Project acronym:	**PrEstoCloud**
Project full name:	**Proactive Cloud Resources Management at the Edge for efficient Real-Time Big Data Processing**
Grant agreement number:	**732339**

D3.1 Communication Broker: Iteration 1

Deliverable Editor:	**Nenad Stojanovic (Nissatech)**
Other contributors:	
Deliverable Reviewers:	**CNRS and ICCS**
Deliverable due date:	**31/12/2017**
Submission date:	**31/03/2018**
Distribution level:	**Public**
Version:	**1.0**

This document is part of a research project funded by the Horizon 2020 Framework Programme of the European Union

Change Log

Version	Date	Amended by	Changes
0.1	18.12.2017	Nissatech	ToC
0.2	31.01.2018	Nissatech	Section 2
0.3	15.02.2018	Nissatech	Section 3, Appendix 1 and 2
0.4	28.02.2018	Nissatech	Section 4, Appendix 3
0.5	09.03.2018	Nissatech	Section 5
0.6	26.03.2018	Nissatech	Executive summary, Section 1, Ready for review
1.0	31.03.2018	Nissatech	Final version

Table of Contents

List of Figures

List of Abbreviations

AMQP	Advanced Message Queuing Protocol
CA	Certificate Authority
CLI	Command-line Interface
CPU	Central Processing Unit
GUI	Graphical User Interface
JAAS	Java Authentication and Authorization Service
JDBC	Java Database Connectivity
JMS	Java Message Service
JMX	Java Management Extensions
JSON	JavaScript Object Notation
HTTP	Hypertext Transfer Protocol
MOM	Message-Oriented Middleware
MQTT	Message Queuing Telemetry Transport
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
RAM	Random Access Memory
REST	Representational state transfer
RPC	Remote Procedure Call
RSS	Rich Site Summary
SAN	Storage Area Network
SASL	Simple Authentication and Security Layer
SHA	Secure Hash Algorithms
SSL	Secure Sockets Layer
STOMP	Simple (or Streaming) Text Orientated Messaging Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
WAN	Wide Area Network
WSIF	Web Services Invocation Framework
XMPP	Extensible Messaging Presence Protocol

Executive Summary

This deliverable is a result of a PrEstoCloud project task T3.1, "Communication broker for real-time data streams".

We started the work by comparing ActiveMQ and RabbitMQ. We analyzed several parameters like:

- scalability types (ways to distribute broker)
- security support (TLS and access)
- cross protocol communication (MQTT and AMQP)
- programming languages support
- monitoring the work of broker and error reporting
- cloud infrastructure providers
- RPC and tutorials for installation and basic usage.

We concluded that RabbitMQ and ActiveMQ have almost same features implemented in different ways. We choose RabbitMQ mainly because of the following:

- RabbitMQ, unlike ActiveMQ, use AMQP by default, so it has support for converting AMQP to MQTT and vice versa
- RabbitMQ has much more documents, materials, examples, etc. to look for and the community of RabbitMQ is larger
- Availability of an official docker image of RabbitMQ

The next step was to make research about the full set of functionalities provided by RabbitMQ. The lessons learnt from this research were used to make decisions how to implement and configure the PrEstoCloud broker. Here are some examples of these decisions:

- in our broker we use our TOPIC exchange (presto.cloud) with multiple queues;
- our presto.cloud exchange is durable, no auto-delete and external. This is set in definition.json files. Beside this exchange we use multiple queues;
- we also have implemented automatic recovery from network failures, etc.

After this research, we have developed the PrEstoCloud broker with all needed functionality. First, we started by implementing RabbitMQ on local machine and testing basic functionality and plugins. We added management and learnt how to use it. After that we added MQTT plugin and implemented java libraries for communication with broker. We used RabbitMQ amqp-client 5.2.0 for consumer and eclipse.paho.client.mqttv3 1.2.0 for producer.

Thereafter we started to use Docker and put broker in container. Because of distribution part we created two node of broker on different ports and connected them with federation. In the end we added autorecovery and started some performance testing of pilot federated broker in order to make broker stable and to check if it delivers all messages. This help us to remove some bugs and to have stable version.

Next installation guide is optimized in way we have much preparation files and less commands.

Finally, we have created the maven libraries that can be used for communication with the broker.

The main conclusion is that the selected broker can satisfy the functional requirements and can be efficiently implemented.

1. Introduction

1.1 Scope and context of the document

This deliverable reflects a work performed on WP3/Task 3.1, "Communication broker for real-time data streams". The objective is to set up a Communication Broker layer that will undertake the responsibility of relaying data streams on and off the PrEstoCloud platform. This document provides the results of the 1st iteration cycle. The 2nd version will be delivered in M30.

The main objective of this deliverable is to develop a 1st version of the communication broker. The consortium iteratively converged into the 1st release of the communication broker. We started from an analysis of the most widely used brokers and selected RabbitMQ as a basis for PrEstoCloud. During the iterative process of understanding functionalities provided by RabbitMQ and objectives and requirements of the PrEStoCloud platform, the broker was conceptualized, developed, tested, refined and improved.

The primary inputs for this deliverable are results from Task 3.1. Additionally, in order to develop the broker, different deliverables have been taken into consideration. In particular, the development of the broker has considered inputs from D2.2, D2.3 and D2.4.

Although in the list of requirements provided in the deliverable D2.2 there was only one requirement related to the communication broker (FR-41 Ability to send / retrieve events to / from the Communications Broker), during the work on the development of the broker we added two additional requirements based on the detailed specification of use cases (D7.2):

- supporting the heterogeneity in the communication (edge devices – cloud services);
- supporting the federation of the brokers in order to support complex communication topologies.

1.2 Structure of the document

In section 2 we present the results of the comparison of RabbitMQ and ActiveMQ and justify the decision that we made.

Chapter 3 contains documentation part that helped us with making decisions and implementation of broker.

In chapter 4, we explain how we created broker with all needed functionality.

In Chapter 5 we describe the usage of the broker in the PrEstoCloud.

The deliverable includes several appendixes:

Appendix 1 is about RabbitMQ.

In Appendix 2 the detailed information about the distributed RabbitMQ is provided.

Appendix 3 summarizes steps to be done to install and use RabbitMQ on Ubuntu.

2. Comparison of ActiveMQ and RabbitMQ

In this section we compare Activisms [1] and RabbitMQ [11]. We do not consider Kafka[1] due to issues with scalability in WAN environment. Official Kafka documentation [25] states:

> *„It is generally not advisable to run a single Kafka cluster that spans multiple datacenters as this will incur very high replication latency both for Kafka writes and Zookeeper writes and neither Kafka nor Zookeeper will remain available if the network partitions."*

We have defined two phases for attaining knowledge and practical skills in order to become familiar with RabbitMQ and successfully implement large scale federation which will meet project requirements - research phase and experimentation phase.

- *Research phase* - based on project requirements, we have identified following research topics and questions about MQTT broker before proceeding to experimentation phase:
  1. Scalability types – federation, shovel, cluster in WAN, LAN context //Ways to distribute broker
  2. Security, Access lists in distributed environment
  3. Cross Protocol Communication (MQTT, AMQP)
  4. Support from various programming languages (Client libraries)
  5. Monitoring broker health, error reporting (GUI tools preferred)
  6. Support by cloud infrastructure providers
  7. Remote Procedure Call (RPC)

- *Experimentation phase* - the goal of experimentation is to set up broker federation in local environment and try out basic functionalities examined during the research phase. Defined experiments are:
  1. Setting broker federation to use combination of federated and un-federated topics;
  2. Setting up cluster of multiple brokers in local environment and try to include it into the federation from experiment 1;
  3. Setup security using TLS protocol;
  4. Developing Java Client library to make it possible for partners to start experimenting with broker.

In the rest of this section we compare ActiveMQ and RabbitMQ with respect to previously defined criteria. For each criterion (at the end of the corresponding sub-section), we clarify the decision(s) taken for the implementation of the PrEstoCloud broker. In the next section we explain what and how has been implemented.

2.1 Criterion 1 - Comparison considering scalability issues

2.1.1 ActiveMQ & criterion 1

The content of this section is based on the following references: [7], [8], [10].

There are various topologies that you can employ with ActiveMQ, where clients are connected to message brokers in various ways like:

- peer based;

[1] https://kafka.apache.org/

- client server;
- hub and spoke.

To create distributed queues or topics we need to have the message brokers communicate with each other.

Clustering is a large topic and often means different things to different people. Here is the list of clustering that ActiveMQ support:

- MasterSlave
  - JDBC Master Slave
  - KahaDB Replication (Experimental)
  - Pure Master Slave
  - Shared File System Master Slave
- Networks of Brokers
- Replicated Message Store

Master/Slave for High Availability

Basically it means that all messages are replicated across each broker in the master/slave cluster.

Master/Slave works by having some form of replication; each message is owned by every broker in the logical cluster. A master/slave cluster then acts as one logical message broker which could then be connected via store and forward to other brokers.

In Master/Slave, queues and topics are all replicated between each broker in the cluster. So each broker in the cluster has exactly the same messages available at any time so if a master fails, clients failover to a slave and you don't lose a message.

Master Slave Type	Requirements	Pros	Cons
Shared File System Master Slave	A shared file system such as a SAN	Run as many slaves as required automatic recovery of old masters	Requires shared file system
JDBC Master Slave	A Shared database	Run as many slaves as required automatic recovery of old masters	Requires a shared database. Also relatively slow as it cannot use the high performance journal

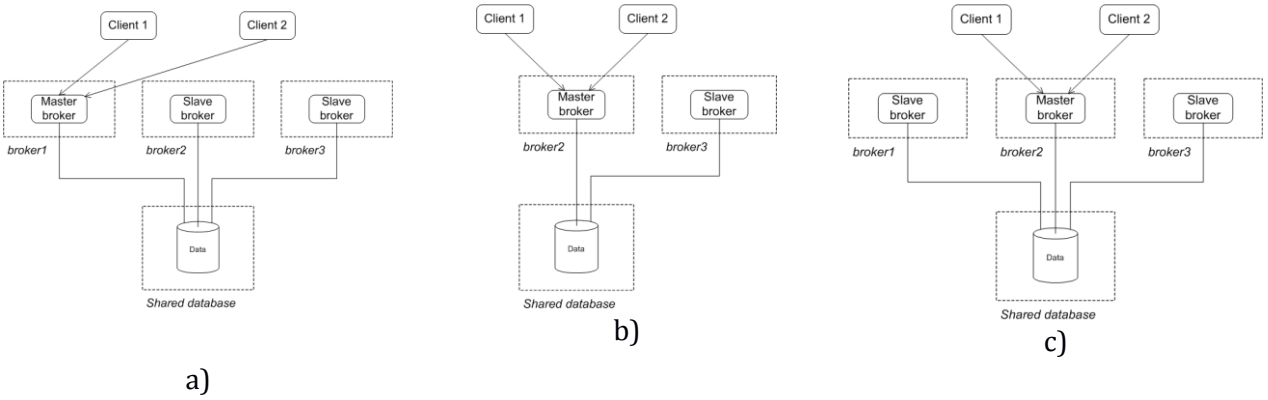

Figure 1: JDBC Master/Slave[2]: a) Initial State b) after Master Failure c) after Master Restart

Store and forward - networks of brokers

It means the messages travel from broker to broker until they reach a consumer; with each message being owned by a single broker at any point in time.

ActiveMQ uses consumer priority, this means that local JMS (Java Message Service) consumers are always higher priority than remote brokers in a store and forward network.

When we publish a message on a queue, it is stored in the persistent store of the broker that the publisher is communicating. Then if that broker is configured to store/forward to other brokers and clients, the broker will send it to one of these clients. This dispatch algorithm continues until the message is finally dispatched and consumed by a client.

At any point in time the message will only exist in one broker's store until it is consumed. Note that messages are only distributed onto other brokers if there is a consumer on those brokers.

For topics the above algorithm is followed except, every interested client receives a copy of the message - plus ActiveMQ will check for loops (to avoid a message flowing infinitely around a ring of brokers).

Replicated Message Stores

It can reduce the risk of message loss to provide either a High Availability backup or a full Disaster Recovery solution capable of surviving a data centre failure.

2.1.2 RabbitMQ & criterion 1

The content of this section is based on [14].

There are three ways to make the RabbitMQ broker itself distributed: with clustering, with federation, and using the shovel. There is no need to pick a single approach - you can connect clusters together with federation, or the shovel, or both.

Connecting brokers with the shovel is conceptually similar to connecting them with federation (the shovel works at a lower level), because of that we has next comparison.

Federation / Shovel	Clustering

[2] https://access.redhat.com/documentation/en-US/Fuse_ESB_Enterprise/7.1/html/Fault_Tolerant_Messaging/files/FMQMasterSlaveJDBC.html

Brokers are logically separate and may have different owners.	A cluster forms a single logical broker.
Brokers can run different versions of RabbitMQ and Erlang[3].	Nodes must run the same version of RabbitMQ, and frequently Erlang.
Brokers can be connected via unreliable WAN links. Communication is via AMQP (optionally secured by TLS), requiring appropriate users and permissions to be set up.	Brokers must be connected via reliable LAN links. Communication is via Erlang internode messaging, requiring a shared Erlang cookie.
Brokers can be connected in whatever topology you arrange. Links can be one- or two-way.	All nodes connect to all other nodes in both directions.
Chooses Availability and Partition Tolerance from the CAP theorem[4].	Chooses Consistency and Partition Tolerance from the CAP theorem.
Some exchanges in a broker may be federated while some may be local.	Clustering is all-or-nothing.
A client connecting to any broker can only see queues in that broker.	A client connecting to any node can see queues on all nodes.

2.2 Criterion 2 - Comparison considering security, access lists in distributed environment

2.2.1 ActiveMQ & criterion 2

The content of this section is based on [2]. ActiveMQ security guide can be found at [22].

ActiveMQ 4.x and greater provides pluggable security through various different providers.

The most common providers are:

- JAAS for authentication
- a default authorization mechanism using a simple XML configuration file.

Simple Authentication Plugin with Anonymous access

```
<simpleAuthenticationPlugin anonymousAccessAllowed="true">
 <users>
  <authenticationUser username="system"
  password="manager" groups="users,admins"/>
  <authenticationUser username="user"
  password="password" groups="users"/>
  <authenticationUser username="guest"
  password="password" groups="guests"/>
 </users>
</simpleAuthenticationPlugin>
```

[3] https://www.erlang.org/

[4] https://en.wikipedia.org/wiki/CAP_theorem

Authorization

In ActiveMQ we use a number of operations which can be associated with user roles and either individual queues or topics. Additionally, wildcards can be used to attach to hierarchies of topics and queues.

- Read - You can browse and consume from the destination
- Write - You can send messages to the destination
- Admin - You can lazily create the destination if it does not yet exist. This allows you fine grained control over which new destinations can be dynamically created in what part of the queue/topic hierarchy

Queues/Topics can specified using the ActiveMQ Wildcards syntax.

Based on [5]: ActiveMQ Artemis version allows sets of permissions to be defined against the queues based on their address. An exact match on the address can be used or a wildcard match can be used using the wildcard characters '#' and '*'.

Seven different permissions can be given to the set of queues which match the address. Those permissions are:

- createDurableQueue - This permission allows the user to create a durable queue under matching addresses.
- deleteDurableQueue - This permission allows the user to delete a durable queue under matching addresses.
- createNonDurableQueue - This permission allows the user to create a non-durable queue under matching addresses.
- deleteNonDurableQueue - This permission allows the user to delete a non-durable queue under matching addresses.
- send - This permission allows the user to send a message to matching addresses.
- consume - This permission allows the user to consume a message from a queue bound to matching addresses.
- manage - This permission allows the user to invoke management operations by sending management messages to the management address.

http://activemq.apache.org/how-do-durable-queues-and-topics-work.html
For each permission, a list of roles who are granted that permission is specified. If the user has any of those roles, he/she will be granted that permission for that set of addresses.

Let's take a simple example:

```
<security-setting match="globalqueues.europe.#">
   <permission type="createDurableQueue" roles="admins"/>
   <permission type="deleteDurableQueue" roles="admins"/>
   <permission type="createNonDurableQueue" roles="admins, guests, users"/>
   <permission type="deleteNonDurableQueue" roles="admins, guests, users"/>
   <permission type="send" roles="admins, users"/>
   <permission type="consume" roles="admins, users"/>
</security-setting>
```

More information can be found at: http://activemq.apache.org/how-do-durable-queues-and-topics-work.html

TLS transport

When messaging clients are connected to servers, or servers are connected to other servers (e.g. via bridges) over an untrusted network then ActiveMQ allows that traffic to be encrypted using the Transport Layer Security (TLS) transport.

Basic user credentials

ActiveMQ ships with a security manager implementation that reads user credentials, i.e. user names, passwords and groups information from properties files on the classpath called users.properties and groups.properties. This is the default security manager.

users.properties file is basically just a set of key value pairs that define the users and their password, like so:

```
system=manager
user=password
guest=password
```

groups.properties defines what groups these users belong too where the key is the groupname and the value is a comma-seperated list of users, like so:

```
admins=system
users=system,user
guests=guest
```

If you wish to use this security manager, then users, passwords and groups can easily be added into these files.

In order for cluster connections to work correctly, each node in the cluster must make connections to the other nodes. The username/password they use for this should always be changed from the installation default to prevent a security risk.

Authentication Example

- login.config http://svn.apache.org/repos/asf/activemq/trunk/activemq-unit-tests/src/test/resources/login.config
- users.properties http://svn.apache.org/repos/asf/activemq/trunk/activemq-unit-tests/src/test/resources/org/apache/activemq/security/users.properties
- groups.properties http://svn.apache.org/repos/asf/activemq/trunk/activemq-unit-tests/src/test/resources/org/apache/activemq/security/groups.properties

2.2.2 RabbitMQ & criterion 2

The content of this section is based on [12] and [18],

Access Control

Default Virtual Host and User

When the server first starts running, and detects that its database is uninitialized or has been deleted, it initializes a fresh database with the following resources:

- a virtual host named /
- a user named guest with a default password of guest, granted full access to the / virtual host.

A "guest" user can only connect via localhost.

When a RabbitMQ client establishes a connection to a server, it specifies a virtual host within which it intends to operate. A first level of access control is enforced at this point, with the server checking whether the user has any permissions to access the virtual hosts, and rejecting the connection attempt otherwise.

RabbitMQ distinguishes between configure, write and read operations on a resource. The configure

operations create or destroy resources, or alter their behavior. The write operations inject messages into a resource. And the read operations retrieve messages from a resource.

In order to perform an operation on a resource the user must have been granted the appropriate permissions for it.

Topic Authorisation

As of version 3.7.0, RabbitMQ supports topic authorisation for topic exchanges. Topic authorisation targets protocols like STOMP and MQTT, which are structured around topics and use topic exchanges under the hood.

Topic authorisation is an additional layer on top of existing checks for publishers. Publishing a message to a topic-typed exchange will go through both the basic.publish and the routing key checks. The latter is never called if the former refuses access.

Topic authorisation can also be enforced for topic consumers. Note that it works different for different protocols.

Internal (default) authorisation backend supports variable expansion in permission patterns. Three variables are supported: username, vhost, and client_id. Note that client_id only applies to MQTT. For example, if tonyg is the connected user, the permission ^{username}-.* is expanded to ^tonyg-.*

If a different authorisation backend (e.g. LDAP, HTTP, AMQP) is used, please refer to the documentation of those backends.

If a custom authorisation backend is used, topic authorisation is enforced by implementing the check_topic_access callback of the rabbit_authz_backend behavior.

Alternative Authentication and Authorisation Backends

Authentication and authorisation are pluggable. Plugins can provide implementations of:

- authentication ("authn") backends
- authorisation ("authz") backends

It is possible for a plugin to provide both. For example the internal, LDAP and HTTP backends do so.

Some plugins, for example, the Source IP range one[5], only provide an authorisation backend. Authentication is supposed to be handled by the internal database, LDAP, etc.

Combining Backends

It is possible to use multiple backends for authn or authz using the auth_backends configuration key. When several authentication backends are used then the first positive result returned by a backend in the chain is considered to be final. This should not be confused with mixed backends (for example, using LDAP for authentication and internal backend for authorisation).

Authentication

RabbitMQ has pluggable support for various Simple Authentication and Security Layer (SASL) authentication mechanisms. There are three such mechanisms built into the server: PLAIN, AMQPLAIN, and RABBIT-CR-DEMO, and one - EXTERNAL - available as a plugin. You can also implement your own authentication mechanism by implementing the rabbit_auth_mechanism behaviour in a plugin.

[5] https://github.com/gotthardp/rabbitmq-auth-backend-ip-range

The built-in mechanisms are:

- PLAIN - SASL PLAIN authentication. This is enabled by default in the RabbitMQ server and clients, and is the default for most other clients.
- AMQPLAIN - Non-standard version of PLAIN as defined by the AMQP 0-8 specification. This is enabled by default in the RabbitMQ server, and is the default for QPid's Python client.
- EXTERNAL - Authentication happens using an out-of-band mechanism such as x509 certificate peer verification, client IP address range, or similar. Such mechanisms are usually provided by RabbitMQ plugins.
- RABBIT-CR-DEMO - Non-standard mechanism which demonstrates challenge-response authentication. This mechanism has security equivalent to PLAIN, and is not enabled by default in the RabbitMQ server.

Per AMQP 0-9-1 spec, authentication failures should result in the server closing TCP connection immediately. However, with RabbitMQ clients can opt in to receive a more specific notification using the authentication failure notification extension to AMQP 0-9-1.

Credentials and Passwords

RabbitMQ supports multiple authentication mechanisms. Some of them use username/password pairs. These credential pairs are then handed over to an authentication backends that perform authentication. One of the backends, known as internal or built-in, uses internal RabbitMQ data store to store user credentials. When a new user is added using rabbitmqctl, her password is combined with a salt value and hashed.

As of version 3.6.0, RabbitMQ can be configured to use several password hashing functions:

- SHA-256
- SHA-512
- MD5[6] (only for backwards compatibility)
- SHA-256 is used by default. More algorithms can be provided by plugins.

Credential Validation

Starting with version 3.6.7 it is possible to define a credential validator. It only has effect on the internal authentication backend and kicks in when a new user is added or password of an existing user is changed.

Validators are modules that implement a validation function. To use a validator, it is necessary to specify it and its additional settings in the config file. There are three credential validators available out of the box:

- rabbit_credential_validator_accept_everything: unconditionally accepts all values. This validator is used by default for backwards compatibility.
- rabbit_credential_validator_min_password_length: validates password length
- rabbit_credential_validator_password_regexp: validates that password matches a regular expression

Custom Credential Validators

Every credential validator is a module that implements a single function behaviour, rabbit_credential_validator. Plugins therefore can provide more implementations.

6 https://en.wikipedia.org/wiki/MD5

Credential validators can also validate usernames or apply any other logic (e.g. make sure that provided username and password are not identical).

Passwordless Users

Internal authentication backend allows for users without a password or with a blank one.

Authentication Using TLS (x509) Certificates

It is possible to authenticate connections using x509[7] certificates and avoid using passwords entirely. The authentication process then will rely on TLS peer certificate chain validation.

2.3 Criterion 3 - Comparison considering cross protocol communication

2.3.1 ActiveMQ & criterion 3

The content of this section is based on [6].

ActiveMQ is a message broker which supports multiple wire level protocols for maximum interoperability.

- AMQP
- AUTO
- MQTT
- OpenWire
- REST
- RSS and Atom
- Stomp
- WSIF
- WS Notification
- XMPP

There is no clear way to convert AMQP to MQTT or vice versa.

2.3.2 RabbitMQ & criterion 3

The content of this section is based on [13].

RabbitMQ supports several messaging protocols, directly and through the use of plugins. The supported protocols are:

- AMQP 0-9-1, 0-9 and 0-8, and extensions
- STOMP
- MQTT
- AMQP 1.0
- HTTP

More information can be found at:

- https://www.rabbitmq.com/mqtt.html#overview
- https://blogs.sap.com/2016/02/21/uniting-amqp-and-mqtt-message-brokering-with-rabbitmq/

[7] https://en.wikipedia.org/wiki/X.509

2.4 Criterion 4 - Comparison considering support from various programming languages

2.4.1 ActiveMQ & criterion 4

The content of this section is based on [9].

ActiveMQ is a message broker written in Java with JMS, REST and WebSocket interfaces, however it supports protocols like AMQP, MQTT, OpenWire and STOMP that can be used by applications in different languages.

The following client libraries are supported:

- .NET
- C (defunct)
- C++
- Erlang
- Go
- Haskell
- Haxe (defunct)
- Jekejeke Prolog
- NetLogo
- Node.js
- Perl 5
- Pike
- Python
- Racket
- Ruby on Rails
- Tcl/Tk

2.4.2 RabbitMQ & criterion 4

The content of this section is based on [15].

RabbitMQ is officially supported on a number of operating systems and several languages. In addition, the RabbitMQ community has created numerous clients, adaptors and tools. Some of them are mentioned here:

- Java and Spring
- .NET
- Ruby
- Python
- PHP
- Objective-C and Swift
- Other JVM Languages
    - Scala
    - Groovy and Grails
    - Clojure
    - JRuby
- JS
- C/C++
- GO
- Unity 3D

2.5 Criterion 5 - Comparison considering monitoring broker health, error reporting

2.5.1 ActiveMQ & criterion 5

The content of this section is based on [4].

You can monitor ActiveMQ using the Web Console by pointing your browser at:

http://localhost:8161/admin

We note here that only works if you open the browser from the same machine that the broker is installed on.

The other possibility is to use the JMX[8] support to view the running state of ActiveMQ. For more information see the file docs/WebConsole-README.txt in the distribution.

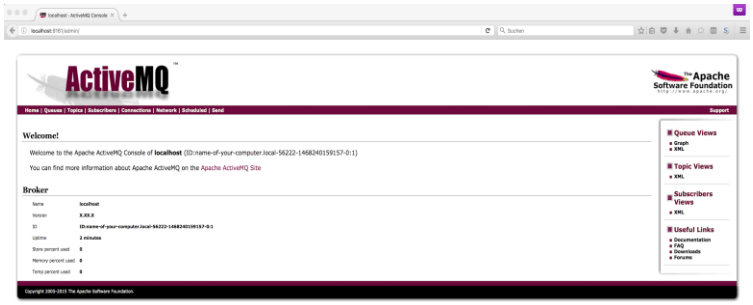

Figure 2: UI of ActiveMQ web console

ActiveMQ web console has next tabs: Home, Queues, Topics, Subscribers, Connections, Scheduled, Send.

2.5.2 RabbitMQ & criterion 5

The content of this section is based on [16] and [17].

The RabbitMQ management plugin provides a starting point for monitoring RabbitMQ metrics. One limitation, however, is that only up to one day's worth of metrics are stored. Storing historical metrics can be an important tool to determine the root cause of issues affecting your users or to plan for future capacity.

RabbitMQ metrics are made available through the HTTP API via the api/queues/vhost/qname endpoint. It is recommended to collect metrics at 60 second intervals because more frequent collection may place too much load on the RabbitMQ server and negatively affect performance.

RabbitMQ supported metrics are: Memory, Queued Messages, Un-acked Messages, Messages Published, Message Publish Rate, Messages Delivered, Message Delivery Rate… Other Message Stats.

The following is an alphabetised list of third-party tools to collect RabbitMQ metrics. These tools have the capability to monitor the recommended system and RabbitMQ metrics. Most of them are open source plugins from GitHub:

- AppDynamics
- collectd
- DataDog

8 http://activemq.apache.org/jmx.html

- Ganglia
- Graphite
- Munin
- Nagios
- New Relic
- Prometheus
- Zabbix
- Zenoss

The rabbitmq-management plugin provides an HTTP-based API for management and monitoring of a RabbitMQ server, along with a browser-based UI and a command line tool, rabbitmqadmin.

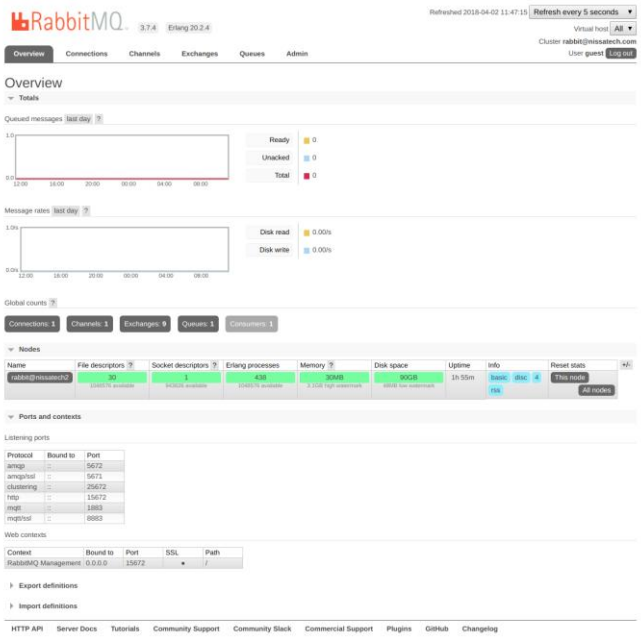

Figure 3: UI of RabbitMQ management

2.6 Criterion 6 - Comparison considering support by cloud infrastructure provider

Over Docker containerization both can be put on any cloud platform.

2.6.1 ActiveMQ & criterion 6

ActiveMQ does not have official image on Docker. There are some popular community images https://hub.docker.com/search/?q=activemq.

It can also be put on cloud over bitnami - https://bitnami.com/stack/activemq. Bitnami Cloud Images are currently available for Google Cloud Platform, Amazon Web Services, Oracle Cloud Infrastructure Classic, Microsoft Azure, CenturyLink, and 1&1 Cloud Platform.

2.6.2 RabbitMQ & criterion 6

The content of this section is based on [11], [23] and [24].

The following companies provide technical support and/or cloud hosting of open source RabbitMQ: Pivotal Software, CloudAMQP, Google Cloud Platform. RabbitMQ can also be deployed in AWS,

Microsoft Azure and Heroku (https://elements.heroku.com/addons/rabbitmq-bigwig).

RabbitMQ has official image on Docker hub - https://hub.docker.com/_/rabbitmq/.

Additionally, RabbitMQ also has bitnami support – https://bitnami.com/stack/rabbitmq**Fehler! Hyperlink-Referenz ungültig.**

2.7 Criterion 7 - Comparison considering RPC

The content of this section is based on [3].

2.7.1 Introduction

Unlike systems based on a RPC pattern, messaging systems primarily use an asynchronous message passing pattern with no tight relationship between requests and responses. Most messaging systems also support a request-response mode but this is not a primary feature of messaging systems.

Technologies using RPC (Remote Procedure Calls) are called tightly coupled distributed applications. Using RPC, one application can call the other application. There are many disadvantages of tightly coupled technologies, a higher maintenance cost being the most common. Another disadvantage is when one application calls another application through RPC, the other application must be available to receive the call or else the whole architecture fails. Figure 2 shows the architecture of two tightly coupled distributed applications.

Figure 4: Tightly coupled distributed applications

2.7.2 ActiveMQ & criterion 7

Information about implementing RPC with ActiveMQ request-response messages can be found at [21].

2.7.3 RabbitMQ & criterion 7

The content of this section is based on [19] and [20].

You can send request-response messages using the RabbitMQ transport by implementing a Remote Procedure Call (RPC) scenario with RabbitMQ.

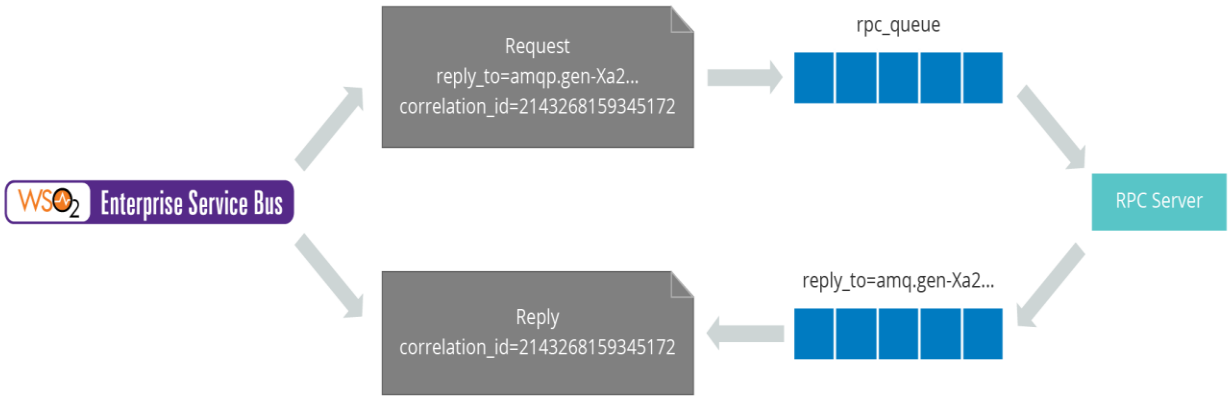

Figure 5: RabbitMQ and RPC

2.8 Conclusion

Like we see RabbitMQ and ActiveMQ have almost same features implemented in different ways. But we choose RabbitMQ mainly because of the following differences:

- Cross protocol communication
  RabbitMQ, unlike ActiveMQ, use AMQP by default, so it has support for converting AMQP to MQTT and vice versa. This is feasible with MQTT plugin - RabbitMQ MQTT Adapter. It makes it possible for MQTT clients to interoperate with AMQP 0-9-1, AMQP 1.0, and STOMP clients.
  ActiveMQ can convert AMQP to JMS, and MQTT protocol will automatically map between JMS and MQTT clients, but there is no concrete proof or example how easy implementation is.

- Documentation, materials and plugins
  During the research we find out that RabbitMQ has much more documents, materials, examples, etc. to look for. Community of RabbitMQ is larger. Their site has better structure, so it is faster to find an answers. Also RabbitMQ has better documentation about all available plugins. ActiveMQ has only description about some important plugins.

- Official docker image of RabbitMQ (see criterion 2.6)

3. Theoretical considerations

This section contains documentation part that helped us with making decisions and implementation of broker. References in the title were used to create this section.

3.1 MQTT and AMQP

The content of this section is based on [26].

For PrEstoCloud project we need to use both MQTT and AMQP protocols. RabbitMQ use AMQP 0-9-1 by default, but MQTT 3.1.1 is implemented with MQTT plugin. The MQTT plugin builds on top of RabbitMQ core protocol's entities: exchanges and queues. Messages published to MQTT topics use a topic exchange (amq.topic by default) internally. Subscribers consume from RabbitMQ queues bound to the exchange. MQTT adapter expects for exchange to be a topic type.

Note that MQTT uses slashes ("/") for topic segment separators and AMQP uses dots. This plugin translates patterns under the hood to bridge the two, for example, watch/<id> becomes watch.<id> and vice versa. This has one important limitation: MQTT topics that have dots in them won't work as expected and are to be avoided, the same goes for AMQP routing keys that contains slashes. Topics in RabbitMQ are called routing key.

AMQP Wildcards [27]	MQTT Wildcards [28]
* (star) can substitute for exactly one word.	+ (plus) can substitute for single level
# (hash) can substitute for zero or more words.	# (hash) can substitute for multiple level

So in our broker we use our TOPIC exchange named presto.cloud with multiple queues.

3.2 Topic Exchange

The content of this section is based on [29].

Topic exchanges route messages to one or many queues based on matching between a message routing key and the pattern that was used to bind a queue to an exchange. The topic exchange type is often used to implement various publish/subscribe pattern variations. Topic exchanges are commonly used for the multicast routing of messages.

Topic exchanges have a very broad set of use cases. Whenever a problem involves multiple consumers/applications that selectively choose which type of messages they want to receive, the use of topic exchanges should be considered.

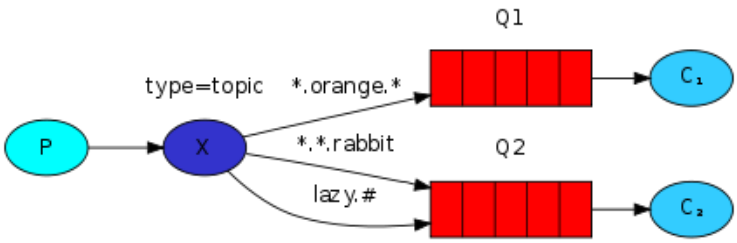

Figure 6: Example of Topic exchange with different topics

Besides the exchange type, exchanges are declared with a number of attributes, the most important of which are:

- Name
- Durability (exchanges survive broker restart)
- Auto-delete (exchange is deleted when last queue is unbound from it) and
- Arguments (optional, used by plugins and broker-specific features).

Our presto.cloud exchange is durable, no auto-delete and external. This is set in definition.json files. Beside this exchange we use multiple queues.

3.3 Queues

The content of this section is based on [29].

Queues in the AMQP model are very similar to queues in other message- and task-queueing systems: they store messages that are consumed by applications. Queues share some properties with exchanges, but also have some additional properties:

- Name
- Durable (the queue will survive a broker restart)
- Exclusive (used by only one connection and the queue will be deleted when that connection closes)
- Auto-delete (queue that has had at least one consumer is deleted when last consumer unsubscribes)
- Arguments (optional; used by plugins and broker-specific features such as message TTL, queue length limit, etc.)

Before a queue can be used it has to be declared. Declaring a queue will cause it to be created if it does not already exist.

Applications may pick queue names or ask the broker to generate a name for them. An AMQP broker can generate a unique queue name on behalf of an app. In our project we use the name created by broker by default, but user can also set his queue names.

User can change and adapt all parameters for queue declaration to his need, but default setup is durable, no exclusive, no auto delete queue that expires after it has been unused for 2 days.

We also have implemented automatic recovery from network failures. We will discuss this further down.

3.4 Connection Recovery

The content of this section is based on [30].

Network connection between clients and RabbitMQ nodes can fail. RabbitMQ Java client supports automatic recovery of connections and topology (queues, exchanges, bindings, and consumers). The automatic recovery process for many applications follows the following steps:

1. Reconnect
2. Restore connection listeners
3. Re-open channels
4. Restore channel listeners
5. Restore channel basic.qos setting, publisher confirms and transaction settings

Topology recovery includes the following actions, performed for every channel:

1. Re-declare exchanges (except for predefined ones)
2. Re-declare queues
3. Recover all bindings
4. Recover all consumers

Topology recovery involves recovery of exchanges, queues, bindings and consumers. It is enabled by default when automatic recovery is enabled. Topology recovery can be disabled explicitly if needed: factory.setTopologyRecoveryEnabled(false);. To disable or enable automatic connection recovery, you should use the factory.setAutomaticRecoveryEnabled(boolean) method.

If recovery fails due to an exception (e.g. RabbitMQ node is still not reachable), it will be retried after a fixed time interval (default is 5 seconds). The interval can be configured factory.setNetworkRecoveryInterval(10000); this is 10s.

3.4.1 When Will Connection Recovery Be Triggered?

Automatic connection recovery, if enabled, will be triggered by the following events:

- An I/O exception is thrown in connection's I/O loop
- A socket read operation times out
- Missed server heartbeats are detected
- Any other unexpected exception is thrown in connection's I/O loop

whichever happens first.

Channel-level exceptions will not trigger any kind of recovery as they usually indicate a semantic issue in the application (e.g. an attempt to consume from a non-existent queue).

3.4.2 Recovery Listeners

It is possible to register one or more recovery listeners on recoverable connections and channels. When connection recovery is enabled, connections returned by ConnectionFactory#newConnection and Connection#createChannel implement com.rabbitmq.client.Recoverable, providing two methods with fairly descriptive names:

- addRecoveryListener
- removeRecoveryListener

Currently we need to cast connections and channels to Recoverable in order to use those methods.

A RecoveryListener receives notifications about completed automatic connection recovery. Because we use generated queue names from broker, we use this listener for manual topology recovery.

3.5 TLS

The content of this section is based on [31] and [32].

Transport Layer Security (TLS) is a protocol that provides confidentiality and data integrity between two communicating applications. It's the most widely deployed security protocol used today, and is used for Web browsers and other applications that require data to be securely exchanged over a network, such as file transfers, VPN connections, instant messaging and voice over IP.

TLS evolved from Netscape's Secure Sockets Layer (SSL) protocol and has largely superseded it, although the terms SSL or SSL/TLS are still sometimes used. According to the protocol specification, TLS is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. The Record Protocol provides connection security, while the Handshake Protocol allows the server and client to authenticate each other and to negotiate encryption algorithms and cryptographic keys before any data is exchanged.

TLS 1.2 is the current version of the protocol, and as of this writing, the Transport Layer Security Working Group of the IETF is working on TLS 1.3 to address the vulnerabilities that have been exposed over the past few years, reduce the chance of implementation errors and remove features no longer needed. TLS 1.3 is still a draft and has not been finalized yet. We use 1.2 version of TLS.

3.6 Federation

The content of this section is based on [33], [34] and [35].

Federation allows an exchange or queue on one broker to receive messages published to an exchange or queue on another (the brokers may be individual machines, or clusters). Communication is via AMQP (with optional TLS).

The high-level goal of the federation plugin is to transmit messages between brokers without requiring clustering. This is useful for various reasons:

- Loose coupling - The federation plugin can transmit messages between brokers (or clusters) in different administrative domains:
  - they may have different users and virtual hosts;
  - they may run on different versions of RabbitMQ and Erlang.
- WAN-friendly - The federation plugin uses AMQP 0-9-1 to communicate between brokers, and is designed to tolerate intermittent connectivity.
- Specificity - A broker can contain federated and local-only components - you don't need to federate everything if you don't want to.
- Scalability - Federation does not require $O(n^2)$ connections between n brokers, which should mean it scales better.

A federated exchange links to other exchanges (called upstream exchanges). Logically, messages published to the upstream exchanges are copied to the federated exchange, as though they were published directly to it. The upstream exchanges do not need to be reconfigured and they do not have to be on the same broker or in the same cluster. All of the configuration needed to establish the upstream links and the federated exchange is in the broker with the federated exchange and there is nothing to prevent a federated exchange being 'upstream' from another federated exchange. Because all of this we use this type of federation.

One typical use would be to implement massive fanout - a single "root" exchange in one broker (which need not be federated) can be declared as upstream by many other federated exchanges in other brokers. In turn, each of these can be upstream for many more exchanges, and so on.

Federated queues provides a way of balancing the load of a single queue across nodes or clusters. A federated queue links to other queues (called upstream queues). It will retrieve messages from upstream queues in order to satisfy demand for messages from local consumers. The upstream queues do not need to be reconfigured and they do not have to be on the same broker or in the same cluster.

When using a federation in a cluster, all the nodes of the cluster should have the federation plugin enabled. Information about federation upstreams is stored in the RabbitMQ database, along with users, permissions, queues, etc. There are three levels of configuration involved in federation:

- Upstreams: each upstream defines how to connect to another broker.
- Upstream sets: each upstream set groups together a set of upstreams to use for federation.
- Policies: each policy selects a set of exchanges, queues or both, and applies a single upstream or an upstream set to those objects.

In practice, for simple use cases you can almost ignore the existence of upstream sets, since there is an implicitly-defined upstream set called all to which all upstreams are added.

3.6.1 Federated exchanges parameters

The content of this section is based on [36] and [37].

3.6.1.1 *uri*

The AMQP URI(s) for the upstream. This field can either be a string, or a list of strings. If more than one string is provided, the federation plugin will randomly pick one URI from the list. This can be used to connect to an upstream cluster. To connect to multiple URIs simultaneously use multiple upstreams.

The syntax of an AMQP 0-9-1 URI is defined by the following ABNF rules.

```
amqp_URI       = "amqp://" amqp_authority [ "/" vhost ] [ "?" query ]
amqp_authority = [ amqp_userinfo "@" ] host [ ":" port ]
amqp_userinfo  = username [ ":" password ]
username       = *( unreserved / pct-encoded / sub-delims )
password       = *( unreserved / pct-encoded / sub-delims )
vhost          = segment
```

query can has some of next parameters:

Parameter name	Description
cacertfile, certfile, keyfile	Paths to files to use in order to present a client-side TLS certificate to the server. Only of use for the amqps scheme.
verify, server_name_indication	Only of use for the amqps scheme and used to configure verification of the server's x509 (TLS) certificate. It is highly recommended to use both values.
auth_mechanism	SASL authentication mechanisms to consider when negotiating a mechanism with the server. This parameter can be specified multiple times.
heartbeat	Heartbeat timeout value in seconds (an integer) to negotiate with the server.
connection_timeout	Time in milliseconds (an integer) to wait while establishing a TCP connection to the server before giving up.
channel_max	Maximum number of channels to permit on this connection

We use next uri:

amqps://guest:guest@192.168.85.112:35671?cacertfile=/home/testca/cacert.pem&certfile=/home/client/cert.pem&keyfile=/home/client/key.pem&verify=verify_peer&fail_if_no_peer_cert=true&server_name_indication=nissatech2.com

So we use AMQPS protocol to connect to IP address of upbroker with default user credentials on matching port with TLS.

3.6.1.2 *prefetch-count*

The maximum number of unacknowledged messages copied over a link at any one time. Default is **1000**. We use this number.

3.6.1.3 *reconnect-delay*

The duration (in seconds) to wait before reconnecting to the broker after being disconnected. Default is 1 and we use it.

3.6.1.4 *ack-mode*

Determines how the link should acknowledge messages. If set to on-confirm (the default), messages are acknowledged to the upstream broker after they have been confirmed downstream. This handles network errors and broker failures without losing messages, and is the slowest option. We use this approach because this benefits.

If set to on-publish, messages are acknowledged to the upstream broker after they have been published downstream. This handles network errors without losing messages, but may lose messages in the event of broker failures.

If set to no-ack, message acknowledgements are not used. This is the fastest option, but may lose messages in the event of network or broker failures.

3.6.1.5 *trust-user-id*

Determines how federation should interact with the validated user-id feature. If set to true, federation will pass through any validated user-id from the upstream, even though it cannot validate it itself. If set to false or not set, it will clear any validated user-id it encounters. You should only set this to true if you trust the upstream server (and by extension, all its upstreams) not to forge user-ids. We set it on false.

3.6.1.6 *exchange*

The name of the upstream exchange. Default is to use the same name as the federated exchange. We use the same name.

3.6.1.7 *max-hops*

The maximum number of federation links that a message published to a federated exchange can traverse before it is discarded. Default is 1 and we use it. Note that even if max-hops is set to a value greater than 1, messages will never visit the same node twice due to travelling in a loop. However, messages may still be duplicated if it is possible for them to travel from the source to the destination via multiple routes.

3.6.1.8 *expires*

The expiry time (in milliseconds) after which an upstream queue for a federated exchange may be deleted, if a connection to the upstream broker is lost. The default is 'none', meaning the queue should never expire. This setting controls how long the upstream queue will last before it is eligible for deletion if the connection is lost. We set this value to 3 600 000, or 1h.

3.6.1.9 *message-ttl*

The expiry time for messages in the upstream queue for a federated exchange (see expires), in milliseconds. Default is 'none', meaning messages should never expire, and we use it.

3.6.1.10 *ha-policy*

Determines the "x-ha-policy" argument for the upstream queue for a federated exchange (see expires). This is only of interest when connecting to old brokers which determine queue HA mode using this argument. Default is 'none', meaning the queue is not HA.

3.7 Docker

The content of this section is based on [38] and [43].

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight because they don't need the extra load of a hypervisor, but run directly within the host machine's kernel. If we compare a Docker container with a traditional VM, we see that Docker saves us from a Guest OS. We do not need any Guest OS in order to run or test our application.

Docker, in general, is composed of 5 core components:

- Docker Image and Dockerfile,
- Docker Daemon,
- Docker Client,
- Docker Host,
- Docker Registry and
- Docker Hub.

We interact with Docker through Docker Client, we type any command to Docker Client that we wish to run for manipulating containers. Docker Daemon is the component which manages the Docker images and containers on our local machine. It manages and manipulates containers using commands received from Docker Client.

As we said that applications run in containers. We make a Docker container of our application by making a Docker Image of our application. This Docker image is made by writing a Dockerfile. Dockerfile is a recipe or blueprint of a Docker container. Dockerfile is composed of different commands for making a Docker image.

Basically Docker containers can wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries – anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment.

Containers and virtual machines have similar resource isolation and allocation benefits -- but a different architectural approach allows containers to be more portable and efficient. Docker containers include the application and all of its dependencies - but share the kernel with other containers, running as isolated processes in user space on the host operating system. Docker containers are not tied to any specific infrastructure: they run on any computer, on any infrastructure, and in any cloud.

An example of raspberry pi can be found at: https://www.raspberrypi.org/blog/docker-comes-to-raspberry-pi/.

4. Implementation

This section provides implementation details based on decisions described in previous section. Here we explain how we created broker with all needed functionality. The section also contains links to git code of maven libraries that can be used for communication with the broker. Finally, the section includes information about our proposed topology and future plans.

4.1 Overview

First we started by implementing RabbitMQ on local machine and testing basic functionality and plugins. We added management and learn how to use it. After that we added MQTT plugin and implemented java libraries for communication with broker. This is described in section 5.

Final version of these libraries can be found at:

- https://git.nissatech.com/presto/RabbitMQ-AMQP-message-consumer.git
- https://git.nissatech.com/presto/RabbitMQ-MQTT-message-producer.git

We use RabbitMQ amqp-client 5.2.0 for consumer and eclipse.paho.client.mqttv3 1.2.0 for producer. We has some problem because stable version of Paho MQTT Java client has some known issue with client disconnecting and thread exit but we workaround that (https://github.com/eclipse/paho.mqtt.java/issues/402). We also use manual acknowledgement after processing the message. After successful usage we implemented TLS with openssl CA on both sides.

Then we start to use Docker and put broker in container. Because of distribution part we create two node of broker on different ports and connect them with federation. There was some problems because federated broker act like client so we needed to add client's certificates (https://serverfault.com/questions/580570/rabbitmq-federation-with-ssl-client-certificates-not-working).

In the end we add autorecovery and start some performance testing of pilot federated broker in order to make broker stable and check if it delivers all messages. This help us to remove some bugs and to have stable version.

Next installation guide is optimized in way we have much preparation files and less commands.

4.2 Installation guide

Note: all commands are for Ubuntu operation system.

4.2.1 Step 1 – Install Docker

$ sudo apt-get update

$ sudo apt-get install apt-transport-https ca-certificates curl software-properties-common

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

$ sudo apt-key fingerprint 0EBFCD88

$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

$ sudo apt-get update

$ sudo apt-get install docker-ce

$ sudo docker --version

if last command return the number of version everything went fine.

4.2.2 Step 2 – Install Kitematic v0.17.3

This is optional step because this app is UI for Docker, all can be seen from terminal also.

$ sudo dpkg -i ./Kitematic_0.17.3_amd64.deb

$ sudo groupadd docker

$ sudo gpasswd -a ${USER} docker

$ sudo service docker restart

$ newgrp docker

$ kitematic

4.2.3 Step 3 – Prepare files for easier configuration of broker

We need next files for configuration:

- **openssl.cnf** - configuration file for openssl, contains where and how to generate certs.

- **start.sh** - creating directories and certificates autority for later use.
- **Dockerfile** – here we upgrade the latest official RabbitMQ image with needed plugins and configuration.
- **definitions.json** - determinate structure and component of broker (create presto.cloud exchange, default user).
- **prepareServerUp.sh** - script for creating server certificates for TLS comunication.
- **prepareServerDown.sh** - beside that, federated (down) broker also need client certificates for comunication with other node, so this is additional part to prepareServerUp.
- **nissarabbitUp.sh** - run our image in container for up broker.
- **nissarabbitDown.sh** - run our image in container for down broker (on different ports, with other name and preparation file).

4.2.3.1 *Content of files*

4.2.3.1.1 Openssl.cnf

```
[ ca ]
default_ca = testca
[ testca ]
dir = .
certificate = $dir/cacert.pem
database = $dir/index.txt
new_certs_dir = $dir/certs
private_key = $dir/private/cakey.pem
serial = $dir/serial
default_crl_days = 7
default_days = 365
default_md = sha256
policy = testca_policy
x509_extensions = certificate_extensions
[ testca_policy ]
commonName = supplied
stateOrProvinceName = optional
countryName = optional
emailAddress = optional
organizationName = optional
organizationalUnitName = optional
domainComponent = optional
[ certificate_extensions ]
basicConstraints = CA:false
[ req ]
default_bits = 2048
default_keyfile = ./private/cakey.pem
default_md = sha256
prompt = yes
distinguished_name = root_ca_distinguished_name
x509_extensions = root_ca_extensions
[ root_ca_distinguished_name ]
commonName = hostname
[ root_ca_extensions ]
basicConstraints = CA:true
keyUsage = keyCertSign, cRLSign
[ client_ca_extensions ]
basicConstraints = CA:false
```

```
keyUsage = digitalSignature,keyEncipherment
extendedKeyUsage = 1.3.6.1.5.5.7.3.2
[ server_ca_extensions ]
basicConstraints = CA:false
keyUsage = digitalSignature,keyEncipherment
extendedKeyUsage = 1.3.6.1.5.5.7.3.1
```

additional explanation can be found at [39].

4.2.3.1.2 start.sh

```
mkdir testca
cd testca/
mkdir certs private
chmod 700 private
echo 01 > serial
touch index.txt
cp ../openssl.cnf .

# Create a self-signed certificate that will serve a certificate authority (CA). The private key is located under "private".
openssl req -x509 -config openssl.cnf -newkey rsa:2048 -days 365 -out cacert.pem -outform PEM -subj /CN=MyTestCA/ -nodes

#Encode our certificate with DER.
openssl x509 -in cacert.pem -out cacert.cer -outform DER
```

4.2.3.1.3 Dockerfile

```
#use latest official RabbitMQ image
FROM rabbitmq:latest

#add needed plugins (MQTT, Management and Federation)
RUN rabbitmq-plugins enable --offline rabbitmq_management
RUN rabbitmq-plugins enable --offline rabbitmq_mqtt
RUN rabbitmq-plugins enable --offline rabbitmq_federation
RUN rabbitmq-plugins enable --offline rabbitmq_federation_management

#install needed programs for later use in container, and creating file system on docker container
RUN apt-get update \
    && apt-get install nano \
    && apt-get install openssl -y  \
    && mkdir -p /home/ \
    && mkdir -p /home/server \
    && mkdir -p /home/testca

#add definition of broker to RabbitMQ config file
RUN echo "management.load_definitions = /etc/rabbitmq/definitions.json" >> /etc/rabbitmq/rabbitmq.conf

#change default mqtt exchange to our
RUN echo "mqtt.exchange=presto.cloud" >> /etc/rabbitmq/rabbitmq.conf
```

```
#open ports for mqtt and management
EXPOSE 15671 15672 1883 8883
```

4.2.3.1.4 definitions.json

```
{"rabbit_version":"3.7.4",
    "users":[{
      "name":"guest",
      "password_hash":"K7TaTzB1RbNPn/1rW/YQih+zaO9uO9QwTZ8yjheKawNEY+at",
      "hashing_algorithm":"rabbit_password_hashing_sha256",
      "tags":"administrator"
    }],
    "vhosts":[{"name":"/"}],
    "permissions":[{
      "user":"guest",
      "vhost":"/",
      "configure":".*",
      "write":".*",
      "read":".*"
    }],
    "topic_permissions":[],
    "parameters":[],
    "global_parameters":[{
      "name":"cluster_name",
      "value":"rabbit@nissatech.com"
    }],
    "policies":[],
    "queues":[],
    "exchanges":[{
      "name":"presto.cloud",
      "vhost":"/",
      "type":"topic",
      "durable":true,
      "auto_delete":false,
      "internal":false,
      "arguments":{}
    }],
    "bindings":[]
}
```

4.2.3.1.5 prepareServerUp.sh

```
set -eu

#
# Prepare the server's stuff.
#
cd /home/server

# Generate a private RSA key.
openssl genrsa -out key.pem 2048

# Generate a certificate from our private key.
```

```
openssl req -new -key key.pem -out req.pem -outform PEM -subj
/CN=$(hostname)/O=server/ -nodes

# Sign the certificate with our CA.
cd /home/testca
openssl ca -config openssl.cnf -in /home/server/req.pem -out
/home/server/cert.pem -notext -batch -extensions server_ca_extensions

# Create a key store that will contain our certificate.
cd /home/server
openssl pkcs12 -export -out keycert.p12 -in cert.pem -inkey key.pem -
passout pass:MySecretPassword

# Make files visible to broker
cd /home &&  chmod 755 testca/cacert.pem server/cert.pem server/key.pem
```

4.2.3.1.6 prepareServerDown.sh additional content

```
#
# Prepare the client's stuff.
#
mkdir /home/client
cd /home/client

# Generate a private RSA key.
openssl genrsa -out key.pem 2048

# Generate a certificate from our private key.
openssl req -new -key key.pem -out req.pem -outform PEM -subj
/CN=$(hostname)/O=client/ -nodes

# Sign the certificate with our CA.
cd /home/testca
openssl ca -config openssl.cnf -in /home/client/req.pem -out
/home/client/cert.pem -notext -batch -extensions client_ca_extensions

# Create a key store that will contain our certificate.
cd /home/client
openssl pkcs12 -export -out key-store.p12 -in cert.pem -inkey key.pem -
passout pass:MySecretPassword

# Make files visible to broker
cd /home &&  chmod 755 testca/cacert.pem server/cert.pem server/key.pem
client/cert.pem client/key.pem
```

4.2.3.1.7 Scripts for running Docker image

nissarabbitDown.sh

```
sudo docker run -d \
    --name="nissarabbit" \
```

```
    --hostname="nissatech.com"\
    -e RABBITMQ_NODENAME="rabbit" \
    -e RABBITMQ_DEFAULT_USER="guest" \
    -e RABBITMQ_DEFAULT_PASS="guest" \
    --publish="5671:5671" \
    --publish="5672:5672" \
    --publish="15672:15672" \
    --publish="1883:1883" \
    --publish="8883:8883" \
    --publish="4369:4369" \
    --publish="25672:25672" \
    -v
/home/nissatech/Desktop/definitions.json:/etc/rabbitmq/definitions.json
\
    -v /home/nissatech/Desktop/testca:/home/testca \
    -v      /home/nissatech/Desktop/prepareServerDown.sh:/home/prepare-
server.sh \
    nrabbit
```

nissarabbitUp.sh

```
sudo docker run -d \
    --name="nissarabbit2" \
    --hostname="nissatech2.com" \
    -e RABBITMQ_NODENAME="rabbit" \
    -e RABBITMQ_DEFAULT_USER="guest" \
    -e RABBITMQ_DEFAULT_PASS="guest" \
    --publish="35671:5671" \
    --publish="35672:5672" \
    --publish="45672:15672" \
    --publish="31883:1883" \
    --publish="38883:8883" \
    --publish="34369:4369" \
    --publish="55672:25672" \
    -v
/home/nissatech/Desktop/definitions.json:/etc/rabbitmq/definitions.json
\
    -v /home/nissatech/Desktop/testca:/home/testca \
    -v        /home/nissatech/Desktop/prepareServerUp.sh:/home/prepare-
server.sh \
    nrabbit
```

4.2.4 Step 4 – Run everything!

```
$ bash start.sh                                    #create directories
                                                   and CA.

$ sudo docker build -t nrabbit - <Dockerfile       #create docker image
```

		from latest officialrabbitmqimage with defines upgrades.
$	bash nissarabbitDown.sh	#run nrabbitin container.
$	bash nissarabbitUp.sh	#run nrabbitin container.
$	sudo docker exec -it nissarabbit[2] /home/prepare-server.sh	#run prepare-server scripts in containers, creating needed certs. Server certs for both and client certs only for down.
$	sudo docker exec -it nissarabbit[2] nano /etc/rabbitmq/rabbitmq.conf	#open config file for change
add	listeners.ssl.default=5671 mqtt.listeners.ssl.default=8883 ssl_options.depth=2 ssl_options.verify=verify_peer ssl_options.fail_if_no_peer_cert=true ssl_options.keyfile=/home/server/key.pem ssl_options.certfile=/home/server/cert.pem ssl_options.cacertfile=/home/testca/cacert.pem ssl_options.versions.1 = tlsv1.2 ssl_options.versions.2 = tlsv1.1 ssl_options.honor_cipher_order = true ssl_options.honor_ecc_order= true	# opening TLS ports for both protocols, adding TLS file paths to options and adding limitation to version of TLS and force server's TLS implementation to dictate its preference (cipher suite order) to avoid malicious clients that intentionally negotiate weak cipher suites in preparation for running an attack on them.
$	sudo docker restart nissarabbit[2]	#restarting brokers to run new configuration

Protocol	Bound to	Port
amqp	::	5672
amqp/ssl	::	5671
clustering	::	25672
http	::	15672
mqtt	::	1883
mqtt/ssl	::	8883

Figure 7: Management show that this ports now use SSL

4.2.5 Step 5 – add Federation do down node

```
  #adding federation policy for exchanges which name starts with
  "presto.", and use federation-upstream that we will create.

  sudo docker exec nissarabbit rabbitmqctl set_policy --apply-to
$ exchanges presto-federation "^presto\." '{"federation-
  upstream":"NissaPresto"}

  #adding upstream with AMQP/SSL connection to address of other broker,
  with needed login data and on corresponding port, with default "/"
  vhost, with client certs. The upstream queue will last one hour before
  it is eligible for deletion if the connection is lost.

  sudo docker exec nissarabbit rabbitmqctl set_parameter federation-
  upstream NissaPresto
$ '{"uri":"amqps://guest:guest@192.168.85.112:35671?cacertfile=/home/test
  ca/cacert.pem&certfile=/home/client/cert.pem&keyfile=/home/client/key.p
  em&verify=verify_peer&fail_if_no_peer_cert=true&server_name_indication=
  nissatech2.com","ack-mode":"on-confirm","trust-user-
  id":false,"expires":3600000}'
```

Now we can use our broker.

4.2.6 Step 6 – RabbitMQ test TLS

Create **generate-client-keys.sh** on with next content:

```
set -eu

#
# Prepare the client's stuff.
#
mkdir client
cd client

# Generate a private RSA key.
```

```
openssl genrsa -out key.pem 2048

# Generate a certificate from our private key.
openssl req -new -key key.pem -out req.pem -outform PEM -subj
/CN=$(hostname)/O=client/ -nodes

# Sign the certificate with our CA.
cd ../testca
openssl ca -config openssl.cnf -in ../client/req.pem -out
../client/cert.pem -notext -batch -extensions client_ca_extensions

# Create a key store that will contain our certificate.
cd ../client
openssl pkcs12 -export -out key-store.p12 -in cert.pem -inkey key.pem -
passout pass:MySecretPassword

# Create a trust store that will contain the certificate of our CA.
openssl pkcs12 -export -out trust-store.p12 -in ../testca/cacert.pem -
inkey ../testca/private/cakey.pem -passout pass:rabbitstore

# Make files visible to code
chmod 755 key-store.p12 trust-store.p12


$ bash generate-client-keys.sh

$ openssl s_client -connect localhost:[3]5671 -cert
  client/cert.pem -key client/key.pem -CAfile testca/cacert.pem

$ openssl s_client -connect localhost:[3]8883 -cert
  client/cert.pem -key client/key.pem -CAfile testca/cacert.pem
```

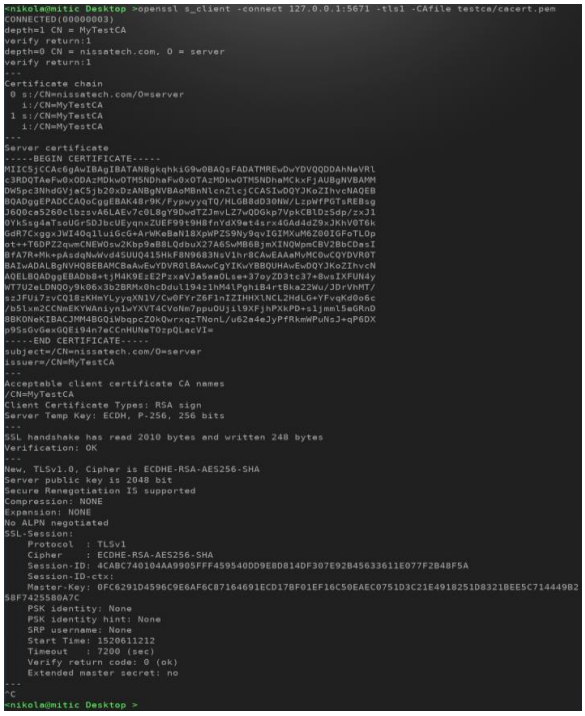

Figure 8: How test looks from terminal

4.2.7 Some additional commands for using RabbitMQ over Docker

```
$ sudo docker {start, restart, stop} nissarabbit
```

Starting, restarting and stop container.

```
$ sudo docker cp definitions.json nissarabbit:/etc/rabbitmq/definitions.json
$ sudo docker diff nissarabbit
$ sudo docker exec -it nissarabbit cat /etc/rabbitmq/enabled_plugins
```

First command copy definitions file to container's filesystem on path /etc/rabbitmq. Second one shows all path of files in container filesystem. Third shows content of file "enabled_plugins". There should be rabbitmq_management, rabbitmq_federation, rabbitmq_federation_management and rabbitmq_mqtt.

```
$ sudo docker exec nissarabbit rabbitmqctl add_user admin <password>
$ sudo docker exec nissarabbit rabbitmqctl set_user_tags admin administrator
$ sudo docker exec nissarabbit rabbitmqctl set_permissions -p / admin ".*" ".*" ".*"
```

Creating admin user

```
$ sudo docker exec nissarabbit rabbitmq-plugins enable <name of plugin>
```

Installing additional plugins to broker

```
$ sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged
```

Forgotten Acknowledgment

```
$ sudo rabbitmqctl list_{consumers, queues, exchanges, bindings, hashes, ciphers}
```

Listing Consumers, Queues, Exchanges, Bindings, Hashes, Ciphers

```
$ sudo docker images
$ sudo docker rmi <IMAGE ID>
```

The first command lists all images in our docker. There is rabbitmq : latest and our nrabbit : latest. And second command is for deleting a specific image.

```
$ sudo docker ps -a
```

```
$ sudo docker rm <NAME>
```

The first command lists all containers in our docker. There is our nissarabbit and nissarabbit2. And second command is for deleting a specific container.

```
$ sudo docker exec nissarabbit rabbitmqctl list_exchanges name
  policy | grep presto-federation
$ sudo docker exec nissarabbit rabbitmqctl eval
  'rabbit_federation_status:status().'
```

With this we check our federation policy over exchanges and federation status.

```
$ sudo apt-get autoremove --purge docker-engine
$ sudo rm -rf /var/lib/docker
```

Uninstall Docker

4.3 Implemented structure and future plans

We run consumer to create a queue to downnode (nissarabbit), and then run producer with the same topic (only difference with AMQP dots and MQTT slashes) to send messages to upnode (nissarabbit2). After upbroker gets a message, it forwards the message to downbroker through the established connection. If there is no messages for forwarding, downbroker every 30s re-establish the connection to check federation exchange status. Implemented structure is shown in Figure 9.

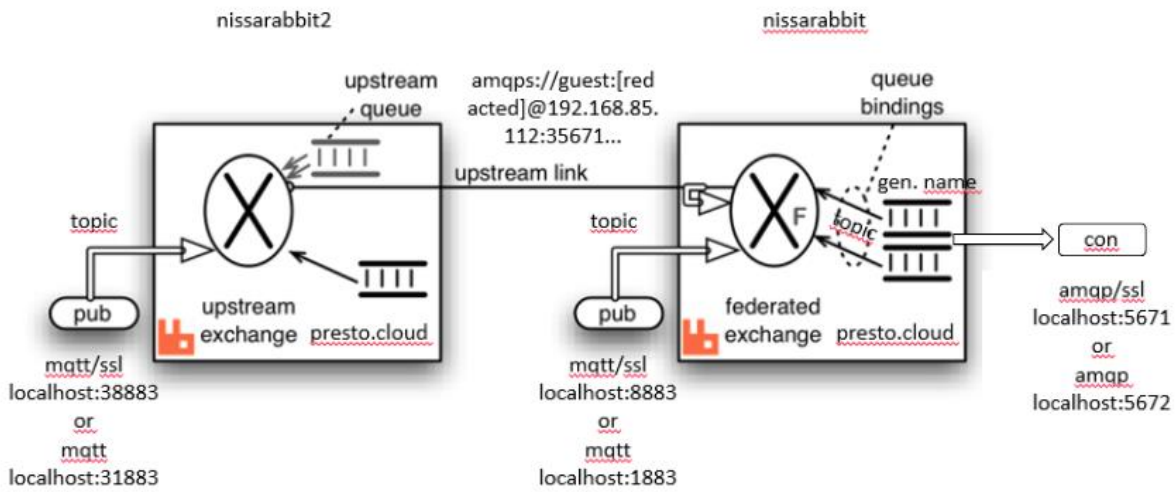

Figure 9: Implemented structure

Upnode in federation forwards every MQTT message that it receives from the producer to downnode like AMQP message and then consumer gets it from downnode with AMQP protocol. For all communication in broker (producer to node, node to node, node to consumer) we use TLS protocol with our generated certificates and keys. Node to node always use TLS, but producer and consumer can choose this optionally. In producers and consumers, we provide reconnection if something happen

with network connection and ensure persistence and delivery if some component go down. (Now we are testing this).

We run some test like:

- Start consumer and producer, after some period of time we stop consumer and wait that producer sends thousands of messages. Consumer after connecting again can get all messages without any problem.
- We connect more consumer on same queue, and they work in round robin, circular one after another.
- Run communications for 24h with both sides.
- We also try test where we restart broker during communication, and all parts can survive restart and auto connect without any problem.

For now, we implemented a federation of two nodes, we know how to connect more of them in way we need. Probably it will be topology where we will have one main broker and make him downnode for more downnode brokers, so all published messages will be forwarded to this one broker, where consumers will get and process messages. This is shown in Figure 10.

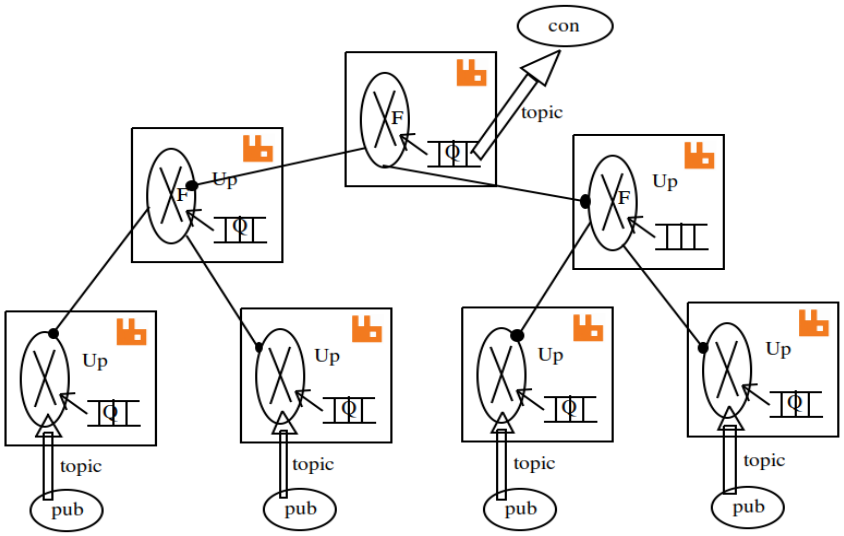

Figure 10: Proposed topology

5. Usage of the broker in the PrEstoCloud

In this section we describe the usage of the broker in the PrEstoCloud.

5.1 Message format

We agreed on the following format for a message:

- device_id: hostname (for example, on Linux OS can be found by typing *hostname* in terminal)
- device_type: type of the device, indicates whether it is a server, a camera, Android device
- timestamp: Unix timestamp (seconds)
- parameter_name: a name of a parameter
- parameter_value; a measured value for the parameter (in float) in that particular moment (for the specified timestamp).

For now, list of parameters names is:

- system.cpu.system(%) - system activities on CPU
- system.cpu.user(%) - user activities on CPU
- system.cpu.iowait(%) - IO activities on CPU
- system.ram.used (GB) - used RAM
- system.ctxt.switches - number of context switches
- system.ram.free(GB) - free RAM

Here we give an example of the messages in JSON.

```
{
    "device_Id": "Android_24",
    "device_type": "Android",
    "timestamp": 123456789,
    "parameter_name": "free RAM",
    "parameter_value": 2048
}
```

5.2 Topics

We would like to note that this is still an open issued discussed by the consortium. Here we provide our proposals.

Our first suggestion about the topics was that there should be one topic that every device would use to send relevant data.

Second suggestion is there should be one topic for every device, and every topic should be bound to separate queue. Thus, the load balancing is better (every device should have its own queue). This way it is easy to get the data for the device of interest. The list of topics is known when the list of devices is available.

Topic name is the name of the device extracted from the netdata tool (https://github.com/firehol/netdata/wiki), for example *nissatech-Lenovo-ideapad-700-15ISK.* Netdata is a scalable, realtime monitoring tool that works on any Linux kernel. Also, there is one separate topic "ids" that is bound to separate queue that keeps ids of all available devices.

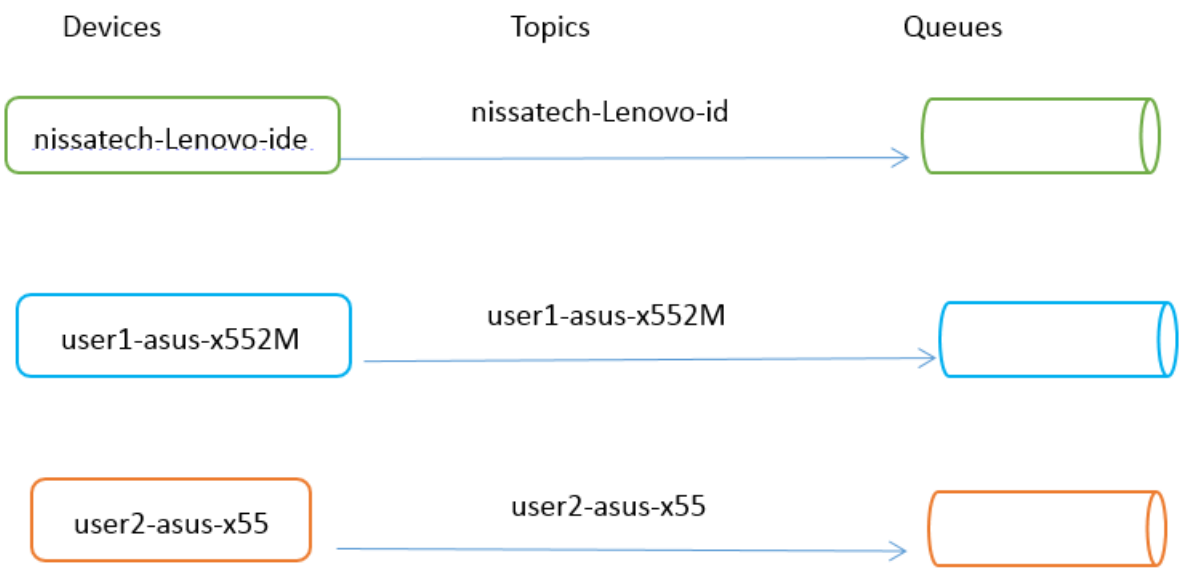

Figure 11: Topics in PrEstoCloud

This is a current and temporary solution, it can change due to requests and needs of the other participants on the project.

Third suggestion could be that we can use the flexibility of RabbitMQ and possibility that topic can consists of more topic levels. Each topic level is separated by a forward slash (topic level separator for MQTT protocol). Format would be device_type/device_id/parameter_name.

Here are some examples of topics:

- laptop/nissatech-Lenovo-ideapad-ISK/system-cpu-user
- camera/camera1/system-ram-used

So for example, if one would like to have all parameters of one device, topic would be **.camera1.** (MQTT format, dots instead of slashes).

If one would like to have values for free ram of all devices that would be *#.system-ram-free*. (In that case it would be mandatory to transform names of the parameters so they do not have dots in their names because of possible problems during conversion.

Here we give an example for a user cpu activity for all laptops: *laptop.*.system-cpu-user*.

5.3 An example

Here we give an example of the typical communication:

1. Firstly, every publisher sends its own id to topic "ids" when it connects to broker. If the client already knows for which devices wants to collect the data, he does not even need to subscribe to "ids" topic (this step can be skipped)
2. Subscriber, if interested in that topic, after subscribe creates a queue for that device
3. When queue is created, it can receive data relevant for that device

This is shown in Figure 12. In the next subsection, we show how each step has been implemented.

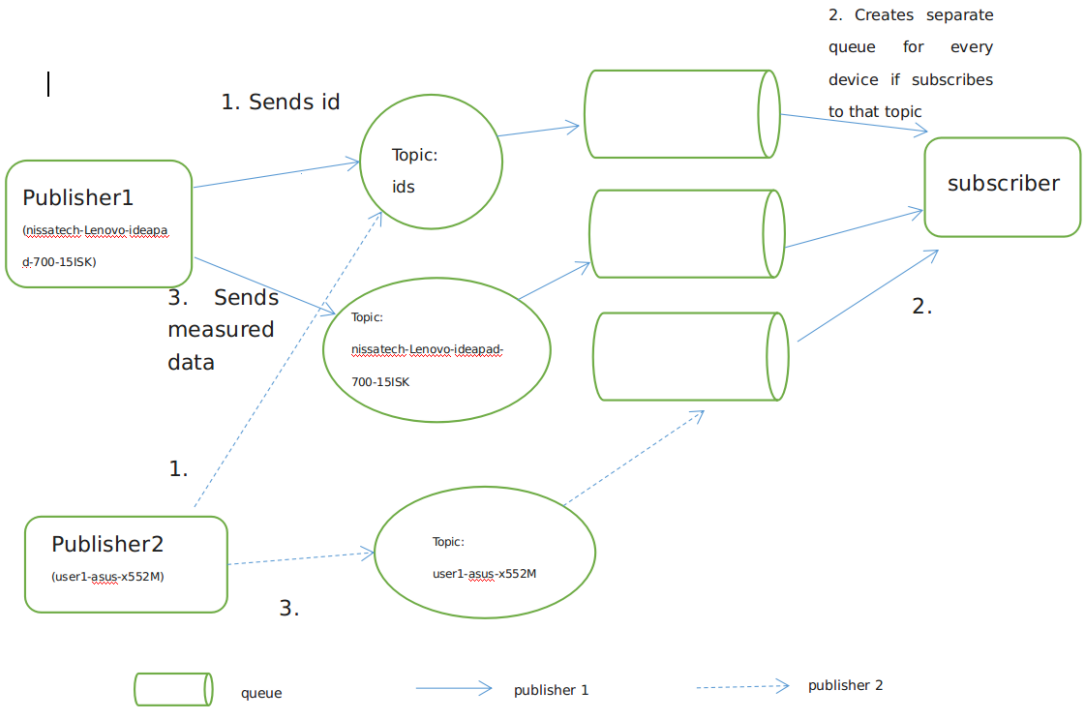

Figure 12: An example

5.4 Current procedure with sending data and creating topics

In this section we give some code-examples that demonstrate the key functionality related to the usage of the broker that has been implemented.

In order the broker to be aware of which devices are available and to be capable to create a topic and a queue for every device, one queue is created for transfer ids of the devices (topic name: *ids*).

```
//producer publishes id to topic ids
producer.publish(deviceId, "ids", 1, false);
```

```
//consumer creates a queue for ids
java.util.function.Consumer<String> consumerFunction = message -> createDataQueue(message);

messageConsumer = new MessageConsumer(brokerUri, "ids", consumerFunction, false);
```

Since we are implementing application that is in charge of prediction of the parameters behavior, our application for every received id creates a separate subscriber that has its own queue and its own topic, as already described.

```
//deviceId is the topic name
```

```
DataConsumer dataConsumer = new DataConsumer(brokerUri, deviceId);
dataConsumer.subscribe();
```

For every subscriber back up is created (we keep topics and queue names in backup file)

```
public void backUp(DataConsumer dataConsumer) {
  String topic = dataConsumer.getTopic();
  String queueName = dataConsumer.getQueueName();
  File dir = new File(file);
  dir.mkdir();

  try (BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(
      new FileOutputStream(file + topic)))) {

      writer.write(topic + "\n");
      writer.write(queueName);
  }

   catch (Exception e) {
     throw new SubscriberBackupException("Could not create back up for subscriber", e);
  }
}
```

If the connection fails, the subscriber automatically reconnects to topic it was connected to before the fail.

If our application that is in charge of receiving data from the broker fails, after the new start, the back up for every subscriber is restored. This way we can be sure that data that was in queues when the crash happened is received. Without this back up, new queues would be created and data would be lost.

```
private void createSubscribersFromBackup(File dir) {
  for (File file : dir.listFiles()) {
    DataConsumer dataConsumer = consumerDataBackup.getBackup(brokerUri, file);
    dataConsumer.subscribe();
  }
}
```

```
public DataConsumer getBackup(String brokerUri, File fileName) {
  try (BufferedReader reader = new BufferedReader(new InputStreamReader(
      new FileInputStream(fileName)))) {
    String topic = reader.readLine();
    String queueName = reader.readLine();

    DataConsumer dataConsumer = new DataConsumer(brokerUri, topic);
```

```
    dataConsumer.setQueueName(queueName);

    return dataConsumer;

  } catch (Exception e) {
    throw new SubscriberBackupException("Could not get back up for subscriber", e);
  }
}
```

6. Appendix 1 - RabbitMQ

The content of this section is based on [40] and [41].

RabbitMQ is an open source message-queueing software called a *message broker* or *queue manager.* Simply said; It is a software where queues can be defined, applications may connect to the queue and transfer a message onto it.

Figure 13: The basic architecture of a message queue

A message can include any kind of information. It could, for example, have information about a process/task that should start on another application (that could be on another server), or it could be just a simple text message. The queue-manager software stores the messages until a receiving application connects and takes a message off the queue. The receiving application then processes the message in an appropriate manner.

RabbitMQ default protocol is Advanced Message Queuing Protocol (AMQP), so the basic element of broker is: exchanges, routes and queues. This is shown in Figure 14. In the rest of this section, the key concpets are introduced.

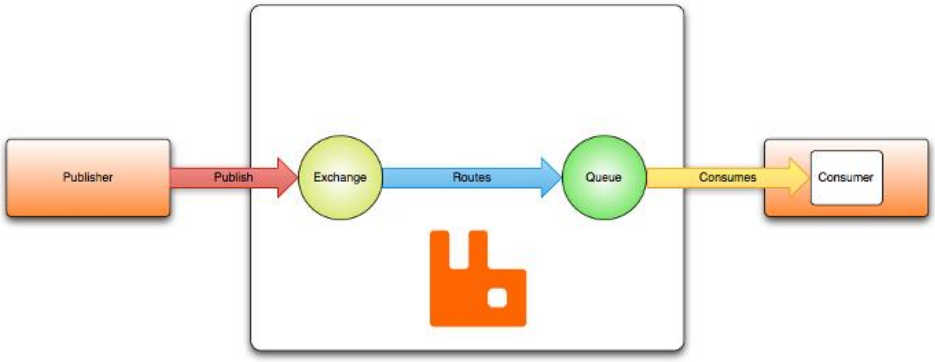

Figure 14: Broker with basic elements

6.1 Exchanges and Exchange Types

Exchanges are AMQP entities where messages are sent. Exchanges take a message and route it into zero or more queues. The routing algorithm used depends on the *exchange type* and rules called *bindings*.

The following exchange types are supported: Direct, Topic, Header and Fanout.

6.1.1 Default Exchange

The default exchange is a direct exchange with no name (empty string) pre-declared by the broker. It has one special property that makes it very useful for simple applications: every queue that is created is automatically bound to it with a routing key which is the same as the queue name.

For example, when you declare a queue with the name of "search-indexing-online", the AMQP 0-9-1 broker will bind it to the default exchange using "search-indexing-online" as the routing key. Therefore, a message published to the default exchange with the routing key "search-indexing-online" will be routed to the queue "search-indexing-online". In other words, the default exchange makes it

seem like it is possible to deliver messages directly to queues, even though that is not technically what is happening.

6.1.2 Direct Exchange

A direct exchange delivers messages to queues based on the message routing key. A direct exchange is ideal for the unicast routing of messages (although they can be used for multicast routing as well). Here is how it works:

- A queue binds to the exchange with a routing key K
- When a new message with routing key R arrives at the direct exchange, the exchange routes it to the queue if K = R

Direct exchanges are often used to distribute tasks between multiple workers (instances of the same application) in a round robin manner. When doing so, it is important to understand that, in AMQP 0-9-1, messages are load balanced between consumers and not between queues.

A direct exchange can be represented graphically as follows:

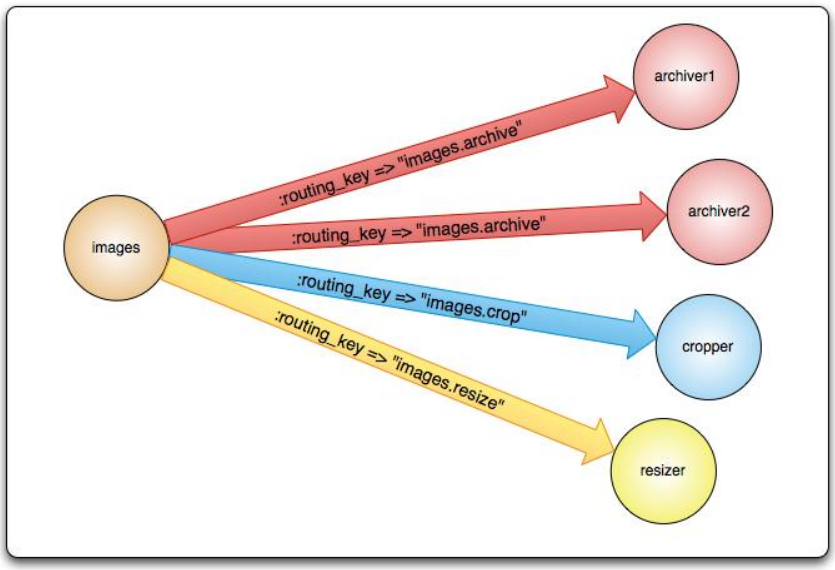

Figure 15: Direct exchange routing

6.1.3 Fanout Exchange

A fanout exchange routes messages to all of the queues that are bound to it and the routing key is ignored. If N queues are bound to a fanout exchange, when a new message is published to that exchange a copy of the message is delivered to all N queues. Fanout exchanges are ideal for the broadcast routing of messages.

A fanout exchange can be represented graphically as follows:

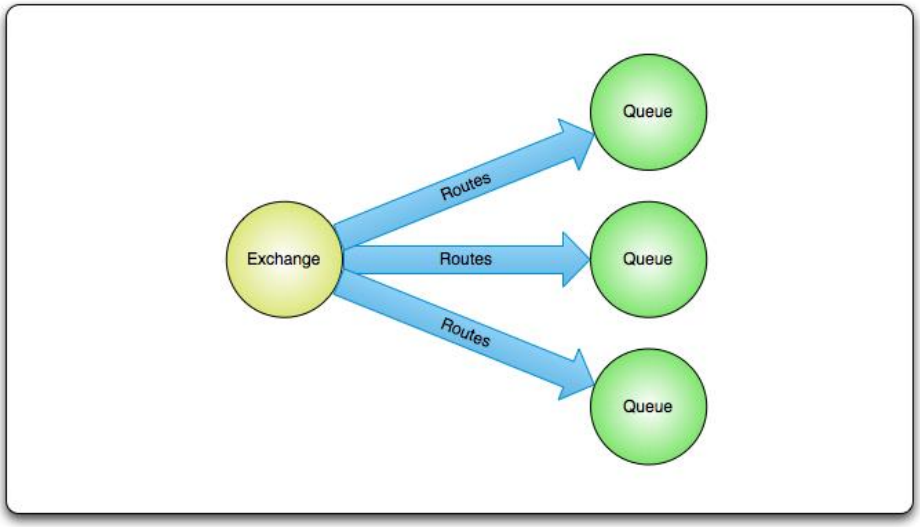

Figure 16: Fanout exchange routing

6.1.4 Topic Exchange

This has been explained in 3.2.

6.1.5 Headers Exchange

A headers exchange is designed for routing on multiple attributes that are more easily expressed as message headers than a routing key. Headers exchanges ignore the routing key attribute. Instead, the attributes used for routing are taken from the headers attribute. A message is considered matching if the value of the header equals the value specified upon binding.

It is possible to bind a queue to a headers exchange using more than one header for matching. In this case, the broker needs one more piece of information from the application developer, namely, should it consider messages with any of the headers matching, or all of them? This is what the "x-match" binding argument is for. When the "x-match" argument is set to "any", just one matching header value is sufficient. Alternatively, setting "x-match" to "all" mandates that all header pairs (key, value) must match.

Headers exchanges can be looked upon as "direct exchanges on steroids". Because they route based on header values, they can be used as direct exchanges where the routing key does not have to be a string; it could be an integer or a hash (dictionary) for example.

Header exchange is ideal for content based publish/subscriber. In way that our producer can put key-value (KV) pairs in the header of the message that describe the content. So consumer can receive only needed messages.

But because of MQTT we need to use TOPIC exchange!

We can make some connection so MQTT sends to TOPIC exchange, that exchange forward everything to HEADER exchange that is upstream and then consumer collect data from federated exchange by header filtering. This will be consider for second release of this deliverable.

6.2 Queues

6.2.1 Queue Names

Queue names starting with "amq." are reserved for internal use by the broker. Attempts to declare a queue with a name that violates this rule will result in a channel-level exception with reply code 403 (ACCESS_REFUSED).

6.2.2 Queue Durability

Durable queues are persisted to disk and thus survive broker restarts. Queues that are not durable are called transient. Not all scenarios and use cases mandate queues to be durable.

Durability of a queue does not make *messages* that are routed to that queue durable. If broker is taken down and then brought back up, durable queue will be re-declared during broker startup, however, only *persistent* messages will be recovered.

6.3 Routing

6.3.1 Port Access

SELinux, and similar mechanisms may prevent RabbitMQ from binding to a port. When that happens, RabbitMQ will fail to start. Firewalls can prevent nodes and CLI tools from communicating with each other. You should make sure the following ports can be opened:

- 4369: epmd, a peer discovery service used by RabbitMQ nodes and CLI tools
- 5672, 5671: used by AMQP 0-9-1 and 1.0 clients without and with TLS
- 25672: used by Erlang distribution for inter-node and CLI tools communication and is allocated from a dynamic range (limited to a single port by default, computed as AMQP port + 20000). See networking guide for details.
- 35672-35682: used by CLI tools (Erlang distribution client ports) for communication with nodes and is allocated from a dynamic range (computed as Erlang dist port + 10000 through dist port + 10010). See networking guide for details.
- 15672: HTTP API clients and rabbitmqadmin (only if the management plugin is enabled)
- 61613, 61614: STOMP clients without and with TLS (only if the STOMP plugin is enabled)
- 1883, 8883: (MQTT clients without and with TLS, if the MQTT plugin is enabled
- 15674: STOMP-over-WebSockets clients (only if the Web STOMP plugin is enabled)
- 15675: MQTT-over-WebSockets clients (only if the Web MQTT plugin is enabled)

It is possible to configure RabbitMQ to use different ports and specific network interfaces.

7. Appendix 2 - Distributed RabbitMQ brokers

The content of this section is based on [14] and [42].

AMQP and the other messaging protocols supported by RabbitMQ via plug-ins (e.g. STOMP), are (of course) inherently distributed - it is quite common for applications from multiple machines to connect to a single broker, even across the internet.

Sometimes however it is necessary or desirable to make the RabbitMQ broker itself distributed. There are three ways in which to accomplish that: with clustering, with federation, and using the shovel. This is described below.

Note that you do not need to pick a single approach - you can connect clusters together with federation, or the shovel, or both.

7.1 Bindings

Bindings are rules that exchanges use (among other things) to route messages to queues. To instruct an exchange E to route messages to a queue Q, Q has to be bound to E. Bindings may have an optional routing key attribute used by some exchange types. The purpose of the routing key is to select certain messages published to an exchange to be routed to the bound queue. In other words, the routing key acts like a filter.

Having this layer of indirection enables routing scenarios that are impossible or very hard to implement using publishing directly to queues and also eliminates certain amount of duplicated work application developers have to do.

If AMQP message cannot be routed to any queue (for example, because there are no bindings for the exchange it was published to) it is either dropped or returned to the publisher, depending on message attributes the publisher has set.

7.2 Clustering

Clustering connects multiple machines together to form a single logical broker. Communication is via Erlang message-passing, so all nodes in the cluster must have the same Erlang cookie. The network links between machines in a cluster must be reliable, and all machines in the cluster must run the same versions of RabbitMQ and Erlang.

Virtual hosts, exchanges, users, and permissions are automatically mirrored across all nodes in a cluster. Queues may be located on a single node, or mirrored across multiple nodes. A client connecting to any node in a cluster can see all queues in the cluster, even if they are not located on that node.

Typically you would use clustering for high availability and increased throughput, with machines in a single location.

7.3 Federation

The content of this section is based on [8].

Federation allows an exchange or queue on one broker to receive messages published to an exchange or queue on another (the brokers may be individual machines, or clusters). Communication is via AMQP (with optional TLS), so for two exchanges or queues to federate they must be granted appropriate users and permissions.

Federated exchanges are connected with one way point-to-point links. By default, messages will only be forwarded over a federation link once, but this can be increased to allow for more complex routing

topologies. Some messages may not be forwarded over the link; if a message would not be routed to a queue after reaching the federated exchange, it will not be forwarded in the first place.

Federated queues are similarly connected with one way point-to-point links. Messages will be moved between federated queues an arbitrary number of times to follow the consumers.

Typically you would use federation to link brokers across the internet for pub/sub messaging and work queueing.

7.4 The Shovel

Connecting brokers with the shovel is conceptually similar to connecting them with federation. However, the shovel works at a lower level. Whereas federation aims to provide opinionated distribution of exchanges and queues, the shovel simply consumes messages from a queue on one broker, and forwards them to an exchange on another. Typically you would use the shovel to link brokers across the internet when you need more control than federation provides. Dynamic shovels can also be useful for moving messages around in an ad-hoc manner on a single broker.

8. Appendix 3 - Install and use RabbitMQ on Ubuntu

8.1 Step 1 – Install Erlang

$ wget https://packages.erlang-solutions.com/erlang-solutions_1.0_all.deb
$ sudo dpkg -i erlang-solutions_1.0_all.deb
$ sudo apt-get update
$ sudo apt-get install erlang erlang-nox
$ erl -eval 'erlang:display(erlang:system_info(otp_release)), halt().' -noshell

8.2 Step 2 – Install RabbitMQ Server

$ echo 'deb http://www.rabbitmq.com/debian/ testing main' | sudo tee /etc/apt/sources.list.d/rabbitmq.list
$ wget -O- https://www.rabbitmq.com/rabbitmq-release-signing-key.asc | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install rabbitmq-server

8.3 Step 3 – Manage RabbitMQ Service

$ sudo systemctl enable rabbitmq-server
$ sudo systemctl start rabbitmq-server

---

localhost:5672 AMQP Connection

$ sudo systemctl stop rabbitmq-server

8.4 Step 4 – Create Admin User in RabbitMQ

$ sudo rabbitmqctl add_user admin <password>
$ sudo rabbitmqctl set_user_tags admin administrator
$ sudo rabbitmqctl set_permissions -p / admin ".*" ".*" ".*"

8.5 Step 5 – Setup RabbitMQ Web Management Console

$ sudo rabbitmq-plugins enable rabbitmq_management

---

localhost:15672 username:"guest" password:"guest" or created admin

---

http://localhost:15672/cli - download rabbitmqadmin

8.6 Step 6 – Run Code from Intellj

8.7 Step 7 - Enabling MQTT Plugin- http://www.rabbitmq.com/mqtt.html

$ sudo rabbitmq-plugins enable rabbitmq_mqtt

---

localhost:1883 MQTT Connection

8.8 Uninstall

$ sudo apt-get purge --auto-remove rabbitmq-server

$ sudo apt-get purge --auto-remove erlang

More information can be found at:

- https://www.howtoinstall.co/en/ubuntu/xenial/rabbitmq-server?action=remove
- https://www.howtoinstall.co/en/ubuntu/xenial/erlang?action=remove

8.9 Additional commands

8.9.1 Listing Consumers, Queues, Exchanges, Bindings, Hashes, Ciphers.

$ sudo rabbitmqctl list_{ consumers, queues, exchanges, bindings, hashes, ciphers}

8.9.2 Forgotten Acknowledgment

$ sudo rabbitmqctl list_queues name messages_ready messages_unacknowledged

More information can be found at:

- http://www.rabbitmq.com/cli.html
- https://www.rabbitmq.com/rabbitmqctl.8.html

References

[1] http://activemq.apache.org/

[2] http://activemq.apache.org/security.html

[3] http://opensourceforu.com/2015/12/an-introduction-to-apache-activemq/

[4] http://activemq.apache.org/getting-started.html

[5] https://activemq.apache.org/artemis/docs/1.0.0/security.html

[6] http://activemq.apache.org/protocols.html

[7] http://activemq.apache.org/how-do-distributed-queues-work.html

[8] http://activemq.apache.org/clustering.html

[9] http://activemq.apache.org/cross-language-clients.html

[10] http://activemq.apache.org/topologies.html

[11] https://www.rabbitmq.com/

[12] https://www.rabbitmq.com/access-control.html

[13] https://www.rabbitmq.com/protocols.html

[14] https://www.rabbitmq.com/distributed.html

[15] https://www.rabbitmq.com/devtools.html

[16] https://www.rabbitmq.com/monitoring.html

[17] http://www.rabbitmq.com/management.html

[18] https://www.rabbitmq.com/authentication.html

[19] https://www.rabbitmq.com/tutorials/tutorial-six-java.html

[20] https://docs.wso2.com/display/EI610/Remote+Procedure+Call%28RPC%29+with+RabbitMQ

[21] http://activemq.apache.org/how-should-i-implement-request-response-with-jms.html

[22] https://access.redhat.com/documentation/en-US/Fuse_ESB/4.4.1/html-single/ActiveMQ_Security_Guide/index.html

[23] https://www.rabbitmq.com/how.html#cloud-configurations

[24] https://www.rabbitmq.com/resources/RabbitMQ_MessagingInTheCloud_VMworld_2010_MS.pdf

[25] https://kafka.apache.org/documentation/

[26] https://www.rabbitmq.com/mqtt.html#custom-exchanges

[27] https://www.rabbitmq.com/tutorials/tutorial-five-java.html

[28] https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices

[29] https://www.rabbitmq.com/tutorials/amqp-concepts.html

[30] http://www.rabbitmq.com/api-guide.html#recovery

[31] https://www.rabbitmq.com/ssl.html

[32] http://searchsecurity.techtarget.com/definition/Transport-Layer-Security-TLS

[33] https://www.rabbitmq.com/federation.html

[34] https://www.rabbitmq.com/federated-exchanges.html

[35] https://www.rabbitmq.com/federated-queues.html

[36] https://www.rabbitmq.com/federation-reference.html#upstreams

[37] https://www.rabbitmq.com/uri-query-parameters.html

[38] https://social.technet.microsoft.com/wiki/contents/articles/34082.understanding-docker-for-absolute-beginners.aspx

[39] https://github.com/openssl/openssl/blob/master/apps/openssl.cnf

[40] https://www.rabbitmq.com/tutorials/amqp-concepts.html

[41] https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html

[42] https://www.rabbitmq.com/clustering-ssl.html

[43] https://docs.docker.com/engine/docker-overview/

Tutorials

- https://www.rabbitmq.com/tutorials/tutorial-one-java.html
- https://www.youtube.com/watch?v=deG25y_r6OY
- https://www.youtube.com/watch?v=XjuiZM7JzPw
- https://dzone.com/articles/connect-rabbitmq-using-scala
- https://www.playframework.com/modules/rabbitmq-0.0.9/home
- http://activemq.apache.org/hello-world.html
- http://tech.lalitbhatt.net/2014/08/activemq-introduction.html
- https://www.youtube.com/watch?v=oaegBVoVvlQ
- https://www.youtube.com/watch?v=3T-lDT-vALE
- https://www.playframework.com/modules/camel-0.2/home
- https://www.youtube.com/watch?v=OVXnwZ3gbD4&list=PL7aRHNGCnFZUF2dH7J7bqeU9-GDdiHF-R

Additional material used for this deliverable:

- https://stackoverflow.com/questions/15150133/jms-and-amqp-rabbitmq
- https://stackshare.io/stackups/activemq-vs-kafka-vs-rabbitmq
- https://www.ekito.fr/people/mqtt-benchmarks-rabbitmq-activemq/
- http://vasters.com/blog/From-MQTT-to-AMQP-and-back/
- http://activemq.apache.org/features.html
- http://activemq.apache.org/faq.html
- http://activemq.apache.org/how-do-i-preserve-order-of-messages.html
- https://activemq.apache.org/maven/apidocs/overview-summary.html