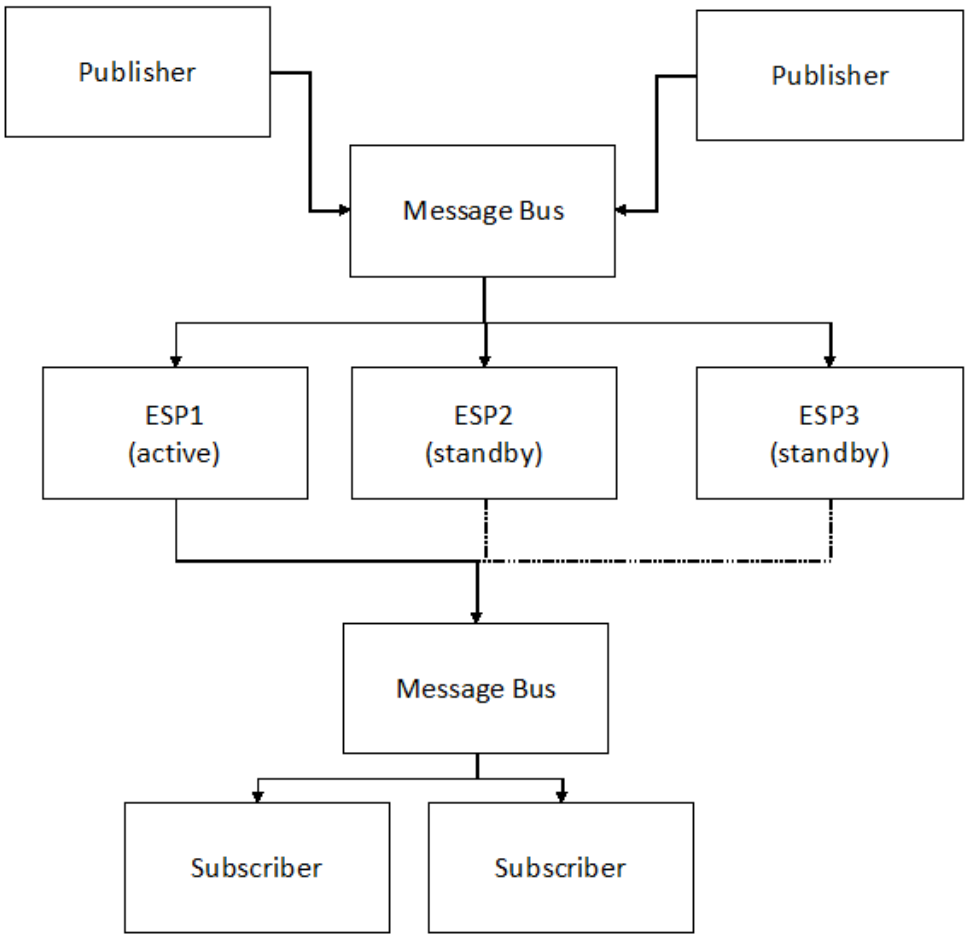
The state of any event stream processing engine in a 1+N cluster ESP servers is either “active” or “standby.” The state of a message bus with respect to another message bus in a redundant pair is either “primary” or “secondary.” When traversing a message bus, event blocks are mapped one-to-one to appliance messages. Each payload message contains exactly one event block. Thus, the terms “message” and “event block” are here used interchangeably. A payload appliance message encapsulates the event block and transports it unmodified.

The following diagram shows how an engine integrates with message buses to provide failover. It shows two separate message buses:

- one between publishers and event stream processing engines
- one between event stream processing engine and subscribers

The two buses do not have to reside on separate appliances. Regardless of whether publishers and subscribers use the same or different appliances, there are two messaging appliances for each virtual messaging appliance — a primary and secondary for messaging appliance failover.

***Figure 1*** *Engine Integration with Message Buses*




In this diagram, ESP1 is the active engine on start-up. ESP2 and ESP3 are standby engines that receive published event blocks. The standby engines do not send processed event blocks to the subscriber message bus, as depicted with dotted arrows. The event stream processing message bus connector for subscribe services is connected to the fabric. A standby engine does not send event blocks to the message bus unless it becomes active on failover.

For simplicity, this diagram illustrates a single cluster of 1+N redundant event stream processing engines. However, there can be multiple clusters of event stream processing engines, each subscribing and publishing on a different set of topics. A single publisher can send messages to multiple clusters of event stream processing engines. A single subscriber can receive messages from multiple event stream processing engines.

When the active engine in a failover cluster fails, the standby appliance connectors are notified. Then one of standby engines becomes the new active engine. The fabric tells the new active

connector the ID of the last message that it received on the window-specific “out” topic. The new active connector begins sending data on that “out” topic with ID + 1.

When appliance connectors are inactive, they buffer outbound messages (up to a configurable maximum) so that they can find messages starting with ID+1 in the buffer if necessary.

**Note:** Failover support is available only when the message format configured on the failover-enabled connectors is set to “`binary`”.

All event stream processing engines in a 1+N failover cluster must implement the same model. It is especially important that all engines in the cluster use the same engine name to coordinate the names on which messages are exchanged through the message bus.

Publishers and subscribers can continue to use the publish/subscribe API even when they are subscribing or publishing through the message bus for failover.

Enabling persistent messaging on the message bus implies the following:

- The message bus guarantees delivery of messages to and from its clients using its proprietary acknowledgment mechanisms. Duplicate message detection, lost message detection, retransmissions, and lost ACK handling are handled by the message bus.
- In general, a topic is a distribution mechanism for publishing messages that are delivered to multiple subscribers. Upon re-connection of any client and its re-subscription to an existing topic, the message bus replays all the messages that it has persisted for that topic. The number of messages or time span covered depends on the configuration of the message bus.
- At the start of the day, the message bus should be purged of all messages on related topics. Message IDs must be synchronized across all connectors.

The event stream processing engines are deployed in a 1+N redundant manner. This means the following:

- All the event stream processing engines in the 1+N cluster receive messages from the publishers.
- Only the active event stream processing engine in the 1+N cluster publishes messages to the subscribers.
- One or more backup event stream processing engines in a 1+N cluster might be located in a remote data center, and connected over the WAN.

The message bus provides a mechanism to signal to an event stream processing engine that it is the active engine in the cluster. The message bus provides a way for an engine, when notified that it is active, to determine the last message published by the previously active engine. The newly active engine can resume publishing at the appropriate point in the message stream.

Sequence numbering of messages is managed by the event stream processor’s connectors for the following purposes:

- detecting duplicates
- detecting gaps
- determining where to resume sending from after a fail-over

An event stream processing engine that is brought online resynchronizes with the day’s published data and the active engine. The process occurs after a failure or when a new engine is added to a 1+N cluster.

Event stream processing engines are deployed in 1+N redundancy clusters. All engines in the cluster subscribe to the same topics on the message bus, and hence receive exactly the same data. However, only one of the engines in the cluster is deemed the active engine at any time. Only the active engine publishes data to the downstream subscribers.

# Restoring Failed Active Engine State after Restart

When you manually bring the failed active event stream processing engine back online, it is made available as a standby to the engine in the cluster that is currently active. If the message bus is operating in "direct" mode, then persisted messages on the topic do not replay. The standby engine remains out-of-sync with other engines with injected event blocks. When the message bus is in "persistence" or "guaranteed" mode, it replays as much data as it has persisted on the "in" topic when a client reconnects. The amount of data that is persisted depends on message bus configuration and disk resources. In many cases, the data persisted might not be enough to cover one day of messages.

# Understanding Topic Naming

Topic names are mapped directly to windows that send or receive event blocks through the fabric. Because all event stream processing engines in a 1+N cluster implement the same model, they also use an identical set of topics on the fabric. However, to isolate publish flows from subscribe flows to the same window, all topic names are appended with an "in" or "out" designator. This enables clients and appliance connectors to use appliance subscriptions and publications, where event blocks can flow only in one direction.

Client applications use the standard URL format, which includes a *host*:*port* section. No publish/subscribe server exists, so *host*:*port* is not interpreted literally. It is overloaded to indicate the target 1+N cluster of event stream processing engines. All of these event stream processing engines have the same engine name, so a direct mapping between *host*:*port* and engine name is established to associate a set of clients with a specific 1+N engine cluster.

Create this mapping by configuring each connector with a "`urlhostport`" parameter that contains the *host*:*port* section of the URL passed by the client to the publish/subscribe API. This parameter must be identical for all appliance connectors in the same 1+N failover cluster.

# Available Transport Options

The following transport options are supported by the publish/subscribe API so that failover can be introduced to an existing implementation without reengineering the subscribers and publishers:

- Rabbit MQ
- Solace
- Tervela

> **IMPORTANT** Tervela transport is no longer supported or included in SAS Viya 3.5 revision 24w44. No updates to this functionality are available for earlier releases.

- Kafka

When you use the transport message bus for publish/subscribe, the publish/subscribe API uses the message bus API to communicate with the messaging appliance. It does not establish a direct TCP connection to the event stream processing publish/subscribe server.

Engines implement Rabbit MQ, Solace, Tervela, or Kafka connectors in order to communicate with the message bus. Like client publishers and subscribers, they are effectively subscribers and publishers. They subscribe to the message bus for messages from the publishers. They publish to the message bus so that it can publish messages to the subscribers.

These message buses support using direct (that is, non-persistent) or persistent messaging modes.

- Rabbit MQ connectors implement non-persistence by declaring non-durable auto-delete queues. They implement persistence by declaring durable non-auto-delete queues. The durable queues require explicit message acknowledgment, which the connector does not do. Messages are read but not consumed from the queue when they are not acknowledged.
- Solace connectors can use either direct or persistent messaging.
- Tervela connectors require that Tervela fabrics use persistent messaging for all publish/subscribe communication between publishers, subscribers, and .
- Kafka is persistent by default (subject to the cluster log.retention parameters). Connectors, adapters, and publish/subscribe clients that consume from a Kafka partition can specify the offset from which to begin consuming.

# Message Sequence Numbers

The message IDs used to synchronize engine failovers are generated by the engine. They are inserted into an event block when that event block is injected into the model. This ID is a 64-bit integer that is

unique within the scope of its project or query or window. Therefore, this ID is unique for the connector. When redundant engines receive identical input, this ID is guaranteed to be identical for an event block that is generated by different engines in a failover cluster.

The message IDs used to synchronize a rebooted engine with published event blocks are generated by the inject method of the Rabbit MQ, Solace, Tervela, or Kafka publisher client API. They are inserted into the event block when the event block is published into the appliance by the client. This ID is a 64-bit integer that is incremented for each event block published by the client. Configure the `useclientmsgid` parameter on the connector in order to use these message sequence numbers.

# Requirements for Failover

## Required Software Components

Note the following requirements when you implement failover:

- The model must implement the required Solace, Tervela, RabbitMQ, or Kafka publish and subscribe connectors. The subscribe connectors must have "`hotfailover`" configured to enable failover.
- Client subscriber applications must use the Solace, Tervela, RabbitMQ, or Kafka publish/subscribe API provided with SAS Event Stream Processing.

  For C or C++ applications, the Solace, Tervela, RabbitMQ, or Kafka transport option is requested by calling `C_dfESPpubsubSetPubsubLib()` before calling `C_dfESPpubsubInit()`.

  For Python applications, the Solace, Tervela, RabbitMQ, or Kafka transport option is requested by calling `SetPubsubLib()` before calling `Init()`.

  For Java applications, the Solace, Tervela, RabbitMQ, or Kafka transport option is invoked by inserting **`dfx-esp-solace-api.jar`**, **`dfx-esp-tervela-api.jar`**, **`dfx-esp-rabbitmq-api.jar`**, or **`dfx-esp-kafka-api.jar`** into the classpath in front of dfx-esp-api.jar.
- You must install the Solace, Tervela, RabbitMQ, or Kafka run-time libraries on platforms that host running instances of the connectors and clients. The run-time environment must define the path to those libraries (using LD_LIBRARY_PATH on Linux platforms, for example).

## Required Hardware Components

Failover requires the installation of one of the following supported message buses. The message bus must be installed in redundant fashion to avoid a single point of failure.

*Table 1* *Message Bus Requirements*

| Message Bus | Requirements |
|---|---|
| Kafka<br>[open source software-based] | Requires three or more servers for redundant installation of Kafka and associated Zookeeper instances. Provides higher throughput than other software-based buses when reading input messages from multiple Kafka topics into multiple Source windows. See redundancy details here: https://kafka.apache.org/documentation/#replication |
| RabbitMQ<br>[open source software-based] | Requires three or more servers for redundant installation. Recommended only when messages on an input RabbitMQ topic are produced by a single producer, else order is not guaranteed for the consumer (ESP redundant servers in this case). See redundancy details here: https://www.rabbitmq.com/ha.html |
| Solace VMR<br>[commercial software-based] | Requires three or more servers for redundant installation. Provides dedicated commercial level support. See redundancy details here: http://docs.solace.com/Features/VMR-Redundancy.htm<br>**Note:** SAS Event Stream Processing primary and Solace primary are not supported on the same machine. |
| Solace<br>[commercial hardware-based] | Runs on dedicated hardware and supports higher throughput than software-based buses. Provides dedicated commercial level support. See redundancy details here: http://docs.solace.com/Features/Redundancy-and-Fault-Tolerance.htm |
| Tervela<br>[commercial hardware-based] | Runs on dedicated hardware and supports higher throughput than software-based buses. Provides dedicated commercial level support. http://www.tervela.com |

Software-based message buses usually require three or more servers for redundancy. This is required to deal with the classic high availability problem outlined here: https://en.wikipedia.org/wiki/Split-brain_(computing)

A software-based message bus might or might not be co-located with the set of two or more redundant ESP servers. It should not matter whether the software bus or SAS Event Stream Processing is containerized, provided these conditions are met:

- IP connectivity between the ESP server and the message bus is maintained at all times.
- The servers or VMs are appropriately sized (CPU cores and memory) to handle maximum ESP server and message bus load. Typically, this means that all servers are identically sized. Any one of them must be able to handle a full ESP server and message bus load, even if only temporarily after a server failure.

# Using Persist/Restore

The engine persist/restore feature guarantees that a rebooted engine can be fully synchronized with other running event stream processing engines in a failover cluster. Using the persist/restore feature requires that engine state be periodically persisted by any single engine in the failover cluster.

Be aware that a persist operation can be triggered by the model itself, so in a failover cluster, it would generate redundant persist data. Alternatively, a client can use the publish/subscribe API to trigger a persist by an engine. The URL provided by the client specifies *host*:*port*, which maps to a specific failover cluster. The messaging mechanism guarantees that only one engine in the cluster receives the message and executes the persist. On a Rabbit MQ server, this is achieved by having the connector use a well-known queue name. Only a single queue exists, and the first engine to consume the persist request performs the persist action. On Solace appliances, this is achieved by setting Deliver-To-One on the persist message to the metadata topic. On the Tervela Data Fabric this is achieved by sending the persist message to an inbox owned by only one engine in the failover cluster. On a Kafka cluster, the Kafka connector consumes messages on the topic reserved for metadata requests using a global group ID. In this way, only one consumer in the group sees the message.

The persist data is always written to disk. The target path for the persist data is specified in the client persist API method. Any client that requests persists of an engine in a specific failover cluster should specify the same path. This path can point to shared disk, so successive persists do not have to be executed by the same engine in the failover cluster.

The other requirement is that the model must execute a restore on boot so that a rebooted standby engine can synchronize its state using the last persisted snapshot. On start-up, appliance connectors always get the message ID of the last event block that was restored. If the restore failed or was not requested, the connector gets 0. This message ID is compared to those of all messages received through replay by a persistence-enabled appliance. Any duplicate messages are ignored.

# Failover with RabbitMQ

## Failover with RabbitMQ

### Installing and Configuring RabbitMQ

For information about installing RabbitMQ, see "Using the RabbitMQ Connector" in *SAS Event Stream Processing: Connectors and Adapters*.

## Required Message Bus Configuration with Rabbit MQ

You must install the presence-exchange plug-in in order to use the Rabbit MQ server in a 1+N-way failover topology. You can download the plug-in from https://github.com/tonyg/presence-exchange.

## Required Client Configuration with RabbitMQ

A RabbitMQ client application requires a client configuration file named **`rabbitmq.cfg`** in the current directory to provide Rabbit MQ connectivity parameters.

See the documentation of the `C_dfESPpubsubSetPubsubLib()` publish/subscribe API function for details about the contents of these configuration files.

## Topic Naming with RabbitMQ

The topic name format used on Rabbit MQ appliances is as follows:

*`host:port/project/contquery/window/direction`*

*direction* takes the value "I" or "O". Because this information appears in a client URL, it is easy for clients to determine the correct appliance topic. SAS Event Stream Processing appliance connectors use their configured "`urlhostport`" parameter to derive the "*host*:*port*" section of the topic name, and the rest of the information is known by the connector.

## Determining Engine Active/Standby State with RabbitMQ

All SAS Event Stream Processing subscribers declare a Rabbit MQ exchange of type `x-presence`. The exchange is named after the configured exchange name with `_failoverpresence` appended. Then subscribers bind to a queue to both send and receive notifications of bind and unbind actions by all peers.

All engines receive send and receive notifications in the same order. Therefore, they maintain the same ordered list of present event stream processing engines (that is, those that are bound). The first engine in the list is always the active engine. When a notification is received, an engine compares its current active/standby state to its position in the list and updates its active/standby state when necessary.

To enable log messages that indicate which ESP server is active and which is standby, use the `-loglevel esp=info` option when instantiating the ESP server.

## New Engine Active Actions on Failover with Rabbit MQ

When a subscriber connector starts in standby state, it creates a queue that is bound to the out topic that is used by the currently active connector. The subscriber consumes and discards all messages received on this queue, except for the last one received. When its state changes from standby to active, the subscriber does the following:

- extracts the message ID from the last received message

- deletes its `receive` queue
- begins publishing starting with the following message:

```
ID = last message ID + 1
```

The connector can obtain this message and subsequent messages from the queue that it maintained while it was inactive. It discards older messages from the queue.

## Metadata Exchanges with Rabbit MQ

The Rabbit MQ publish/subscribe API handles the `C_dfESPpubsubQueryMeta()` and `C_dfESPpubsubPersistModel()` methods as follows:

- The connectors listen for metadata requests on a special topic named **`"urlhostport/M"`**.
- The client sends formatted messages on this topic in request or reply fashion.
- The request messages are always sent using a well-known queue name with multiple consumers. This is to ensure that no more than one engine in the failover cluster handles the message.
- The response is sent back to the originator, and contains the same information provided by the native publish/subscribe API.

# Testing Failover with RabbitMQ

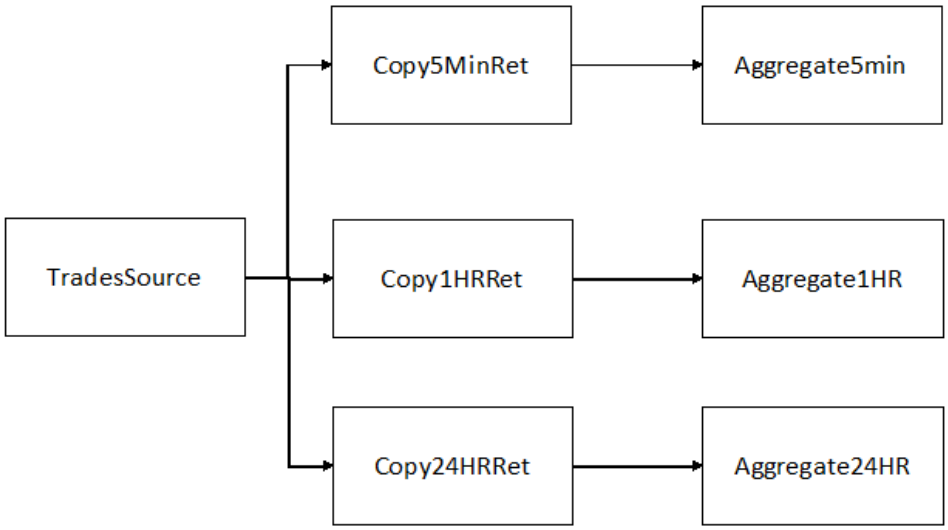
## Setting Up the Model

After you have installed and configured RabbitMQ, you can create a model with the appropriate RabbitMQ connectors and adapters in a failover environment. Setting up failover requires at least two systems.

SAS Event Stream Processing supports homogenous and heterogeneous failover environments. The following example uses two Red Hat Linux systems in a homogeneous failover environment: RH1 and RH2

The following model shows a Source window feeding events to three Copy windows. Each copy window has a different retention policy. Each of those Copy windows feeds events to an Aggregate window.

*Figure 2* *Model Setup*




A connector to the source window (**TradesSource**) enables the model to receive data from a dynamic queue bound to a specific routing key. When data is published by the file and socket adapter using the transport option, it publishes to a dynamic RabbitMQ queue. This queue is subsequently consumed by a connector defined in the Source window. In order to publish to a RabbitMQ exchange, an exchange must be created. If you do not create an exchange manually, the connector creates the exchange automatically. An exchange serves the role of a message router. In this test, the same exchange is also used for subscribing.

The RabbitMQ connector is defined in one of the Aggregate windows (**Aggregrate24HR**) to push events to a second RabbitMQ queue through the exchange. A second file and socket adapter subscribes to events by creating a dynamic queue for an exchange and routing key and writes events to a file.

To ensure that events successfully flow through the model and are written to a CSV file, use **`xml/vwap_xml/trades1M.csv`** on RH2. You can obtain this file from the support web site.

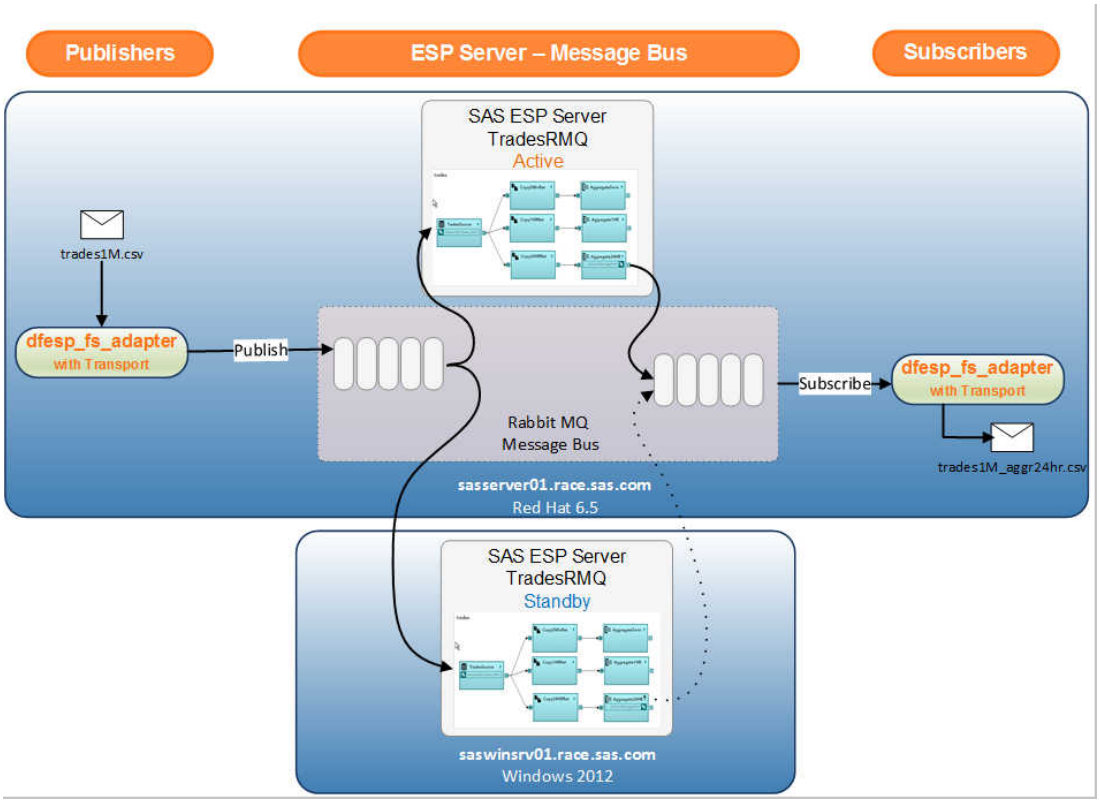
Set up the failover environment as follows:

1. Copy the original project to the second machine (in this case RH2).
2. Modify the SAS Event Stream Processing JSON file to use an ESP server on RH2.
3. Start ESP servers on both machines. Use the `-loglevel esp=info` option to display active and standby messages in the ESP server log.
4. Define a RabbitMQ exchange (**Trades**) with type *topic* and durability *transient*.
5. Define RabbitMQ connectors for a Source window (**TradesSource**) and an Aggregate window (**Aggregate24HR**).
6. With the transport option, use the file and socket adapter to publish events to a Source window (**TradesSource**).

7 Use a file and socket adapter to subscribe from an Aggregate window (**Aggregate24HR**) using the transport option.

The following diagram provides an overview of the failover environment that is created. **TradesRMQ** is the engine instance and trades is the project name (not to be confused with the exchange name). The RabbitMQ message bus, adapters, and one instance of the engine run on RH1. The CSV "event" files, published and subscribed, are also located on RH1. The only thing that runs on RH2 is the second instance of the **TradesRMQ** engine.

***Figure 3*** *Failover Environment*



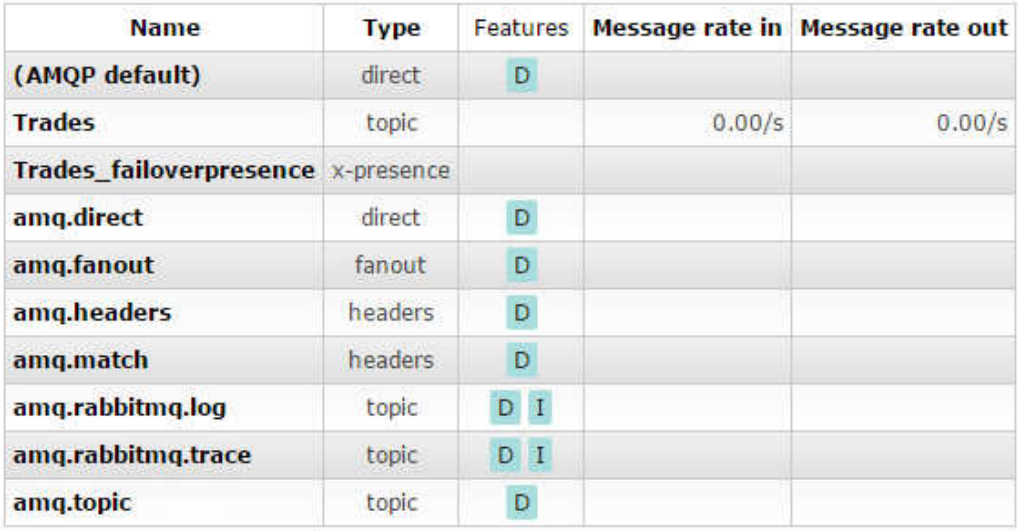

# Testing Parallel Models

After you install and configure RabbitMQ and edit connectors in the models to use RabbitMQ queues, test the models for failover. Set up with two models. Two tests are necessary: one test without a failed server and one test with a failed server.

ESP servers had been started earlier. The next step loaded the modified model in Test mode in SAS Event Stream Processing Studio on each host. It then started the modified model. At this point, the adapters have not been started, so there are no events flowing.

Use the web-based RabbitMQ management tool to monitor and watch queues. After starting the models in Test mode, you can view the list of exchanges using the management tool, as seen in the following figure.

*Figure 4* *Viewing Exchanges with the RabbitMQ Management Tool*

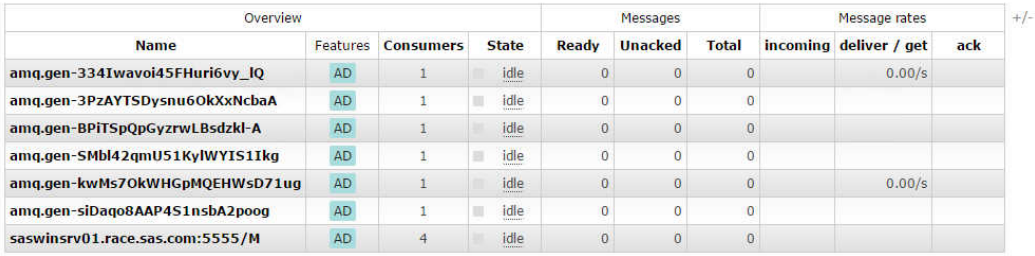


| Name | Type | Features | Message rate in | Message rate out |
|---|---|---|---|---|
| (AMQP default) | direct | D | | |
| Trades | topic | | 0.00/s | 0.00/s |
| Trades_failoverpresence | x-presence | | | |
| amq.direct | direct | D | | |
| amq.fanout | fanout | D | | |
| amq.headers | headers | D | | |
| amq.match | headers | D | | |
| amq.rabbitmq.log | topic | D I | | |
| amq.rabbitmq.trace | topic | D I | | |
| amq.topic | topic | D | | |

After the model is started, the subscriber connectors automatically create the **Trades_failoverpresence** exchange. The subscriber connector is configured with the `hotfailover` argument, which triggers the creation of the **Trades_failoverpresence** exchange. Notice that the exchange **Type** is **x-presence**. This is the plug-in that was added after you installed RabbitMQ. This mechanism detects what engine is active and when engine binds to a Standby engine if the Active engine fails.

Even though no events have been published to the model, queues have been created. There are queues for connectors, subscribers, failover, and metadata. The metadata includes information about messages. You can view additional information about each queue by selecting the queue name.

*Figure 5* *Viewing Queue Information with the RabbitMQ Management Tool*



| Overview | | | | Messages | | | Message rates | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Features | Consumers | State | Ready | Unacked | Total | incoming | deliver / get | ack |
| amq.gen-334Iwavoi45FHuri6vy_lQ | AD | 1 | idle | 0 | 0 | 0 | | 0.00/s | |
| amq.gen-3PzAYTSDysnu6OkXxNcbaA | AD | 1 | idle | 0 | 0 | 0 | | | |
| amq.gen-BPiTSpQpGyzrwLBsdzkl-A | AD | 1 | idle | 0 | 0 | 0 | | | |
| amq.gen-SMbl42qmU51KylWYIS1Ikg | AD | 1 | idle | 0 | 0 | 0 | | | |
| amq.gen-kwMs7OkWHGpMQEHWsD71ug | AD | 1 | idle | 0 | 0 | 0 | | 0.00/s | |
| amq.gen-siDaqo8AAP4S1nsbA2poog | AD | 1 | idle | 0 | 0 | 0 | | | |
| saswinsrv01.race.sas.com:5555/M | AD | 4 | idle | 0 | 0 | 0 | | | |

If you specified `-loglevel esp=info` at ESP server start-up, then messages appear in the log regarding active/standby status. For example, if the first ESP server was started and the model was loaded on RH1, then a message similar to the following is displayed:

```
2016-04-29T12:13:15,875; WARN; 00000053; DF.ESP; (dfESPrmqConnector.cpp:570);
dfESPrmqConnector::goActive(): Hot failover state switched to active
```

Similarly, when the ESP server is started and the model is loaded on the second machine, then a message similar to the following is displayed:

```
2016-04-29T12:16:40,909; INFO; 00000050; DF.ESP; (dfESPrmqConnector.cpp:1746);
dfESPrmqConnector::start(): Hot failover state switched to standby
```

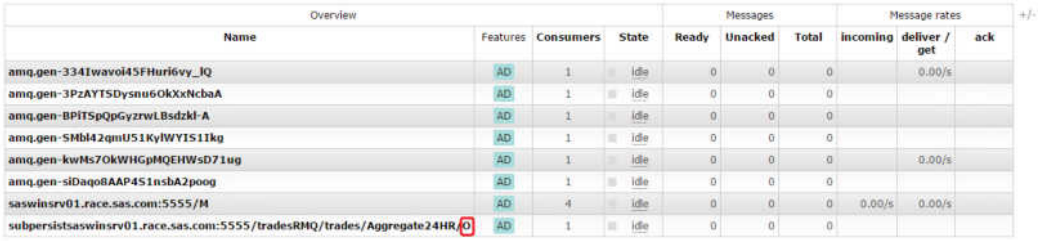
After the model is running in test mode, start the file and socket adapter to subscribe to the **Aggregate24HR** window within the model using the transport option. Use the following command:

```
$DFESP_HOME/bin/dfesp_fs_adapter -k sub -h "dfESP://host:port/tradesRMQ/
trades/Aggregate24HR?snapshot=false" -f trades1M_aggr24hr.csv -t csv -l rabbitmq
```

The `-l rabbitmq` argument on the command line specifies the transport type. The adapter looks for RabbitMQ connection information in the **`rabbitmq.cfg`** in the current directory. Start the subscriber before you start the publisher to ensure that all events are captured from the model.

The subscriber adapter creates a new associated queue that is specified as the "Output" queue. This is indicated by the last character in the queue name. Events processed by the **Aggregate24HR** window are placed in this queue where the adapter retrieves them and writes them to the **`trades1M_aggr24hr.csv`** file.

***Figure 6*** *Verifying That a Queue Is Specified as the Output Queue*



| Overview | | | | Messages | | | Message rates | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Features | Consumers | State | Ready | Unacked | Total | incoming | deliver / get | ack |
| amq.gen-334Iwavoi45FHuri6vy_IQ | AD | 1 | idle | 0 | 0 | 0 | | 0.00/s | |
| amq.gen-3PzAYTSDysnu6OkXxNcbaA | AD | 1 | idle | 0 | 0 | 0 | | | |
| amq.gen-BPiTSpQpGyzrwLBsdzkl-A | AD | 1 | idle | 0 | 0 | 0 | | | |
| amq.gen-SMbl42qmU51KylWYIS11kg | AD | 1 | idle | 0 | 0 | 0 | | | |
| amq.gen-kwMs7OkWHGpMQEHWsD71ug | AD | 1 | idle | 0 | 0 | 0 | | 0.00/s | |
| amq.gen-siDaqo8AAP4S1nsbA2poog | AD | 1 | idle | 0 | 0 | 0 | | | |
| saswinsrv01.race.sas.com:5555/M | AD | 4 | idle | 0 | 0 | 0 | 0.00/s | 0.00/s | |
| subpersistsaswinsrv01.race.sas.com:5555/tradesRMQ/trades/Aggregate24HR/O | AD | 1 | idle | 0 | 0 | 0 | | | |

Now that the model is running in test mode and a subscriber adapter has been started, you can start a file and socket adapter to publish events into the model. Again specify the `-l rabbitmq` argument to specify the RabbitMQ message bus. Use the following command:

```
$DFESP_HOME/bin/dfesp_fs_adapter -k pub -h "dfESP://host:port/tradesRMQ/
trades/TradesSource" -f trades1M.csv -t csv -l rabbitmq
```

Typically, the block parameter is specified to reduce overhead, but in this scenario, blocking was not specified. This results in a blocking factor of one. This results in slow throughput.

As events flow through the model, it is possible to review the number of event blocks incoming to the queue and delivered to the connector.

***Figure 7*** *Reviewing the Number of Event Blocks Delivered to the Connector*

| Overview | | | | Messages | | | Message rates | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Features | Consumers | State | Ready | Unacked | Total | incoming | deliver / get | ack |
| amq.gen-334Iwavoi45FHuri6vy_IQ | AD | 1 | idle | 0 | 0 | 0 | | 0.00/s | |
| amq.gen-3PzAYTSDysnu6OkXxNcbaA | AD | 1 | running | 0 | 0 | 0 | 2,906/s | 2,927/s | |
| amq.gen-BPiTSpQpGyzrwLBsdzkl-A | AD | 1 | running | 0 | 0 | 0 | 2,853/s | 2,853/s | |
| amq.gen-SMbl42qmU51KylWYIS11kg | AD | 1 | running | 0 | 0 | 0 | 2,853/s | 2,853/s | |
| amq.gen-kwMs7OkWHGpMQEHWsD71ug | AD | 1 | idle | 0 | 0 | 0 | | 0.00/s | |
| amq.gen-siDaqo8AAP4S1nsbA2poog | AD | 1 | running | 0 | 0 | 0 | 2,906/s | 2,928/s | |
| saswinsrv01.race.sas.com:5555/M | AD | 4 | idle | 0 | 0 | 0 | 0.00/s | 0.00/s | |
| subpersistsaswinsrv01.race.sas.com:5555/tradesRMQ/trades/Aggregate24HR/O | AD | 1 | running | 0 | 0 | 0 | 2,853/s | 2,853/s | |

After all events are published, use an HTTP request to the HTTP port on each ESP server to confirm that both engine instances processed the same number of records. You can also acquire this information using the UNIX curl command.

*Figure 8* *Using the curl Command to Confirm Number of Engine Instances Processed*


Each model should have processed the same number of records and all window counts should match.

Finally, review the file that contains the events from the **Aggregate24HR** window using the subscribe adapter.

```
[sasinst@sasserver01 ~]$ ll trades1M*
-rw-r-r-. 1 sasinst sas  98345849 Apr 15 10:41 trades1M_aggr24hr.csv
```

# Failover Testing

To ensure that processing continues when one of the ESP servers fail, repeat the previous test and terminate the active ESP server in the middle of processing events. Suppose that the active ESP server running on RH1 was terminated by using the Ctrl-C interrupt. SAS Event Stream Processing Studio issues an error message that it can no longer communicate with the ESP server.

Several queues associated with the terminated ESP server are removed, but the subscriber output queue is still delivering messages to the adapter.

*Figure 9* *Verifying That the Subscriber Output Queue Is Delivering Messages to the Adapter*

| Overview | | | | Messages | | | Message rates | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Features | Consumers | State | Ready | Unacked | Total | incoming | deliver / get | ack |
| amq.gen-X1UzgtM6KqKHB8fpgRQrwQ | AD | 1 | idle | 0 | 0 | 0 | | 0.00/s | |
| amq.gen-aCPYRpzdjDj8CYCle0Z65A | AD | 1 | running | 0 | 0 | 0 | 5,247/s | 5,237/s | |
| saswinsrv01.race.sas.com:5555/M | AD | 2 | idle | 0 | 0 | 0 | 0.00/s | 0.00/s | |
| subpersistsaswinsrv01.race.sas.com:5555/tradesRMQ/trades/Aggregate24HR/O | AD | 1 | running | 0 | 0 | 0 | 4,139/s | 4,139/s | |

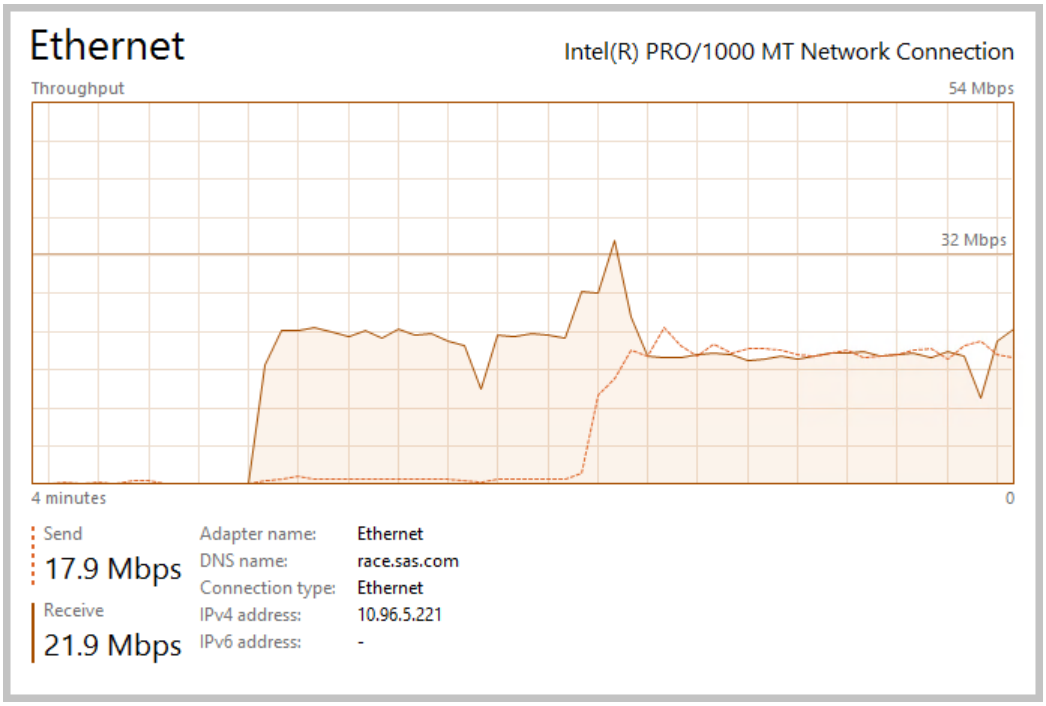
If you specified `-loglevel esp=info` at ESP server start-up, the following messages confirm that failover occurred and that the standby engine is now the active engine.

```
2016-04-29T12:27:37,946; INFO; 00000053; DF.ESP; (dfESPrmqConnector.cpp:342);
 dfESPrmqConnector::sendSerializedBlock():
Standby sending buffered messages, last message id = 184211, current message id = 259306

2016-04-29T12:27:51,387; INFO; 00000053; DF.ESP; (dfESPrmqConnector.cpp:383);
dfESPrmqConnector::sendSerializedBlock():
Standby synced, current message id = 259306

2016-04-29T12:27:51,446; WARN; 00000053; DF.ESP; (dfESPrmqConnector.cpp:570);
 dfESPrmqConnector::goActive(): Hot failover state switched to active
```

A second way to confirm that the RH2 ESP server has switched active servers is to review the network traffic associated with RH2.

The following figure shows the host traffic received when events start flowing through the RH2 engine. The received traffic, ~20MB/sec, represents the events retrieved from the publish queue (that is, sent to the model). After the failover occurs, you can see that sent traffic spikes to ~18MB/sec, which represents the traffic sent from the connector on RH2 to the queue on RH1.

***Figure 10*** *Reviewing Network Traffic to Confirm Server Switch*




Checking the subscriber file created by the file and socket adapter shows that it has been replaced. The size is the same as the file created when no failover scenario was created.

```
[sasinst@sasserver01 ~]$ ll trades1M*
-rw-r-r-. 1 sasinst sas  98345849 Apr 15 10:49 trades1M_aggr24hr.csv
```

The output file from the first test is renamed. Use the Linux `diff` command to compare to the one created in the failover scenario. The results show that the files are identical, indicating a successful failover.

You can repeat the same test with the RH2 ESP server as the active server. Terminating the RH2 server should produce the same result. The model on the RH1 engine should continue to run and the adapter on the RH1 machine should continue to write events to the CSV file.

# Failover with Solace

## Required Appliance Configuration with Solace

For information about the minimum configuration required by a Solace appliance used in a 1+N-way failover topology, see "Using the Solace Systems Connector" in *SAS Event Stream Processing: Connectors and Adapters*.

## Required Client Configuration with Solace

A Solace client application requires a client configuration file named **`solace.cfg`** in the current directory to provide appliance connectivity parameters.

See the documentation of the `C_dfESPpubsubSetPubsubLib()` publish/subscribe API function for details about the contents of these configuration files.

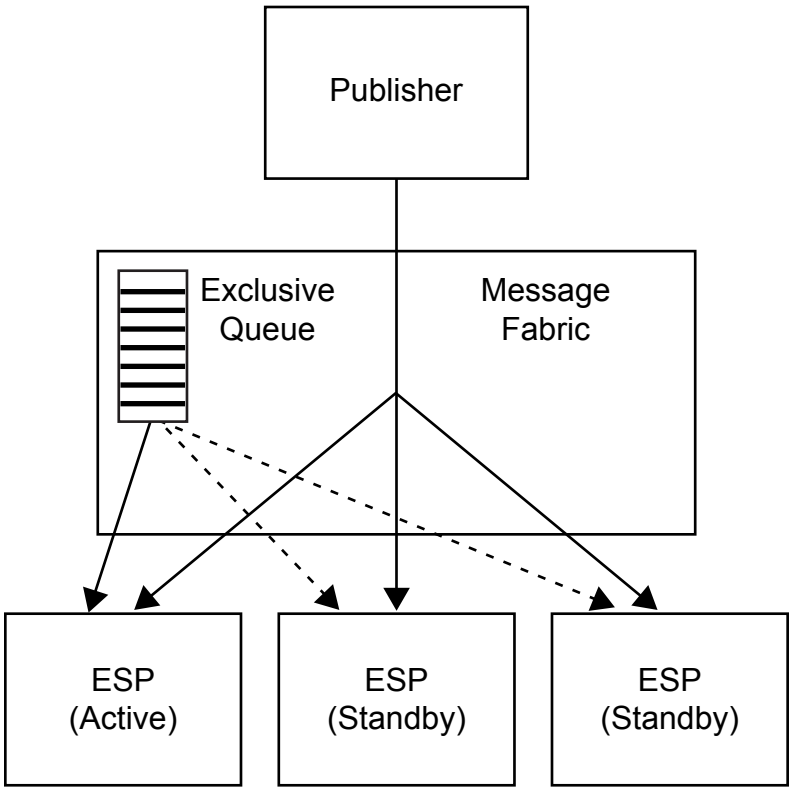
## Topic Naming with Solace

The topic name format used on Solace appliances is as follows: *`host:port/project/contquery/window/direction`*, where *direction* takes the value "I" or "O". Because all this information is present in a client URL, it is easy for clients to determine the correct appliance topic. SAS Event Stream Processing appliance connectors use their configured "`urlhostport`" parameter to derive the "*host*:*port*" section of the topic name, and the rest of the information is known by the connector.

## Determining Engine Active/Standby State with Solace

For Solace appliances, an exclusive messaging queue is shared amongst all the engines in the 1+N cluster. The queue is used to signal active state. No data is published to this queue. It is used as a semaphore to determine which engine is the active at any point in time.

*Figure 11* *Determining Active State*




Engine active/standby status is coordinated among the engines using the following mechanism:

1. When a SAS Event Stream Processing subscriber appliance connector starts, it acts as a queue consumer. It tries to bind to the exclusive queue that has been created for the engine cluster.
2. If the connector is the first to bind to the queue, it receives a “Flow Active” indication from the messaging appliance API. This signals to the connector that it is now the active engine.
3. As other connectors bind to the queue, they receive a “Flow Inactive” indication. This indicates that they are standby engines, and should not be publishing data onto the message bus.
4. If the active engine fails or disconnects from the appliance, one of the standby connectors receives a “Flow Active” indication from the messaging appliance API. Originally, this is the second standby connector to connect to the appliance. This indicates that it is now the active engine in the cluster.

# New Engine Active Actions on Failover with Solace

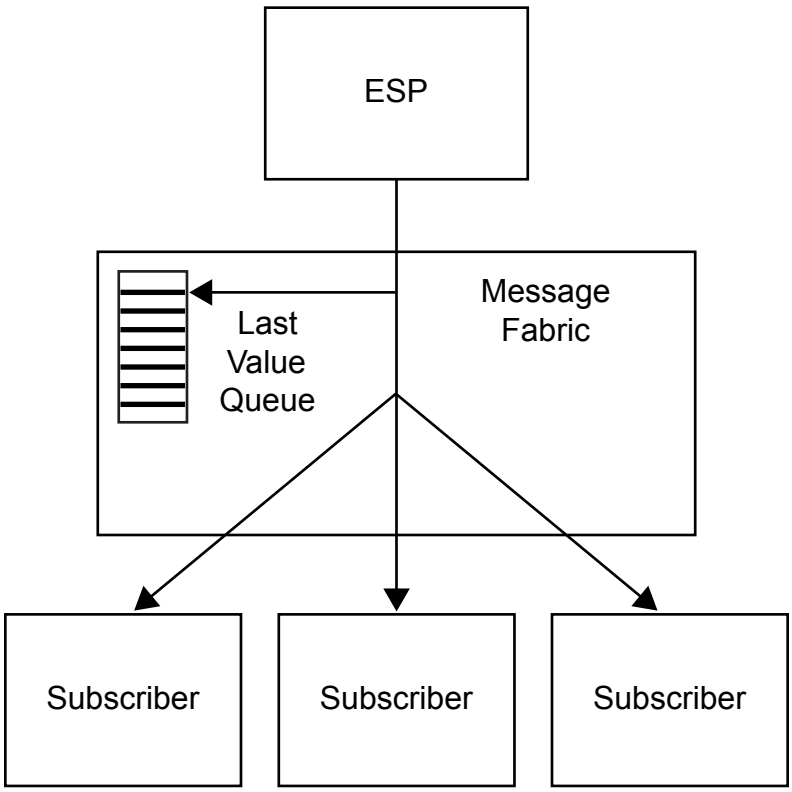
The newly active engine determines, from the message bus, the last message published by the previously active engine for the relevant window. To assist in this process, guaranteed messaging Last Value Queues (LVQs) are used.

LVQs are subscribed to the same “out” topics that are used by the appliance connectors. An LVQ has the unique characteristic that it maintains a queue depth of one message, which contains the last message published on the topic to which it subscribed. When the engine can publish messages as

"direct" or "guaranteed", those messages can always be received by a guaranteed messaging queue that has subscribed to the message topic. Thus, the LVQ always contains the last message that an engine in the cluster published onto the message bus.

When an engine receives a "Flow Active" indication, it binds to the LVQ as a browser. It then retrieves the last message published from the queue, saves its message ID, disconnects from the LVQ, and starts publishing starting with message ID = the saved message ID + 1. The connector can obtain this message and subsequent messages from the queue that it maintained while it was inactive. It can ignore newly received messages until the one with ID = saved message ID + 1 is received.

***Figure 12*** *Last Value Queues*




# Metadata Exchanges with Solace

The Solace publish/subscribe API handles the `C_dfESPpubsubQueryMeta()` and `C_dfESPpubsubPersistModel()` methods as follows:

- The connectors listen for metadata requests on a special topic named **`"urlhostport/M"`**.
- The client sends formatted messages on this topic in request or reply fashion.
- The request messages are always sent using Deliver-To-One. This is to ensure that no more than one engine in the failover cluster handles the message.
- The response is sent back to the originator, and contains the same information provided by the native publish/subscribe API.

# Failover with Tervela

**IMPORTANT** Tervela transport is no longer supported or included in SAS Viya 3.5 revision 24w44. No updates to this functionality are available for earlier releases.

## Required Client Configuration with Tervela

A Tervela client application requires a client configuration file named **`client.config`** in the current directory to provide appliance connectivity parameters.

See the documentation of the `C_dfESPpubsubSetPubsubLib()` publish/subscribe API function for details about the contents of these configuration files.

## Required Appliance Configuration with Tervela

A Tervela appliance used in a 1+N-way failover topology requires the following configuration at a minimum:

- A client user name and password to match the connector's `tvauserid` and `tvapassword` configuration parameters.
- The inbound and outbound topic strings and associated schema. (See topic string formats described previously.)
- Publish or subscribe entitlement rights associated with a client user name described previously.

## Topic Naming with Tervela

The topic name format used on Tervela appliances is as follows: "`SAS.ENGINES.`*`engine.project.contquery.window.direction`*", where *direction* takes the value "IN" or "OUT". SAS Event Stream Processing appliance connectors know this information, so it is easy for them to determine the correct appliance topic.

Clients must be able to map the "*host*:*port*" section of the received URL to the engine section of the topic name. This mapping is obtained by the client by subscribing to a special topic named `SAS.META.`*`host:port`*. The SAS Event Stream Processing appliance connectors use their configured "`urlhostport`" parameter to build this topic name. They publish a metadata message to the topic that includes the "*host*:*port*" to engine mapping. Only after receiving this message can clients send or

receive event block data. SAS Event Stream Processing appliance connectors automatically send this message when the SAS Event Stream Processing model is started.

# Determining Engine Active/Standby State with Tervela

When using the Tervela Data Fabric, engine active/standby status is signaled to the engines using the following mechanism:

1. When an SAS Event Stream Processing subscriber appliance connector starts, it attempts to create a "well-known" Tervela inbox. It uses the engine name for the inbox name, which makes it specific to the failover cluster. If successful, that connector takes ownership of a system-wide Tervela GD context, and becomes active. If the inbox already exists, another connector is already active. The connector becomes standby and does not publish data onto the message bus.
2. When a connector becomes standby, it also connects to the inbox, and sends an empty message to it.
3. The active connector receives an empty message from all standby connectors. It assigns the first responder the role of the active standby connector by responding to the empty message. The active connector maintains a map of all standby connectors and their status.
4. If the active connector receives notification of an inbox disconnect by a standby connector, it notifies another standby connector to become the active standby, using the same mechanism.
5. If the active engine fails, the inbox also fails. At this point the fabric sends a `TVA_ERR_INBOX_COMM_LOST` message sent to the connected standby connectors.
6. When the active standby connector receives a `TVA_ERR_INBOX_COMM_LOST` message, it becomes the active engine in the failover cluster. It then creates a new inbox as described in step 1.
7. When another standby connector receives a `TVA_ERR_COMM_LOST` message, it retains standby status. It also finds the new inbox, connects to it, and send an empty message to it.

# New Engine Active Actions on Failover with Tervela

Active Tervela appliance connectors own a cluster-wide Tervela GD context with a name that matches the configured "`tvaclientname`" parameter. This parameter must be identical for all subscribe appliance connectors in the same failover cluster. When a connector goes active because of a failover, it takes over the existing GD context. This allows it to query the context for the ID of the last successfully published message, and this message ID is saved.

The connector then starts publishing starting with message ID = the saved message ID + 1. The connector can obtain this message and subsequent messages from the queue that it maintained

while it was inactive. Alternatively, it can ignore newly received messages until the one with ID = saved message ID + 1 is received.

## Metadata Exchanges with Tervela

The Tervela publish/subscribe API handles the `C_dfESPpubsubQueryMeta()` method as follows:

- On start-up, appliance connectors publish complete metadata information about special topic "`SAS.META.host:port`". This information includes the "`urlhostport`" to engine mapping needed by the clients.
- On start-up, clients subscribe to this topic and save the received metadata and engine mapping. To process a subsequent `C_dfESPpubsubQueryMeta()` request, the client copies the requested information from the saved response(s).

The Tervela publish/subscribe API handles the `C_dfESPpubsubPersistModel()` method as follows.

- Using the same global inbox scheme described previously, the appliance connectors create a single cluster-wide inbox named "engine_meta".
- The client derives the inbox name using the received "`urlhostport`" - engine mapping, and sends formatted messages to this inbox in request or reply fashion.
- The response is sent back to the originator, and contains the same information provided by the native publish/subscribe API.

# Failover with Kafka

## Required Message Bus Configuration with Kafka

When you use a Kafka cluster in a failover topology, the Kafka connectors must have access to an Apache Zookeeper cluster. The connectors require this access in order to monitor the presence of other event stream processing servers in the failover group. When Zookeeper is installed as part of the Kafka cluster installation, you can use it for monitoring by configuring *`host:port`* on the Kafka connectors. You can download Zookeeper from https://zookeeper.apache.org.

**Note:** The Zookeeper configuration must specify a value for `tickTime` no greater than 500 milliseconds. If Zookeeper is already in use, this setting might conflict with client requirements. If there are conflicts, a separate Zookeeper installation is required for failover.

**Note:** Do not configure Kafka connectors or adapters or client transports with partition = –1. 1+N-way failover is supported only when all components read or write a single fixed partition. If your model includes multiple publisher connectors reading from different partitions, then those streams must not be joined in the model. They must propagate to different subscriber connectors. This

ensures that all subscriber connectors in the set of failover engines receive events in exactly the same order.

# Required Client Configuration with Kafka

An event stream processing publish/subscribe client application that uses Kafka requires a client configuration file named **kafka.cfg**. This configuration file must exist in the current directory to provide Kafka connectivity parameters. See the documentation of the `C_dfESPpubsubSetPubsubLib()` publish/subscribe API function for details about the contents of these configuration files.

# Topic Naming with Kafka

The topic name format used on Kafka clusters is as follows:

*host_port.project.contquery.window.direction*

The *direction* takes the value **"I"** or **"O"**. Because this information appears in a client URL, clients can easily determine the correct appliance topic. Kafka connectors use their configured "urlhostport" parameter to derive the "*host:port*" section of the topic name. The rest of the information is known by the connector. To meet Kafka topic naming requirements, the configured urlhostport string must replace ':' with '_'.

# Determining Engine Active/Standby State with Kafka

All subscriber Kafka connectors that are enabled for failover require connectivity to a Zookeeper cluster. The first subscriber connector to start up within an event stream processing server implements a Zookeeper watcher. When necessary, the connector also creates a root "**/ESP**" zNode. Then it creates a leaf node that is both sequential and ephemeral. It creates the node using the path "**/ESP/server-n_*seq***", where *seq* specifies an incrementing integer that is appended by Zookeeper. All other ESP servers in the failover group follow the same process. Thus, each server is represented in Zookeeper by a unique zNode. Each server implements a Zookeeper watcher. The first server to connect to Zookeeper has the smallest path (as identified by the *seq* field).

Status changes related to these Zookeeper nodes are logged to the event stream processing server console as info-level messages. When a watcher receives a zNode status change, it does the following:

- gathers all the current "**/ESP**" child nodes
- finds the path with the greatest path that is less than its own
- begins watching the zNode that owns that path

The watched zNode is the active engine. If a path was not found, the watcher itself has the smallest path and becomes the active engine.

The result is that the server with the smallest path is always the active engine. The server with the next smallest path (if there is one) is the watcher of the active engine. That server becomes the active engine when the active engine fails. In this way, no more than one zNode is watched. The zNode is watched only by one other zNode, which keeps Zookeeper chatter to a minimum. The Zookeeper `tickTime` configuration parameter must be no greater than 500 milliseconds. This enables the watcher to detect a failed active engine within one second.

# New Engine Active Actions on Failover with Kafka

A standby engine server runs a Zookeeper watcher that watches the active server. When its state changes from standby to active, each subscriber Kafka connector does the following:

- queries the outbound Kafka topic for the current offset into the partition that the subscriber is configured to write to
- creates a consumer and consumes the message at `current_offset - 1` from that partition
- stops the consumer
- extracts the message ID from the event block in the received message
- begins publishing starting with the following message:

```
ID = message_ID + 1
```

Suppose that the next event block produced by the subscribed window contains an ID greater than that. In that case, the connector publishes all messages in the gap from the queue that it maintained while it was in standby state. It then discards older messages from the queue. Then it resumes publishing messages produced by the subscribed window.

# Metadata Exchanges with Kafka

The Kafka publish/subscribe API handles the `C_dfESPpubsubQueryMeta()`, `C_dfESPpubsubPersistModel()`, and `C_dfESPpubsubQuiesceProject()` methods as follows:

- The connectors running in a failover group listen for metadata requests on a special topic named "*urlhostport*`.M`". They consume the topic using a global group ID, such that only one consumer in the group processes any message sent on that topic.
- The client sends formatted messages on this topic in request-reply fashion.
- The request contains an additional field that specifies a GUID-based topic on which to send the response. Since only the requester is listening on this topic, the response is guaranteed to be received by the requester and only the requester.
- The response contains the same information that is provided by the native publish/subscribe API.

# Publisher Adapter Failover with Kafka

Failover for C and Java publisher adapters uses the Kafka failover mechanism described previously. Thus, the publisher adapter must publish event blocks to the ESP server using the Kafka transport. It also must be able to access a Kafka broker and a Zookeeper cluster. Client configuration parameters must be configured in a file named kafka.cfg.

To use the C adapter, the librdkafka and zookeeper client libraries must be installed. For more information, see "Using the Kafka Connector and Adapter" in *SAS Event Stream Processing: Connectors and Adapters*.

To use the Java adapter, the native Kafka Java client JAR files and Apache Zookeeper Java client JAR files must be installed. For more information, see "Using Alternative Transport Libraries for Java Clients" in *SAS Event Stream Processing: Publish/Subscribe API Reference*.

You must define the following parameters in kafka.cfg to support a failover enabled publisher adapter:

***Table 2*** *Kafka Parameters*

| Parameter | Description |
|---|---|
| `hotfailover` | enables or disables hot failover. Valid values are **`true`** and **`false`**. |
| `numbufferedmsgs` | specifies the maximum number of messages to buffer on a standby adapter. |
| `zookeeperhostport` | specifies the Zookeeper cluster *host*:*port*. |
| `failovergroup` | a string defined identically for all publisher adapters that belong to the same failover group |

To guarantee a successful switchover from standby adapter to active adapter, you must ensure the following:

- All publisher adapters in the failover group must be configured identically in order to ensure that they receive identical input from the publishing source. When the source is a message bus, the associated topic or queue must deliver identical messages to all clients.
- All publisher adapters in the failover group must begin publishing at approximately the same time in order to reduce buffering requirements on standby adapters. The maximum number of buffered event blocks is set by the `numbufferedmsgs` parameter.

Following these steps guarantees that all adapters in the failover group build the same sequence of SAS Event Stream Processing event blocks to publish to the ESP server. It also guarantees that the unique message ID associated with each event block is assigned identically on all adapters in the group.

When you restart a failed publisher adapter, it must have access to the complete source data. It needs this access in order to get back in sync with already running adapters in the same group. If it

reads incomplete data, then it assigns message IDs to event blocks incorrectly. Thus, when it becomes active, the input stream to the ESP server contains missing or duplicate event blocks.