OpenAMQ Clustering
High-Availabilty and Partitioning

version 2.1

iMatix Corporation <openamq@imatix.com>

Revised: 2006/12/14

Contents

1 Cover

1.1 State of this Document

This document is a technical whitepaper.

1.2 Copyright and License

1.3 Authors

This document was written by Pieter Hintjens.

1.4 Abstract

We analyse the requirements for OpenAMQ local-area and wide-area clustering and we design a generic clustering model based on simple and easy to manage elements. The clustering model provides two main functionalities: the use of high-availability pairs of servers and the ability to partition groups of client applications, for reasons of scalability and geographic distribution.

2 Introduction

2.1 Scope and Requirements

This document describes the requirements, architecture, and design of the OpenAMQ clustering implementation. The OpenAMQ clustering implementation covers two main requirements:

1. The use of high-availability pairs to create automatic failover systems in which applications can switch to a backup server if a primary server crashes.
2. The interconnection of servers or high-availability pairs into larger architectures that can support distributed request-response and pub-sub scenarios.

2.2 Out-of-Scope

We do not attempt to support these functions:

1. The use of 'active backups'. In a HA pair, the backup server is inactive and does no useful work until the primary server goes offline.
2. The load-balancing of applications between servers. While applications can be spread-out between servers (or HA pairs), this happens manually, through the technique of "partitioning", i.e. the grouping of applications into well-defined blocks, each served by a specific server or HA pair.
3. The handling of persistent messages or transactions in any specific way. While our clustering design is compatible with persistent messaging and transactions, these introduce specific concerns (such as the processing of messages during failover) which we do not address at present.

2.3 Technical Terminology

These terms have significance within the context of this document:

Partition A group of AMQP client applications that are served by a single OpenAMQ server, or a pair of servers that collaborate as a "high-availability pair".

Partitioned Network A network of OpenAMQ servers or HA pairs (which we call a "node"). Each node serves a well-defined set of client applications, where applications are tied to a particular node. (A single application may work with multiple nodes, but this must be done explicitly, there is no concept of load-balancing between nodes.)

Node A server or HA pair in a partitioned OpenAMQ network. A node is either one server or two servers that collaborate as a high-availability pair in a primary-backup relationship designed to provide fast failover in case of failure of the primary server.

Cluster A general term that covers any network architecture involving more than one OpenAMQ server in a collaborative relationship.

2.4 Scenarios

These are typical scenarios that we wish to support in OpenAMQ clustering:

- A set of applications working with a single message broker need a backup broker than can take over in case of problems with the primary broker. The failover needs to happen rapidly, and automatically, so that ongoing business is not interrupted.
- A regional location is connected to a central location by a slow satellite link. The regional applications need to access market data and business services that are provided centrally. Traffic across the satellite link must be optimised so that messages to multiple subscribes are sent only once.
- A customer requires a message broker installed at their location so that local applications can inter-operate; these applications also subscribe to data feeds from a central server.

2.5 Basic Functionality

We can organise the clustering functionality to two layers:

1. High-availability, in which two servers collaborate to ensure that a set of applications are well-served.
2. Partitioning of applications into multiple groups so that we can create larger networks of servers.

We look at each of these aspects in more detail.

2.5.1 High-Availability

High-availability covers the following scenario:

- A set of applications are working with a server on a computer system.
- Either the software or the hardware crashes.
- The applications detect the failure and reconnect to a backup server that is ready and running.

When we look at implementing high-availability, we must consider a number of aspects:

1. How many servers do we need to create a satisfactory level of redundancy?
2. What happens to messages published during the failover process?
3. What level of support is required from client applications (that is, the AMQP client API)?
4. Does the high-availability design place restrictions on the network architecture?
5. How do we avoid the "split-brain scenario" in which both servers of a HA pair believe they are the active server?

Each of these questions has good answers, which we will look at later. In gross terms, we will aim to keep the HA design as simple as possible, since any additional complexity creates its own unreliability.

2.5.2 Application Partitioning

Partitioning means "divide and conquer", in a simple and direct sense. When we have a large number of applications, it can be more useful to divide these across a set of servers, rather than concentrate them at a single point.

The alternative to partitioning is "load-balancing", in which a group of applications is shared across multiple servers.

Whether or not we need load-balancing is a very important question, because traditional clustering is about load-balancing as much as it is about high availability, and rather a lot less about partitioning. (Few middleware products implement partitioning, which is the basis for grids, wide-area networks, and other interesting organisations).

Traditionally, load-balancing is a performance option. This makes sense in servers that have severe limitations - perhaps limited by the number of file handles (=sockets) that a single process can open, or simply by an inefficient protocol and implementation that does not allow scaling.

In our experience, AMQP and OpenAMQ are fast enough that load-balancing is not a key requirement. However, we can also divide typical messaging into two domains, request-response and publish-subscribe, and say that while we do not need to use performance boosting architectures for request-response processing, which is usually low on volume and high on reliability, we do potentially need performance-boosting architectures for publish-subscribe.

On analysis, we discover that we can use partitioning as an effective and simple model for performance improvement, in a publish-subscribe scenario. This works by using a 'fanout' architecture (a tree or star model) in which messages are distributed from a central point to multiple servers, each serving a well-defined set of applications.

Partitioning thus covers these needs:

1. High-performance pub-sub architectures, for market-data and other very high-volume scenarios.
2. Geographic partitioning of applications, e.g. between central and regional offices.

2.6 General Architecture

2.6.1 The Network "Node"

We construct a partitioned network out of nodes that are either standalone servers, or high-availability pairs. From the perspective of the network, whether a node is a single server or a HA pair makes zero difference; this is a purely local decision based on the need for service continuity at that node.

To simplify discussion, we use the term "node" to cover either of one OpenAMQ server, or a HA pair of OpenAMQ servers, collaborating to serve a single set of client applications.

2.6.2 Partitioning Models

These are a number of plausible network architectures, in which each node in the network is a HA pair or a single stand-alone server:

1. A star network, with a single central node and many distributed nodes. This would typically be used to send information from a central point to regional networks, each serving a set of local applications and users.
2. A tree network, with one top-level node, a small number of second-level nodes, and more nodes connected to these, in a tree hierarchy. This would typically be used to create extremely-high volume data publishing networks (capable of serving over a million messages per second).
3. A loose network, with ad-hoc relationships between nodes organised at regional, national, and global levels. This would typically be used for real-life global organisations with diverse information streams, each involving a set of nodes.

We do not favour any of these organisations, but partitioning provides the tools to build them all, and others that we have not yet considered.

2.6.3 Message Pull and Push

AMQP is easily compatible with organising the flow of messages between partitions. The main semantics for message flow between two partitions are:

1. The receiving partition can subscribe to a set of messages.
2. The sending partition can forward a set of messages.

These are typical "pull" and "push" models. Within the context of AMQP we can define exactly what each of these mean.

Subscription, in AMQP terms, means creating a private queue and binding this to one or more exchanges with specific binding conditions. Messages published to those exchanges, which match the conditions, are copied into the private queue where they can be consumed. This is how ordinary applications subscribe to messages; this is also how partitions pull messages from each other.

We can in fact construct a full partitioned network using only pull mechanisms. However, this means that for each pair of connected partitions we have to configure subscriptions at each side. The administrative cost for this can be significant.

The push mechanism is a useful alternative because it lets us configure the connection between the partitions at one node only (both pull and push). This means that adding a partition to an existing network can be done without any configuration changes to the existing nodes.

To forward messages usefully we need some clear semantics. From our analysis it makes sense to forward messages on a per-exchange basis under one of these (configurable) conditions:

- Forward all messages, but process local bindings additionally.
- Forward only messages that do not match any local bindings.

There are other possibilities (such as forwarding all messages) but we do not have use cases for those.

3 Architecture and Design

3.1 Layers of Operation

We break the high-availability and partitioning architecture into a number of specific layers, each solving one problem:

1. The general ability to connect pairs of servers together. This is a core requirement for both high-availability and partitioning. We call this "Server Peering".
2. The ability for peers to exchange arbitrary non-application information. This is a requirement for high-availability (so that peers can agree on who is the active server). We call this "Internal Messaging".
3. A layer in each server that handles the high-availability functionality. This is called the "High-availability Controller", or HAC.
4. A layer in each server that pushes messages to or pulls messages from another server, on behalf of its own client applications. We call this the "Message Transfer Agent", or MTA.
5. The use of globally-unique names for reply queues, so that messages can be routed successfully across a partitioned network. We call this "Global Routing".

We explicitly will NOT look at:

- The ability to automatically define the topology, called "discovery". In our current OpenAMQ use cases, the cluster topology is defined manually by the system administrator.
- The ability to spread large numbers of clients across many servers, called "load balancing". First, current OpenAMQ implementations are capable of handling very large loads; secondly we can use partitioning to support more clients if necessary.
- The ability to use clustered servers in any other role, for example as part of a message persistence scheme.
- The ability to replicate the state of objects such as exchanges and shared queues between servers.

3.2 Server Peering

The basic relationship between servers is called a "peering" and consists of an asymmetric relationship from one server to another.

A peering goes in one direction, from a 'client' host onto a 'target' host. To create a symmetric relationship between two servers, we define one peering in each direction.

Peerings are internal objects, not configured by administrators, nor visible to them, but created by other parts of the OpenAMQ server as needed. Peerings have these properties:

- A peering will automatically detect network failures and target host failures, and will suspend and reconnect as required.
- A peering acts as a remote client, with the only difference from the perspective of the target host is that peerings may use their own authentication data.
- A peering uses a single configured network route. If two peers talk over multiple network interfaces (for redundancy), this will imply multiple peerings.

A peering automatically creates a private queue on the target host and consumes messages off this queue. Incoming messages are delivered to whatever server entity created the peering.

3.3 Internal Messaging

Internal messaging is the exchange of information between entities at each side of a peering. AMQP does not provide any semantics for such exchanges, so we must build our own on top of AMQP.

The simplest way to add internal messaging semantics to AMQP is to use the exchange concept as a routing point. That is, when one entity wishes to send status information to another, across a peering, it would create a content (a message) and publish that to a specific exchange at the destination server.

While the AMQP specifications include a "tunnel" class that could also be used for this purpose, the semantics of publishing via an exchange are simple and generic enough that we will use that mechanism.

Internal messaging is thus handled by an exchange called "amq.peer", and using a set of specific binding criteria and messages that allow the creation of simple request-response and pub-sub messaging models between peer entities. (Note that such messages never leave the high-availability pair.)

3.4 High-availability Controller (HAC)

The HAC is an entity that operates within each server. There is no external controller process.

The two HACs in a HA pair use peering and peer messaging to talk to each other. They exchange messages that include:

- Their own current status (active or passive).
- The number of clients they have connected (must be zero for passive servers).
- Their server identity ('name').

Each HAC implements an algorithm that continually detects the state of the other HAC and decides whether to become passive or active. The two HACs do not use the identical algorithm - there is a "primary HAC" and a "backup HAC" and the algorithm is slightly different in each case.

3.4.1 High-Availability Requirements

We can list our requirements for a high-availability architecture:

1. The time for a failover must be under 60 seconds and preferrably under 10 seconds.
2. Failover must happen automatically, while recovery must happen manually, so that application service interruptions are minimise.
3. We need a single backup server to ensure an acceptable level of security. Multiple backup servers creates extra complexity that creates its own risks.
4. We assume that the HA pair is connected by a reliable network, possibly by multiple independent networks.
5. We assume that the HA backup server is passive, not doing any work until called upon to replace the HA primary server. That is, we do not require an active-backup server (which also adds complexity without necessarily adding any value).
6. We assume that the HA backup and HA primary server are equally capable, or at least that the HA backup can carry the full application load. There is no attempt to load-balance the application load across multiple backup servers.
7. We need simple semantics for how client applications should work with a HA pair.

8. We need clear instructions for network architects on how to avoid designs that could result in a "split-brain" syndrome in which the HA pair becomes split into two nodes through specific network failure scenarios.
9. The startup sequence of a HA pair must not matter; that is, there must be no timing-dependent issues at startup.
10. It must be possible to make planned stops and restarts of either server without stopping the client applications.
11. Operators must be able to monitor both servers at all times.

3.4.2 General Principles

The HAC has three internal states:

- Pending: the server has started and is waiting for its peer to start.
- Active: the server has become the active server for the partition.
- Passive: the server is the passive (inactive failover) server for the partition.

When a server is running in HA mode, and a client application attempts to connect, what happens depends on the the HAC state, and the type of client:

- If the client is the other peer in a HA pair, or a console user, it is allowed to connect as usual, irrespective of the HAC state.
- If the HAC is pending, normal clients are held in a connection 'opening' state until the HAC leaves the pending state, or the client times out.
- If the HAC is active, the client connection is accepted as usual.
- If the HAC is passive, the client connection is closed.

If a server becomes passive, and has normal clients, it disconnects those clients. Console users and HA peers are not disconnected.

3.4.3 Primary HAC Behaviour

The primary HAC has these states and transitions:

- (Primary Pending) - backup server comes online and is pending or passive —> Primary Active. Meaning: normal HA startup.
- (Primary Active) - backup server comes online and is active —> Primary Passive. Meaning: primary has been restarted, and backup has taken over active role.
- (Primary Passive) - backup server has gone offline and there is at least one connected client or client attempting to connect —> Primary Active. Meaning: primary has been restarted, and now backup server has gone down.
- (Primary Passive) - backup server is passive —> Primary Active. Meaning: operator has forced backup server into passive mode to allow HA pair to resume normal configuration.

All other events result in no state changes.

3.4.4 Backup HAC Algorithm

The backup HAC has these states and transitions:

- (Backup Pending) - primary server comes online and is pending or active —> Backup Passive. Meaning: normal HA startup.
- (Backup Passive) - primary server has gone offline and a client attempts to connect —> Backup Active. Meaning: primary has crashed or stopped and backup is still correctly connected to network.
- (Backup Passive) - primary server is passive —> Backup Active. Meaning: operator has forced primary server into passive mode to force HA pair into failover configuration.
- (Backup Active) - primary server has gone offline and backup server has no connected clients —> Backup Passive. Meaning: backup is not connected to network any longer.

Other events result in no state changes.

3.4.5 Failover Scenarios

Here are a number of scenarios that will result in failover happening. Each starts from the same point, two HA servers, an active primary and a passive backup:

1. The primary server process or system crashes, and the applications reconnect to the backup server.
2. The primary server is disconnected from the network long enough for at least one application to disconnect (perhaps using agressive heartbeating) and reconnect to the backup server while the primary server is offline.
3. The operator forces the primary server into passive mode using the console.

3.4.6 Recovery Process

There are several ways to recover from a failover. First, the operators can always restart the primary server, with no impact on applications. The primary server will not become active; it will defer to the backup server.

Next, the operators can either stop and restart the backup server, or they can use the operator console to force it to go into passive mode.

3.4.7 Normal Shutdown Process

The normal shutdown process is to either:

1. Stop the passive server and then stop the active server, or
2. Stop both servers in any order but within a few seconds of each other.

Stopping the active and then the passive server with any intervening delay will force applications to disconnect, then reconnect, then disconnect again, which may disturb users.

3.4.8 Split-Brain Prevention

The algorithm for selecting the active/passive server eliminates split-brain situations in all usual cases.

In some unusual cases, split-brain situations can still happen, and we can document these cases, and avoid them.

The basic rule is that a peer will not become the active server unless it can see client applications. For both

servers to become active, we need a network configuration in which the servers cannot see each other, but each server can see a group of client applications.

A typical scenario would a HA pair distributed between two buildings, where each building also had a set of applications, and there was a single network link between both buildings. Breaking this link would create two sets of client applications, each with half of the HA pair, and each HA peer would become active.

To prevent split-brain situations, we must connect HA peers using a dedicated network link, which can be as simple as plugging them both into the same switch.

We must not split a HA pair into two islands, each with a set of applications. While this may be a common type of network architecture, we use partitioning, not high-availability, in such cases.

3.4.9 Client-side Semantics

Our goal with high-availability is to make the client semantics very simple:

1. Client applications should know both HA peer addresses. In the client APIs we allow the "hostname" to be specified as a list. This list would contain two server names/addresses.
2. The client API will attempt to connect to either of these, in an undefined order.
3. The server will either hold the connection (if the server is pending), accept it (if the server is active, or the connection is 'special'), or reject it (if the server is passive).
4. If the client cannot connect to the first server, it will attempt to connect to the second.
5. The client may, optionally, retry this connection sequence, with appropriate pauses, before abandoning.

Note that we do NOT use the AMQP Connection.Redirect semantics, and we do not provide the client with alternative hosts to connect to (the known-hosts field of the Connection.Open method). Both these semantics have proven complex and error-prone in the past, and today appear to be unnecessary.

3.5 Message Transfer Agent (MTA)

3.5.1 Requirements

We can list our requirements for message transfer between partitions:

1. Message transfer must work equally well over slow WAN connections as fast LAN connections. That is, message traffic must be optimised so that a message destined for multiple recipients at a node is sent only once to that node.
2. Message transfer must be easy to configure, operate, and debug.
3. We must support a distributed publish-subscribe fanout model in which data streams are distributed from a central source to a distributed network of nodes.
4. We must also support a distributed service request-response model in which services and the applications which use them can be put onto different partitions.
5. It must be possible to configure a working message transfer scenario from a single node, i.e. without no configuration changes on the target node.
6. We want to be able to create multiple independent message routing flows, so that different application architectures can be implemented within a single OpenAMQ server.

3.5.2 General MTA Design

The general MTA design is this:

- An MTA is attached to an exchange. An MTA instance always operates on a single, well-specified exchange.
- An MTA pulls or pushes messages from/to an eopnymous exchange on another partition.
- MTAs are configured objects that have the same lifespan as the exchanges they operate on.

3.5.3 Message "Pull" Processing

The message pull functionality is implemented as follows:

- The MTA connects to the specified partition using a peering.
- The MTA (or the peering, on behalf of the MTA) creates a temporary private queue and consumes off this queue.
- When the exchange receives a new binding request (from an application), it passes this to the MTA, which makes an identical binding request on the remote partition, for the MTA's private queue.
- The MTA thus receives all messages at the remote partition which correspond to the binding criteria that were specified.
- When the exchange destroys a binding, it also tells the MTA. Note: AMQP does not currently have any semantics for unbinding a queue, so any current MTA implementation will gradually become inefficient, with old unused bindings.
- Incoming messages are published to the local exchange and thus to any local queues that requested them.

3.5.4 Message "Push" Processing

The message push functionality is implemented as follows:

- The MTA connects to the specified partition using a peering.
- The MTA tells the local exchange under what conditions it wants to receive messages.
- When the exchange passes a message to the MTA, the MTA publishes it on the remote partition.

The conditions for message forwarding are:

1. Forward all messages, but process local bindings additionally.
2. Forward only messages that do not match any local bindings.

We assume that the connection between partitions is perhaps slow, but reliable. That is, we do not attempt to queue and forward messages in case of a network failure - messages will simply be dropped.

3.5.5 Current settings for MTA

Currently we have three properties used to specify MTA's behaviour:

1. forward-mode - possible values are 'all' (forward all messages) or 'none' (don't forward messages)
2. consume-mode - possible values are 'all' (pull all messages) or 'none' (don't pull messages)
3. copy - possible values are '1' (send published messages to local consumers as well as to remote server) or '0' (send published messages to remote server only)

3.6 Global Routing

3.6.1 Routing Keys

To enable request-response across partitions, using message transfer agents, we must ensure that reply-to values are global. We do not specifically need service names to be global although this may be useful at some stage.

AMQP routing semantics are based primarily on a "routing key". To make global routing work, we need to ensure that routing keys are unique across all interconnected servers.

The simplest way to create global routing keys is to use a naming convention that includes unique server-dependent information. We propose a convention that has been well-proven in the Internet domain:

```
queuename@hostname
```

To implement global routing, we must:

1. Ensure that every server is properly configured with a unique identity.
2. Use the server identity when naming temporary private queues (which are the basis for replies).

3.6.2 Default Routes

Although we do not have a concrete use case for this, we are analysing the use of default routes for messages, based on the routing keys that follow a systematic queuename@hostname format. This would allow automatic message transfer without the need for manual MTA configuration for each message stream.

4 Configuration and Operation

We look at how the clustering and inter-clustering functions explained in this document are configured for real scenarios.

4.1 HAC Configuration

The configuration data for a HA pair must specify:

- The primary and backup server identities (their 'names').
- For each of these two servers, one or more internal network addresses, used for connections between servers.

Operation of the HA pair is as follows:

- When a server process is started, we must tell it what configuration data to load, and whether or not it is half of a HA pair. A server must know whether it is the primary or backup server.
- The HAC in a particular server can be interrogated via the console, started and stopped. This is also how we test the failover and recovery mechanisms without stopping and starting processes.

4.2 MTA Configuration

The configuration for an MTA needs to specify:

- Authentication information on the central server(s) so that the remote server can connect.
- On the remote server(s), the name of each exchange to operate on, and the central server to federate it with.
- For each MTA, whether or not to pull messages.
- For each MT, whether or not to push messages, and under what conditions.