The implementation of the Notification Channel Replacement from CORBA to ActiveMQ as an underliying technology requiered many aspects to be considered. Hereunder are the details of said implementation.

There are two versions of ActiveMQ: Classic and Artemis. The chosen version for the implementation of the project was Artemis, mainly because it is the newest version and it is preferred to always work with the newest technology.

Main details about ActiveMQ Artemis:

- It is a high-performance, non-blocking architecture for the next generation of event-driven messaging applications.
- High availability using shared storage or network replication
- Simple & powerful protocol agnostic addressing model
- 100% open source software.
- Apache ActiveMQ Artemis is designed with usability in mind.
- Written in Java, and compatible with many other different languages such as C++ and Python.
- Amazing performance.
- ActiveMQ Artemis can be ran stand-alone, integrated in a JEE application server, or embedded inside another product.
- High availability solution with automatic client failover so zero message loss or duplication in event of server failure can be guaranteed.

More important information about ArtemisMQ, relevant to this project can be found here.

Implemented in C++, using the Qpid Proton

involved a great amount of knowledge about the broker used and the C++ implementation

ActiveMQ Artemis

[Insertar documentación de Artemis acá]

Apache ActiveMQ Artemis implements the AMQP 1.0 specification. Any client that supports the 1.0 specification will be able to interact with Apache ActiveMQ Artemis.

IP:

As a default, on the Linux platform, if you have IPV6 support enabled in your kernel, the Java Runtime (since 1.4 version) will use IPV6 sockets to communicate. That's perfectly fine, except that your network may not be configured for IPV6. So everything will be dropped…and nothing will ever be received on the consumer side (which maybe even does not support IPV6).

There's a solution to this problem: Force IPV4 stack to be used by the Java Runtime. This can be done using a system property:

-Djava.net.preferIPv4Stack=true

IMPORTANT:

When creating the broker, this must be done using:

/bin/artemis create --allow-anonymous --silent --force --user almamgr --password alma123 --port-offset 0 --data ./data --allow-anonymous --no-autotune --verbose --aio --java-options -Djava.net.preferIPv4Stack=true {target_dir}/mybroker

Update:

${ARTEMIS_HOME}/artemis create --autocreate --default-port 5672 --java-options -Djava.net.preferIPv4Stack=true --user almamgr --password alma123 --port-offset 0 --require-login --verbose --force {target_dir}/almabroker

And this will create a broker that uses IPV4.

It is important for the "auto-create" to be allowed, so that queues can be automatically created using AMQP links.

In the case of the broker created for this project, it is called "mybroker2".

Clusters:

Apache ActiveMQ Artemis provides very configurable state-of-the-art clustering model where messages can be intelligently load balanced between the servers in the cluster, according to the number of consumers on each node, and whether they are ready for messages.

Apache ActiveMQ Artemis also has the ability to automatically redistribute messages between nodes of a cluster to prevent starvation on any particular node.

Bridges and routing:

Apache ActiveMQ Artemis bridges can be configured with filter expressions to only forward certain messages, and transformation can also be hooked in.

Apache ActiveMQ Artemis also allows routing between queues to be configured in server side configuration. This allows complex routing networks to be set up forwarding or copying messages from one destination to another, forming a global network of interconnected brokers.

Server JVM Settings

The run scripts set some JVM settings for tuning the garbage collection policy and heap size. We recommend using a parallel garbage collection algorithm to smooth out latency and minimise large GC pauses.

By default Apache ActiveMQ Artemis runs in a maximum of 1GiB of RAM. To increase the memory settings change the -Xms and -Xmx memory settings as you would for any Java program.

Broker configuration file

The configuration for the Apache ActiveMQ Artemis core server is contained in broker.xml. This is what the FileConfiguration bean uses to configure the messaging server.

Address Model

Addressing Model

Apache ActiveMQ Artemis has a unique addressing model that is both powerful and flexible and that offers great performance. The addressing model comprises three main concepts: addresses, queues, and routing types.

Address

An address represents a messaging endpoint. Within the configuration, a typical address is given a unique name, 0 or more queues, and a routing type.

Queue

A queue is associated with an address. There can be multiple queues per address. Once an incoming message is matched to an address, the message will be sent on to one or more of its queues, depending on the routing type configured. Queues can be configured to be automatically created and deleted.

Routing Types

A routing type determines how messages are sent to the queues associated with an address. An Apache ActiveMQ Artemis address can be configured with two different routing types.

Table 1. Routing Types

If you want your messages routed to... Use this routing type...

A single queue within the matching address, in a point-to-point manner. Anycast

Every queue within the matching address, in a publish-subscribe manner. Multicast

By default is Anycast

The following has to be added to the etc/broker.xml file of the broker created:

```
<address-setting match="/#">

    <auto-create-addresses>true</auto-create-addresses>
    <auto-delete-addresses>true</auto-delete-addresses>
    <default-address-routing-type>MULTICAST</default-address-routing-type>

</address-setting>
```

AMPQ Clients

QPid Proton:

Java JMS 1.1 Client (Apache QPid JMS based on Qpid Proton)

Reactive C++ Client (Apache Qpid Proton)

Reactive Python Client (Apache Qpid Proton)

According to the Requirements

#
Title
User Story
Importance
Status

Notes

1

API must be available in C++, Java and Python

Developers use those three languages to implement the different parts of ALMA software, therefore the API must be provided in those three languages

Must Have

qpid supports all 3 languages

2

The Notification Channel API must hide as much as possible of the underlying technology

The API must be common for all the technologies used to implement this API. This includes CORBA at the beginning

Must have

3

The possibility of setting the quality of service and administrative properties of channels must exist.

Each channel has different QoS requirements based on the role they play in whole ALMA system. The user must be able to set this up using the Configuration Database

Must Have

4

The API must be implemented in a high-performance manner to reduce the chances of events being discarded, if the QoS settings are set properly.

Each client (publisher or subscriber) must perform accordingly to prevent slow participants. The sending and reception of events should be done as quickly as possible. Each client consuming events from the API is responsible to handle each event timely, otherwise events could be lost.

Must Have

5

Notification Channel must decouple publishers from subscribers.

Must Have

6

Notification channel shall deliver events based on best-effort or reliable way. Depending of the QoS configuration.

Must Have

7
Events should be delivered to consumers in a timely manner
The events must be sent and received with the lowest latency possible. Acceptable values are in order of 100 [ms]
Must Have

8
Notification channel must support re-connection of clients
If the Event Channel goes down, the Channel must be re-created automatically. All the clients connected to the Event Channel shall reconnect to it.
Must Have

9
Notification Channels must provide introspection
The services associated, the channels and the clients must offer operational performance parameters introspection, in a way they can be monitored
Desirable

10
Notification channel should be fault tolerant
Problems in any network path should not affect communications that do not involve that path Events should be queued, up to some limit, in network paths that do have a problem. Data transmission between a publisher and consumer should not require the data be staged in a third place, like a central server.
Must have

11
Maintain event order
Events between the same publisher and consumer should be delivered in the same order
Must have

Original Classes and Changes Made:

include

ConsumerTimer.h
?

File CorbaNotifyUtils.h

CORBA Erase (?

File ORBTask.h

CORBA Erase (?

File pDataConsumer.h

add private methods for management of qpid CosNotifyChannelAdmin change

File pDataSupplier.h

add private methods for management of qpid CosNotifyChannelAdmin change

File QoSProps.h

none yet

File SupplierTimer.h

?

File TimevalUtils.h

none

scr