SAS® Event Stream Processing 4.3: Advanced Topics

Starting Streamviewer from the Java Command Line

Follow these instructions if you prefer to start Streamviewer from the Java command prompt. You should know which database system you want to use before starting Streamviewer. See "Planning for Streamviewer" in SAS Event Stream Processing: Visualizing Event Streams with Streamviewer for more information.

Start Streamviewer by running the **`streamviewer-4.3.jar`** file, which is located in the **`/lib`** directory of your SAS Event Stream Processing installation (**`$DFESP_HOME/lib`**). Streamviewer requires a Java Virtual Machine (JVM) of 1.7 or later.

Use one of the following commands to start Streamviewer, depending on the database that you want to use for the Streamviewer configuration. See "Command Arguments for Streamviewer" in SAS Event Stream Processing: Visualizing Event Streams with Streamviewer for more information about the values for each command line argument.

H2

```
java -Dserver.port=<http_port> -Dspring.datasource.url="jdbc:h2:<database_file>" -Dfile.encoding=UTF-8
-jar streamviewer-4.3.jar
```

Note: By default, H2 creates a trace file alongside the database file. If you do not want the trace file to be created, add the '`trace_level_file=0`' specification to the `-Dspring.datasource.url` argument, like this:

```
java -Dserver.port=<http_port> -Dspring.datasource.url="jdbc:h2:<database_file>;trace_level_file=0"
-Dfile.encoding=UTF-8 -jar streamviewer-4.3.jar
```

PostgreSQL

```
java -Dserver.port=<http_port> -Dspring.datasource.url="jdbc:postgresql:<host>:<port>/<db_name>"
-Dspring.datasource.username=<username> -Dspring.datasource.password=<password> -Dfile.encoding=UTF-8
```

```
-jar streamviewer-4.3.jar
```

MariaDB

```
java -Dserver.port=<http_port> -Dspring.datasource.url="jdbc:mariadb://<host>:<port>/<db_name>"
-Dspring.datasource.username=<username> -Dspring.datasource.password=<password> -Dfile.encoding=UTF-8
-jar streamviewer-4.3.jar
```

MySQL

```
java -Dserver.port=<http_port> -Dspring.datasource.url="jdbc:mariadb://<host>:<port>/<db_name>"
-Dspring.datasource.username=<username> -Dspring.datasource.password=<password> -Dfile.encoding=UTF-8
-jar streamviewer-4.3.jar
```

SQLServer

```
java -Dserver.port=<http_port> -Dspring.datasource.url="jdbc:jtds:sqlserver://<host>:<port>"
-Dspring.datasource.username=<username> -Dspring.datasource.password=<password> -Dfile.encoding=UTF-8
-jar streamviewer-4.3.jar
```

Sybase

```
java -Dserver.port=<http_port> -Dspring.datasource.url="jdbc:jtds:sybase://<host>:<port>"
-Dspring.datasource.username=<username> -Dspring.datasource.password=<password>
-Dspring.jpa.properties.hibernate.dialect=org.hibernate.dialect.SybaseDialect -Dfile.encoding=UTF-8
-jar streamviewer-4.3.jar
```

Oracle

```
java -Dserver.port=<http_port> -Dspring.datasource.url="jdbc:oracle:thin:@//<host>:<port>/<service_name>"
-Dspring.datasource.username=<username> -Dspring.datasource.password=<password> -Dfile.encoding=UTF-8
-jar streamviewer-4.3.jar
```

Teradata

```
java -Dserver.port=<http_port> -Dspring.datasource.url="jdbc:teradata://<host>/USER=<username>,
PASSWORD=<password>,DATABASE=<db_name>"
-Dspring.jpa.properties.hibernate.dialect=org.hibernate.dialect.TeradataDialect
-Dfile.encoding=UTF-8 -jar streamviewer-4.3.jar
```

Implementing 1+N-Way Failover

Overview to 1+N-Way Failover

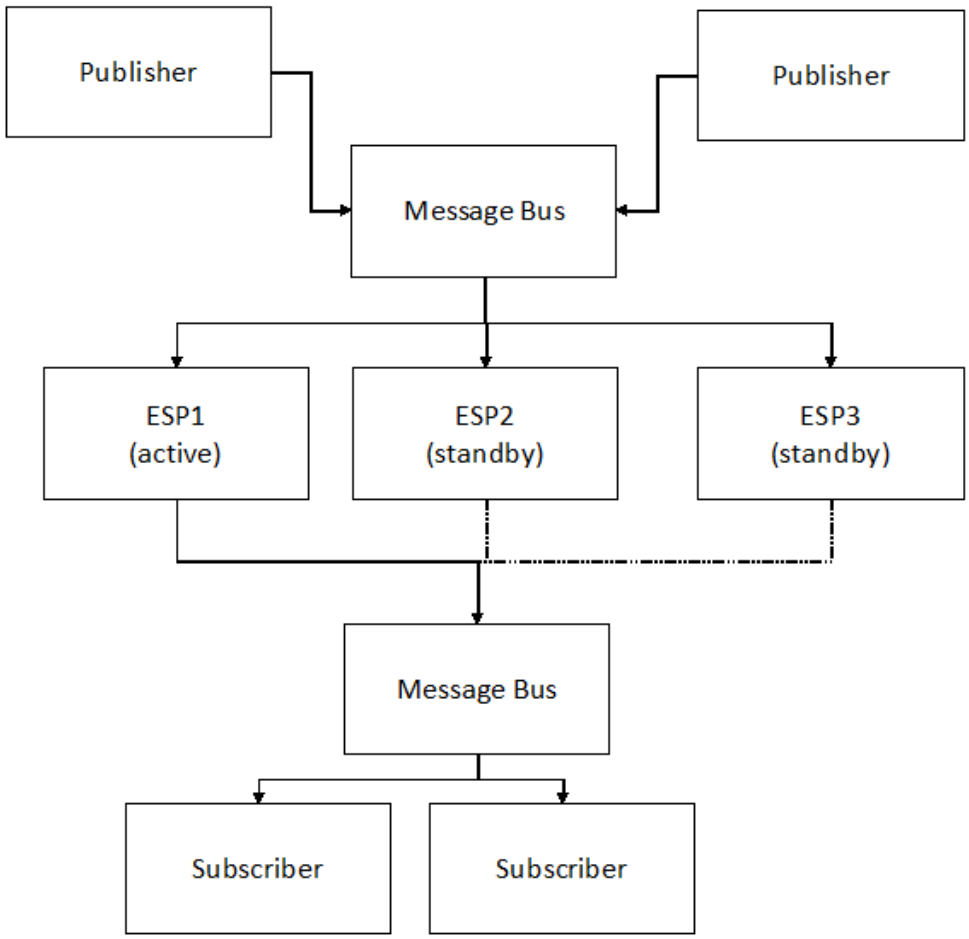
SAS Event Stream Processing can use message buses to provide 1+N-Way Failover. A message bus connector or adapter can be configured to exchange CSV, JSON, or data in other text formats across the message bus. However, when enabled for 1+N-Way failover, messaging publishers and subscribers must exchange binary event blocks. This is because only binary event blocks contain the required message IDs. When traversing a message bus, event blocks are mapped one-to-one to appliance messages. Each payload message contains exactly one event block. A payload appliance message encapsulates the event block and transports it unmodified.

This topic uses the terms “message” and “event block” interchangeably. The term active/standby identifies the state of any event stream processors in a 1+N cluster of event stream processors. The term primary/secondary identifies the state of a message bus with respect to another message bus in a redundant pair. The terms 1+N, failover, cluster, and combinations of these terms are used interchangeably.

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 (ESPs) and a second between ESPs and subscribers. In actual deployments, 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.

Engine Integration with Message Buses



In this diagram, ESP1 is the active engine (on start-up at least). ESP2 and ESP3 are standbys that receive published event blocks. The standbys do not send processed event blocks to the subscriber message bus. This distinction is depicted with dotted arrows. The event stream processing message bus connector for subscribe services is connected to the fabric. A standby does not actually send event blocks to the message bus until one of them becomes the active on failover.

All ESPs in a 1+N failover cluster must implement the same model because they are redundant. It is especially important that all ESPs in the cluster use the same engine name. This is because the engine name is used to coordinate the topic names on which messages are exchanged through the message bus.

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

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

- native
- Rabbit MQ

- Solace
- Tervela
- Kafka

When you use the message bus for publish/subscribe, the event stream processing 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, ESPs, and subscribers.
- Kafka is persistent by default (subject to the cluster log.retention parameters). ESP connectors, adapters, and publish/subscribe clients that consume from a Kafka partition can specify the offset from which to begin consuming.

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.
- 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 ESPs are deployed in a 1+N redundant manner. This means the following:

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

For simplicity, the reference architecture diagram illustrates one cluster of 1+N redundant ESPs. However, there can be multiple clusters of ESPs, each subscribing and publishing on a different set of topics. A single publisher can send messages to multiple clusters of ESPs. A single subscriber can receive messages from multiple ESPs.

The message bus provides a mechanism to signal to an ESP that it is the active ESP in the cluster. The message bus provides a way for an ESP, when notified that it is active, to determine the last message published by the previously active ESP. The newly active ESP 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 an ESP fail-over

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

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

Requirements

Required Software Components

Note the following requirements when you implement 1+N-way failover:

- The ESP model must implement the required Solace, Tervela, RabbitMQ, or Kafka publish and subscribe connectors. The subscribe connectors must have "`hotfailover`" configured to enable 1+N-way failover.
- Client publisher and 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. SAS Event Stream Processing does not ship any appliance standard API libraries. The run-time environment must define the path to those libraries (using LD_LIBRARY_PATH on Linux platforms, for example).

Required Hardware Components

ESP 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.

Kafka [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 ESP source windows. See redundancy details here: https://kafka.apache.org/documentation/#replication
RabbitMQ [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 [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
Solace [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 [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 well known, and 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 may or may not be collocated with the set of two or more redundant ESP servers. It should not matter whether the software bus or ESP is containerized, provided these conditions are met:

- IP connectivity between 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 expected ESP and message bus load. Typically this means all servers are identically sized since any one of them must be able to handle a full ESP and message bus load, even if only temporarily after a server failure.

Topic Naming

Topic names are mapped directly to engine (ESP) windows that send or receive event blocks through the fabric. Because all ESPs 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 ESP appliance connectors to use appliance subscriptions and publications, where event blocks can flow only in one direction.

Current client applications continue to use the standard ESP 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 ESPs. All of these ESPs 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 ESP cluster.

You create this mapping by configuring each ESP appliance 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.

Failover Mechanisms

If the active engine (ESP) in a failover cluster fails, the standby ESP appliance connectors are notified. Then one of them becomes the new active ESP. 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`**”.

Restoring Failed Active ESP State after Restart

When you manually bring a failed active ESP back online, it is made available as a standby when another ESP in the cluster is currently active. If the message bus is operating in “direct” mode, persisted messages on the topic do not replay. The standby ESP remains out-of-sync with other ESPs 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.

Using ESP Persist/Restore

To guarantee that a rebooted engine (ESP) can be fully synchronized with other running ESPs in a failover cluster, use the ESP persist/restore feature with an appliance in “guaranteed” mode. This requires that ESP state is periodically persisted by any single ESP in the failover cluster. A persist can be triggered by the model itself, but in a failover cluster this generates redundant persist data.

Alternatively, a client can use the publish/subscribe API to trigger a persist by an ESP engine. The URL provided by the client specifies *host*:*port*, which maps to a specific ESP failover cluster. The messaging mechanism guarantees that only one ESP 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 ESP 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 ESP 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 ESP 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 ESP in the failover cluster.

The other requirement is that the model must execute a restore on boot so that a rebooted standby ESP 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.

Message Sequence Numbers

The message IDs that are used to synchronize ESP failovers are generated by the ESP 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/query/window, and therefore unique for the connector. When redundant ESP 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 ESP 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.

Failover with RabbitMQ

Installing and Configuring RabbitMQ

For information about installing RabbitMQ, see “Using the Rabbit MQ 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. ESP 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 ESP Active/Standby State with RabbitMQ

All ESP 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 ESP peers.

All ESPs receive send and receive notifications in the same order. Therefore, they maintain the same ordered list of present ESPs (that is, those that are bound). The first ESP in the list is always the active ESP. When a notification is received, an ESP 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 XML server.

New ESP 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/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 ESP 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 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.

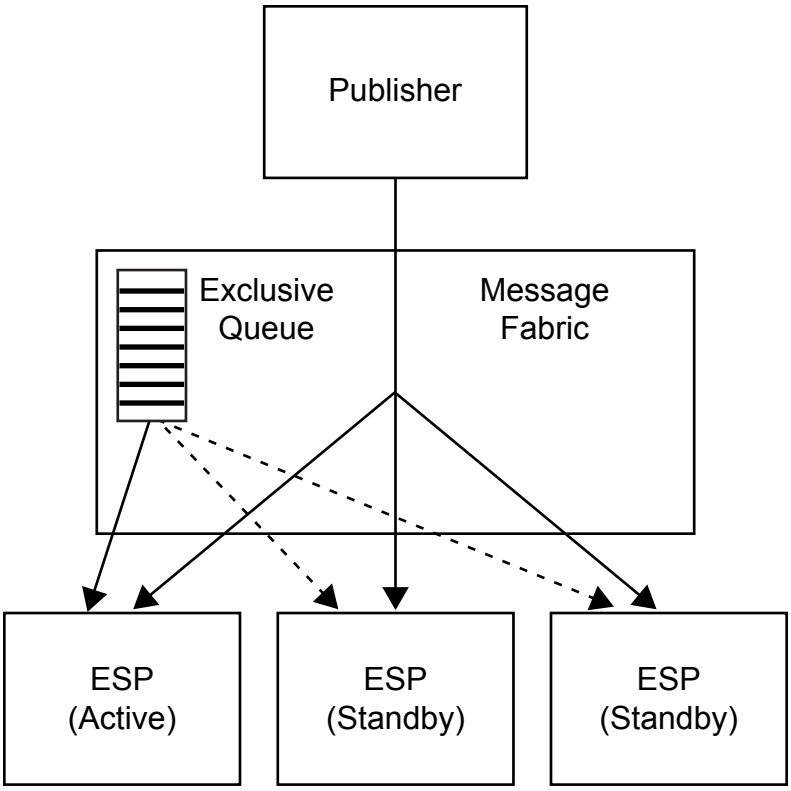
See the documentation of the `C_dfESPpubsubSetPubsubLib()` publish/subscribe API function for details about the contents of these configuration files.

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. ESP 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 ESP Active/Standby State with Solace

For Solace appliances, an exclusive messaging queue is shared amongst all the engines (ESPs) 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 ESP is the active at any point in time.



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

1. When an ESP subscriber appliance connector starts, it tries, as a queue consumer, to bind to the exclusive queue that has been created for the ESP 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 ESP.
3. As other connectors bind to the queue, they receive a “Flow Inactive” indication. This indicates that they are standby ESPs, and should not be publishing data onto the message bus.
4. If the active ESP 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 ESP in the cluster.

New ESP Active Actions on Failover with Solace

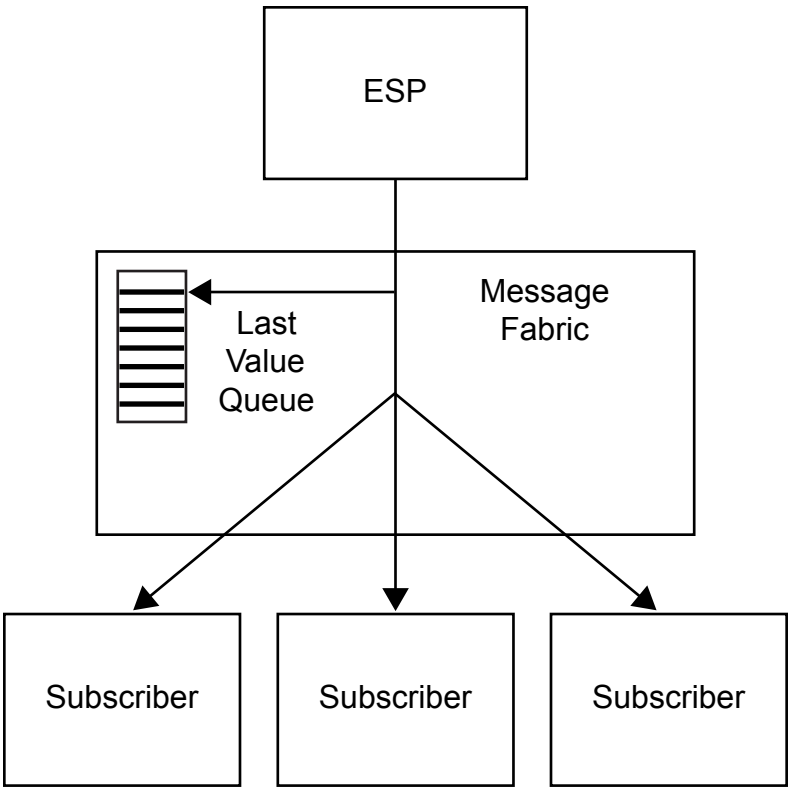
The newly active engine (ESP) determines, from the message bus, the last message published by the previously active ESP for the relevant window. To assist in this process, guaranteed messaging Last Value Queues (LVQs) are used.

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 ESP 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 ESP in the cluster published onto the message bus.

When an ESP 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.

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/reply fashion.
- The request messages are always sent using Deliver-To-One. This is to ensure that no more than one ESP 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

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". ESP 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 ESP 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. ESP appliance connectors automatically send this message when the ESP model is started.

Determining ESP Active/Standby State with Tervela

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

1. When an ESP 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 ESP 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 ESP 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 ESP 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/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 1+N Way Failover topology, the SAS Event Stream Processing 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.

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 ESP 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 ESP. If a path was not found, the watcher itself has the smallest path and becomes the active ESP.

The result is that the server with the smallest path is always the active ESP. The server with the next smallest path (if there is one) is the watcher of the active ESP. That server becomes the active ESP when the active ESP 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 ESP within one second.

New ESP Active Actions on Failover with Kafka

A standby ESP 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 an ESP 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.

Example: Testing Failover with RabbitMQ

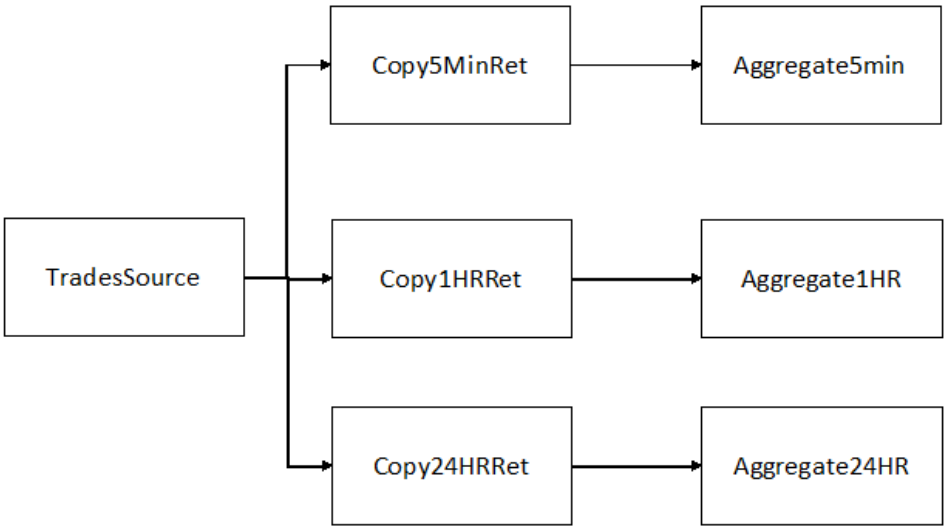
The following example sets up failover in SAS Event Stream Processing with RabbitMQ and tests a failover scenario. This example uses SAS Event Stream Processing Studio to create models.

SAS Event Stream Processing Model Setup

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.



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.

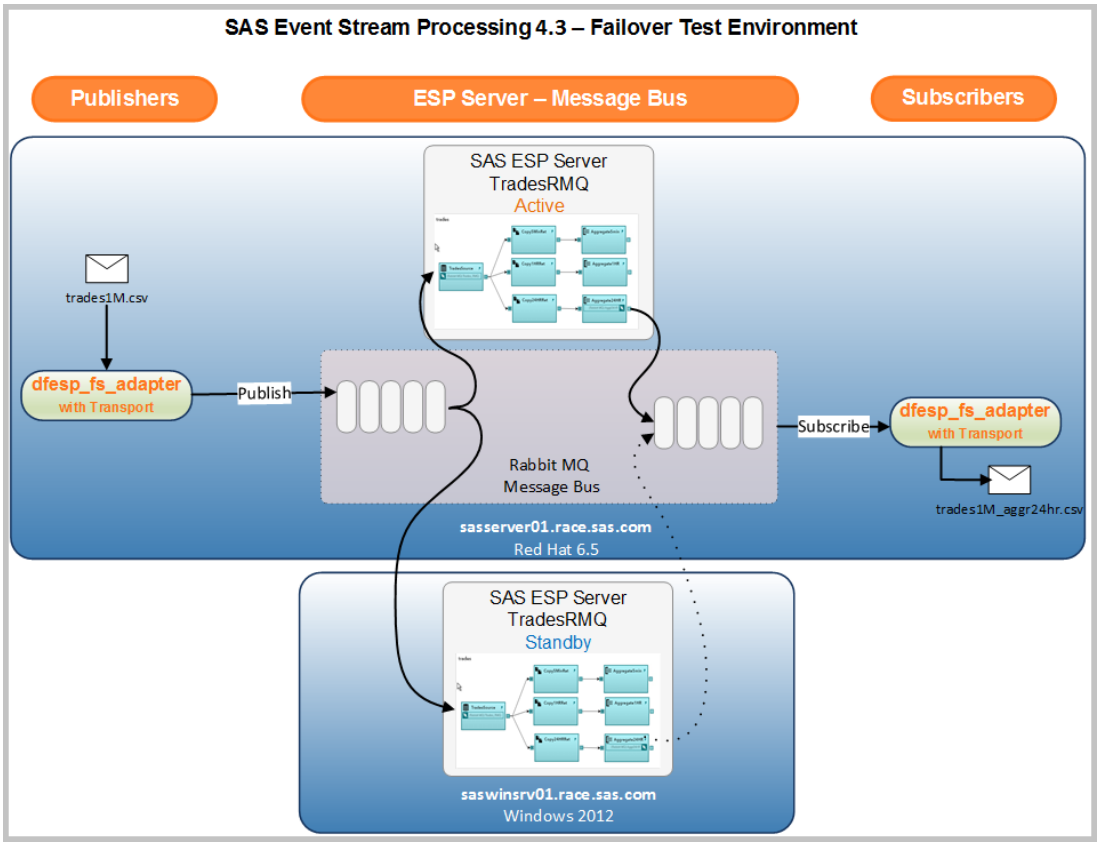
The RabbitMQ connector is defined in one of the aggregate windows (**Aggregrate24HR**) to push events to a second RabbitMQ queue through the exchange. A second file and socket adapter subscribes to events by creating a dynamic queue for an exchange and routing key and writes events to a file.

You can use**`$DFESP_HOME/examples/xml/vwap_xml/trades1M.csv`** on RH2 to ensure that events successfully flow through the model and are written to a CSV file.

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 Studio JSON file to use an ESP server on RH2.
3. Start XML servers on both machines. Use the `-loglevel esp=info` option to display active and standby messages in the XML 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.



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.

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.

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 XML 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
```

Notice the `-l rabbitmq` argument on the command line, which 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.

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-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	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.

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-SMbl42qmU51KylWYIS1Ikg	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 the events have been published, use an HTTP request to the HTTP admin 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.

sasserver01:5556/SASESP/windows?count=true

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<windows>
   <window-aggregate contquery="trades" count="3068" name="Aggregate1HR" project="tradesRMQ"/>
   <window-aggregate contquery="trades" count="3608" name="Aggregate24HR" project="tradesRMQ"/>
   <window-aggregate contquery="trades" count="1941" name="Aggregate5min" project="tradesRMQ"/>
   <window-copy contquery="trades" count="359951" name="Copy1HRRet" project="tradesRMQ"/>
   <window-copy contquery="trades" count="1000000" name="Copy24HRRet" project="tradesRMQ"/>
   <window-copy contquery="trades" count="29951" name="Copy5MinRet" project="tradesRMQ"/>
   <window-source contquery="trades" count="1000000" name="TradesSource" project="tradesRMQ"/>
 </windows>
```

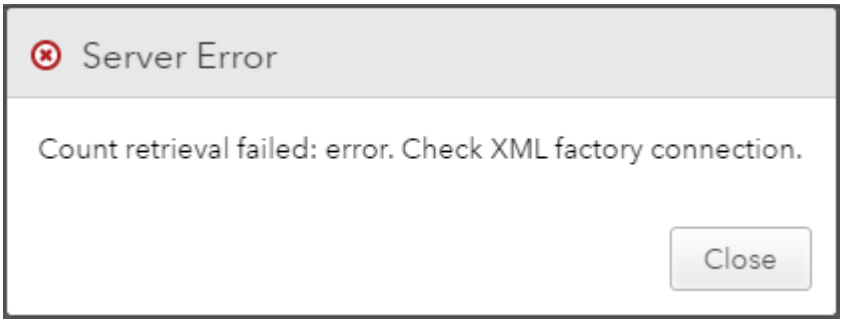
Each model should have processed the same number of records and all window counts should match.

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 a 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.

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 XML server start-up, the following messages confirm that failover occurred and that the standby engine is now the active engine.

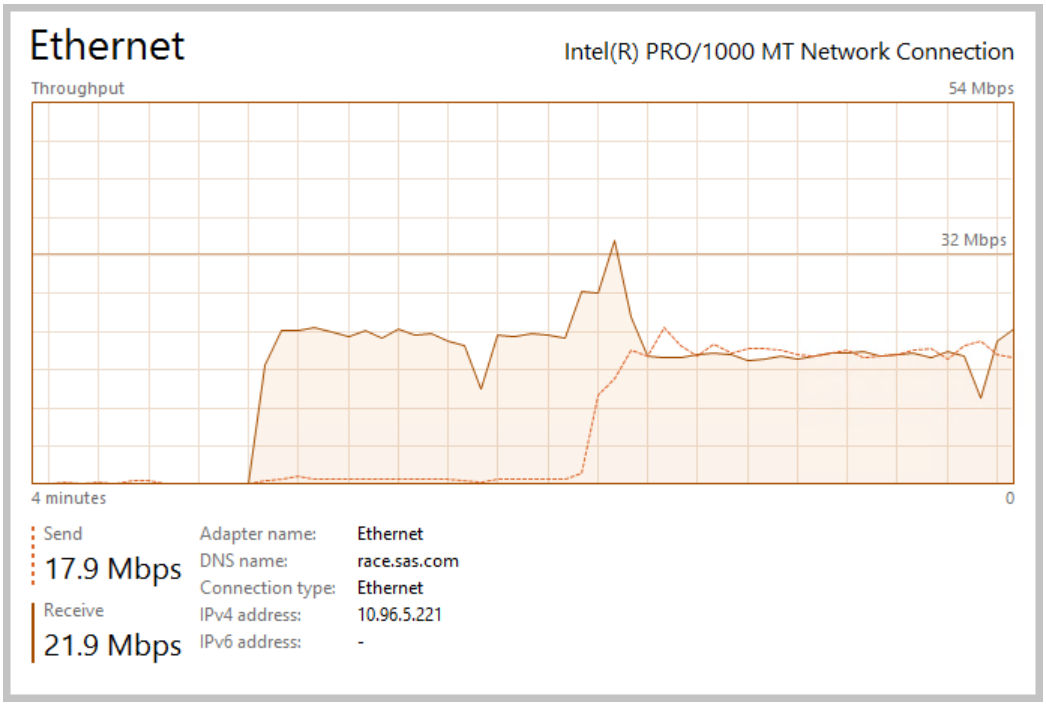
```
2016-04-29T12:27:37,946; INFO; 00000053; DF.ESP; (dfESPrmqConnector.cpp:342);
 dfESPrmqConnector::sendSerializedBlock():
Standby sending buffered messages, last message id = 184211, current message id = 259306
```

```
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.



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.

Running an Event Stream Processing Engine in a Hadoop YARN Container

Overview to YARN

Hadoop YARN serves as a resource management framework to schedule and handle computing resources for distributed applications. A Resource Manager manages the global assignment of compute resources for an application. A per-application Application Master manages an application's scheduling and coordination. YARN provides processing capacity to applications by allocating containers to them. Containers encapsulate resource elements such as memory, CPU, and so on.

For more information about YARN, see the YARN documentation on the Apache Hadoop website.

Starting the Server in the Hadoop YARN Container

To run an event stream processing server in a Hadoop YARN container, run the following script to implement a YARN client.

$DFESP_HOME/bin/dfesp_yarn_joblauncher -e *localdfesphome* -a *httpadminport*

-u *pubsubport* <-q *yarnqueue* > <-p *yarnpriority* >
<-m *yarnmemory* > <-c *yarncores* >

Argument	Description
-e ***localdfesphome***	Specify the **$DFESP_HOME** environment variable setting for SAS Event Stream Processing installations on Hadoop nodes.
-a ***httpadminport***	Specify the value for the **-http-admin** argument on the **dfesp_xml_server** command line. This port is opened when the ESP server is started.
-u ***pubsubport***	Specify the value for the **-pubsub** argument on the **dfesp_xml_server** command line. This port is opened when the ESP server is started.
-t ***httppubsubport***	Specify the value for the **-http-pubsub** argument on the **dfesp_xml_server** command line. This port is opened when the ESP server is started.
-q ***yarnqueue***	Specify the YARN Application Master queue. The default value is **default**.
-p ***yarnpriority***	Specify the YARN Application Master priority. The default value is **0**.
-m ***yarnmemory***	Specify the YARN Memory resource requirement for the Application Master and event stream processing server containers
-c ***yarncores***	Specify the YARN Virtual Cores resource requirement for the Application Master and event stream processing server containers.
-l	Specify that the Application Master should use the **/tmp/ESP/log4j.properties** file found in HDFS.
-o ***consulthostport***	Specify the ***host:port*** of a consult service.

The YARN client submits an Application Master and a script to run an ESP server. The client also passes associated resource requirements to the YARN Resource Manager.

The value of the environment variable **DFESP_HADOOP_PATH** must be the location of Hadoop configuration files on the system where you execute **dfesp_yarn_joblauncher**. This path must also point to the location of the following Hadoop JAR files, which are required by **dfesp_yarn_joblauncher**:

- **hadoop-yarn-client-*.jar**
- **hadoop-yarn-api-*.jar**
- **hadoop-yarn-common-*.jar**
- **hadoop-common-*.jar**
- **hadoop-hdfs-*.jar**
- **hadoop-auth-*.jar**
- supporting JAR files that are usually found in the **hadoop/share/common/lib** directory

The ApplicationMaster submitted by **dfesp_yarn_joblauncher** is copied to and executed from **/tmp/ESP** in HDFS, and is named “sas.esp.clients.yarn.*.*.jar”.

Note: The Application Master is built using Java 1.8, so the Java run-time environment on the Hadoop grid nodes must be compatible with that version. A SAS Event Stream Processing installation must already exist on every node in the grid.

No model XML file is provided. The HTTP client must subsequently manage the model run by the server.

The following command line is executed by the Application Master submitted by **dfesp_yarn_joblauncher**:

dfesp_xml_server -http-admin http-admin *httpadminport* <-pubsub *pubsubport* > <—httppubsub *httppubsubport* >
loglevel esp=info

By default, **dfesp_yarn_joblauncher** sets the following values for the Application Master:

- YARN Queue = default
- YARN Priority = 0
- YARN Resources: Memory = 32768 MB, Virtual Cores = 4

You can override these values through optional arguments to **dfesp_yarn_joblauncher**.

When launched, the Application Master requests one container in which to run the server shell script. That container request specifies the same YARN resources as were requested for the Application Master. Its defaults are 32768 MB and 4 virtual cores, unless you have passed different values to **dfesp_yarn_joblauncher**. YARN might kill any running process at any time when it exhausts its resources. Thus, you should tune these memory and core requirements to match the requirements of the running model.

The event stream processing container runs on a Hadoop node that might or might not be the node running the Application Master container.

You can invoke the **dfesp_yarn_joblauncher** again to launch additional event stream processing servers, which all run independently and have no knowledge of any other servers running on the grid.

Managing the Event Stream Processing Server

On successful start-up, the YARN Resource Manager web application should show an entry where the Name column shows “ESP”. Make a note of the associated application ID. Click the Tracking UI link to show the Application Master parameters. This URL connects to a web server running in the event stream processing Application Master itself.

Note the **Master Host** and **Execution Host** values. These are the nodes where the Application Master and event stream processing server containers are running, respectively. You can drill down to those specific nodes in the Node Manager. There, follow the links for your application ID to find process logs for the Application Master or event stream processing server.

If a failure occurs, the option to view logs depends on whether logging aggregation is enabled on your YARN installation. If it is enabled, log on to any Hadoop node, navigate to the local Hadoop installation, and run “**/bin/yarn logs –applicationId=*ESP_application_id***”. If it is disabled, log on to the individual Hadoop node and find the yarn logs in the local Hadoop installation directories.

To kill containers started by **dfesp_yarn_joblauncher**, log on to any Hadoop node, navigate to the local Hadoop installation, and run **/bin/yarn application –kill *ESP_application_id***.

Connecting to an ESP Server

The http-admin and publish/subscribe ports opened by the ESP server should be reachable by devices outside of the Hadoop grid, if network connectivity is available. The server host name is the name of the Hadoop node where the ESP server is running (shown in the **Execution Host** value displayed by the YARN Resource Manager). The port is the http-admin or publish/subscribe port passed to the **dfesp_yarn_joblauncher** script.

Clients connecting to this ESP server do not know that the server is running in a YARN container on a Hadoop node. Functional behavior is identical to a stand-alone server running outside a Hadoop grid.

Using the Apache Camel Framework

Overview

The Apache Camel framework enables you to integrate different applications into a single, cohesive architecture. Using the Apache Camel framework, you set up routes that contain endpoints:

```
<route id="injectTrades" startupOrder="10">
   <from uri="systemA://someThing"/>
   <to uri="systemB://someOtherThing"/>
</route>
```

The “from” and “to” elements in the previous code are Apache Camel endpoints that refer to components. Apache Camel supports many out-of-box components. It gives you the tools to develop custom components. An Apache Camel Consumer maps to a “from” endpoint, and an Apache Camel Producer maps to a “to” endpoint.

For more information about the Apache Camel framework, see the documentation.

Installing the Apache Camel Framework

In order to use the Apache Camel framework with SAS Event Stream Processing, you must download and install various files. These include Apache components and specific JAR files.

1. Access and install the following Apache components:
   - Apache Camel, which you can download from http://camel.apache.org/
   - Apache Maven, which you can download from https://maven.apache.org/

   Apache Maven is a build environment that you can use to create projects that leverage components from SAS Event Stream Processing. When you install Apache Maven, make sure that you install the **bin** directory subordinate to your Maven install directory in your path.

2. After you have installed the Apache components, install two JAR files into your local Maven repository:
   - ESP API Client JAR
   - ESP Camel JAR

   Note: You can find these JAR files in **$DFESP_HOME/lib**. They contain the client API and the Camel components.

3. Install the JAR files into the Maven repository:

```
$ cd $DFESP_HOME/lib
$ mvn install:install-file -Dfile=dfx-esp-api.jar -DgroupId=com.sas.esp
-DartifactId=dfx-esp-api -Dversion=4.2 -Dpackaging=jar
$ mvn install:install-file -Dfile=dfx-esp-camel.jar -DgroupId=com.sas.esp
-DartifactId=dfx-esp-camel -Dversion=4.2 -Dpackaging=jar
$ mvn install:install-file -Dfile=cas-client-3.06.jar -DgroupID=com.sas.esp
-DartifactID=dfx-cas-auth -Dversion=3.0.6 -Dpackaging=jar
```

4. After you have installed the JAR files, you can reference them from your Maven project object model (pom.xml) file. You must have an entry for both the event stream processing client API and the SAS Event Stream Processing Camel components.

Here is the event stream processing client API entry:

```
<dependency>
```

```
    <groupId>com.sas.esp</groupId>
    <artifactId>dfx-esp-api</artifactId>
    <version>[4.2]</version>
</dependency>
```

Here is the SAS Event Stream Processing Camel entry:

```
<dependency>
<groupId>com.sas.esp</groupId>
    <artifactId>dfx-esp-camel</artifactId>
    <version>[4.2]</version>
</dependency>
```

Here is the CAS authentication entry:

```
<dependency>
<groupID>com.sas.esp</groupID>
<artifactID>dfs-cas-auth</artifactID>
<version>3.0.6</version>
</dependency>
```

Installing the RabbitMQ Library

Configure the Maven project so that SAS Event Stream Processing uses RabbitMQ as an alternative transport library:

1 Install the RabbitMQ API JAR:

```
$ cd $DFESP_HOME/lib
$ mvn install:install-file -Dfile=dfx-esp-rabbitmq-api.jar -DgroupId=com.sas.esp
-DartifactId=dfx-esp-rabbitmq-api -Dversion=3.2 -Dpackaging=jar
```

2 Update the Maven project object model (pom.xml) to include RabbitMQ dependency information:

```
<dependency>
          <groupId>com.rabbitmq</groupId>
          <artifactId>amqp-client</artifactId>
          <version>3.5.6</version>
        </dependency>
        <dependency>
            <groupId>commons-configuration</groupId>
             <artifactId>commons-configuration</artifactId>
             <version>1.10</version>
        </dependency>
        <dependency>
            <groupId>com.sas.esp</groupId>
            <artifactId>dfx-esp-rabbitmq-api</artifactId>
            <version>3.2</version>
        </dependency>
```

Note: You must define the dependency for the RabbitMQ JAR (`dfx-esp-rabbitmq-api`) in the pom.xml file before you define dependency for the ESP API Client JAR (`dfx-esp-api`).

SAS Event Stream Processing Implementation

The SAS Event Stream Processing implementation consists of SAS Event Stream processing Camel Endpoints that are either Consumers (which implement publish/subscribe subscribers) or Producers (which implement publish/subscribe publishers). A Consumer maps to a **`from`** endpoint, and a Producer maps to a **`to`** endpoint.

For example, to receive events from one publish/subscribe server and send them to another, execute the following code:

```
...
<endpoint id="subscribe" uri="esp://espsrv01:46003">
  <property key="project" value="project" />
  <property key="contquery" value="query" />
  <property key="window" value="transform" />
</endpoint>

<endpoint id="publish" uri="esp://espsrv01:47003">
  <property key="project" value="project" />
  <property key="contquery" value="query" />
  <property key="window" value="trades" />
</endpoint>

<route>
  <from uri="ref:subscribe"/>
  <to uri="ref:publish" />
</route>
...
```

The SAS Event Stream Processing components can work with the following formats representing events:

Format	Description
Map<String,Object>	A standard JAVA Map object in which the keys are field names and the values are field values.
List<Map<String,Object>>	A standard JAVA List of JAVA Map objects in which the keys are field names and the values are field values.
XML	Event data in XML.
JSON	Event data in JSON.

A SAS Event Stream Processing Consumer (subscriber) receives events and converts them into one of these formats to send them along the route. A SAS Event Stream Processing Producer (publisher) receives data in one of these formats, converts it into events, and publishes them.

Through these standard formats, SAS Event Stream Processing easily integrates with any of the other components available in the Camel framework and shares data with them. If some transformation is required in order to get the data into the required format, you can use a transformation bean between the endpoints.

Here is an example of transforming the standard comma-separated value event format into one of the supported types:

```
...
<endpoint id="csvData" uri="stream:file">
   <property key="fileName" value="/mnt/data/share/tradesData/trades1M.csv" />
</endpoint>

<endpoint id="inject" uri="esp://espsrv01:46003">
   <property key="project" value="project" />
   <property key="contquery" value="query" />
   <property key="window" value="trades" />
</endpoint>
```

```
<route id="injectTrades" startupOrder="10">
   <from uri="ref:csvData"/>
   <bean ref="csvTransform" method="transform" />
   <to uri="ref:inject"/>
</route>

<bean id="csvTransform" class="com.sas.esp.camel.transforms.CsvTransform">
   <property name="schema" value="id*:int64,symbol:string,currency:
int32,time:int64,msecs:int32,price:double,quant:int32,venue:int32,broker:
int32,buyer:int32,seller:int32,buysellflg:int32" />
   <property name="format" value="xml" />
</bean>

...
```

First create an endpoint called csvData to read the CSV data from a file. You also create an endpoint called inject to inject this data into an ESP source window. The route that you use goes from the file into SAS Event Stream Processing, but you must get the CSV data into one of the supported formats.

To do this, create a bean called csvTransform, which requires a schema and an output format. After that, you can form events from the CSV data and create an XML document to pass along the route. This is consumed by the Producer, which injects the data into the specified source window. The method attribute for any of the SAS Event Stream Processing transformation beans is always a transform.

Using Camel Components in a Maven Project

In order to reference the SAS Event Stream Processing Camel components by URI, you must create the following file:

```
META-INF/services/org/apache/camel/component/esp
```

The file must contain the following line:

```
class=com.sas.esp.clients.camel.EspComponent
```

Note: This is described in further detail at http://camel.apache.org/writing-components.html.

This enables you to reference the event stream processing components with a URI such as the following:

```
<endpoint id="inject" uri="esp://<pub/sub host>:<pub/sub port>">
```

Configuring Endpoints

You must determine the host and port of the SAS Event Stream Processing publish/subscribe server with which you are going to communicate. Use this information to specify the URI of the component. For example, if your publish/subscribe server is running on port 46003 on machine espsrv01, your URI is:

```
esp://espsrv01:46003
```

Because you are always putting events into or getting events out of windows, you also must specify the project, continuous query, and window in which you are interested. You can do this using either of the following methods:

add parameters to the URI

```
esp://espsrv01:46003?project=myproject&contquery=mycq&window=trades
```

specify an endpoint element with properties :

```
<endpoint id="inject" uri="esp://espsrv01:46003">
   <property key="project" value="project" />
   <property key="contquery" value="query" />
   <property key="window" value="trades" />
```

```
</endpoint>
```

Properties That You Can Set on an Endpoint

Property	Description	Consumer	Producer	Valid Values
`project`	The event stream processing project.	x	x	A valid project.
`contquery`	The event stream processing continuous query.	x	x	A valid continuous query.
`window`	The event stream processing window.	x	x	A valid window.
`format`	The data format for this component. This is usually used with a Consumer to specify the format of the event data to send down the route. It can be used with a Producer. If the component receives a message with a body that is a String, the producer uses the format (either XML or JSON) for conversion.	x	x	map, list, XML, JSON
`blocksize`	The size of the event blocks to inject into a source window.		x	Unsigned integer
blob	The name of an event field that is used to contain the entire message body. If this property is set, the Producer receives a message and creates a MapMap<String,Object> where the key is the field indicated by the blob property. The value is the entire message body represented as a String. This data is injected into the appropriate source window. **Note:** When using the blob property, the source window into which the event is being injected must set both 'insert-only=true' and 'autogen-key=true'.		x	A valid event field
`authUser`	The name to use with SASLogon authorization.	x	x	
`oAuthToken`	The OAuth token to use with OAuth authentication.	x	x	

Using Transformation Beans

When data is not in a format immediately usable by the event stream processing components, you must use a transformation bean to convert the data into a usable format. You can use a transformation bean to convert events in the SAS Event Stream Processing CSV format into a usable format. Place the transformation beans in a route between the from and to endpoints.

Here is an example:

```
...
        <route id="injectTrades">
```

```
      <from uri="ref:csvData"/>
      <bean ref="csvTransform" method="transform" />
      <to uri="ref:inject"/>
   </route>

...

<bean id="csvTransform" class="com.sas.esp.clients.camel.transforms.CsvTransform">
   <property name="schema" value="id*:int64,symbol:string,currency:
int32,time:int64,msecs:int32,price:double,quant:int32,venue:int32,broker:
int32,buyer:int32,seller:int32,buysellflg:int32" />
   <property name="format" value="xml" />
   <property name="dateformat" value="ddMMMyyyy HH:mm:ss.S a" />
</bean>
...
```

Each bean takes certain parameters to help it convert the data to be usable by SAS Event Stream Processing. The CSV transformation bean requires the event schema and the output data format. Note that the bean lies between a “from” endpoint that contains a file reference and a “to” endpoint to publish events into a window.

Transformation Beans in the SAS Event Stream Processing Camel Package

Class	Description
com.sas.esp.clients.camel.transforms.CsvTransform	Transforms CSV data into ESP event data schema The event schema for the events represented by the CSV data. format The data format to use to write events. Valid values are map, list, XML, and JSON. dateformat The date format to use.
com.sas.esp.clients.camel.transforms.RssTransform	Transforms RSS data into ESP event data format The data format to use to write events. Valid values are list, XML, and JSON.

Examples

Where to Find Examples

A set of examples is available at **`$DFESP_HOME/examples/java/camel`**.

CSV Injection

The following example reads trade data from a CSV file and injects the trades into the broker surveillance model. It also subscribes to the brokerAlertsAggr window and writes these events to the console in JSON format.

1 Edit **`src/main/resources/esp.properties`** so that it contains your publish/subscribe server information:

```
espServer=esp://espsrv01:46003
```

```
tradesFile=data/trades1M.csv
```

2 Start your ESP server:

```
$ dfesp_xml_server -model file://model.xml -http-admin <http admin port>
-http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup
```

3 Start the project:

```
$ mvn camel:run
```

Distributed Modeling

This example distributes broker surveillance model between two servers. The first server takes the trade data and performs all the dimensional additions (broker info, venue data) to the event. This model is in primary.xml and ends with a functional window called transform. The events generated by transform contain a full set of trade information. The project subscribes to this window in server1 and forwards the events to server2, which looks for the broker alerts. Another route is used to subscribe to brokerAlertsAggr window in server2 and dump the events to the screen in Map format.

1 Edit **`src/main/resources/esp.properties`** so that it contains your publish/subscribe server information:

```
espServer1=esp://espsrv01:46003
espServer2=esp://espsrv01:47003
tradesFile=data/trades1M.csv
```

2 Start your primary event stream processing server:

```
$ dfesp_xml_server -model file://primary.xml -http-admin <http admin port>
-http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup
```

3 Start your secondary event stream processing server:

```
$ dfesp_xml_server -model file://secondary.xml -http-admin <http admin port>
 -http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup
```

4 Start the project:

```
$ mvn camel:run
```

RSS

This example uses the Camel RSS Component to set up a route that reads data from any number of RSS feeds and injects them into SAS Event Stream Processing. You should be able to add any RSS feeds to the route.

```
<route>
   <from uri="rss:http://feeds.reuters.com/reuters/businessNews" />
   <from uri="rss:http://feeds.reuters.com/reuters/topNews" />
   <from uri="rss:http://feeds.reuters.com/reuters/technologyNews" />
   <bean ref="rssTransform" method="transform" />
   <to uri="ref:publishNews" />
</route>

...
```

You can also use a new transformation bean to transform the RSS data into a supported format:

```
<bean id="rssTransform" class="com.sas.esp.clients.camel.transforms.RssTransform">
   <property name="opcode" value="upsert" />
</bean>

...
```

In the following project, the RSS data is keyed by title:

```
<project name='project' pubsub='auto' threads='4'>
   <contqueries>
      <contquery name='cq' trace='src'>
         <windows>
            <window-source name='src'>
               <schema-string>title*:string,author:string,link:string,
description:string,categories:string,pubDate:date</schema-string>
            </window-source>
         </windows>
      </contquery>
   </contqueries>
</project>

...
```

1 Edit **`src/main/resources/esp.properties`** so that it contains your publish/subscribe server information:

```
espServer=esp://espsrv01:46003
```

2 Start your event stream processing server:

```
$ dfesp_xml_server -model file://model.xml -http-admin <http admin port>
-http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup
```

3 Start the project:

```
$ mvn camel:run
```

Weather

This example uses the Camel Weather Component to set up a route that reads weather data for any number of locations and injects it into SAS Event Stream Processing. You should be able to add any locations to the route. The locations can be defined as endpoints as shown in the following example:

```
<endpoint id="cary" uri="weather:foo">
   <property key="location" value="cary,nc"/>
   <property key="mode" value="XML"/>
   <property key="units" value="IMPERIAL"/>
</endpoint>

<endpoint id="morehead" uri="weather:foo">
   <property key="location" value="moreheadcity,nc"/>
   <property key="mode" value="XML"/>
   <property key="units" value="IMPERIAL"/>
</endpoint>

<endpoint id="chapelHill" uri="weather:foo">
   <property key="location" value="chapelhill,nc"/>
   <property key="mode" value="XML"/>
   <property key="units" value="IMPERIAL"/>
</endpoint>

...
```

You can then add these endpoints to your route:

```
<route>
   <from uri="ref:cary"/>
```

```
    <from uri="ref:morehead"/>
    <from uri="ref:chapelHill"/>
    <to uri="ref:publishWeather"/>
</route>

...
```

1 Edit **src/main/resources/esp.properties** so that it contains your publish/subscribe server information:

```
espServer=esp://espsrv01:46003
```

2 Start your event stream processing server:

```
$ dfesp_xml_server -model file://model.xml -http-admin <http admin port>
-http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup
```

3 Start the project:

```
$ mvn camel:run
```

Using SAS Event Stream Processing with a Cloud Infrastructure

Overview to Using SAS Event Stream Processing in the Cloud

SAS Event Stream Processing is cloud-ready. You can deploy it almost anywhere, from data centers to edge devices and from physical machines to clouds. You can implement engines and adapters on different virtual machines.

You can use SAS Event Stream Processing with deployment tools such as BOSH and Chef. This document assumes that you have experience using BOSH and Chef.

Using SAS Event Stream Processing with BOSH

Overview

BOSH is an open-source tool for release engineering, deployment, lifecycle management, and monitoring of distributed systems. BOSH consists of a server tool and a command line interface (CLI). You can use BOSH to provision and deploy software over hundreds of virtual machines. For more information about BOSH, see https://bosh.io/.

BOSH was developed to deploy Cloud Foundry PaaS (platform as a service), but you can also use it to deploy software such as RabbitMQ or Hadoop. BOSH includes a Cloud Provider Interface (CPI) that enables you to extend it to support additional IaaS (infrastructure as a service) providers such as Google Compute Engine and Apache CloudStack.

To use BOSH:

- Create a BOSH release contained in a directory with a pre-defined structure.
- Obtain the suitable *stemcell* file, which is an OS image wrapped with IaaS-specific packaging. A typical stemcell contains an OS skeleton with some common utilities pre-installed, a BOSH Agent, and some configuration files to securely configure the OS.
- Create a deployment manifest.

- Deploy the service in the cloud.

Implementing SAS Event Stream Processing with BOSH

1 Ensure that the BOSH director is running.

2 The BOSH CLI does not interact with IaaS directly. Instead, it interacts with the director, which in turn interacts with the cloud. Thus, you need to point the BOSH CLI to the director:

```
$bosh target [IP_of_bosh_director]
```

3 Get the BOSH release. First, navigate to the appropriate directory:

```
$cd esp-bosh
```

This directory has a structure that is defined by BOSH release:

- The **src** subdirectory contains the data files for the SAS Event Stream Processing installation package and license file needed to run services in VMs.
- The **packages** subdirectory contains the packaging instructions that BOSH uses to install SAS Event Stream Processing binaries.
- The **jobs** subdirectory contains the scripts to start, stop, and restart the event stream processing service.

4 Download the stemcell with the IaaS packaging and the version appropriate for your VM (for example, **ubuntu-trusty**): **$wget --content-disposition https://bosh.io/d/stemcells/bosh-openstack-kvm-ubuntu-trusty-go_agent**

5 Obtain the SAS Event Stream Processing installation package and put it in BOSH's default directory:

```
$mv sas.esp-4.1-lax.tar.gz src/esp/
```

6 Upload the release to the director:

```
$bosh upload release
```

7 Upload the stemcell to the director:

```
$bosh upload stemcell bosh-stemcell-3215-openstack-kvm-ubuntu-trusty-go_agent.tgz
```

8 Tell the director how to deploy the event stream processing service.

First, upload the deployment manifest file. The directory **deployment_manifests** contains two example deployment manifests, one for OpenStack (**manifest_openstack.yml**) and one for Warden (**manifest.ml**). These create containers on the local machine.

manifest_openstack.yml creates a demo of SAS Event Stream Processing deployment on OpenStack. It instantiates five VMs for servers (esp-server/0-4) and one VM for monitoring (esp-monitoring-instance/0). These servers run the ESP server and wait for the injection of models through its http port (31416). The monitoring VM runs the ESP server with a preloaded monitoring model (**file://monitor.xml**).

9 The manifest file must specify the UUID of the director that it uses. Obtain the director's UUID:

```
$bosh status --uuid
```

10 Update the manifest file's value of `director_uuid` so that it is exactly the same as the director's UUID.

11 Upload the deployment manifest:

```
$bosh deployment deployment_manifests/manifest_openstack.yml
```

12 Deploy BOSH:

```
$bosh -n deploy
```

After deployment is completed, run **$bosh vms** to show what VMs are running. You can use the BOSH SSH to access these VMs.

Using SAS Event Stream Processing with Chef

Overview to Chef

Chef is a configuration management tool written in Ruby and Erlang. It uses Ruby to write system configuration recipes in order to streamline the configuration and management of servers. Chef can integrate with cloud-based platforms such as Google Cloud Platform, OpenStack, and Amazon EC2. For more information about Chef, see https://www.chef.io/chef/.

Implementing SAS Event Stream Processing with Chef

Before you begin, verify that you have successfully installed Chef DK on your system with the **chef verify** command.

A Chef Server sits between your system and managed nodes, serving as a hub for configuration data. For the Chef Server, you can either sign up to a hosted server (for example, **https://manage.chef.io/**) or you can run one that you own. The instructions that follow assume that a Chef Server is correctly configured and running. The Chef Server has a web interface that enables you to check its status, upload cookbooks, and perform other tasks.

All Chef installations require a central workspace known as the chef-repo. This is where primitive objects such as cookbooks, roles, environments, data bags, and chef-repo configuration files are stored and managed. Typically, chef-repo is kept under version control. SAS Event Stream Processing includes a sample chef-repo that you can use as a starting point. It contains a **.chef** directory that stores all configuration details and a **pem**file used to access the Chef Server.

The **USERNAME.pem** file contains a private key unique to a user, which should never be shared. The **ORGANIZATION-validator.pem** file contains a private key that is global to the entire organization. This key is used by all nodes and workstations that send requests to the Chef server.

1. Replace the **pem** files with those appropriate for your configuration.
2. Customize a file named **knife.rb**. This file specifies chef-repo-specific configuration details for knife. For more information about **knife.rb**, see https://docs.chef.io/config_rb_knife.html. Use **knife ssl check** to verify the connection between the workstation and the Chef Server.
3. Set up a cookbook, which is the fundamental unit of configuration and policy distribution. Under chef-repo, the **cookbooks** directory is where all cookbooks are stored and managed. Cookbook dependencies are described in each cookbook's metadata file. The metadata file for the SAS Event Stream Processing cookbook is **cookbooks/esp-4.1/metadata.rb**

   This file indicates that **esp-4.1** depends on cookbooks **poise-archive** and **poise-monit**, with specific version requirements. You can download these two cookbooks from the Chef Supermarket using the following commands:

   ```
   $knife cookbook site download [poins-archive]
   ```

   ```
   $knife cookbook site download [poins-monit]
   ```

4. After you download the cookbooks, install them:

   ```
   $knife cookbook install [poins-archive]
   ```

   ```
   $knife cookbook install [poins-monit]
   ```

   The **monit** cookbook is a proactive monitoring and repair tool for Linux systems.

5 Set up a recipe, which is the most important configuration unit in a cookbook. Recipes are stored and managed under the **recipes** directory. A cookbook has a default recipe that can reference other recipes in the same cookbook. Alternatively, it can reference recipes from other cookbooks upon which this cookbook is dependent.

   In the **esp-4.1** cookbook, the default recipe references to three other recipes,

   - **license.rb** copies the license to the node and sets the correct permissions. Make sure that the license file esp_setinit.sas is available under **esp-4.1/files/default** so that this recipe can run correctly.
   - **xml_runtime.rb** creates the group and user to run SAS Event Stream Processing. It also obtains the product installation package, sets up the environment, and configures **monit**.
   - **xml_server.rb** starts the event stream processing service. This service is a running model with pre-configured ports that are defined in **attributes/default.rb** through **monit**.

6 After you develop cookbooks and recipes, test them in a Kitchen. **es-4.1/.kitchen.yml** provides a working example that uses **vagrant** to create a local container for testing. To use this example, you must have installed VirtualBox on your system.

7 After you test cookbooks in the Kitchen, upload them to the Chef Server:

```
$knife cookbook upload -all
```

8 Make sure the node created in the cloud (for example, OpenStack) is accessible through SSH. Then bootstrap that node:

```
$knife bootstrap [NODE_IP] --ssh-user [USER] --sudo --identity-file key.pem --
node-name node1 --run-list 'recipe[esp-4.1]'
```

9 If the node was successfully boot strapped, you should be able to see it on the Chef Server (for example, **https://manage.chef.io** under the nodes tab). When you run the command **knife node list**, it should show the name of the node. You should also be able to verify that the event stream processing service is running on the pre-defined ports.

To update cookbooks, first modify them on your system and then upload them to the Chef Server. After that, you can run the chef-client on the nodes to keep things in sync:

```
$knife ssh [NODE_IP] 'sudo chef-client' --manual-list --ssh-user [USER] --sudo --
identity-file key.pem
```

Setting Up Apache NiFi to Run with SAS Event Stream Processing

Setting Up Apache NiFi

Apache NiFi is a system to automate the flow of data between systems. You can download Apache NiFi from https://nifi.apache.org/. For more information about Apache NiFi, see the FAQ.

1 Download Apache NiFi version 1.0 into a working directory, **$WORK**, on a UNIX system, **$HOST**.

2 Enter the following commands at the command prompt:

   a `cd $WORK`

   b `gunzip nifi-0.x-bin.tar.gz`

   c `tar xvf nifi-0.x-bin.tar`

d **export NIFI_HOME $WORK/nifi-0.x**

e **export PATH=$NIFI_HOME/bin:$PATH**

3 A NAR file is a zipped container file that holds multiple JPEG files and a .xml metadata file. Copy the SAS Event Stream processing NAR file from **$DFESP_HOME/lib** to **$NIFI_HOME/lib**.

4 Edit **$NIFI_HOME/conf/nifi.properties** to change the following line to use an available port:

```
nifi.web.http.port=8080
```

For example, if port 31005 is available, the line reads as follows:

```
nifi.web.http.port=31005
```

5 Start Apache NiFi as follows: **$NIFI_HOME/bin/nifi.sh run**.

6 Start Apache NiFi in a browser: **http://$HOST:*available_port*/nifi**

You should now be able to design your Apache NiFi flows. The SAS Event Stream Processing processors ListenESP and PutESP should be available. When processor updates become available, you need to update only the NAR file in **$NIFI_HOME/lib**.

PutESP Processor

The PutESP processor enables you to publish events from Apache NiFi into an ESP engine. It requires that you specify an ESP project, continuous query, and source window hierarchy for the processor. Given this information, it can accept FlowFiles in AVRO format. For each object in the AVRO input, PutESP takes any data that maps to a field in the ESP source window schema and creates an ESP event. When all the events have been created, they are sent into ESP.

Properties

Pub/Sub Host	The host name of the ESP engine publish/subscribe server.
Pub/Sub Port	The port of the ESP engine publish/subscribe server.
Project	The project into which events are published.
Continuous Query	The continuous query into which events are published.
Source Window	The source window into which events are published.
Record Schema	The AVRO schema for input records. If the schema is not set, it generates an AVRO schema from the publish target source window in the ESP model. You can get the AVRO schema from a running event stream processing model through the REST interface.
Block Size	If this is set, PutESP limits the events sent at one time into ESP engine to this number.
Input Date Format	If this is set, PutESP expects dates in the incoming data to be strings in this format.

Convert Inserts to Upserts	This should be set if you want to convert any opcodes coming in as inserts to upserts before publishing. Allowable Values: ■ **Yes** Convert inserts to upserts. ■ **No** Do not convert inserts to upserts. Note: Defaults to **No**
Authentication User	The SASLogon authentication user.
OAuth Token	The token to use for OAuth authentication.

Relationships

success	If the events are successfully sent to the ESP engine, PutESP sends the originating FlowFile to this relationship.
failure	When PutESP detects a failure in its interaction with the ESP engine, it sends a FlowFile to the failure relationship.

Note: No **Reads Attributes** or **Writes Attributes** specified.

ListenESP Processor

The ListenESP processor enables you to subscribe to any number of windows in a SAS ESP engine and receive events from those windows. The events are collected and converted to AVRO format and sent along the success relationship. The project, continuous query, and window attributes are set on the FlowFile so that downstream processors can act on this information. For example, one can use the **RouteOnAttribute** processor to send events down different paths depending on the originating project, continuous query, and window. You can also specify field values from the ESP events to put on the FlowFile as attributes.

The ListenESP processor requires information for the subscribed ESP windows in order to collect events of interest. You must supply valid ESP engine connection information in the **Pub/Sub Host** and **Pub/Sub Port** properties. For example, if you are running an ESP XML server on a host called *mymachine*,

```
$ $DFESP_HOME/bin/dfesp_xml_server -model file://model.xml -pubsub 28003
```

your connection properties are as follows:

Property	Value	
Pub/Sub Host	mymachine	
Pub/Sub Port	28003	

Once you have entered the connection information, you need to specify regular expressions to determine the subscribed windows from which to get the event data. To get all of an ESP element from the engine, whether it is projects, continuous queries, or windows, you simply use the `.*` regular expression. Because many engines contain a single project and continuous query, this is the simplest way to specify that you want them all. Usually, you do not want to subscribe to all the windows in a model.

The following example subscribes to all windows whose name begins with **`frontRunning`** or whose name is **`brokerAlertsAggr`** in all continuous queries in all projects.

Project	.*
Continuous Query	.*
Window	frontRunning.*|brokerAlertsAggr

If you want to grab field values from each received event, you use the **Attribute Fields** property. This is a comma-separated list of ESP field names. For each event that contains a value for each specified field in this property, values are put onto the FlowFile as an attribute.

The following example tells the processor to put the values of the **`brokerName`** and **`symbol`** fields onto the FlowFile.

Attribute Fields	brokerName, symbol

Properties

Pub/Sub Host	The host name of the ESP engine publish/subscribe server.
Pub/Sub Port	The port of the ESP engine publish/subscribe server.
Project	This is a regular expression used to specify the projects in the engine to which to subscribe.
Continuous Query	A regular expression used to specify the continuous queries in the engine to which to subscribe.
Window	A regular expression used to specify the windows in the engine to which to subscribe.
Snapshot	Set to **Yes** to get a snapshot of the events in the window. Allowable Values: ■ **Yes** Grab initial contents of the window. ■ **No** Do not grab initial contents of the window. Note: Defaults to **No**
Output Date Format	If this is set, ListenESP writes date and timestamp fields as strings using this format instead of as numeric values.
Attribute Fields	These are event field values to put onto the FlowFile as attributes.
Authentication User	The SASLogon authentication user.
OAuth Token	The token to use for OAuth authentication.

Relationships

success	FlowFiles containing ESP events in AVRO format are passed to this relationship upon arrival.
failure	When ListenESP detects a failure in its interaction with the ESP engine, it sends a FlowFile to the failure relationship.

Writes Attributes

project	The project containing the current set of events.
contquery	The continuous query containing the current set of events.
window	The window containing the current set of events.
path	This is set to the model container of the event: project or continuous query.
filename	This is set to the window of the event with a timestamp appended.

Note: No Reads Attributes specified

Using the Cluster Manager

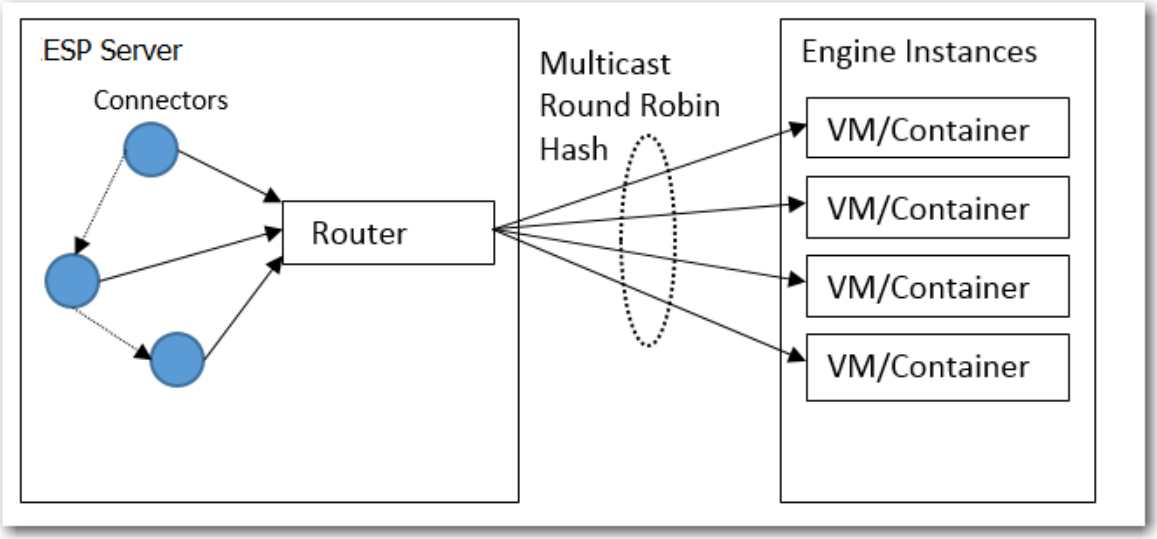
Overview

The Cluster Manager maps sources from edge devices to event stream processing engines that are provisioned within the cloud. By managing the mapping between connectors and engine instances, the Cluster Manager facilitates the elastic deployment of SAS Event Stream Processing in the cloud and eases large-scale deployment.

For example, you could use Cluster Manager when you deploy SAS Event Stream Processing as a service in Cloud Foundry. Typically, you create and manage a pool of VMs (or containers) that have their hardware resources in the cloud. Through a Cloud Foundry service broker API, you could request engine instances to be provisioned on the containers. After you provision those engine instances, you could instruct Cluster Manager to do the following:

- Deploy projects to provisioned engine instances through the administrative REST API.
- Start more than one data sources (through connectors) in an orchestrated fashion.
- Stream events for processing and analyzing through their publish/subscribe API.
- Dynamically add or remove an engine instance to or from the Cluster Manager's control.

The following figure shows that Cluster Manager runs on the ESP server.



Cluster Manager supports the following functions:

- Connector orchestration. In the figure, the dotted lines between connectors represent the order of connector execution.
- Data event routing. Cluster Manager automatically creates a router in the XML Server to map events to corresponding engine instances. Mappings are based on the policy that is defined in the Cluster Manager's configuration file.
- Loading projects to engine instances and dynamically managing engines. A SAS Event Stream Processing deployment in Cloud is intended to be flexible. You can dynamically add more engine instances when you need more processing power. You can remove engine instances when you do not need them.

When you run Cluster Manager, it reads the project XML file and deploys the project to engine instances that are described in the Cluster Manager configuration XML file. Cluster Manager provides a REST API to dynamically add or remove event stream processing engine instances. The routing policy reacts accordingly.

Three event routing policies are supported:

- Multicast policy sends every event to all the engine instances.
- Round Robin policy sends events to engine instances in a round-robin fashion.
- Hash policy hashes the value of some pre-defined fields and uses that value to decide where to send the event.

In Cloud deployment, you sometimes need to stream the same data to different engines. Multicast policy facilities this need. You do not have to deal with each engine instance individually and keep track of the engine instances that are subject to dynamic changes. Multicast policy relies on a multicast destination that serves as a single point of contact.

Round Robin policy distributes event blocks that are received from a connector to engine instances in a round robin fashion.

Hash policy examines each event that it receives from a connector and performs a hash function on the specified fields. It then sends the event to the engine instance according to the hash result. Fields are interpreted as strings, and the hash function might not cause a perfect uniform distribution because of its complexity constraint. Keep in mind that hashing can slow performance and significantly increase memory usage.

Suppose that you want to run an application to analyze stock trading events at real time in order to detect suspicious activity. Cluster Manager can request engine instances that are running in the Cloud and have them perform the following steps:

1 Load projects to engine instances so that trading events that are generated by data sources can be correctly received and processed.

2 Start connectors to read from data sources. The reads follow a pre-defined order (for example, before the trading events are streamed, a blacklist of brokers is read and uploaded to the engines).

3 Configure the connector to route events to the corresponding engine instances.

In an elastic Cloud environment, changing the configuration can be difficult. When a data source is generating trading events too quickly for the corresponding engine instances to process, new instances might need to be created dynamically. Even so, you need to map data sources to engine instances. That mapping usually requires scripted steps, which can be error-prone. You should balance the load in such a way that each engine instance takes a portion of the feeding events from a data source.

Running the Cluster Manager

Before you run Cluster Manager, engine instances must first have been provisioned in the cloud. These engine instances run the basic ESP server with their administrative and publish/subscribe ports open.

Use the following command to run Cluster Manager:

dfesp_cluster_manager -http-admin *port* -pubsub *port* -cluster-manager *model* <-auth> <-output-projects *file*> <-output-routers *file*>

Argument	Description
`-http-admin` ***`port`***	Specify the port for HTTP administrative commands.
`-pubsub` ***`port`***	Specify the port for publish/subscribe commands.
`-cluster-manager` ***`model`***	Specify the XML model for Cluster Manager to use.
`-auth`	(Optional) Enable authentication.
`-output-projects` ***`file`***	(Optional) Write the project's model to the specified file and deploy manually.
`-output-routers` ***`file`***	(Optional) Write the router's model to the specified file and deploy manually.

You can enter this command without arguments in order to obtain basic usage information.

Create a model (for example, **`file://config.xml`**) to specify the configuration file for Cluster Manager. The model must contain a reference to a project XML file that deploys projects into engine instances in the cloud. Internally, Cluster Manager parses the model and generates two configuration files that start connectors and a router on the local event stream processing server. To produce a dump of these two files, use the **`-output-projects`** and **`-output-routers`** arguments to the command that starts the ESP server.

You can use the following REST API to add and remove ESP instances:

- **`PUT /routerEngines/`*`routerId`*`/`*`engineId`***. The ***`routerId`*** is the name of the `<esp-map>` element specified in the configuration file. The ***`engineId`*** is the name of the `<esp-engine>` element specified in the configuration file. The request body must contain the engine definition.

  For example: **`curl -X PUT -d '<esp-engine name="esp3" host="10.37.24.3" port="31417" ha_port="31416"/>' http://localhost:12345/SASESP/routerEngines/esp-map1/esp2`**

- **DELETE /routerEngines/*routerId*/*engineId*.**

Specifying the Cluster Manager Configuration File

What follows is a sample configuration file for Cluster Manager. It contains four major sections:

- the projects to be loaded in engine instances that receive events from data sources, defined in the `<projects>` element
- data sources, defined in the `<raw-sources>` element
- the mapping from raw-sources to the destination windows (as specified in the projects) that are running on engine instances, defined in the `<esp-maps>` element
- a collection of engine instances, defined in the `<esp-clusters>` element

Note: The `<esp-cluster-managers>` container element currently supports only a single cluster manager.

```
<engine>
  <esp-cluster-managers>
        <esp-cluster-manager name='adapter-manager'>
        <!--
             projects definition: "test" references a project
             that is specified through the project-name element
             in a file that is specified by the project-url element
        -->
          <projects>
                  <project name='test' type='reference'>
                    <project-url>file://broker.xml</project-url>
                    <project-name>project</project-name>
                  </project>
          </projects>

          <raw-sources>
            <!--
                Specify a data sources as you would use in any esp-map.
                Connection parameters are identical to connector parameters in
                any typical model.
                Refer to the XML reference for details about specifying connectors.
            -->

            <raw-source name='trades' class='fs'>
              <!-- streaming trades -->
              <properties>
                <property name='type'>pub</property>
                <property name='fstype'>binary</property>
                <property name='fsname'>trades50M256B.bin</property>
              </properties>
            </raw-source>
            <raw-source name='venuesSource' class='fs'>
              <properties>
                <property name='type'>pub</property>
                <property name='fstype'>csv</property>
                <property name='fsname'>venues.csv</property>
              </properties>
            </raw-source>
            <raw-source name='restrictedSource' class='fs'>
              <properties>
```

```
              <property name='type'>pub</property>
              <property name='fstype'>csv</property>
              <property name='fsname'>restricted.csv</property>
            </properties>
          </raw-source>
        </raw-sources>
        <esp-maps>
          <!--
          The esp-map element defines raw-source to esp window connectivity
          and the orchestration of raw-sources. Esp-map has a reference
          to cluster as well as a reference to the model defined in the <projects>
          tag as a project name. An esp-map defines a mapping from a raw-source to
          destination window that is defined in model-ref and that is running on servers
          that are defined in the cluster-ref tag. Currently, the esp-map tag supports
          three types of destinations, multicast-destination, roundrobin-destination,
          and hash-destination. Esp-map also supports connector orchestration.
          -->
          <esp-map name='esp-map1' cluster-ref='openstack' model-ref='test'>
            <map name='venuesMap'>
              <from source='venuesSource'/> <!-- raw-source name -->
              <multicast-destination name='dest2' opcode='insert'>
  <!-- refer to the router destination configuration-->
                <publish-target>
                  <project-func>project</project-func>
                  <contquery-func>query</contquery-func>
                  <window-func>venuesSource</window-func>
                </publish-target>
                </multicast-destination>
            </map>

            <map name='restrictedMap'>
              <from source='restrictedSource'/>
              <multicast-destination name='dest3' opcode='insert'>
                <publish-target>
                  <project-func>project</project-func>
                  <contquery-func>query</contquery-func>
                  <window-func>restrictedSource</window-func>
                </publish-target>
              </multicast-destination>
</map>

            <map name='tradesMap'>
              <from source='trades'/>
              <hash-destination name='dest4' opcode='insert'>
                <publish-target>
                  <project-func>project</project-func>
                  <contquery-func>query</contquery-func>
                  <window-func>trades</window-func>
                </publish-target>
                <fields>
  <!—
  The field to be hashed. The hash value is an integer between 0 and the
  number of engine instances minus one. The router uses the hash value to
  determine which engine instance the event is sent to.
  -->
                  <field name='broker'/>
```

```
            </fields>
          </hash-destination>
        </map>

        <orchestration> <!-- refer to connector orchestration for details-->
          <adapter-groups>
            <adapter-group name='G1'>
              <adapter-entry adapter='venuesSource' state='finished'/>
              <adapter-entry adapter='restrictedSource' state='finished'/>
            </adapter-group>

            <adapter-group name='G2'>
              <adapter-entry adapter='trades' state='finished'/>
            </adapter-group>
          </adapter-groups>
          <edges>
            <edge source='G1' target='G2'/>
          </edges>
        </orchestration>
      </esp-map>
    </esp-maps>

    <esp-clusters>
      <!--
      An esp-cluster is a collection of running engine instances.
      The value of host is the IP address of the instance.
      The value of port is the pubsub port.
      The value of ha_port is the http-admin port.
      -->
      <esp-cluster name='openstack'>
        <esp-engines>
          <esp-engine name='esp1' host='10.37.24.3' port='31415' ha_port='31414'/>
          <esp-engine name='esp2' host='10.37.24.3' port='31417' ha_port='31416'/>
          <esp-engine name='esp3' host='10.37.24.3' port='31419' ha_port='31418'/>
        </esp-engines>
      </esp-cluster>
    </esp-clusters>
  </esp-cluster-manager>
</esp-cluster-managers>
</engine>
```

Using the Cluster Manager with SASLogon

Oauth2 is a protocol that enables Internet users to log on to third-party websites using public accounts without exposing their passwords. SASLogon is an Oauth2 server that is compatible with Cloud Foundry UAA.

To enable SASLogon on the ESP server, enter the following at the command prompt:

dfesp_xml_server –saslogon–url *authorization_server_url*

Authorization occurs after the following:

1. A project is loaded to an engine.
2. SASlogon is enabled through its HTTP admin port.
3. The publish/subscribe client connects to a window in that engine.

Because Cluster Manager deploys projects to remote engines, it must specify an authorized user in the configuration file. Specifically, in the `<esp-cluster>` element of the model, here is how you specify an engine that requires SASLogon:

```
 <esp-engine name='esp1' host='localhost' port='31415' ha_port='31414'>
<auth-user>test_client</auth-user>
</esp-engine>
```

You must create an .authinfo file that can be found by Cluster Manager. Cluster Manager retrieves the password from this file and uses it to access the engine.

For more information, see "Authentication Using a User Name and Password" in SAS Event Stream Processing: Security.