Using SAS Event Stream Processing within a Cloud Infrastructure

Overview
SAS Event Stream Processing is cloud-ready for large-scale distributed services. You can deploy it almost anywhere, from data centers to edge devices and from physical machines to clouds.

Deploying SAS Event Stream Processing to a Self-Installed Kubernetes Cluster

Overview
Use these instructions to build multiple Docker images and then use those images to create a deployment of SAS Event Stream Processing (Full) in Kubernetes. You deploy the ESP server, SAS Event Stream Processing Studio, and Streamviewer.

Important: These instructions assume that you are an experienced Kubernetes administrator.

Before You Begin
Kubernetes is an open-source system to deploy and manage applications in the cloud. Before you deploy SAS Event Stream Processing (Full) to a self-installed Kubernetes cluster, you must meet the following prerequisites:

- A supported version of Docker CE (Community Edition) is installed on your build system. For more information, see https://docs.docker.com/install/
- Access to a Docker registry. The build process pushes Docker images automatically to the Docker registry.
- A Kubernetes cluster is installed and ready for use. kubectl is installed on your build system. This program is required for the run step but not for the build step. For more information, see https://kubernetes.io/docs/tasks/tools/install-kubectl/.
Deployment Steps

1. On your build system, create an rwx directory on the system from which you want to deploy SAS Event Stream Processing (Full). Navigate to that directory, making it your current working directory.

2. Locate your Software Order Email (SOE) and retrieve the `SAS_Viya_deployment_data.zip` file from it. Copy the file to your current working directory.

3. Clone the following repository from GitHub in your current working directory. The `build.sh` script that you use in a later step is located in this repository.

   
   ```
   $git clone git@github.com:sassoftware/sas-container-recipes
   
   ```

4. From your current working directory, log on to the Docker registry:

   ```
   $docker login docker.registry.your_company.com
   
   ```

   Ensure that the values in `$HOME/.docker/config.json` are correct for your computing environment. For more information, see [https://docs.docker.com/registry/](https://docs.docker.com/registry/).

5. From your current working directory, run the `build.sh` script. For example:

   ```
   ./build.sh --type full --docker-registry-namespace esp
   
   ```

   The `build.sh` script creates the following Docker images and pushes them to the Docker registry:

   Table 1  Docker Images

<table>
<thead>
<tr>
<th>Entity</th>
<th>Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS Data Service</td>
<td>sas-viya-sasdatasvrc</td>
</tr>
<tr>
<td>RabbitMQ message bus</td>
<td>sas-viya-rabbitmq</td>
</tr>
<tr>
<td>Postgres DB</td>
<td>sas-viya-pgpoolc</td>
</tr>
<tr>
<td>Reverse proxy server</td>
<td>sas-viya-httpproxy</td>
</tr>
<tr>
<td>SAS data configuration</td>
<td>sas-viya-configuratn</td>
</tr>
<tr>
<td>SAS ESP design studio</td>
<td>sas-viya-espstudio</td>
</tr>
<tr>
<td>SAS ESP Dashboard tool</td>
<td>sas-viya-espstreamviewer</td>
</tr>
<tr>
<td>SAS ESP server</td>
<td>sas-viya-espserver</td>
</tr>
<tr>
<td>SAS core microservices</td>
<td>sas-viya-coreservices</td>
</tr>
<tr>
<td>SAS configuration DB</td>
<td>sas-viya-consul</td>
</tr>
</tbody>
</table>

6. Prebuilt deployment manifests for Kubernetes are also created by `build.sh`:

   ```
   [sas-container-recipes]$ cd builds/full/manifests/
   [manifests]$ tree
   ```
7 On your Kubernetes cluster, run the following commands to deploy the containers:

```bash
$ kubectl apply -f viya-visuals/working/manifests/kubernetes/namespace
...
$ kubectl -n sas-viya apply -f viya-visuals/working/manifests/kubernetes/configmaps
...
$ kubectl -n sas-viya apply -f viya-visuals/working/manifests/kubernetes/ingress
...
$ kubectl -n sas-viya apply -f viya-visuals/working/manifests/kubernetes/secrets
...
$ kubectl -n sas-viya apply -f viya-visuals/working/manifests/kubernetes/services
...
$ kubectl -n sas-viya apply -f viya-visuals/working/manifests/kubernetes/deployments
```

You must configure the Kubernetes cluster for multi-tenancy following the instructions in *SAS Viya Administration: Multi-tenancy*. 
Deploying SAS Event Stream Processing on Amazon EKS

Overview
Amazon Elastic Container Service for Kubernetes (Amazon EKS) runs scalable Kubernetes clusters on Amazon Web Services (AWS). You can deploy SAS Event Stream Processing to Amazon EKS. For more information about Amazon EKS, see Amazon EKS documentation.

Important: These instructions assume that you have experience with Kubernetes clusters and with Amazon EKS.

Before You Start
Set up a Kubernetes cluster and make it ready to accept your deployment. For more information about creating a Kubernetes cluster, start here. For more information about kubectl, see the documentation.

It is highly recommended that you use a preexisting, accessible Amazon Elastic Container Registry (Amazon ECR). For more information about ECR, see the documentation.

Create and Deploy Docker Images
Use the steps provided in "Deployment Steps" to create the Docker images of SAS Event Stream Processing to deploy.

Upload the Docker images to Amazon ECR with the Docker push command. This ensures that the images can be reached and expedites the instantiation of containers.

Deploy Kubernetes Ingress
It is recommended that in production environments, you deploy the NGINX Ingress controller for Kubernetes in order to govern how external users access services running there. Follow these instructions to install the NGINX Ingress controller.

You deploy and expose the NGINX Ingress controller with two services:

- nginx-ingress (for HTTP traffic)
- nginx-ingress-tcp (for TCP traffic)

Note: It is important to configure the system so that TCP traffic does not enable TCP proxy. SAS Event Stream Processing’s publish/subscribe API does not work with the TCP proxy protocol.

Use the following YAML file to configure the NGINX Ingress controller services. Customize parameter values appropriately for your computing environment.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
  namespace: nginx-ingress
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-backend-protocol: "tcp"
    service.beta.kubernetes.io/aws-load-balancer-proxy-protocol: "***
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
```
protocol: TCP
  name: http
- port: 443
  targetPort: 443
  protocol: TCP
  name: https
selector:
  app: nginx-ingress

---
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress-tcp
  namespace: nginx-ingress
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-backend-protocol: "tcp"
    #service.beta.kubernetes.io/aws-load-balancer-proxy-protocol: "*"
spec:
  type: LoadBalancer
  ports:
  - port: 31416
    targetPort: 31416
    protocol: TCP
    name: tcp
  selector:
    app: nginx-ingress

---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ingress
  namespace: nginx-ingress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-ingress
  template:
    metadata:
      labels:
        app: nginx-ingress
    spec:
      serviceAccountName: nginx-ingress
    containers:
    - image: nginx/nginx-ingress:edge
      imagePullPolicy: Always
      name: nginx-ingress
      ports:
      - name: http
        containerPort: 80
      - name: https
        containerPort: 443
      - name: tcp
        containerPort: 31416
---

env:
- name: POD_NAMESPACE
  valueFrom:
    fieldRef:
      fieldPath: metadata.namespace
- name: POD_NAME
  valueFrom:
    fieldRef:
      fieldPath: metadata.name

args:
- -nginx-configmaps=${POD_NAMESPACE}/nginx-config
- -default-server-tls-secret=${POD_NAMESPACE}/default-server-secret

Note: Note that the following line has been commented out in this YAML file:

```
service.beta.kubernetes.io/aws-load-balancer-proxy-protocol: "***"
```

After you deploy the `nginx-ingress` and `nginx-ingress-tcp` services, two AWS Application Load Balancers (ALBs) should be created automatically. By default, AWS uses the Classic Load Balancer, which supports TCP traffic. Make sure that the security groups that are attached to the ALBs are correct.

Use the following YAML file to deploy Ingress to govern HTTP and HTTPS traffic.

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: esp-ingress
spec:
rules:
- host: iot.your_company.com
  http:
    paths:
    - path: /eventStreamProcessing/v1
      backend:
        serviceName: sas-viya-espserver
        servicePort: 31415
```

Use the following YAML file to configure the NGINX Ingress controller for TCP traffic:

```yaml
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-config
namespace: nginx-ingress
data:
  proxy-protocol: "True"
  real-ip-header: "proxy_protocol"
  set-real-ip-from: "0.0.0.0/0"
stream-snippets: |
  upstream pubsub {
    server xx.xx.xx.xx:31416; # specify the esp-viya-espserver’s cluster IP
  }
  server {
    listen 31416;
    proxy_pass pubsub;
  }
```

**Test HTTP and HTTPS Traffic**

To test whether HTTP and HTTPS traffic works as expected, follow these steps:
1 Make the host name `iot.your_company.com` resolvable. If host name is not resolvable, determine the IP address manually.

2 Run the following command on the Kubernetes cluster: `kubectl describe svc nginx-ingress --namespace=nginx-ingress`

3 Find the host name for the load balancer (for example, `id.us-east-1.elb.amazonaws.com`).

4 Run the following command on the Kubernetes cluster to obtain a list of IP addresses:
   ```bash
   $nslookup id.us-east-1.elb.amazonaws.com
   ```

5 Choose one IP address (for example, `10.0.0.10`) and then use the `curl` command to test it:
   ```bash
   $curl -resolve iot.your_company.com:80: 10.0.0.10 http://iot.your_company.com/eventStreamProcessing/v1
   ```
   The command should hit the load balancer's IP, `10.0.0.10:80`, which is routed to the Ingress controller.

   The Ingress controller sees `iot.your_company.com/eventStreamProcessing/v1` as a match to the Ingress rule and forwards the HTTP packet to service `sas-viya-espserver:31415`.

   You should receive an HTTP 200 OK response. After that, you can try to load a model:
   ```bash
   $curl -X PUT --resolve iot.your_company.com:80: 10.0.0.10 http://iot.your_company.com/eventStreamProcessing/v1/projects/project_01 --data-binary @model.xml.
   ```

### Test the TCP Configuration

To test whether the TCP configuration is correct, use a file and socket adapter. Assume that a model has been installed and that data exists in `input/input.json` for testing and that you use the same IP address as before.

```bash
$kubectl describe svc nginx-ingress-tcp --namespace=nginx-ingress
...
$nslookup 10.0.0.10
...
$dfesp_fs_adapter -k pub -h "dfESP://10.0.0.10:31416/project_01/cq_01/input" -f input/input.json -t json -e -b 12 -Q
```

Monitor the ESP server's log to ensure correct operation. The TCP traffic travels this path:

```
Desktop -> TCP ELB external IP:31416 -> Nginx Controller -> Service (sas-viya-espserver) -> Container (sas-viya-
```

**Important:** You must use the same port (31416) in the container for TCP traffic and for the load balancer for external user TCP access. Otherwise, TCP traffic is impeded.

### Other Deployments

SAS Event Stream Processing can be used with deployment tools such as BOSH, Cloud Foundry, and Chef. Contact your SAS representative to initiate a SAS Event Stream Processing deployment using these tools.
Using the Cluster Manager

Overview

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

For example, you can use the 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 can request engine instances to be provisioned on the containers. After you provision those engine instances, you can instruct the Cluster Manager to do the following:

- Deploy projects to provisioned engine instances through the administrative REST API.
- Start more than one data source (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 the Cluster Manager runs on the ESP server.

Figure 1  Cluster Manager

The 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.** The Cluster Manager automatically creates a router in the ESP 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 the 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 the 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. The Cluster Manager provides a REST API to dynamically add or remove event stream processing engine instances. The routing policy reacts accordingly.

**Note:** The ESPENV environment variable is not supported by the Cluster Manager. For example, suppose you run this on the command line:

```
$export ESPENV='GetLogonConfig'
```

The Cluster Manager fails to load the project using this environment variable.

**Routing Policies**

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.
- Durable Hash policy treats the hash value space as a ring. It uses two hash functions: one to hash the ID of a server and the other to hash an event based on fields specified in the XML model.

In Cloud deployment, you sometimes need to stream the same data to different engines. Multicast policy facilitates 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.

Durable Hash policy is suitable for a dynamic environment where servers are on and off unpredictably. When you use Durable Hash, the addition or removal of a server affects only the server; it does not lead to a change in the hash function itself. Events hashed to another server continue to be hashed to that server.

**Running the Cluster Manager**

Before you run the Cluster Manager, provision engine instances in the cloud. Run the basic ESP server with their HTTP and publish/subscribe ports open on these engine instances.

Use the following command to run the Cluster Manager:

```
desp_cluster_manager -httpport <http port> -pubsubport <pubsub port> -cluster-managermodel <model> <-auth> <-output-projects <file> <-output-routers <file>
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-http port</code></td>
<td>Specify the port for HTTP commands.</td>
</tr>
<tr>
<td><code>-pubsub port</code></td>
<td>Specify the port for publish/subscribe commands.</td>
</tr>
<tr>
<td><code>-cluster-manager model</code></td>
<td>Specify the XML model for the Cluster Manager to use.</td>
</tr>
</tbody>
</table>
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 the Cluster Manager. The model must contain a reference to a project XML file that deploys projects into engine instances in the cloud. Internally, the 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/eventStreamProcessing/v1/routerEngines/esp-map1.esp2

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

## Specifying the Cluster Manager Configuration File

What follows is a sample configuration file for the 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.

```xml
<engine>
  <esp-cluster-managers>
    <esp-cluster-manager name='adapter-manager'>
      <projects>
        <project name='test' type='reference'>
          <project-url>file://broker.xml</project-url>
          <project-name>project</project-name>
        </project>
      </projects>
    </esp-cluster-manager>
  </esp-cluster-managers>
  <raw-sources>
    <raw-source name='trades' class='fs'>
      <properties>
        <property name='type'>pub</property>
        <property name='fstype'>binary</property>
      </properties>
    </raw-source>
  </raw-sources>
</engine>
```
<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>
<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'>
    <publish-target>
      <project-func>project</project-func>
      <contquery-func>query</contquery-func>
      <window-func>venuesSource</window-func>
    </publish-target>
  </multicast-destination>
</map>
</esp-map>
<map name='restrictedMap'>
  <from source='restrictedSource'/> <!-- raw-source name -->
  <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>
</esp-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>
      <field name='broker'/>
    </fields>
  </hash-destination>
</map>
<orchestration>
  <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-clusters>
  <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-cluster-manager>
  </esp-cluster-managers>
</engine>

1 Here, test references a project that is specified through the project-name element in a file that is specified by the project-url element.

2 Specify a data source as you would use in any esp-map element. Connection parameters are identical to connector parameters in any model.

3 This element defines the connectivity between raw-source and destination window connectivity and the orchestration of raw-sources. The element references the cluster as well as the model defined in the projects tag. The destination window is defined in the model-ref tag. That window 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.

4 This is the field to be hashed. The hash value is an integer between 0 and N number of engine instances minus one. The router uses the hash value to determine what engine instance receives the event.

5 For more information, see “Orchestrating Connectors” in SAS Event Stream Processing: Connectors and Adapters.

6 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 publish/subscribe port. The value of ha_port is the http port.

**Cluster Redundancy**

Enable cluster redundancy by setting up spare engines. When an engine fails, the router automatically and transparently replaces it with the specified spare engine. The spare engine inherits the failed engine’s name, so that the events hashed to the failed engine are routed to the new engine.
Use the `spare-esp-engines` element to set them up.

Here is an example:

```xml
<esp-cluster name='cluster1' redundancy='2'>
  <esp-engines><!-- one of more esp servers, do not use esp_local, which is a reserved name -->
    <esp-engine name='esp1' host='localhost' port='41003' ha_port='41001' />
    <esp-engine name='esp2' host='localhost' port='41006' ha_port='41004' />
    <esp-engine name='esp3' host='localhost' port='41009' ha_port='41007' />
  </esp-engines>
  <spare-esp-engines>
    <esp-engine name='esp4' host='localhost' port='51003' ha_port='51001' />
  </spare-esp-engines>
</esp-cluster>
```

When you specify more than one spare, the system traverses the list in order.

After you set `redundancy=n`, an event is copied to `n-1` redundant engines when is hashed to an engine. That way, it can survive up to `n-1` simultaneous engine failures.

Note: Currently, only Hash and Durable Hash policies support redundancy.

Note: The addition or removal of a spare engine using the REST API is not supported.

Note: Adding or removing an engine to a cluster does not trigger the swap of a spare 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:

```
desp_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 port.
3. The publish/subscribe client connects to a window in that engine.

Because the 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:

```xml
<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 the Cluster Manager. The Cluster Manager retrieves the password from this file and uses it to access the engine.

For more information, see “Overview” in *SAS Event Stream Processing: Security*.

### Cluster Manager Failover

Enable Cluster Manager failover by setting up a cluster of Cluster Managers. Make one of them the leader and designate the others standby.

Note: The Cluster Manager does not support failover for routing policies other than Hash.
For example, suppose you start a normal Cluster Manager as follows:

dfesp_cluster_manager -http 12347 -pubsub 12348 -cluster-manager file://config.xml

Start two other Cluster Managers and designate them standby:

dfesp_cluster_manager -http 12341 -pubsub 12342 -cluster-manager file://config.xml -standby localhost:12347

dfesp_cluster_manager -http 12343 -pubsub 12344 -cluster-manager file://config.xml -standby localhost:12347

All three Cluster Managers reference the same configuration file. Each uses a different http and pubsub port. The first Cluster Manager that you started becomes the leader.

The –standby parameter sets the Cluster Manager to a standby mode. It notifies the leader of its existence. When the leader fails, one of the standby Cluster Managers automatically becomes the new leader and continues publishing to the remote servers. The new leader traces the last received message by the targets so that it can start sending from the next message. In this way, a message is sent once and only once as long as there is a standby Cluster Manager available.

Notice that the standby Cluster Manager specifies the leader’s host:port as the argument to the –standby parameter. Events published to these standby Cluster Managers are queued in the memory.

The failover mechanism is implemented using HTTP protocol and uses the existing HTTP port. Cluster Manager failover assumes the sources of data are reliable (have their own fault tolerance guarantee) and guarantees that all events are delivered with order once and exactly once. The leader Cluster Manager periodically broadcasts an update message to standby Cluster Managers.

The Cluster Manager has four command line parameters that can be used to customize the configuration. To make the protocol consistent, the standby Cluster Managers should use the same configuration of these parameters as the leader.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticktime</td>
<td>Defines the heartbeat rate in milliseconds. The default value is 1000ms. When the Cluster Manager runs on an edge device, specify a long tick time.</td>
</tr>
<tr>
<td>retries</td>
<td>Defines the number of updates to receive before concluding that the leader is down. The default value is 2. To omit transient failures, set retries to a large value. The failover protocol is less aggressive, but there is a longer failure detection time.</td>
</tr>
<tr>
<td>timeout</td>
<td>Defines the time-out in ms. When a connection is half closed, the initiator of the connection deems the server to be unreachable. An example of a half-closed connection is when a server accepts a connection request but does not respond within the defined time. The default value is 3000.</td>
</tr>
<tr>
<td>buffersize</td>
<td>Specifies the size of the queue. The default value is 1m.</td>
</tr>
</tbody>
</table>

When a standby Cluster Manager fails to receive a number of retries in a row, it assumes that the leader is down. The standby initiates an election immediately until all the standby Cluster Managers reach a consensus choice for the next leader. Then the next leader assumes leadership. This procedure goes on until all Cluster Managers fail.

Inside the Cluster Manager, the publish/subscribe layer maintains a queue for each ESP engine. If an engine is in standby mode, events are queued rather than published.