Ecosystem for COllaborative Manufacturing PrOceSses – Intra- and Interfactory Integration and AutomaTION
(Grant Agreement No 723145)

D6.2 Real-time event broker II

Date: 2018-11-16

Version 1.0

Published by the COMPOSITION Consortium

Dissemination Level: Public

Co-funded by the European Union's Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 723145

Document control page

Document file: D6.2 Real-time event broker II_v1.0.docx
Document version: 1.0
Document owner: CNET

Work package: WP6 - COMPOSITION Collaborative Ecosystem
Task: T6.1 - Real-time event brokering for factory interoperability

Deliverable type: OTHER

Document status: ☒ Approved by the document owner for internal review
☒ Approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Mathias Axling (CNET)	2018-10-01	TOC, initial content
0.11	Mathias Axling (CNET)	2018-10-15	Minor edits, CNET contributions
0.2	Alexandros Nizamis (CERTH)	2018-10-21	CERTH contributions
0.3	Paolo Vergori, Giuseppe Pacelli	2018-10-24	ISMB contributions
0.4	Mathias Axling (CNET)	2018-10-26	Merged content, editing
0.5	Mathias Axling (CNET)	2018-10-28	Additional CNET contributions
0.6	Ignacio Gonzalez Fernandez	2018-10-29	ATOS contribution
0.7	Peter Rosengren (CNET)	2018-11-06	Additional CNET contributions
0.9	Mathias Axling (CNET)	2018-11-13	Ready for peer review
1.0	Peter Rosengren, Mathias Axling (CNET)	2018-11-16	Ready for submission to EC

Internal review history:

Reviewed by	Date	Summary of comments
Alexandros Nizamis, Vagia Rousopoulou (CERTH)	2018-11-14	The quality of the document is high and it can be submitted after small corrections in both content and syntax
Luis Martins (BSL)	2018-11-14	Minor formatting corrections/suggestions. Reviewed and approved for submission.

Index:

1 Executive Summary

This deliverable presents the current state of the design of the real-time event broker infrastructure of the COMPOSITION system at month 26 in the project. During the first year of the project, work was focused on evaluation of alternative implementation mechanisms, choice of protocols and the design of the real-time event broker integration with other components. During the second year of the project, implementation and deployment in test and pilot environments and scalability design have been performed, the information models have been refined, and extensions to the broker has been developed.

The real-time event broker (referred to in this document and other COMPOSITION documentation as the Message Broker) is a principal component in realizing both of COMPOSITION's main goals. The first goal is to integrate data along the value chain inside a factory into one integrated information management system (IIMS) combining physical world, simulation, planning and forecasting data. The goal of the IIMS is to enhance re-configurability, scalability and optimisation of resources and processes inside the factory and optimise manufacturing and logistics processes. Here the broker creates a secure, loosely coupled and scalable way to distribute data in the system at near real-time speeds. The second goal is to create a (semi-)automatic ecosystem, which extends the local IIMS concept to a holistic and collaborative system incorporating and interlinking both the supply and the value chains. This should be able to dynamically adapt to changing market requirements. Here, the broker provides the communication between the actors in this marketplace. These two brokers are separate instances of the same component.

RabbitMQ[1] is the core of this component. It is a widely used, open, standards-based product previously used in FITMAN (EU FP7 2013-2015)[2]. It provides support for multiple protocols and horizontal scalability. COMPOSTION significantly contributes to this open and scalable design. Through adapters developed in COMPOSITION, we extend RabbitMQ with an integrated security framework and using blockchain-based log functionality on multiple levels.

The Message Broker has been deployed as a set of Docker images at the COMPOSITION intra-factory test server and intra- and inter-factory sites.

[1] http://www.rabbitmq.com/
[2] http://catalogue.fitman.atosresearch.eu/catalogue.fitman.atosresearch.eu/enablers/secure-event-management.html

2 Terminology

Table 1: Acronyms and terminology used in this report.

Term	Definition
Agent Container	An agent container is a set of intelligent agents interacting through the same, shared transport protocol and referring to shared platform services such as the Directory Facilitator, DF and the Agent Management Service, AMS.
AMQP	Advanced Message Queuing Protocol, an open standard application layer protocol for message-oriented middleware (ISO/IEC 19464).
Closed Marketplace	COMPOSITION Marketplace owned by one stakeholder and typically offered to a trusted subset of other COMPOSITION stakeholders. The Closed Marketplace can be public or private. A public, closed market will accept join requests by agents living in the Open Marketplace A private, closed marketplace will accept agents only by invitation. A Closed Marketplace is structurally equivalent to the open marketplace A Closed Marketplace is physically separated to the Open Marketplace and has typically a separate infrastructure of shared platform services including the broker, AMS, DF, etc.
COMPOSITION Ecosystem	The supply chain part of a COMPOSITION system, implemented by a COMPOSITION Marketplace and involving suppliers, producers and logistics services.
COMPOSITION Marketplace	A COMPOSITION Marketplace is an agent container.
Integrated Information Management System (IIMS)	The Integrated Information Management System is a digital automation framework that optimizes the manufacturing processes by exploiting existing data, knowledge and tools to increase productivity and dynamically adapt to changing market requirements.
IoT	Internet Of Things
JSON	JavaScript Object Notation is an open-standard human-readable data format.
JSON-LD	JavaScript Object Notation for Linked Data I a standard for embedding metadata in JSON documents, linking them to an RDF model.
Message broker	A message broker is an architectural pattern for message validation, transformation and routing. A message broker can receive messages from multiple destinations, determine the correct destination and route the message to the correct channel. Used interchangeably with “Real-time event broker” in this report.

MQTT	MQ Telemetry Transport or Message Queue Telemetry Transport. A binary, lightweight messaging protocol for small sensors and mobile devices (ISO/IEC PRF 20922).
OPC-UA	OPC Unified Architecture, IEC 62541, is an open, SOA-based, platform-independent machine to machine communication protocol for industrial automation.
RDF-A	Resource Description Framework in Attributes is a W3C Recommendation for embedding metadata in HTML and XML documents types, linking them to an RDF model.
RPC	Remote Procedure Call (Request-response communication)
SSL	Secure Sockets Layer is a standard technology for securing internet connections.
TLS	Transport Layer Security is the successor to version 3 of the SSL protocol,
Virtual Marketplace	A Virtual Marketplace, or group is a "multicast" group of agents interacting with each other in the context of a negotiation. The group can be: • persistent over negotiations or • just be defined for a single negotiation exchange. A Virtual Marketplace lives in, and exploits the infrastructure of the Open Marketplace.
XML	Extensible Markup Language is an open-standard human-readable data format.

3 Introduction

3.1 Purpose, context and scope of this deliverable

This deliverable presents the actions performed, results and planned future work on the design of the real-time event broker infrastructure of the COMPOSITION system. The work has been carried out mainly in Work Package 6 (WP6), “COMPOSITION Collaborative Ecosystem”, and to some extent also in Work Package 2 (WP2), “Use Case Driven Requirements Engineering and Architecture”, in the COMPOSITION work package structure defined by the project specification (COMPOSITION, 2016). The main tasks involved are:

- Task 6.1 “Real-time event brokering for factory interoperability”
- Task 6.2 “Cloud Infrastructures for Inter-Factory Data Exchange”
- Task 6.5 “Brokering and Matchmaking for Efficient Management of Manufacturing Processes”

This report follows up D6.1 “Real-time event broker I”, which provided the status at month 14 in the project. Sections that are still relevant will be only slightly updated.

Information related to this deliverable has been reported in D2.3 “The COMPOSITION Architecture Specification I“, D2.4 “The COMPOSITION Architecture Specification II“ and D6.3 “COMPOSITION Marketplace I”. The elaboration of the broker component has been performed using input from D2.1 “Industrial use cases for an Integrated Information Management System” and D2.2 “Initial requirements specification” as well as the design of other components. The communication design is tightly integrated with the Security Framework, reported in D4.1 “Design of the Security Framework I”, D4.2 “Design of the Security Framework II” and D4.4 “Prototype of the Security Framework I”. This report will include an overview of this integration. Previously used material has been re-used where appropriate.

3.2 Content and structure of this deliverable

The structure of this deliverable is aligned with that of the architecture description deliverable D2.4 “The COMPOSITION Architecture Specification II“. In some cases, information in the architecture description deliverable has been repeated in this one for clarity and readability. In other cases, we have referred to the architecture deliverable. The intent of this structure is to provide a more in-depth view of the Real-time Event Broker, while maintaining the context of the overall system architecture.

Sections present in D6.1 necessary for context will be included in this report. In-depth sections have been moved to appendixes.

The remainder of the document is structured as follows:

Section 4 – Provides an overview of the real-time event broker, or message broker, domain.

Section 5 – Summarises the architectural design concerns relevant to the design of the real-time event broker.

Section 6 – Describes the design decisions taken for the real-time event broker and the functional view.

Section 7 – Describes the work on information models relevant to the real-time event broker; the information view.

Section 8 – Provides an overview of the deployment view.

Section 9 – Addresses the scalability perspective.

Section 10 - Addresses the security perspective.

Section 11 - Presents a summary of the current state of development with conclusions.

4 Background

The architecture inception phase reported in the COMPOSITION project specification (COMPOSITION, 2016) identifies real-time event brokering as a principal component in the IIMS and marketplace design. The role of the component was to support interoperability, message brokering, message translation and annotation, integration with support for hererogenous protcols and formats. It is the primary communication mechanism and provides the communication and integration infrastructure of the COMPOSITION system. The term message broker is used interchangeably with real-time event broker throughout this report.

4.1 Message broker

A message broker is an architectural pattern that decouples the destination of a message from the sender. Messages can be received from multiple destinations and routed to the correct destination while implementing a multitude of messaging patterns (Hohpe & Woolf, 2003). The message broker maintains central control over the flow of messages and provides the means to define interactions between the decoupled components. Typical actions on messages performed by the message broker includes validation, transformation to other formats and representations, and routing of messages to any number of recipients based on topic, headers or content.

This is a common pattern for distributed and asynchronous systems and has been implemented in previous IoT projects (e.g. Hydra[3] (LinkSmart), EBBITS[4], ALMANAC[5]) and commercial platforms (e.g. Microsoft Azure IoT Hub[6], Amazon AWS IoT Message Broker[7]). However, this document will describe some significant additions and integrations that has been performed in COMPOSITION that contribute to added value from the component.

4.2 RAMI4.0 Communication Layer

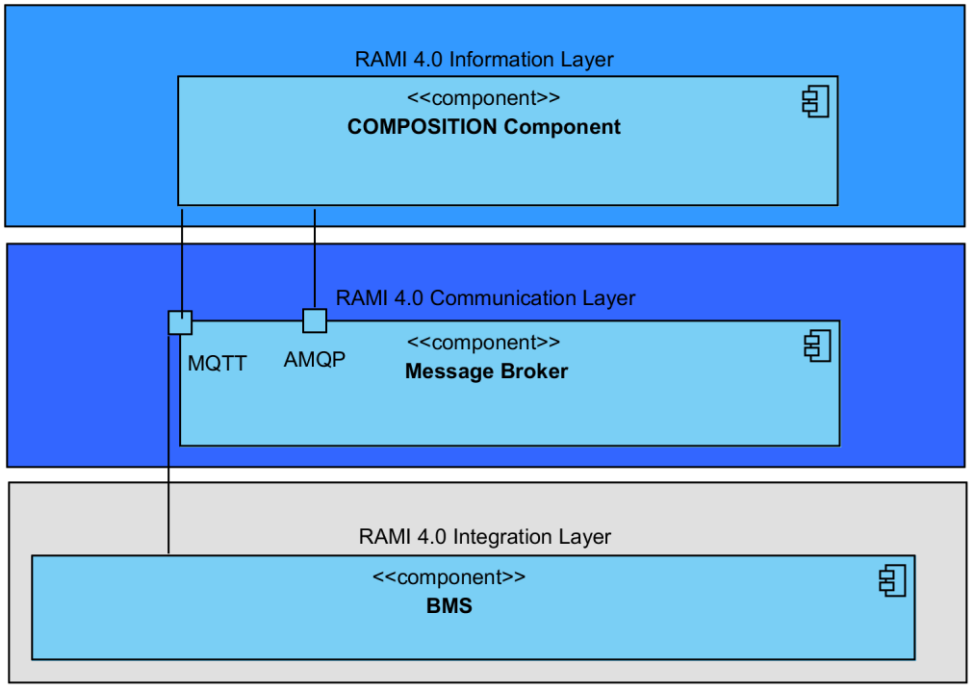

Figure 1: Message Broker in RAMI 4.0 Functional Layers

In the RAMI4.0 context, the message broker belongs in the Communication Layer, which performs transmission of data and files (COMPOSITION D2.3, 2017), see Figure 1. It standardizes the communication

[3] http://cordis.europa.eu/project/rcn/79422_en.html
[4] http://cordis.europa.eu/project/rcn/96598_en.html
[5] http://cordis.europa.eu/project/rcn/109709_en.html
[6] https://azure.microsoft.com/en-us/services/iot-hub/
[7] https://docs.aws.amazon.com/iot/latest/developerguide/iot-message-broker.html

from the Integration Layer, providing uniform data formats, protocols and interfaces in the direction of the Information Layer. It also provisions the services for controlling the Integration Layer. The administrative shell is the virtual representation of an asset describing the data and functions of the asset. Protocols which currently have example mapping to the RAMI4.0 Administrative shell, describing how to realize the shell functionality, are MQTT and UPC-UA (MQTT and AMQP are a part of the OPC UA specification as part of the publish/subscribe extension). In composition, the lightweight MQTT protocol will be used for sensor-machine communication in the intra-factory IIMS. The cloud-based Marketplace will use AMQP, suitable for server communication.

4.3 Protocols

The selected implementation of the communication mechanism supports a number of communication protocols by configuration. We list here the ones that are relevant to the pilot deployments and early exploitation of COMPOSITION.

4.3.1 MQTT

MQ Telemetry Transport or Message Queue Telemetry Transport (MQTT), ISO/IEC 20922, is a simple, lightweight protocol running on TCP/IP using a publish-subscribe model. It is designed to minimise network bandwidth and device resource requirements, making it very suitable for collecting data from edge network devices like sensors. Implementations of MQTT brokers and clients are available on multiple platforms.

MQTT is also available to web browser clients over web sockets[8] for display of real-time data streams in user interfaces. This will be supported in COMPOSITION.

4.3.2 AMQP

The Advanced Message Queuing Protocol (AMQP), ISO/IEC 19464:2014, is an open standard application layer protocol for message-oriented middleware. While not a light-weight protocol like MQTT, AMQP allows for a variety of message queuing and routing patterns (including publish-and-subscribe) while stressing reliability and security. AMQP declares a model, protocol methods, format (application payload is opaque to the broker, however) and type system that broker and client implementations must conform to for different implementations to be interoperable. Both versions 0-9-1 and 1.0 are supported by a number of software vendors.

4.3.3 OPC-UA

OPC Unified Architecture (OPC UA), IEC 62541, is an open, SOA-based, platform-independent machine to machine communication protocol for industrial automation. RAMI4.0 has OPC-UA confirmed as an appropriate design mechanism for the Communication Layer. It is a multi-part specification defining e.g. an Information Model, Services and Security Model. MQTT and AMQP are a part of the OPC UA PubSub specification[9]. There is both a binary and HTTP protocol specified for communication. A connector to OPC-DA and OPC-UA has been developed by COMPOSITION[10] to meet exploitation concerns.

[8] https://www.eclipse.org/paho/clients/js/
[9] https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-14-pubsub
[10] The development stakeholder is a member of the OPC-UA Foundation, which allows use of OPC Foundation source code in commercial products and Distribution of OPC Foundation source code.

5 Design Concerns

The COMPOSITION system has three main stakeholder groups with concerns for the system stakeholders. These groups are the acquirers, the developers/maintainers and the users (D2.3 The COMPOSITION Architecture Specification). The goals and concerns of the acquirers of the COMPOSITION system are stated in the strategic and technical objectives in the project specification (COMPOSITION, 2016). The message broker is instrumental in meeting the following goals:

- Technical Objective 1.1: Innovate and extend the FI-WARE and FITMAN catalogues of Generic Enablers with an innovative CPS-aware library of open, standard connectors specialised for real-time architectures for interoperability in manufacturing to ease the integration and coupling of data, information and knowledge from existing, heterogeneous, sources in the factory.
- Technical Objective 2.1: Design and implement *a Log Oriented Architecture*, based on blockchain technology, ensuring the trusted, secure and automated exchange of supply chain data among all authorized stakeholders, to connect factories and support interoperability and product traceability along the supply chain.
- Technical Objective 2.2: Provide *end-to-end security* from factory floor to cloud services encompassing major mechanisms in a seamless and fully integrated manner including authentication and access control, transport security, as well as system security, while maintaining suitable levels of IPR and knowledge protection.

This puts emphasis on use of open standards, extensibility, and ease of integration of the chosen implementation of the message broker. Multiple protocols and formats should be supported.

The developer/maintainer stakeholders, i.e. the technical partners, have concerns regarding extensibility and compatibility. In so far as possible, compatibility with the existing products is desired, and stakeholders should be able to supply components and services complementing and extending the system on the COMPOSITION aftermarket. The loose coupling between components is desirable if new third-party products are to be integrated in the platform after the end of the project, as part of the COMPOSITION ecosystem. The broker should use open standards, especially consider ones already compatible with the supplied components, and provide support for integration on multiple software platforms. Security should be seamlessly integrated in the entire system and allow for integration of components from external sources into the COMPOSITION platform. The use of open standards is thus a requirement from the security perspective as well.

The user stakeholder group are the pilot partners and future users of the system, whose concerns are mainly expressed in the scenarios, use cases (D2.1 “Industrial Use Cases for an Integrated Information Management System”) and requirements (D2.2 “Initial requirements specification”). These deliverables capture the needs of the manufacturing industry and the priorities of the pilot partners. Security, scalability, extensibility and ease of integration with existing systems are concerns expressed in these requirements[11].

Licensing must allow for commercial usage of individual components or the entire system. Incorporating or applying open source licensing affecting the possibility of commercial exploitation, such as GPL, is explicitly forbidden (COMPOSITION, 2016).

[11] JIRA requirements COM-35 to COM-43in particular relate to the Message Broker.

6 Design of the Message Broker

This section describes the design decisions and internal structure of the Message Broker, interfaces, interactions dependencies, and dependent components. I.e., it provides the functional view of the architecture of the Message Broker.

6.1 Overview

For COMPOSITION purposes it is a required to support the most common IoT Messaging Protocols to integrate data from multiple sources in the intra factory and support flexible component integration. There is also the need to be able to secure the messaging using the services provided by the COMPOSITION security framework. Furthermore, as an intermediary, decoupling system components, the Message Broker also provides the means to manage scalability in a consistent manner. Thus, the general communication mechanism for the system will be data-centric and messaging-based. Factory data is published and subscribing components (performing e.g. processing, analytical or supervisory functions) consume this data without direct addressing between components. This is built using standard message broker components with extensions for security, multi-protocol and multi-format support.

In the COMPOSITION Marketplace, the agents exchange supply-chain formation and execution data, inter-factory data shared with selected stakeholders and reputation data. All communication takes the form of CXL messages. The ability to correctly exchange such messages is the only requirement on a marketplace agent, which allows the actual agent implementation to use any language or platform.

The AMQP protocol will be used for component communication and message routing. It is a very flexible protocol with high-level configurability for different message routing schemes and emulation of other protocols. As MQTT may be transparently used by clients on top of an AMQP broker architecture, this protocol will be used for the data and components in the intra-factory IIMS, most of which already implement MQTT support.

The COMPOSITION Message Broker will be the communication mechanism in both in the intra factory and in the COMPOSITION Marketplace. Note that these will be two completely different instances, but they will provide the same function. They are configured individually, i.e. the components will be the same, but they will be used differently and deployed on different nodes and in different networks.

6.2 Core Message Broker Implementation

RabbitMQ was put forward as the core message broker implementation in the architecture inception phase (COMPOSITION, 2016), and COMPOSITION selected this implementation after evaluation other candidate implementations. The alternative implementations are described in section 12, “Appendix 1: Candidate message broker implementations”.

6.2.1 RabbitMQ

RabbitMQ[12] is the implementation mechanism for the core of the message broker. It is a widely used open source message broker [13] supplied under the Mozilla Public License with an extensible architecture. It implements the AMQP 0-9-1 protocol[14] and can through extension mechanisms, plugins, support the most common messaging protocols, e.g. MQTT, STOMP and XMPP. Extensions and adapters can be written to support other messaging patterns, protocols and security management solutions.

RabbitMQ implements AMQP 0-9-1 and the AMQP concepts of messages, producers, exchanges, queues and consumers, see Figure 2. Each of these exists within an administrative unit called a virtual host. A broker may contain several virtual hosts, and users defined in RabbitMQ can be assigned read, write and administrative rights per host.

[12] https://www.rabbitmq.com/
[13] At the time of writing 35.000 production deployments , https://www.rabbitmq.com/
[14] http://www.amqp.org/sites/amqp.org/files/amqp0-9-1.zip

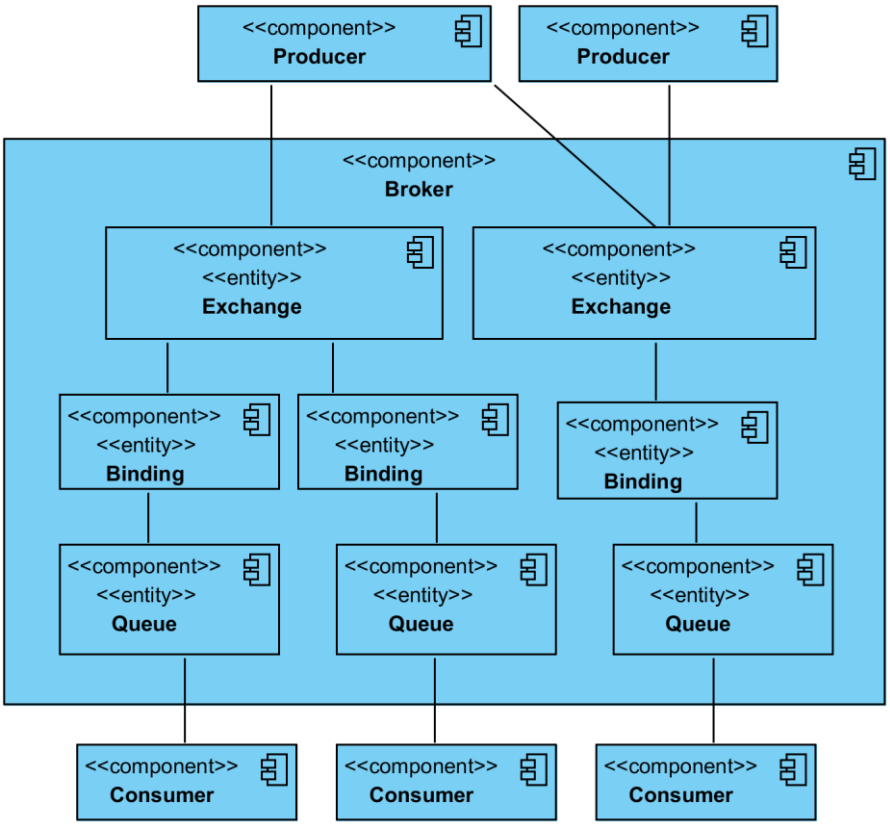

Figure 2: Example of exchanges bindings and queues

A publisher – an application that produces messages - sends a message to an exchange, where it is routed to queues. The message is then pushed to (or pulled by) a consumer – an application that processes messages - for processing. The producer, consumers and the broker may all reside on different brokers. The configuration of exchanges and bindings may be done at setup or dynamically. It also offers a number of scalability mechanisms, described in section 9. A detailed description of these concepts of AMQP 0.9.1, and details of RabbitMQ was included in D6.1 and can be found in section 13, “Appendix 2: RabbitMQ”.

The SEM[15] (Secure Event Messaging) Specific Enabler (SE) developed in FITMAN (EU FP7 2013-2015) is built on top of RabbitMQ. As stated in (COMPOSITION, 2016), COMPOSITION will extend and build on the RabbitMQ multi queuing approach as developed in FIWARE and FITMAN. COMPOSTITION extends RabbitMQ with blockchain technology and Keycloak [16]identity and access management. COMPOSITION will also provide micro services for real-time message and protocol translation under high loads, where necessary.

6.2.2 Composition Message Broker

As was explained above the COMPOSITION Message Broker will be the communication mechanism in both in the intra factory and in the COMPOSITION market place. Note that these will be two completely different instances, but they will provide the same function. They are configured individually, i.e. the components will be the same, but they will be used differently and deployed on different nodes and in different networks. In this section we describe the core functionality and design of the Message Broker.

To increase the scalability of the Composition Message Broker solution we will implement a set of microservices which each has a dedicated task to perform. This will make it more efficient to process the different message queues and to scale out queue processing if necessary. Furthermore, we will implement a Fog/Edge Computing Architecture where processing can take place both at the edge level (i.e. the gateways) and in the cloud back end, see Figure 3.

[15] http://www.fiware4industry.com/?portfolio=secure-event-management-sem
[16] http://www.keycloak.org/

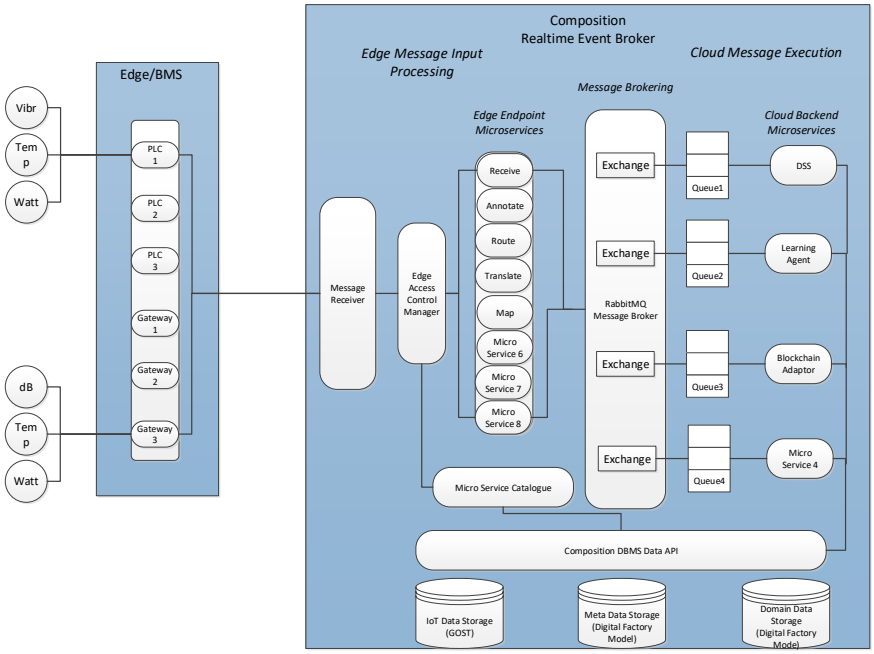

Figure 3: Schematic microservice architecture

6.3 Intra-factory Event Broker

Figure 4 depicts the components connected in the intra-factory scenario and the role the Message Broker plays.

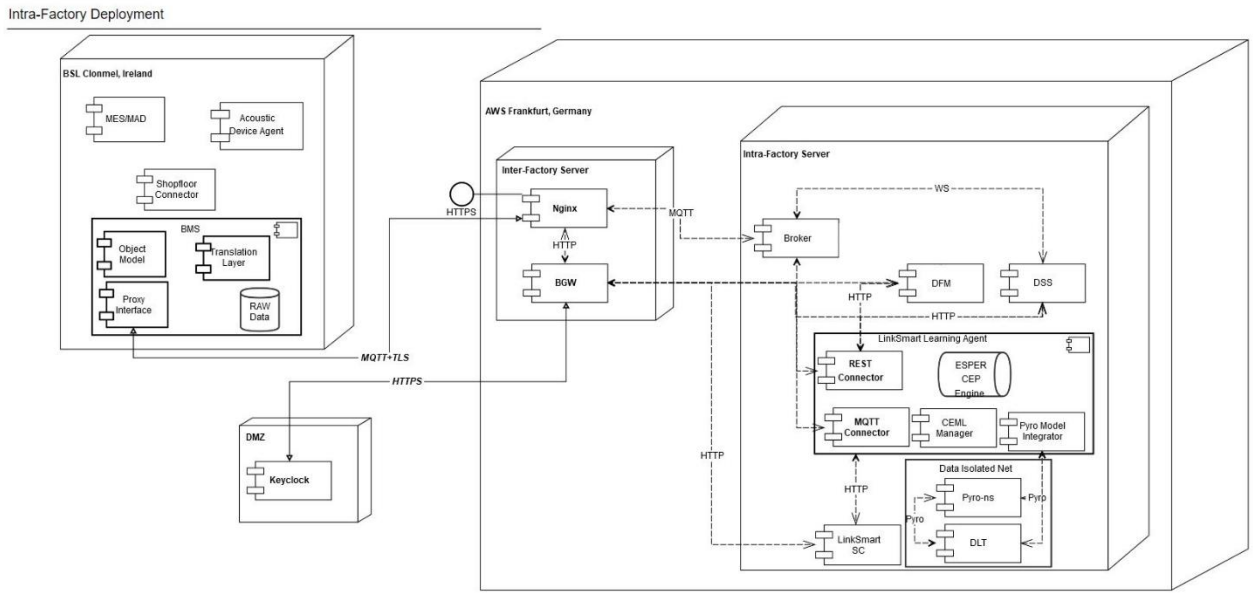

Figure 4: Intra-factory components

In task 6.1 the broker-based system has been characterized based on the design concerns described in section 5. In a first iteration it has also been tested and deployed in its first draft of the Docker container that later has been ported to the deployment environment, embedding the components in its final form. The

brokering system will be the main distribution point for the intra-factory scenario. In this way, there are few entry points to secure, whereas the reliability is demanded to the chosen encapsulation. In fact, even though the majority of the event brokers themselves do not implement recovery mechanisms, the Docker containers in which COMPOSITION components are deployed, allow for restart on crash functionalities to be available. This compromise provides enough reliability for the links, even without implementing either an actual recovery or a retransmission mechanism, for a scenario where scalability is the number one demand. Nevertheless, being also a single point of reference, makes it also a single point of failure for which work package 4 has proceeded by securing it by design, in order to minimize denial of service attacks.

In the following a simplified version of the information flow is depicted in Figure 4, in order to make clear the centrality of this component:

1. The information is produced from legacy sensors, aggregated by existing machineries and from newly deployed sensors at the shop floor level.
2. The information is buffered before being inputted to the Building Management System component that acts as a gateway for the intra-factory COMPOSITION ecosystem.
3. The information is transformed in actual data by the Building Management System that translates sensor levels based on each of their references into usable data with a corresponding measurement unit and proceeds by normalizing them.
4. The Building Management System is registered and authenticated against Keycloak with a token based access that allows an open authentication thanks to the mediation of the COMPOSITION security framework. It is, therefore, allowed to access topics that have registered on the LinkSmart catalogue at any time.
5. The Building Management System publishes each aggregated and consolidated sensor value to the corresponding topic through the broker-based system.
6. The broker based system will dispatch in real time the published data to each of the subscribers that are allowed to subscribe to the corresponding topic. By design, retained messaged are available, even if for sensor data it is function that is not usually required.
7. At runtime and in the final deployment every message that passes through the event broker should be signed by the sender and it is demanded to the receiver to verify it against the public key of each component. Ideally it will be store in one of the LinkSmart available catalogues, but at the time this deliverable is published the catalogue with the public keys has not been populated, yet.
8. Data is received through the event broker and almost consumed in real time by the designated subscribers.

Every component that needs to exchange information within the COMPOSITION intra-factory communication layer will be virtually demanded to use the event broker, registering a scope-based topic. COMPOSITION intra-factory components will leverage on this interconnection scalability, capability and most important without the burdensome of securing yet another communication channel that would not benefit from the enhancements of the security framework that mediates access token renewals and credentials retaining.

The COMPOSITION ecosystem uses two implementations of the same broker-based software, in order to distribute messages in both the intra and inter factory environment. As specified above, the former leverages on MQTT for implementing messages retaining and also demanding performance enhancements, the latter uses AMQP, sacrificing strictly related bandwidth performance over reliability and link recovery options.

Exceptions exists, as long as they maintain an equal or greater level of security. For instance, the Deep Learning Toolkit acts in this scenario as a hidden container that is not reachable though the broker. Its container is reachable only by the Learning Agent that is in charge of initializing, instantiating and triggering its functions though a remote procedure calls like system. Therefore, the two components are connected with Pyro and mapped on private logical link. Multiple instances of the Deep Learning Toolkit are instantiated at the same time and handled by the Learning Agents that is also in charge of gathering and filtering the results and propagating them to subscribers via the broker.

The Building Management System, as shown in Figure 4, is in charge of interconnecting two different words: the shop floor layer and the COMPOSITION intra-factory layer. The former is the layer in which the information is generated from both legacy and novel sensors, the latter is the broker-based interconnection system where all COMPOSITION components are built on the top of and on which their communications rely on.

The Digital Factory Model (DFM), from task 3.2 Integrated Digital Factory Models, contains a representation of the intra-factory real components (e.g. production lines, products, sensors, etc.); this information will be used both to propagate the input coming from the physical devices to the other components and to build the topics in the Message Broker to be subscribed for live data.

6.4 Inter-factory Market Event Broker

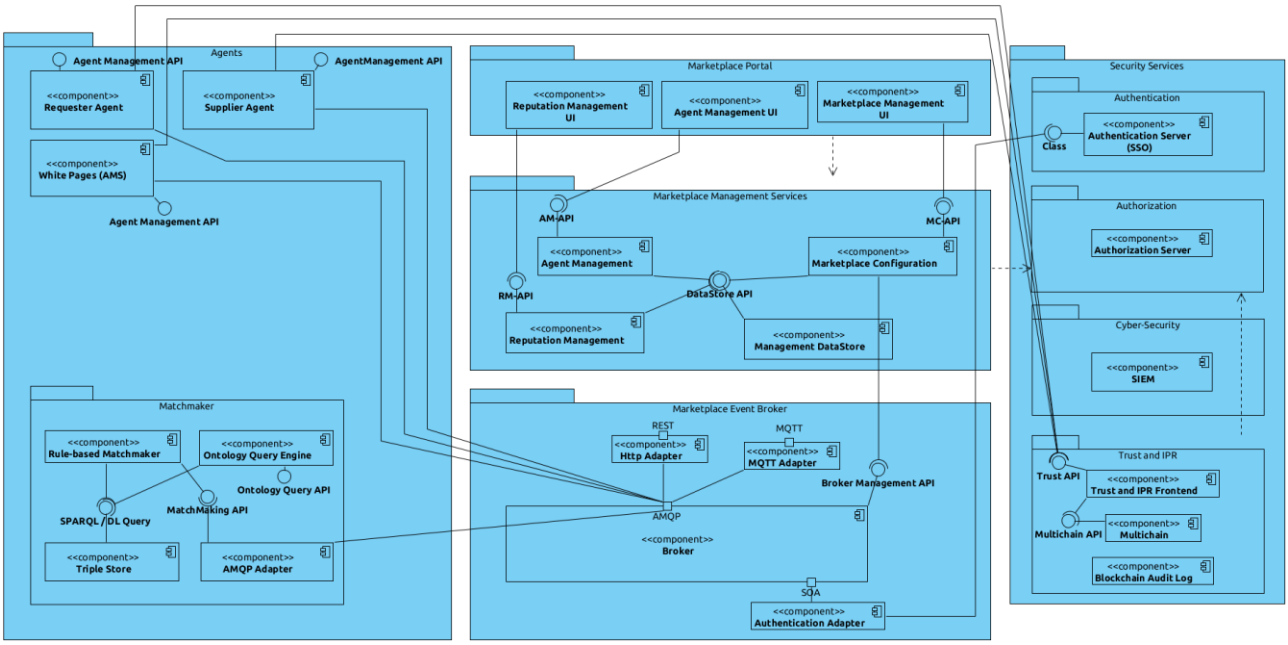

Figure 5: Marketplace components

The message broker instance used in the COMPOSITION collaborative marketplace, see Figure 5, is a separate instance (could be a cluster or federation for scalability purposes) that does not process intra-factory data. However, it uses most of the same COMPOSITION extensions. The Market Event Broker uses AMQP as primary communication protocol.

Agents use the Real-time event broker as a main hub for communication among them I.e. to send or receive any message. Agents in a Closed Marketplace use a separate broker while agents in a Virtual Marketplace communicate via the broker in the Open Marketplace.

The Real-time event broker is in general transparent to message content, as it only provides message dispatching, and mechanism for security protections are addressed as described in section 10.

In the first implementation, only the open marketplace is taken into account. It is therefore possible to exchange messages based on topics, using the Contract-NET interaction protocol. Virtual and closed marketplaces will be further implemented as a direct evolution of the open one.

6.5 COMPOSITION Extensions to RabbitMQ

6.5.1 RAAS

The RabbitMQ Authentication and Authorization Service (RAAS) provides the possibility to override the in-built authentication and authorization of RabbitMQ with an external mechanism. In COMPOSITION, integration with Keycloak has been implemented for authorization and the authorization service EPICA is to follow. This allows for integrated security in all parts of the system, and specifically a way to have a common security mechanism when deploying RabbitMQ federations – which is not otherwise possible.

6.5.2 Blockchain Adapter

The blockchain adapter, or Blockchain Audit Log, allows publishers and subscribers of messages to the Message Broker to log and verify the information by using the COMPOSITION blockchain, which cannot be tampered with. The adapter will be integrated with the Message Broker. The primary uses are:

- PKI infrastructure
    - Publish public keys to the blockchain
- Asset tracking blockchain
    - Asset tracking values from positioning system can be collected in a blockchain in the UC-BSL-3 use case. A demonstrator of this concept has been built, however, the decision on whether to implement this is still pending.
- Agent CXL message log
    - Hashes of the messages published to the Message Broker can be stored in the blockchain so that message authenticity can be verified.
- Marketplace Reputation Model
    - The reputation values assigned to other agents after business interactions can be stored in the blockchain.

The work on the COMPOSITION blockchain will be described in detail in deliverable D4.3 The COMPOSITION Blockchain.

6.5.3 Marketplace Data Sharing

As described in the COMPOSITION description of action, the system will provide mechanisms to share data from the intra-factory IIMS with chosen stakeholders in the marketplace. A factory may elect to share certain data with partners across the supply chain on a permanent basis or a single interaction, e.g., inventory data or scrap container fill levels. The data owner agent will route required information to the right recipient agents, through dedicated CXL messages. A sequence diagram illustrating the negotiation between agents using CXL to set up the data exchange can be seen in Figure 6.

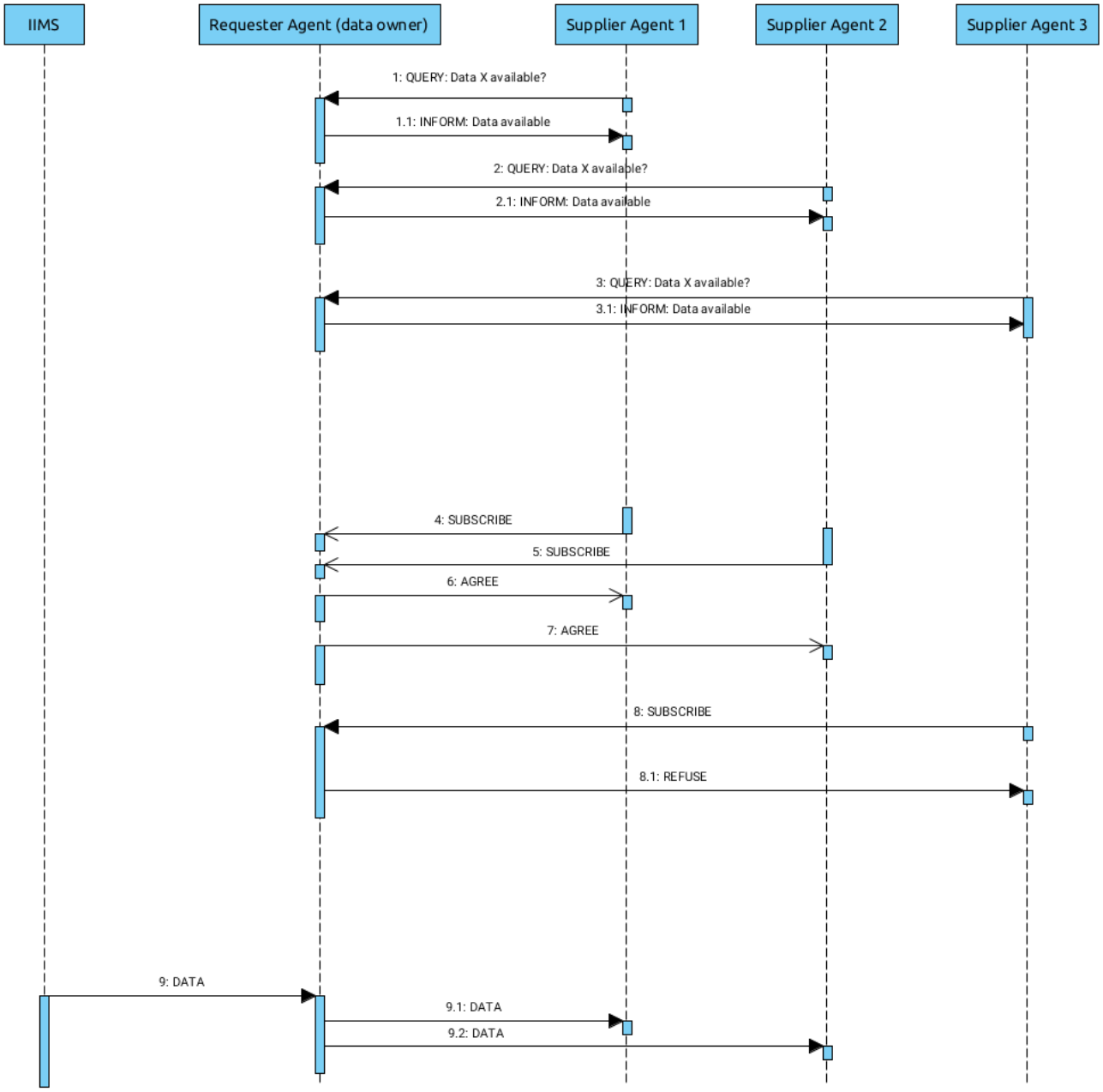

Figure 6: Data routing information flow

This data sharing mechanism is realized through the Message Broker (Figure 7). Integrated access control provided by the Security Framework makes it possible to set up an exclusive message queue for a business partner at the Marketplace Broker. Only the approved actors in the marketplace may publish and/or read data from the queue. The messages sent can also be secured by the possibility to store a hash of each message in the distributed blockchain ledger. As with all CXL messages, the agreement to share data itself may also be stored in the ledger to keep a non-repudiable audit trail of agreements.

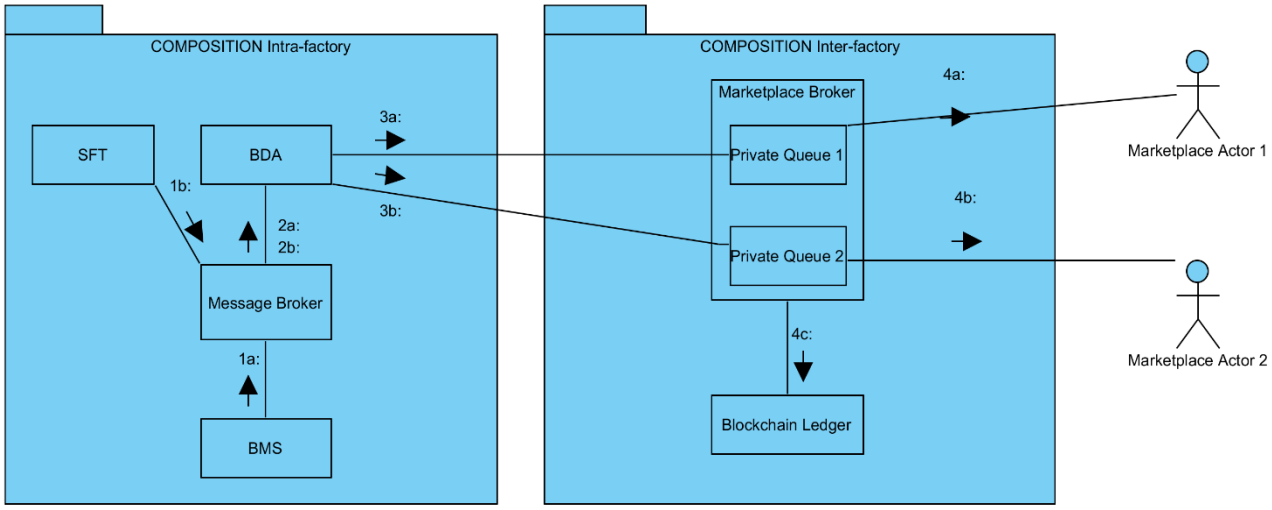

Figure 7: Simplified model of the marketplace data exchange design

6.5.4 REST Adapter

When the broker is used for inter-component communication, logical addressing of components can be used – a component identifier instead of a network address and port – decoupling components and providing a consistent way to address and find them for other components. Authentication and authorization can also be managed in a uniform manner via the broker. As extensibility is a concern for the developer stakeholders, it is desirable to use the broker for component communication. COMPOSITION components use either messaging (using MQTT or AMQP) or REST APIs. Routing the REST calls through the broker would make the most use of the integrated identity management and blockchain integration in COMPOSITION as well as introduce a level of decoupling of components, logical addressing of services and centralized management.

We have developed a transparent adapter for the request-response communication for HTTP REST services in COMPOSITION, corresponding to the SOAP tunnelling in (Milagro et al, 2008). This provides decoupling of services, logical addressing of services, discovery and an integrated security solution for HTTP, MQTT and AMQP communication. RabbitMQ already provides the mechanisms for RPC (Remote Procedure Call) style request-response messaging, including facilities for sending responses directly to the client channel without a client queue[17].

Drawbacks are that the load and dependency on the broker increases. Also, for this specific purpose, there may be better messaging solutions to build on, but these would not bring the benefits of the COMPOSITON extensions and centralized management.

When the broker is used for inter-component communication, logical addressing of components can be used – a component identifier instead of a network address and port – decoupling components and providing a consistent way to address and find them for other components. As mentioned above, authentication and authorization can also be managed in a uniform manner via the broker. As extensibility is a concern for the developer stakeholders, it is desirable to use the broker for all component communication. De-coupled, message-based communication suits the data-centric nature of the COMPOSITION system well, where several components independently subscribe to the same information. However, some exchanges are more suited for request-response interaction, e.g. REST APIs used for querying or administration. An adapter for RabbitMQ has been developed provide transparent request-response messaging (tentatively named “RabbitHole”). A bit simplified, this routes HTTP requests through an HTTP Proxy, resolves the base URL to a queue where the HTTP request is put. Clients (the REST services) may subscribe to the requests directed at them and return the response without exposing any public HTTP ports. The RPC Executer handles the request-response transparently to the service, see Figure 8. This implements request and response buffering, work queues, load balancing, logical addressing and the RAAS provides integrated security for all calls. Further work (outside the scope of COMPOSITION) will extend this to a general purpose microservice execution framework.

[17] https://www.rabbitmq.com/direct-reply-to.html

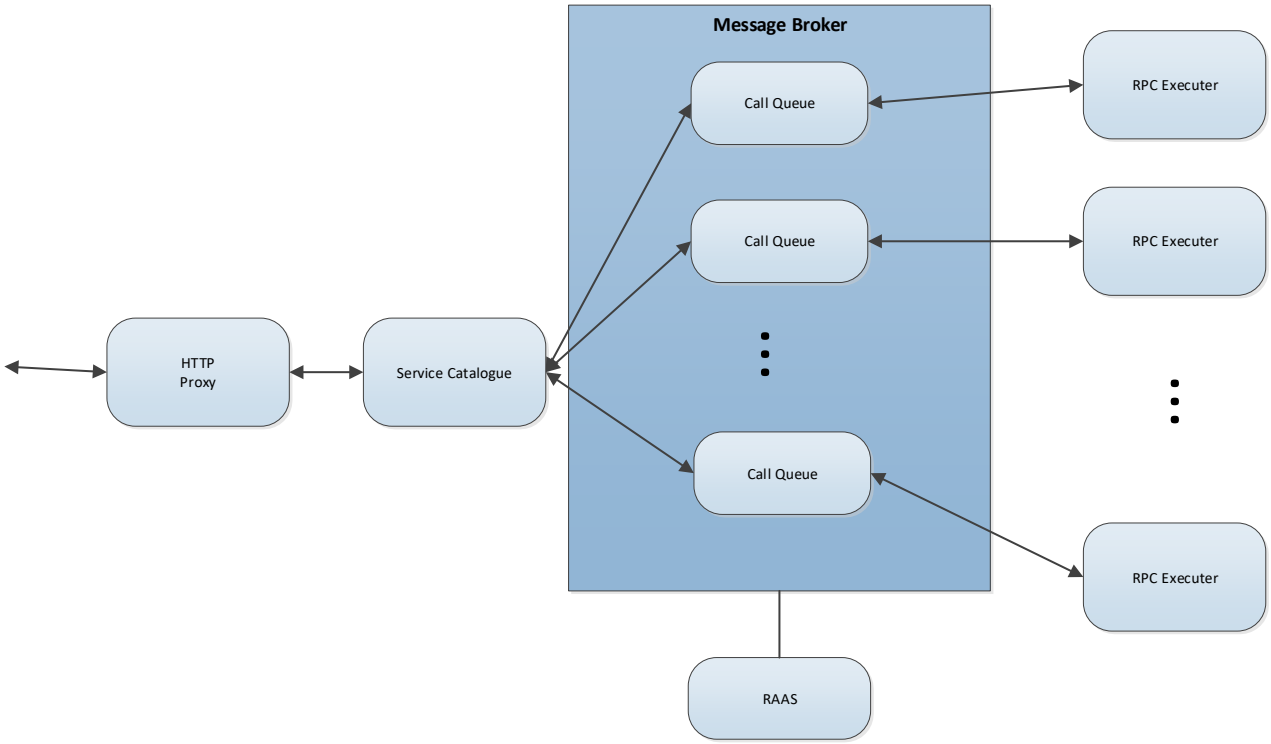

Figure 8: RPC over AMQP

6.5.5 Future work

The REST Adapter can be seen as a special case of an infrastructure and framework for activating microservices using the Message Broker for activation choreography. On publisher side, there is the activation microservices, where the REST accepts an HTTP call and sends this to a configured exchange to be put on a work queue. This message contains all the information from the HTTP call. However, this type of message could also be activated from e.g. a timer, file system trigger or email, by activation (or choreography) microservices. On the subscriber side, the REST Executer is but one type of microservice that could be configured to process messages from a queue and return the results.

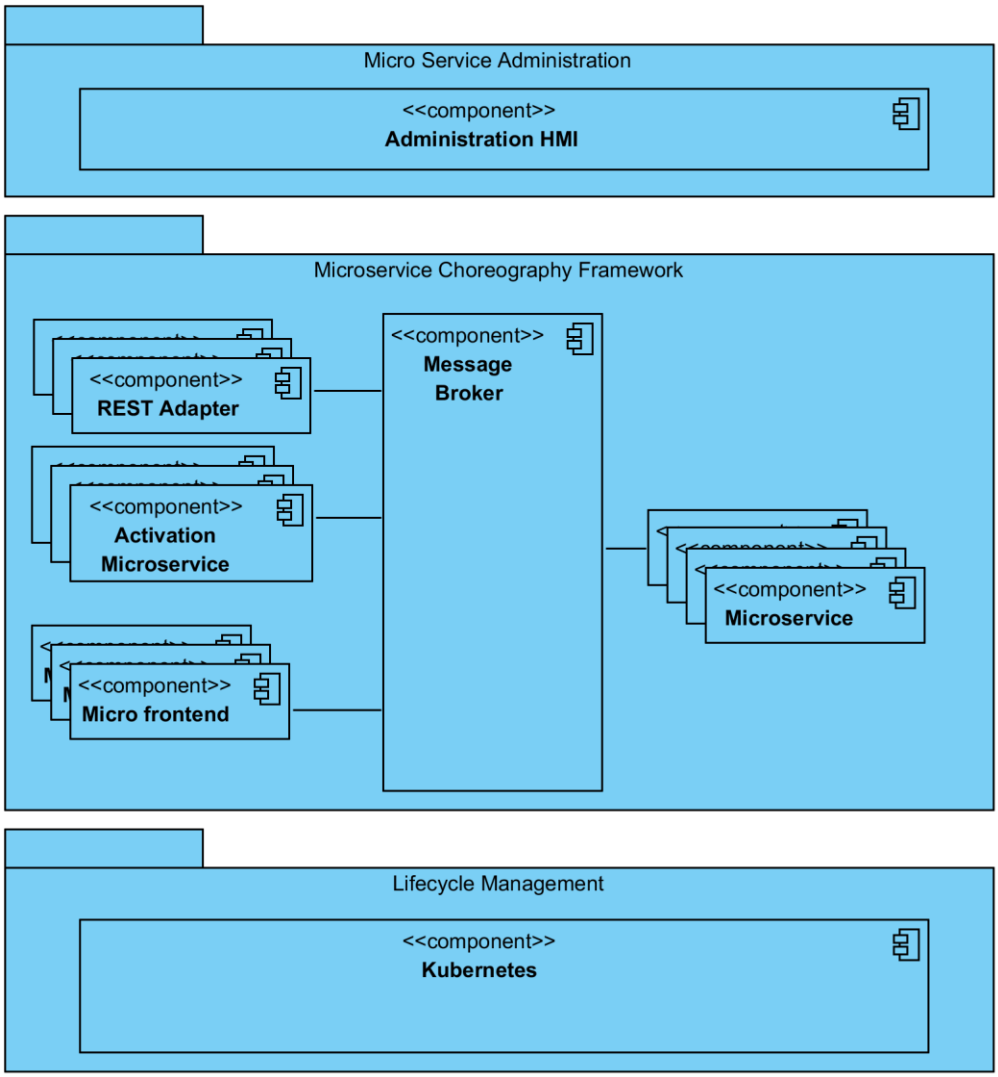

Figure 9: Microservice Framework

If the messages are standardized and the Message Broker communication details are hidden/virtualized/abstracted in a framework, the microservices may be written, managed and orchestrated using a simple web IDE, very similar to e.g. Azure Functions[18] or AWS Lambda[19]. However, the framework would not be tied to any specific cloud infrastructure. Kubernetes or Docker Swarm can be used to handle to handle the infrastructure for microservice container lifecycles, load balancing and scaling.

It is our belief that a lightweight, standardized microservice framework (see Figure 3 and Figure 9) that can be deployed on any cloud platform or physical servers would be a significant addition to the platform and an exploitable asset in the area of I4.0.

[18] https://azure.microsoft.com/en-us/services/functions/
[19] https://aws.amazon.com/lambda/

7 Information View

The format of the messages relayed by the message broker is opaque to the broker (these are described in (COMPOSITION D2.3, 2017)). However, for the components to declare the semantics and syntax of the content, schemas for message broker topics and headers have been defined. These schemas are closely related to the information models of COMPOSITION projects, such as the Digital Factory Model and the Collaborative Manufacturing Services Ontology.

7.1 Inter-factory Market Event Broker

As previously stated in deliverable (COMPOSITION D2.3, 2017 and updated in COMPOSITION D6.3, 2018), agents only communicate through CXL, which has been designed in order to be compliant with FIPA-ACL. Messages not compliant with CXL language should never be exchanged and they will not be processed.

A CXL message is composed of parameters identifying the message purpose, sender and language, and a variable payload whose content depends on the message type, encoded according to an explicitly pre-defined ontology specified by the ontology parameter. The payload depends on both the communication protocol adopted by the agents and on the reference ontology specified.

The CXL messaged are represented in JSON, with ontology expressions encoded in JSON-LD. The JSOM schema for CXL has been provided in D6.3 “The COMPOSITION Marketplace I”. A well-defined set of data-formats for communication is available in (COMPOSITION D2.3, 2017), section 5.4.1.4. The predefined schema for message validation shall be in general available to any agent and the pre-defined schemas for messages should be accepted by all agents. This is part of the specification of a COMPOSITION Marketplace compliant agent.

7.1.1 AMQP Message Routing

The agent communication in the marketplace uses AMQP, which is a “programmable” protocol, offering dynamic control over routing topology (exchanges, bindings and queues) to the consumers and publishers (the agents). The agents use fanout and direct exchanges and thus have no need for a pre-defined set of topics or headers for these exchange types.

Two main modes of operation are supported and used by the agents:

- Dispatch of a message to all the agents:

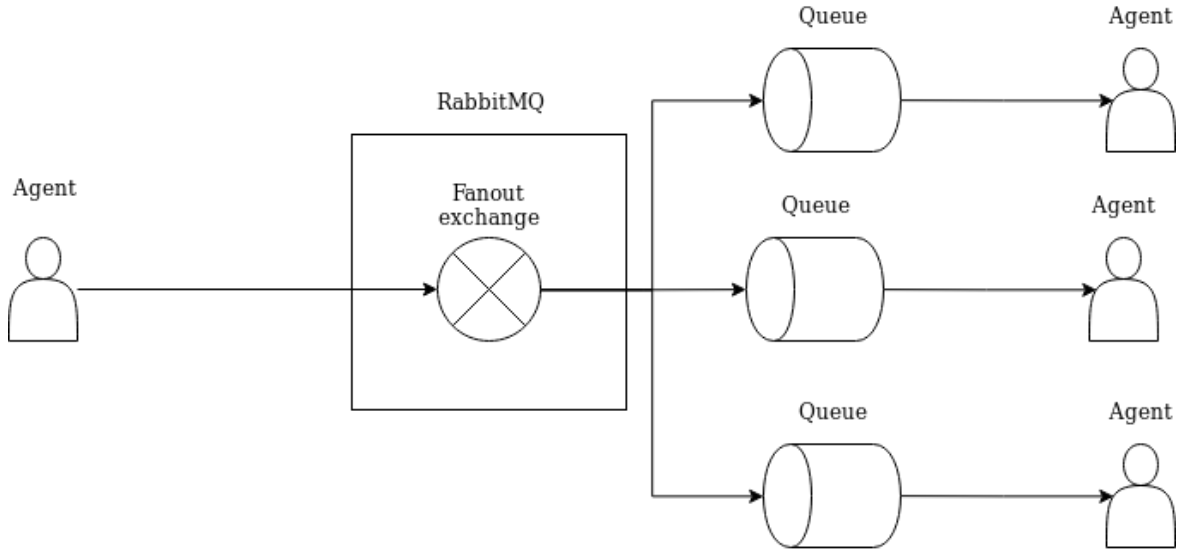

Figure 10: Fanout exchange

In Figure 10 it is shown as the agent willing to send a broadcast message publishes the message on an exchange of type ‘fanout’, which will take care of dispatching it to the proper queues. These are named after the agent identifier (unique within the marketplace) in order to be dynamically created and destroyed at need.

- Dispatch of a message to a single agent:

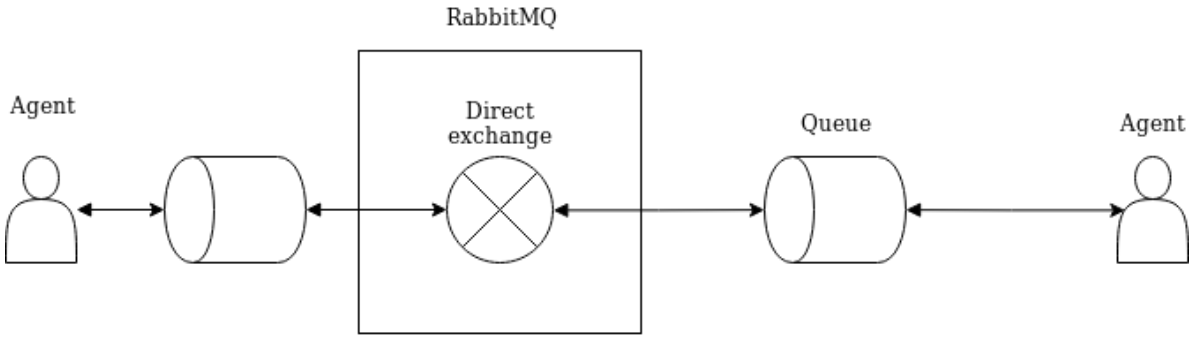

Figure 11: Direct exchange

In Figure 11 it is shown as the agent willing to send a direct message publishes it on an exchange of type 'direct', which will take care of dispatching it to the proper queue. Such queue, as for the previous example, is named after the identifier of the agent which the message is intended for.
The Real-time event broker supporting the marketplace does not need any special configuration, except for the configurations included within its runtime environment, i.e. the Portainer UI made available within the local Docker installation hosting the various components. The agents configure the necessary exchanges, queues and bindings.

7.2 Intra-factory Real-time Event Broker

In the intra-factory deployment of the message broker, MQTT is used for the distribution of sensor data and processed information. This protocol is suited for sensor data and it is also supported by the core components. In the pilot installations the metadata is provided out-of-band, by the DFM. There is no metadata annotation of the data. However, should this be a requirement for a particular installation or additional component, all this data can be made available in-band using e.g. JSON-LD format.

Sensor and analysis data exchanged in the intra-factory system is structured according to the OGC SensorThings Data Model and represented in JSON format, also some data from Digital Facgtory Model is managed in XML format..

7.2.1 MQTT Topic structure

In the intra-factory scenario there is the need to create a hierarchy that defines the topics structure for the event broker. As the JSON representation of OGC SensorThings Data Model [REF] is used for the internal data, the topic structure described in this standard is used with minor adaptations.

COMPOSITION topics use a common structure for all components publishing data:

- a topic root that will use the "Composition" tag as identifier;
- the discriminative dichotomy identifier of the intra or inter factory component (*<SW-CODE>);*
- the component instance id that is in charge of generating the data *(<SW-ID>);*
- the standard used for the data produced (<*STD-CODE*>), which will be "OGC";
- The version of the standard used (<*VER-NO*>), which will be "1.0"

In the following manner: *<SYSTEM>/<SW-CODE>/<SW-ID>/<STD-CODE>/<VER-NO>*

The OGC SensorThings Topic is constructed as

/<RESOURCE-PATH>/<COLLECTION-NAME>

E.g. *Composition/BMS/NXW_51/OGC/1.0/Datastreams(ds_1)/Observations*, to subscribe to all new observations published for Datastream ds_1.

COMPOSITION has put the resource id in a separate topic level. Also, MQTT uses slashes ("/") for topic segment separators while the AMQP 0.9.1 topic separator is the dot ".". The dot should not be used in topics when using RabbitMQ. Thus, topic structure used is as follows:

<SYSTEM>/<SW-CODE>/<SW-ID>/<STD-CODE>/<VER-NO>/<RESOURCE-PATH>/<COLLECTION-NAME>

E.g. *Composition/BMS/NXW_51/OGC/1_0/Datastreams/ds_1/Observations*

Concerning the formal representation, an upper camel case notation has been used and it is strongly recommended to be adopted by all components, in order to provide consistency among publishing and subscribing topics.

7.3 Resource Catalogue

The topic structure based on the OGC SensorThings Data Model only describes the data from a sensor perspective. The components that subscribe to the data streams need to know which factory asset, represented in the DFM, the data stream is associated with. This is provided by a mapping in the DFM to the OGC SensorThings model.

The COMPOSITION Digital Factory Model's Assets List is used as a Resource Catalogue for the project's needs. Other COMPOSITION components such as Learning Agent, DSS, UIs etc. that needs information about datastreams are connected in a factory asset can get the information from this Resource Catalogue and DFM APIs RESTful services. This catalogue will provide to these components the IDs of the MQTT topics in BMS in order to receive the requested data coming from assets' sensors.

For the implementation of Asset List, the B2MML package related to the assets has been updated for the COMPOSITION purposes. Every factory asset, for example a machine has installed built-in sensors or newly deployed sensors in order to cover the project use cases' requirements. A sensors list has been created and added on the assets as the modelling of sensors is mandatory. Sensors related to the pilots such as vibration sensors installed in machines, light barriers, built in sensors in machines and bin's fill level monitoring sensors are modelled. As soon as a sensor deployed in COMPOSITION (before or after the system is put in use), information on how to identify data (OGC ST Observations) coming from this sensor are added to the DFM and distributed to other parts of the system.

The BMS will connect to a sensor. The BMS assigns a datastream id to the data from the sensor and publishes this to COMPOSITION as an OGC ST observation. Other components can query the DFM to find out which datastream id is used for the information they are interested in and subscribe to live data or query for historical data. These ids will be the sensors id coming from Sensors List that belongs to a factory asset (machine or a bin etc.).

Furthermore, the described assets schema that contains the above described sensors list can be easily used by project components which are familiar with OGC SensorThings.

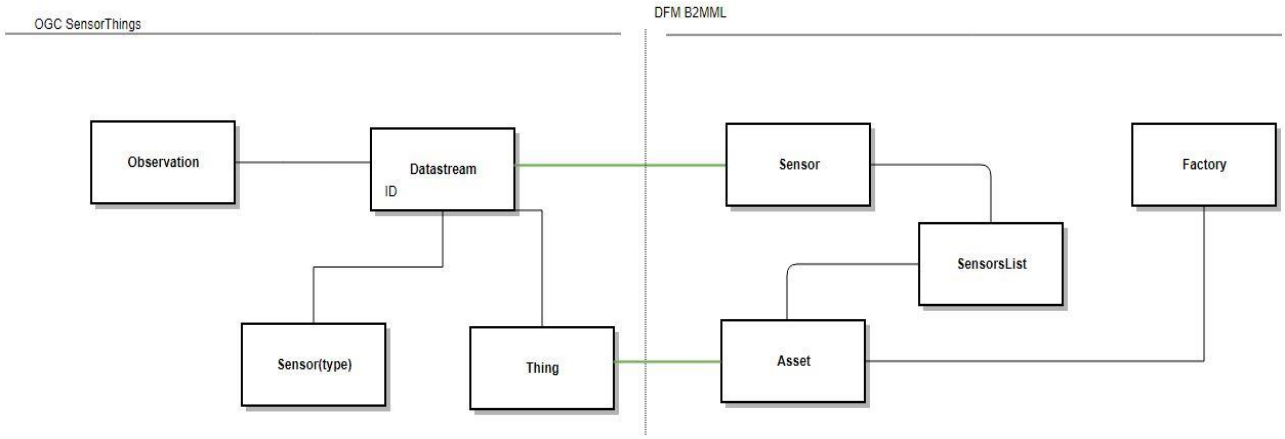

Figure 12: DFM Assets and OGC SensorThings Mapping

A DFM's Asset class is equivalent with the Thing class of OGC SensorThings. A Sensor class of DFM is mapped to a SensorThings Datastream. For example, a machine in the production line of a factory is considered as an asset/thing. For this machine are available some sensors/datastream. A component such as the Learning Agent of COMPOSITION which uses SensorThings is able to find available datastreams in a

machine of digital factory instance of COMPOSITION by using the aforementioned mapping between the two data models. The automatic mapping is enabled by DFM API services.

If a component sent a request to a DFM instance for a machine's datastream coming from the installed sensors in this machine for a specific use case (e.g. an Oven) and the DFM API will inform the component for the datastreams IDs that the component can use in order to read sensors data from BMS. A simplified example of the DFM API service's response in JSON format is presented below in Figure 13:


```
{
   "sensor 1": {
      "dataStreams": [
         "1-0"
      ],
      "type": "Log Events"
   },
   "sensor 2": {
      "dataStreams": [
         "2-1",
         "2-2",
         "2-3"
      ],
      "type": "Acoustic Sensor"
   },
   "asset": "UC2_Oven_ID"
}
```


Figure 13: Simplified response from DFM

8 Deployment view

The COMPOSITION Ecosystem uses Amazon Web Services (AWS)[20] as the infrastructure for cloud components of the system. All selected computing, storage, and networking AWS resources operate in Frankfurt (eu-central-1) region, providing low latency across Europe. AWS guarantees data privacy[21] and is compliant to European Union's General Data Protection Regulation (GDPR)[22]. Detailed specifications have been presented in D2.4 "The COMPOSITION Architecture Specification II".

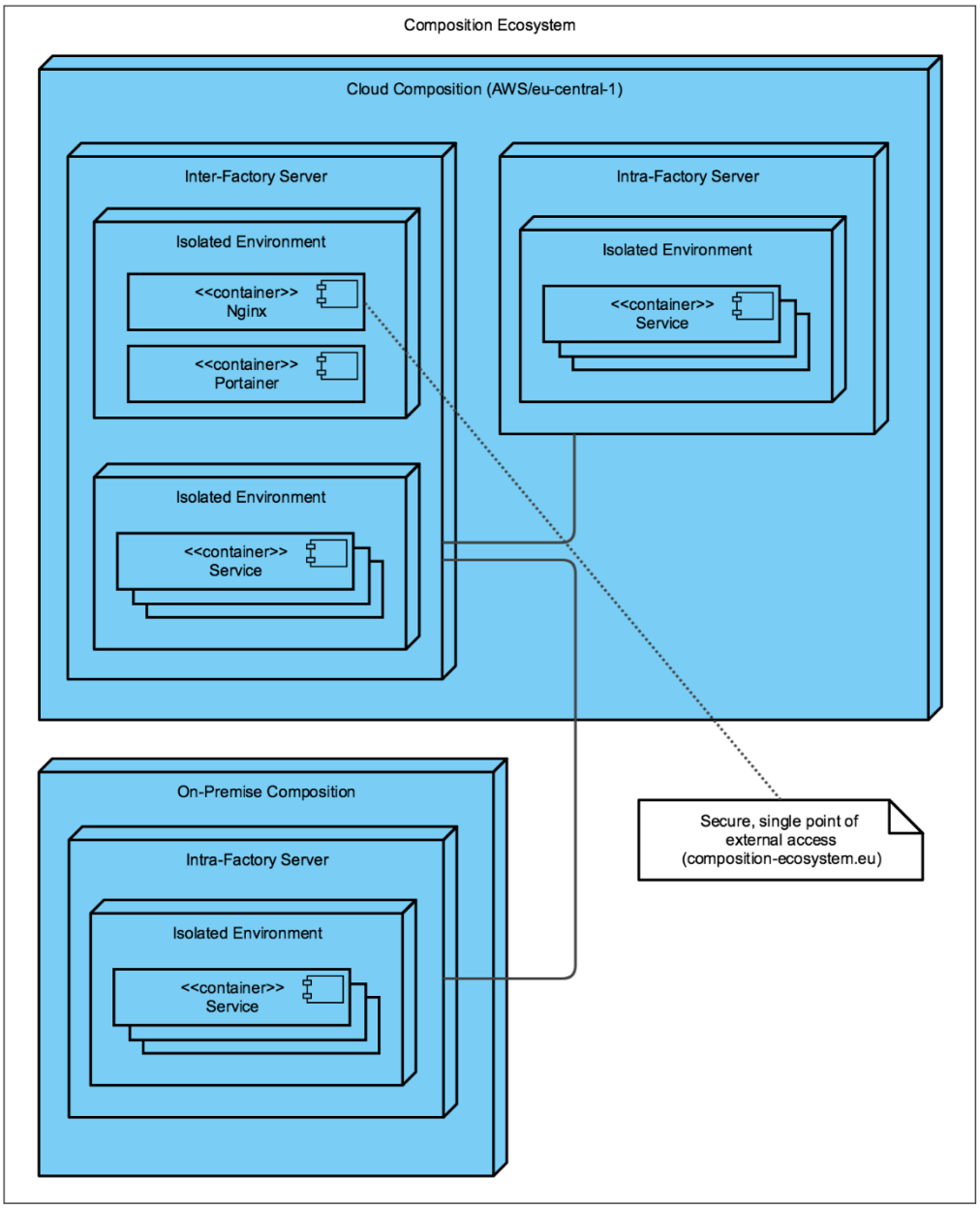

Figure 14: Current COMPOSITION production servers: all components are deployed as Docker containers, external traffic is secured by TLS

The unit of deployment in COMPOSITION for all cloud software components is a Docker container, which provides portability and isolation. The Inter-Factory server hosts docker containers for instances of the Docker management tool Portainer and Nginx, which proxies requests to internal components. Nginx also secures all

[20] https://aws.amazon.com/
[21] https://aws.amazon.com/compliance/data-privacy-faq/
[22] https://aws.amazon.com/compliance/gdpr-center/

external traffic by TLS (using a Let's Encrypt certificate), including AMQP and MQTT traffic to and from the Message Broker, see Figure 14. (The BMS publishes sensor data to the Message Broker by MQTT.)

9 Scalability Perspective

The message broker is a principal component in both the intra- and inter-factory parts of the COMPOSITION system. It must be responsive and fault-tolerant and be able to handle a large workload with many communicating components and large amounts of data. The exact workload will depend on the specific deployment scenario. The message broker cannot be a bottleneck or a single point of failure. The scalability of the message broker has been studied in WP6 and results have been previously reported in D6.3 COMPOSITION Marketplace I and D2.4 The COMPOSITION architecture specification II. This section describes the scalability concerns for the message broker and design decisions and mechanisms that can be employed to address these concerns.

9.1 Scalability Overview

Scalability is discussed in D2.4 The COMPOSITION architecture specification II, which has a more thorough section on this subject. For clarity, we provide a summary here.

We define the performance of a component as the capability to handle a specific workload given a specific set of resources, e.g. CPU cycles, memory and disk space. An instance of the message broker can increase the maximum workload it can handle by expanding its quantity of consumed resources. The ability to do this is called scalability (Lehrig, Eikerling, & Becker, 2015). The resources can be increased in two ways, called scaling up and scaling out. To scale up, or scale vertically, is to increase overall application capacity by increasing the resources within existing nodes, e.g. increase memory or CPU of the existing node. To scale out, or scale horizontally, is to increase overall application capacity by adding nodes, e.g. adding an additional message broker that shares the workload.

In D2.4, we have identified the following attributes that may affect the workload of the message broker and thus the need for increased resources (scaling):

- Factory IIMS
    - The number of concurrently reporting sensors/field devices
    - The frequency of the reporting from sensors
    - The number of concurrently reporting internally generated data streams from e.g. Learning Agent, DSS and DLT
    - The frequency of the reporting from internally generated data streams
- Marketplace
    - The number of marketplaces
    - The number of stakeholders in a marketplace
    - The number of concurrent agent negotiations
    - The number of participants in each negotiation
    - The number of data sharing agreements between marketplace stakeholders

Like all other COMPOSITION components, the message broker unit of deployment is a Docker image that may be deployed as one or more containers on a host running in-premises or in the cloud. Docker supports control of both horizontal and vertical scaling of the services offered by a component.

9.2 RabbitMQ

Based on prior experience (ALMANAC project, PICASO project, FITMAN SEM) and published performance figures[23][24] (Fernandes, 2013) (Maciej, Krzysztof, & Aleksander, 2014) it is currently estimated that a single RabbitMQ instance, scaled vertically to adequate performance, will likely suffice in the pilot scenarios. However, the broker component will need to scale to large real-world scenarios.

[23] https://www.rabbitmq.com/blog/tag/performance/
[24] http://underthehood.meltwater.com/blog/2016/09/01/rabbitmq-performance/

The broker will have to provide support for a high number of sensors and near real-time updates of processed data in the intra-factory system, and very large number of interacting agents in the Open Marketplace.

As mentioned in (COMPOSITION D2.4, 2018), the centralized message broker architectural pattern of communication can introduce a possible bottleneck or a single point of failure in the system. To distribute the message broker – scale out - by adding nodes is a well-tried configuration to deal with scalability of the broker. Depending on the communication patterns, this will also be applicable in COMPOSITION. RabbitMQ is also available as highly scalable cloud services[25].

If a node in the scaled-out message broker fails, the choice of technique will favour one of two properties of the distributed component, availability or consistency. Availability ensures that every request is delivered and receives a non-error response, but it may not contain the most recent message. Consistency is to be prioritized if it is required that every client will receive the most recent message in a stream (or an error).

9.2.1 RabbitMQ Scalability

This section will describe the techniques available to implement horizontal scaling of RabbitMQ by distributing the message broker: clustering, federation and “the shovel”. These approaches to message broker distribution may be combined, e.g. using clusters connected with federation or ”the shovel”. Thus, a desirable degree of throughput and resilience to failure, with preserved consistency where needed, may be achieved.

9.2.1.1 Clustering

A RabbitMQ cluster connects multiple distributed nodes together, to form a single logical broker. The nodes must run the same version of RabbitMQ. All nodes in the cluster are connected to all other nodes. Cluster nodes communicate via Erlang message-passing and should be located on single low latency network (LAN) with reliable communication.

Exchanges[26] and bindings are shared and automatically mirrored across all nodes in a cluster. Queues may be mirrored but are located on a single node by default. Creating a queue for a client will only create a new process in one broker in the cluster. A client connecting to any node can see queues on any node in the cluster. Published messages are replicated on all mirrored queues and consumed messages are removed from all nodes, so replicating a queue also replicates the queue work load on all nodes.

RabbitMQ clusters are used to increase the throughput of a broker, prioritizing consistency. Clustering solves the bottleneck problem, but since all nodes are in a single location, the single point of failure remains.

9.2.1.2 Federation

With federation, an exchange or queue on one broker can be set up to receive messages published to an exchange or queue on another, logically separate, broker. (Note that a single logical broker in this case may be a cluster, as described in the previous section.) These are typically located on different networks and communicate over the internet via AMQP (with SSL encryption). Using AMQP connections requires users and permissions to be set up on both servers. Unlike a cluster, brokers in a federation can be connected in any topology, with links between brokers going in one direction, or both. Federated and local exchanges and queues may co-exist in the same broker.

Federated exchange links are one-to-one, in one direction. Messages will be forwarded over this link only if a binding to a queue on the federated exchange exists. A client connecting to any broker can only see queues in that broker, and messages will be sent between federated queues to where the consumers are connected.

Federations are typically used to link brokers across the internet to maximize availability for publish-subscribe messaging and work queueing. The integrated security provided by COMPOSITION Security Framework will facilitate the set-up of federated message brokers with shared user management
In the Open marketplace, federations between brokers belonging to different stakeholders is a viable way to scale out the system.

[25] https://www.cloudamqp.com/
[26] And other entities, e.g. virtual hosts, users, permissions, runtime parameters, et c.

9.2.1.3 "The Shovel"

"The shovel" is similar to federation. However, while federation distributes exchanges and queues across brokers, "the shovel" simply specifies how messages should be moved. "The shovel" works at low level, consuming messages from a queue and re-publishing them at an exchange, usually at another, logically separate, broker. Shovels may be configured statically at startup or dynamically, at runtime, depending on the level of control desired. Communication is through AMQP (with TLS), in a local network or across the internet, with high tolerance for network failures.

"The shovel" is an alternative to federation with a more fine-grained control and lower level of abstraction and may also be used as an alternative to a specific client application to implement a desired communication pattern.

9.2.1.4 Central Authentication and Authorization

The COMPOSITION integration with Keycloak by RAAS, described in section 10, overrides the built in RabbitMQ user management. This creates a unified authentication and authorization system for all brokers in a COMPOSITION system deployment. This simplifies the implementation of the scaling techniques described in this document, as we can manage users for all brokers from one Keycloak system, whether in a federation or connected with "the shovel".

9.3 Scalability Design

The Message Broker is the central communication hub in both the intra- and inter-factory scenarios and must scale well in a number of scenarios. This section builds on the scalability design reported in D6.3 "The COMPOSITION Marketplace I". Choosing a scalability design for the message broker requires analysis of the usage pattern and how messages are distributed in the specific scenario and utilizes on the design of the AMQP protocol. The message broker consists of one or several brokers distributed on one or more nodes. In a broker, exchanges receive and route messages to queues based on bindings with different filters. There is no fixed limit to the number of exchanges and queues in a broker. We have identified are two types of configuration which can be used to address scalability for the broker, which are referred to as routing topology and broker topology.

Broker topology deals with the distribution of logical brokers on nodes, by the built-in support for clustering (one logical broker on separate nodes) or federation (different logical brokers on separate nodes).

Routing topology deals with the connections of exchanges and queues by bindings and the distribution of these on brokers. This topology can be set up dynamically on existing brokers by the AMQP protocol (and RabbitMQ extensions). The clients (consumers and producers of messages) can control the routing topology at run-time.

RabbitMQ allows exchange-to-exchange bindings, routing messages from one exchange directly to a secondary exchange. Clients would then only bind to the secondary exchange, and the number of client queues and number of connects and disconnects at the secondary exchange would not affect the primary exchange. This is a viable way to scale out the system for a large number of agents in the marketplace. (A closed marketplace could require that stakeholders provide the resources for running a broker node.)

Routing topology design could e.g. favour many fanout exchanges or fewer exchanges and more use of routing. Fanout exchanges are slightly faster than the other types of exchanges for multiple recipients, e.g. topic and header exchanges. However, the difference is not a deciding factor in the choice of topology.

9.3.1 Intra-factory

The intra-factory message broker has to handle data streams from shop-floor sensors and analytical components within COMPOSITION. As MQTT is used, only publish-subscribe routing of messages is available and the run-time configuration of routing topology in AMQP is not available via MQTT.

Neither the number of publishers nor the number of consumers is expected to be very large in the intra-factory setting. However, the frequency may be high causing the amount of data to be large. To handle this, a broker topology with a cluster setup is most suitable to manage high loads. Queues for consumers will only be created on the cluster node that the consumer is connected to.

9.3.2 Inter-factory

This section will discuss examples of possible scaling strategies for the marketplace, where the Message Broker manages the agent CXL communication. Also in this case, the design of broker topology is the primary way to ensure scalability for the marketplace. However, the agents of different stakeholders may set up the routing topology to suit their specific requirements and communication patterns.

Growth in the number of marketplaces is typically handled by adding nodes to the broker topology. A Closed Marketplace typically has a separate infrastructure from the Open Marketplace, whereas a Virtual Marketplace shares the infrastructure of the Open Marketplace. Marketplaces are logically separated; no messages are exchanged between marketplaces. Virtual marketplaces are set up by actors already in the Open Marketplace. Each Closed marketplace will be handled by a separate Message Broker. Open Marketplace and Virtual Marketplaces will use clustering.

In the cluster, load-balancing techniques may be used to distribute agents among the nodes so that the (non-mirrored) queues created by the agents is evenly distributed on the nodes,

Growth in the number of stakeholders in a marketplace may be handled by a routing topology which creates a secondary exchange for each specific stakeholder (Figure 15). The secondary exchange has an exchange binding to the primary exchange, which can be a fanout exchange. The consumers and producers (Agents) connected to the secondary exchange only create bindings and queues on one broker in the cluster when they connect. The secondary exchange may be a topic or header exchange.

The secondary stakeholder exchange will always exist, whether the stakeholder agents connect or disconnects. It will receive messages from all exchanges that the stakeholder has an interest in. Whenever a consumer (agent) connects, it simply has to declare its queue and bind that queue to the stakeholder exchange using the desired topic filter.

A similar topology may be created by using either the shovel or federation with an upstream broker (primary) and a federated broker (secondary). These may be two separate broker nodes using different infrastructure. The messages to a queue declared in the federated broker are buffered in a queue created in broker the upstream exchange. If each connected stakeholder provides the infrastructure for the broker where the secondary exchange resides, the system can scale very well.

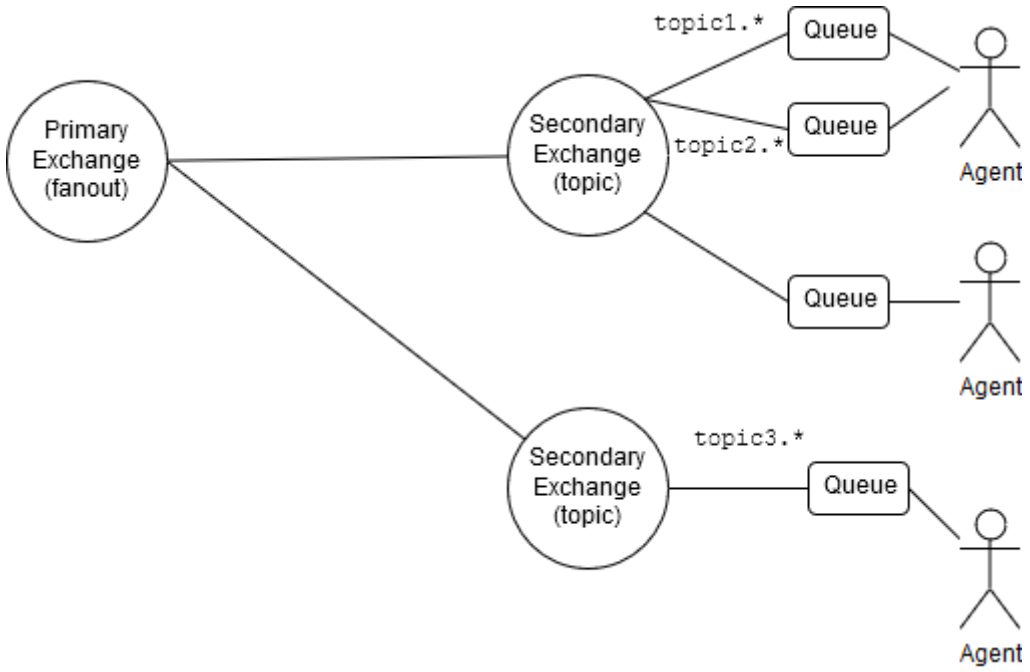

Figure 15: Primary and secondary exchange routing topology

The number of concurrent agent negotiations taking place will increase the number of messages being sent. In the above topology, the queues will be at the secondary exchanges and messages published to the exchange will be propagated to the primary and to all secondary exchanges. The primary/secondary broker topology deployed in a RabbitMQ cluster will handle a very large number of concurrent negotiations. Should

the message flow require even more resources, a broker topology using a federation in a connected graph (each one a cluster), where an exchange for the negotiation will exist on one broker node in the federation only for the duration of the negotiation (Figure 16). The number of participants in each negotiation will likely not be a limiting factor for the described topology.

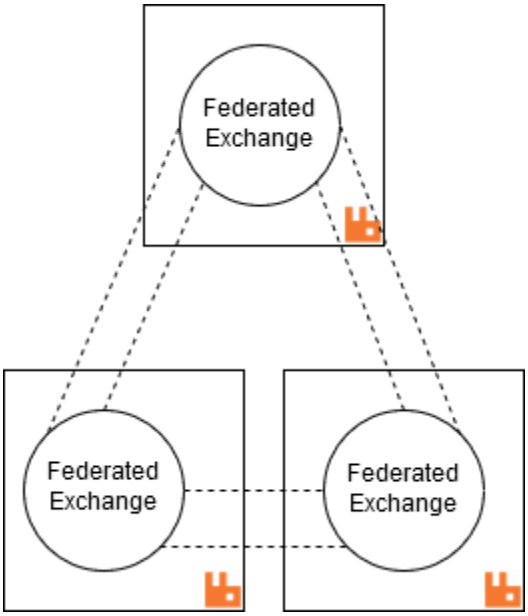

Figure 16: Federated exchanges broker topology

An exchange that only the involved parties can access can be set up for each data sharing agreement (Figure 17). At most this will result in a number of exchanges on the scale of $O(n^2)$ to the number of stakeholders. If one exchange is created for a stakeholder to publish to, and exchange to exchange bindings (or shovels) are defined for each recipient of data to the secondary exchanges described above (Figure 18), the number of exchanges will relate to the number of data sharing agreements by O(n). The sender will control the exchange to exchange bindings or shovels. The data sharing may need to use a separate logical broker (cluster) in the marketplace depending on the load.

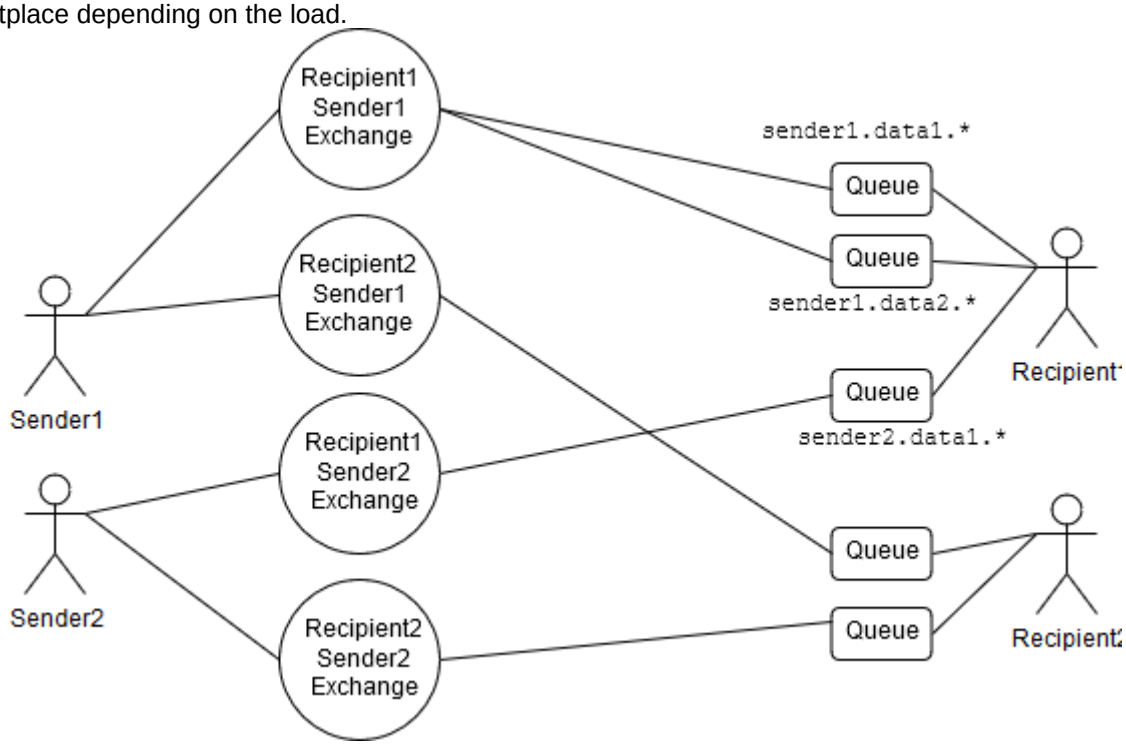

Figure 17: Data sharing using one exchange per data sharing agreement

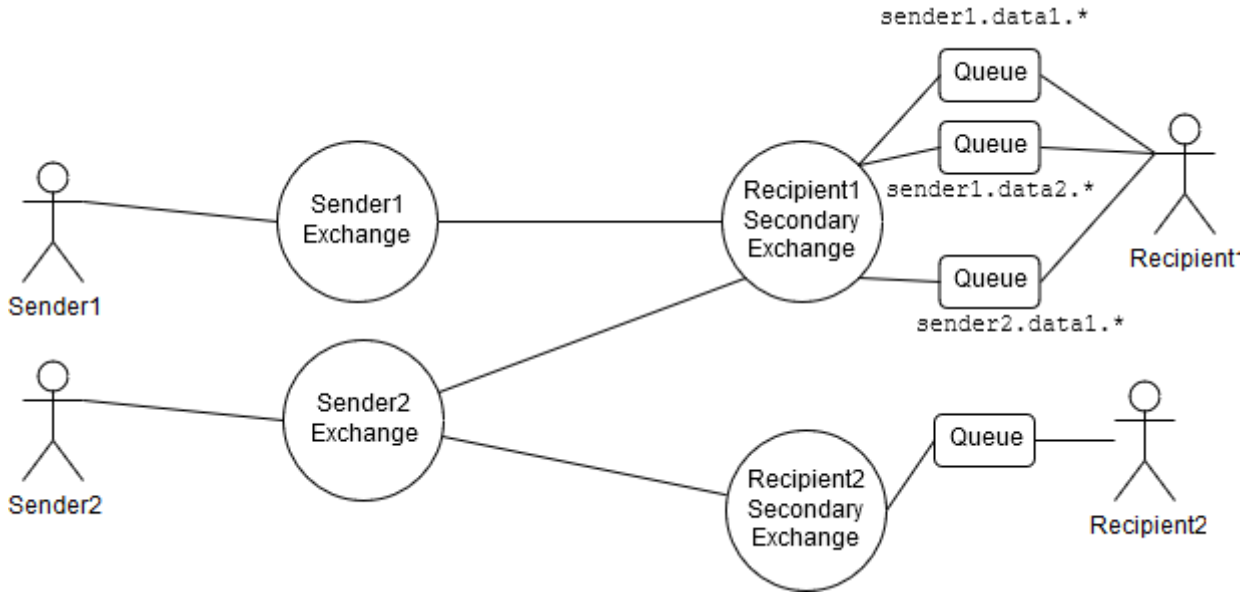

Figure 18: Data sharing using sender and recipient exchanges

10 Security Perspective

To provide an integrated security solution for COMPOSITION, an adapter allowing the authentication and authorization mechanisms of RabbitMQ to be managed by Keycloak and Authorization Service (EPICA) is being developed. With the use of rabbitmq-auth-backend-http[27] community plugin RabbitMQ built-in authentication and authorization can be overridden and managed from outside by referring the requests to other components.

The adapter in development, RabbitMQ Authentication and Authorization Service (RAAS), is a web-service developed in Node.js and exposes the following endpoints, required by the plugin:

- https://server:port/raas/auth/user: Used to authenticate a user providing username and password.
- https://server:port/raas/auth/vhost: Used to authorize access to a virtual host.
- https://server:port/raas/auth/resource: Used to authorize access to a resource.
- https://server:port/raas/auth/topic: Used to authorize access to a topic.

The adapter returns always HTTP 200 OK and one of the following:

- allow
- deny
- allow [list of tags] (only for https://server:port/auth/user)

All communication between RabbitMQ and the adapter is encrypted using TLS cryptographic protocol, provided by Nginx Reverse Proxy.

The adapter will manage everything related with the access tokens obtained from Keycloak when a user login RabbitMQ.

The same security system can thus be used for intra-factory business user identity, marketplace partners and system components.

RabbitMQ configured protocols use SSL/TLS cryptographic protocols for communication with publishers and subscribers. The default non-secured communication ports will be disabled to ensure all communication is encrypted.

All messages flowing between publishers and subscribers will be signed using JSON Web Signature[28] (JWS) standard.

An adapter for the blockchain distributed trust mechanism allows the integrity and non-repudiation of broker messages, publishers will store the digital fingerprint of the data transmitted and the subscribers will have the possibility to check the digital fingerprint of the data received.

The achievement of the integrated security solution for COMPOSITION has been performed by the configuration, installation and deployment of the different components (described in D4.2 Design of Security Framework II and D4.4 Prototype of the Security Framework I) which provide the securization mechanisms for assuring a complete authorization and authentication protection.

Regarding the deployment, the message broker has been deployed at the production server in the pilot environments:

- Inter-factory
    - Docker container: rabbitmq-inter
- Intra-factory
    - Docker container: rabbitmq-intra

Both of them use AMQP with authorization based on vhosts and operations on resources as temporary solution until the EPICA component is integrated for authorization purposes, and MQTT plugin is installed for its

[27] https://github.com/rabbitmq/rabbitmq-auth-backend-http
[28] https://tools.ietf.org/html/rfc7515

support. Once integrated, EPICA will be able to handle authorization issues, extracting all the needed information directly from Keycloak tokens, for finally performing the matching with internally stored authorization policies. For more information about EPICA, the reader is advised to consult (COMPOSITION D4.4, 2018), more precisely Section 3.1.

The following sections, 10.1 and 10.2, will present the differences between Inter-factory and Intra-factory event brokers' configuration and related security components.
RabbitMQ will only support default AMPQ[29] protocol over TLS in Inter-factory deployment.

10.1 Inter-factory Market Event Broker

A blockchain adapter will be developed to store and retrieve the public keys needed by the subscribers to verify the signature of the message received. Each participant in COMPOSITION will need to deploy a blockchain node to get access to the public key and thus be able to verify the signature of messages. The blockchain used will be the same as the one used to allow the integrity and non-repudiation of broker messages and mentioned in the previous section.

Figure 19 depicts the architecture and the relation between the message broker and the security components.

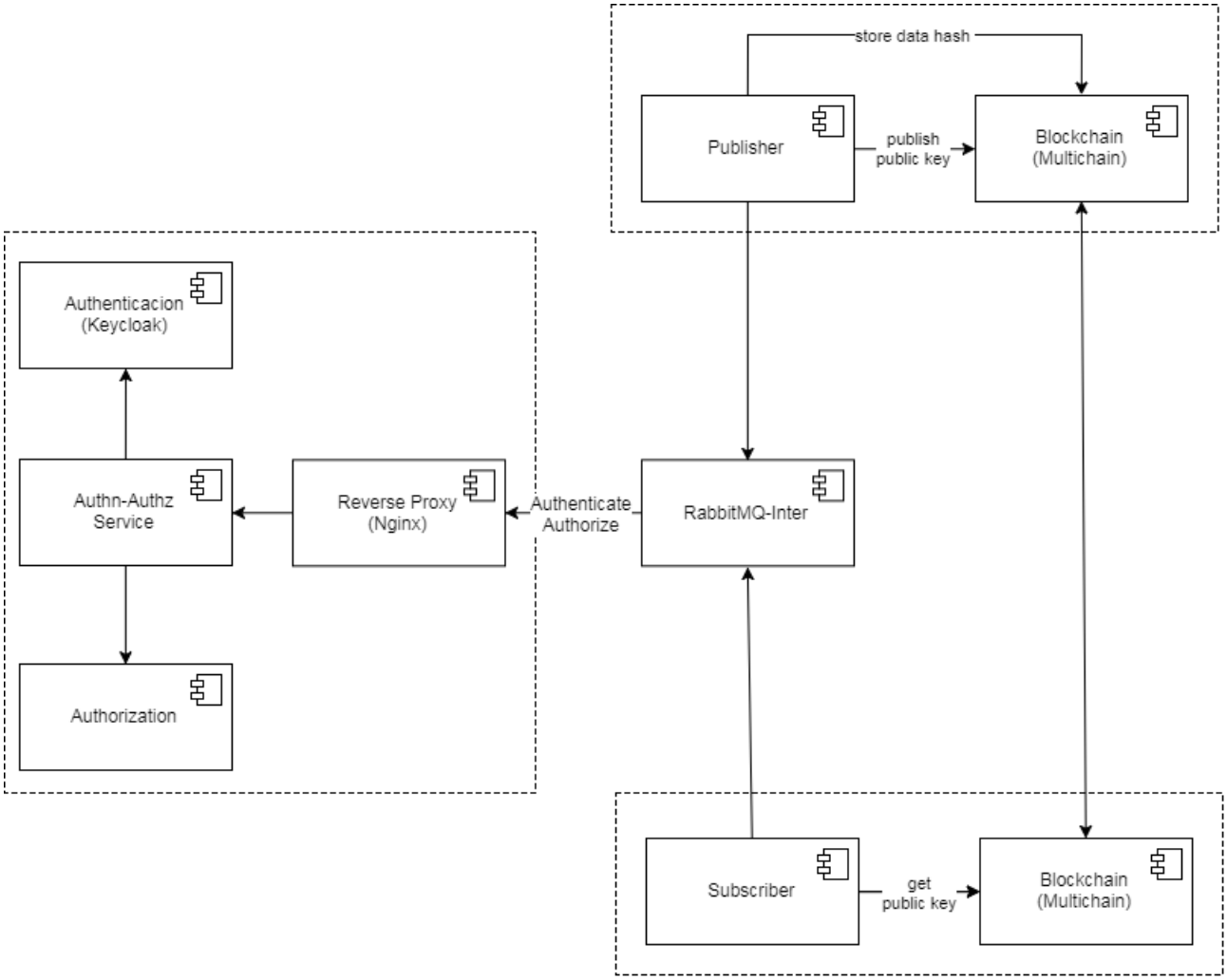

Figure 19: Inter-factory Market Event Broker security architecture

[29] https://www.amqp.org/

10.2 Intra-factory Real-time Event Broker

In this case, RabbitMQ will support two different messaging protocols; default AMPQ protocol and MQTT[30] protocol, both over TLS.

In order to make available the public keys needed by the subscribers to validate the message signatures, LinkSmart will be used for such task.

Figure 20 gives an overview of the architecture and the relation between the message broker and the security components.

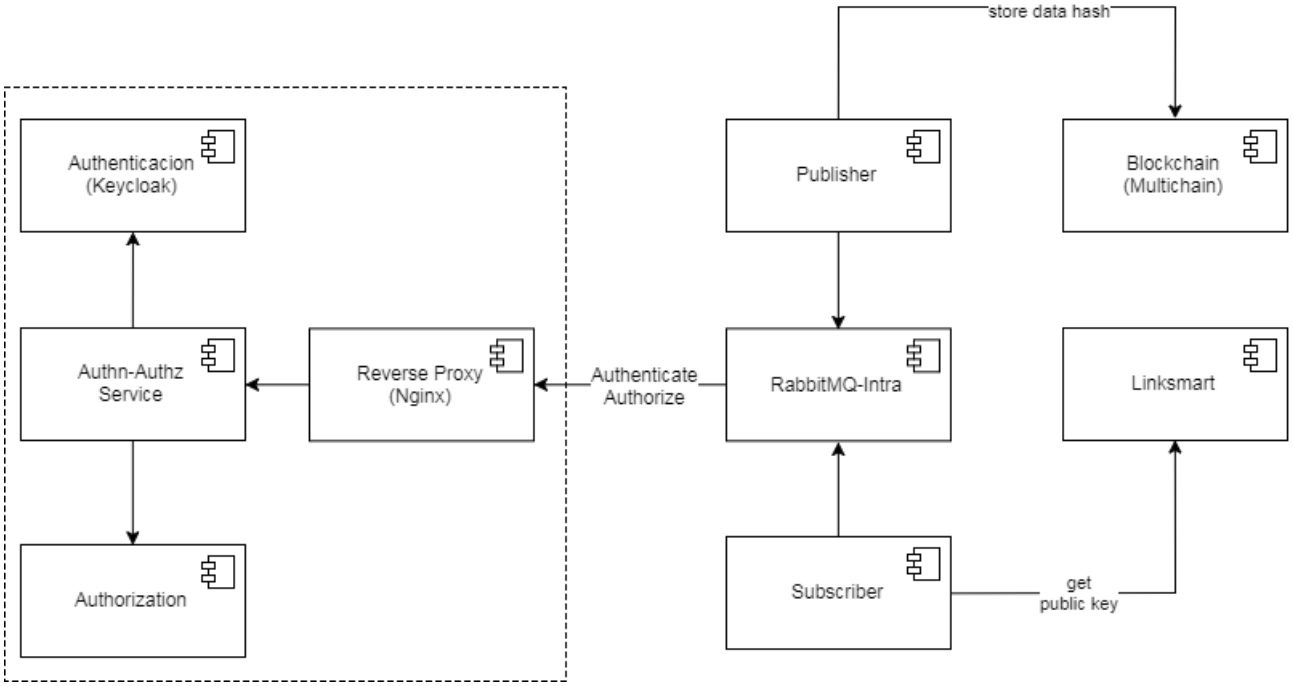

Figure 20: Intra-factory Real-time Event Broker security architecture

[30] http://mqtt.org/

11 Summary and conclusions

The real-time event broker is a principal component in the COMPOSITION architecture. It integrates the heterogenous components using standard protocols, allowing for future extensibility through low coupling. The multi-protocol support allows the inter- and intra-factory COMPOSITION systems to use the most appropriate protocol for the task and a multitude of communication patterns. During design, MQTT was selected for factory sensor and real-time analysis data and AMQP was selected for internet communication between heterogenous systems in the COMPOSITION Marketplace.

In the first year, the project evaluated different broker implementations. This resulted in the candidate from the architecture inception phase, RabbitMQ, being confirmed as the communication mechanism implementation. While RabbitMQ is not the fastest message broker available, it is standards-based, easy to configure and maintain, well tested in production, robust, scalable and highly extensible. The general-purpose applicability, plugin architecture and extension mechanisms allow for built-in multiprotocol support and tight integration with the important COMPOSITION goals of end-to-end security and blockchain-based log-oriented architecture.

The investigation into scalability techniques for RabbitMQ has found design solutions to the concerns for bottlenecks and a centralized point of failure. However, these findings have to be evaluated against a set if concrete scalability scenarios in each case. The design to choose will depend on the concrete scenario at hand. For the pilot deployments, the configuration described in this report will be used.

Message brokers with security framework integration has been deployed at the COMPOSITION pilot servers. The design of topic schemas for the intra- and inter-factory message brokers has been finalized.

Significant extensions and application to RabbitMQ undertaken in COMPOSITION are

- RAAS integration of Security Framework replacing RabbitMQ proprietary authentication and authorization,
- blockchain audit log,
- use of AMQP together with the Security Framework for marketplace data sharing,
- use of AMQP for platform-independent agent ACL interchanges,
- REST-tunnelling over AMQP,
- proposal for AMQP-based microservice orchestration extending the REST-tunnelling concept

12 Appendix 1: Candidate message broker implementations

This section provides a brief overview of alternative message broker implementations that were considered as evaluated as complements or substitutes for the intra- and inter-factory Message Broker.

12.1.1 Mosquitto

Eclipse Mosquitto is an open source, Eclipse licensed (EDL/EPL[31]) message broker that implements versions 3.1 and 3.1.1 of the MQTT protocol. While lightweight and fast, it did not provide the extensibility, reliability (durable queues) or configurability required.

12.1.2 Kafka

Kafka is built to process real-time streams of data in a horizontally scalable, fault-tolerant and very fast manner. It does not implement a standard protocol; integration with Kafka is made through proprietary producer, consumer, stream processor and connector APIs. Kafka is distributed under Apache License[32] and widely deployed in large production environments. Kafka could be a complement for the sensor platform in deployments that handle a very large number of sensors (e.g. large scale fully automated production with a large number of robots reporting movement and power consumption from every motor).

12.1.3 ZeroMQ

ZeroMQ [33](a.k.a. ØMQ, 0MQ, or zmq) is a fast concurrency framework providing transport sockets for in-process, inter-process, TCP, and multicast communication. Multiple patterns are possible, e.g. fan-out, pub-sub, task distribution, and request-reply, but require programming. It is provided as APIs (not a standardized protocol) for multiple platforms. It was considered for agent communication but is LGPLv3 licensed.

12.1.4 ActiveMQ

Apache ActiveMQ[34] is a message broker - the one most similar to RabbitMQ of the considered alternative implementations. Released under Apache 2.0 License, and written in Java with JMS[35], REST and WebSocket interfaces, it also supports protocols AMQP and MQTT. RabbitMQ was favoured for known ease-of-use and configurability, once use AMQP (instead of JMS for agents) and MQTT had been decided.

[31] https://www.eclipse.org/org/documents/epl-v10.php
[32] http://www.apache.org/licenses/LICENSE-2.0
[33] http://zeromq.org/
[34] http://activemq.apache.org/
[35] https://en.wikipedia.org/wiki/Java_Message_Service

13 Appendix 2: RabbitMQ

13.1.1.1 Producers

A producer is an application that sends messages to an exchange. The producer may be any application written in any programming language, using an AMQP client API. The producer sets the attributes and contents of the message, including routing information, and sends the message to an exchange on a broker host. The producer specifies whether messages should be persisted or transient, and what should happen with messages that cannot be routed to a queue.

13.1.1.2 Messages

An AMQP message consists of a header with attributes and application data. Attributes consist of key-value pairs. The properties consist of optional applications-specific properties and a set of standard message delivery annotations defined by the AMQP specification, e.g. message id, correlation id, time to live, delivery mode, priority, routing key and header dictionary.

The routing key or header dictionary are “addressing” attributes set by the producer to specify which queue(s) a message should be distributed to by the exchange. The delivery mode attribute of a message can be declared persistent by the publisher – it is transient by default. The message must then be persisted between server restarts.

The application data is the actual content of the message, a byte array which is not inspected by the broker. It is entirely application-specific and could be e.g. UTF-8 encoded text, XML, JSON, or Protocol Buffer byte format. AMQP defines an optional type system for specifying content and encoding type of the application data.

13.1.1.3 Exchanges

Messages are sent from a producer to an exchange. Exchanges are defined per message broker host and are responsible for routing the messages to queues. The way the messages are routed depends on the routing keys or headers set by the producer, the queue binding and the type of exchange.

Exchanges can be configured as durable, temporary or auto delete when created. Durable exchanges will survive server restarts and will remain in the broker until explicitly deleted. Temporary exchanges exist until RabbitMQ is shutdown. Auto deleted exchanges are deleted when the last producer or binding are removed from the exchange.

The dead letter exchange is an AMQP extension provided by RabbitMQ. The default behavior of any exchange is to drop messages for which there is no binding providing a matching queue. The dead letter exchange will capture messages that cannot be delivered, which will be an important part of operational management of the system.

13.1.1.3.1 Direct exchange

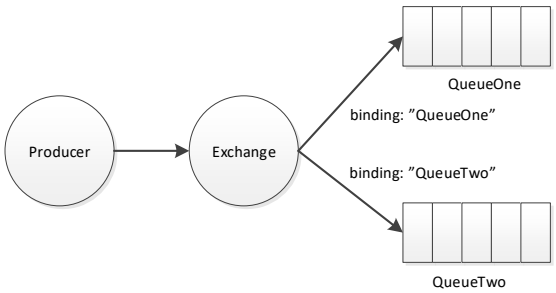

Figure 21: Direct exchange

A direct exchange delivers messages to queues based on the message routing key, see Figure 21. A message is routed to the queues whose binding key is an exact match to the routing key of the message,

e.g. a message with the routing key “log” would be delivered to all queues with the binding key “log”. A common practice is to use the queue name as routing key. If there is no matching binding, the message is discarded. AMQP specifies that an unnamed default exchange must be implemented and that this must be a direct exchange. All queues must be bound to the unnamed exchange using the queue name as routing key.

13.1.1.3.2 Fanout exchange

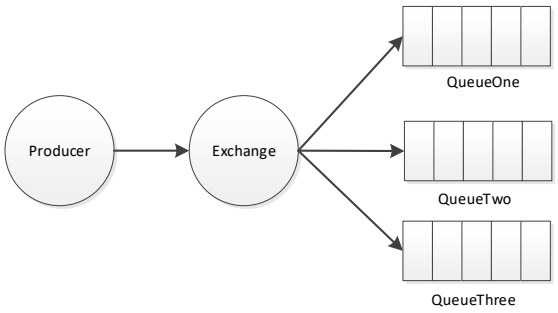

Figure 22: Fanout exchange

In a fanout exchange, messages are routed to all queues that are bound to the exchange. Any routing keys or headers are ignored. This is a useful pattern when broadcasting to several consumers that may process the message in different ways, e.g. logging, notification and aggregation.

13.1.1.3.3 Topic exchange

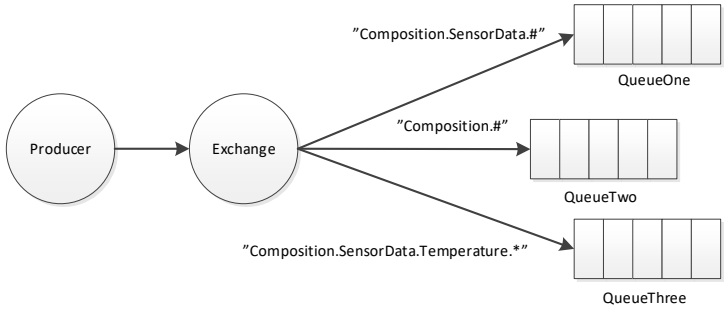

Figure 23: Topic exchange

The topic exchange, see Figure 23, uses the routing key to distribute messages to queues. A topic routing key consists of zero or more words separated by dots “.”, e.g. “Composition.KLE.SensorData”. The binding defines a routing pattern by the same rule, where “*” is used as a wildcard for a single word and “#” is used a wildcard for zero or more words. If the binding for one or several queues matches the routing key, the message is distributed to these queues. This is very similar to the topic hierarchy and matching in MQTT (exchanging “/” for “.”). A typical use for topic exchanges is to implement a publish-subscribe messaging pattern.

13.1.1.3.4 Headers exchange

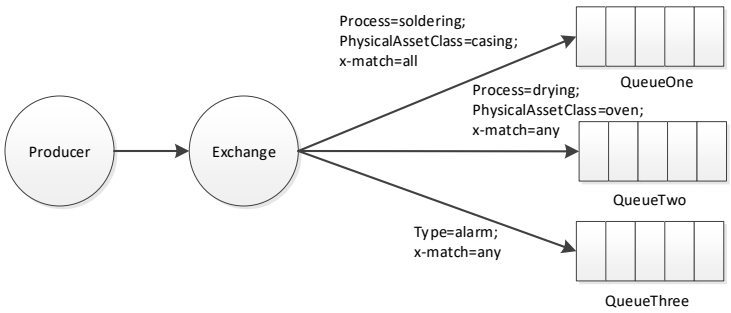

Figure 24: Headers exchange

The headers exchange, see Figure 24, allows for slightly more flexible routing than topic exchanges. The routing key is not used in header exchanges, instead the message headers attribute, containing keys-value pairs, is used. The queue binding specifies the header keys to be matched and (optionally) the values that these should have. If the binding does not specify a value for the header key, it is sufficient for the key to be present in the message header for the binding key to match the message key. If the binding specifies a value for a key, the message header key must match this value. The binding attribute “x-match” specifies whether the logical “AND” or “OR” should be used when combining the matches of header binding keys. If “x-match=all” is specified, all key-value pairs in the binding must match the header for the message to be routed to that queue. The value “x-match=any” indicates that if any of the key-value pairs in the binding matches one or more in the message header, the message will be routed to that queue.

13.1.1.4 Consumers

Any application that receives messages from a queue is a consumer and is identified by the broker by a consumer tag string. The messages can be delivered to the consumer by the AMQP push API or fetched by the consumer using the AMQP pull API. It is possible to register more than one consumer per queue or declare one consumer as the exclusive consumer for the queue. The consumer can send acknowledgement messages back to the host to indicate whether the message has been received or rejected.

13.1.1.5 Queues

Queues are named first-in-first-out buffers in a message broker host that store messages in memory or on disk. The messages are kept in the queue until a consumer connects. The messages are then delivered (in sequence) to the receiving application. The queues can be shared or private to a consumer. When a queue is shared, the name is usually defined by the client, whereas when it is private to the consumer, the server will provide the name. An exclusive queue is associated with a current connection and will be deleted when the consumer disconnects. If the queue is defined as durable, the queue will persist between server restarts. Non-persistent messages may be lost, however.

13.1.1.6 Bindings

A binding is the relation between a queue and an exchange that defines how messages should be routed from the exchange to the queue. Bindings are created or destroyed by applications over time to shape the message flow to queues. When a message arrives at the exchange the message attributes - routing key or header dictionary – set by the producer are evaluated to see if the binding has a match. If the binding matches, the message is copied to the queue. How the matching is done depends on the type of exchange.

13.1.1.7 Messaging Scenarios

AMQP is developed to be a programmable protocol where multiple communication patterns can be set up by the producers and consumers without direct configuration of the server. The main messaging scenarios that the Message Broker will support are the following:

Simple queueing: Where messages are queued between producer and consumer, see fig below, acting as a buffer.

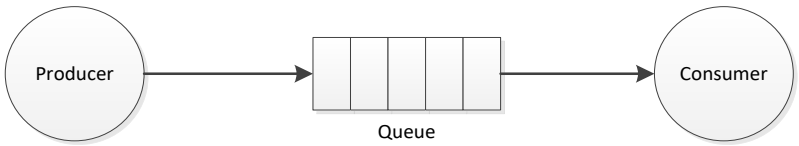

Figure 25: Simple queuing.

Publish/Subscribe: A common pattern for message based architectures in the case of a producer publishes messages typically with a topic pattern and the consumers subscribe to different patterns.

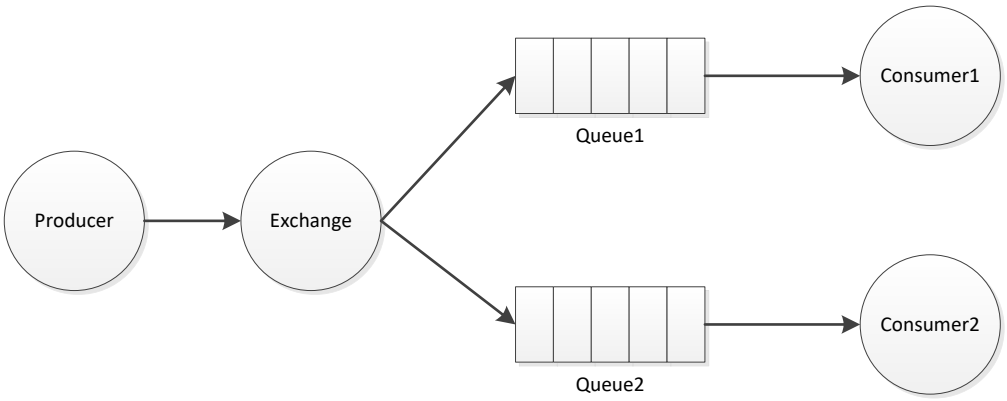

Figure 26: Publish-subscribe

RPC (Remote Procedure Calls): In this case the message broker is used as an exchange and queue for procedure calls. This is useful both for ensuring security as well providing mechanism to manage scalability.

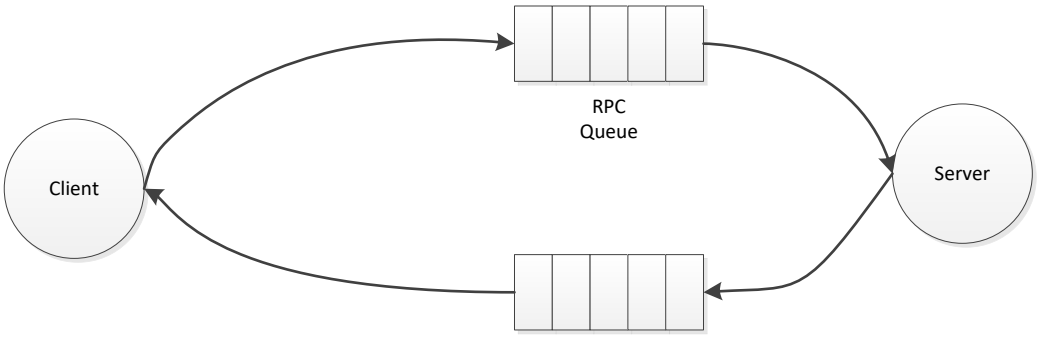

Figure 27: Remote Procedure Call

Competing consumers: Multiple concurrent consumers process messages received on the same message queue. Multiple messages can be processed concurrently to balance the workload and optimize throughput, thereby improving scalability and availability.

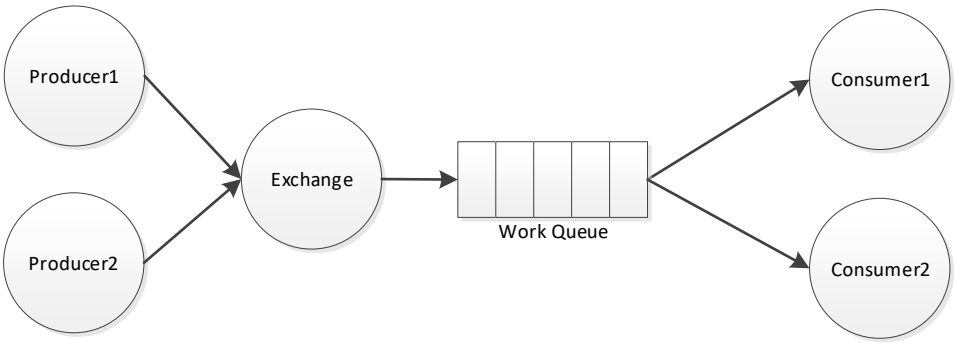

Figure 28: Competing consumers

13.1.1.8 Extensibility

RabbitMQ allows its extension through a variety of plugins that are included with the product or through the implementation of custom ones.

- Available plugins: Some of the plugins bundled with RabbitMQ are described in Table 2, while the complete list can be found at the RabbitMQ web site[36].

Table 2: RabbitMQ bundled plugins

Name	Description
rabbitmq_auth_backend_ldap	Authentication / authorisation plugin using an external LDAP server
rabbitmq_management	A management / monitoring API over HTTP, along with a browser-based UI.
rabbitmq_mqtt	An adapter implementing the MQTT[37] 3.1 protocol.
rabbitmq_stomp	Provides STOMP[38] protocol support in RabbitMQ.

In addition to the mentioned bundled plugins there are available for downloading a set of plugins developed by the RabbitMQ community, these plugins can be found at the RabbitMQ web site[39].

- Custom plugins: As mentioned previously, RabbitMQ allows also the custom implementation of plugins. For the implementation of a plugin knowledge is necessary in Erlang/OTP[40] system and design principles.
  - Erlang: General-purpose, concurrent, functional programming language used to build scalable real-time systems with requirements on high availability.
  - OTP: Set of Erlang libraries and design principles providing middle-ware to develop these systems

13.1.1.9 Platforms

RabbitMQ is available for several platforms, including Windows, MacOS, Linux, BSD and UNIX. It is also available as a Docker image and as Software-as-a-Service cloud offerings.

[36] https://www.rabbitmq.com/plugins.html
[37] http://mqtt.org/
[38] https://stomp.github.io/
[39] https://www.rabbitmq.com/community-plugins.html
[40] https://www.erlang.org/

13.1.1.10 Performance

Performance depends on messaging patterns, message size, persistence and other factors. However, RabbitMQ performs well and is highly scalable. Performance figures ranges from approximately 25000 messages per second on a typical single node deployment (Azure B1 virtual machine), to reports of 10^6 messages per second using a 30-node cluster[41].

13.1.1.11 Licensing

RabbitMQ is distributed under the Mozilla Public License (MPL)[42], a free and open source software license that permits free use, modification, distribution, and exploitation. It entails no limitations to exploitability for the COMPOSITION platform.

[41] https://content.pivotal.io/blog/rabbitmq-hits-one-million-messages-per-second-on-google-compute-engine
[42] https://www.mozilla.org/en-US/MPL/

14 List of Figures and Tables

14.1 Figures

14.2 Tables

15 References

Bondi, A. (2000). Characteristics of scalability and their impact on performance. Proceedings of the second international workshop on Software and performance - WOSP '00.

COMPOSITION. (2016). GRANT AGREEMENT 723145 — COMPOSITION: Annex 1 Research and innovation action.

COMPOSITION. (2017). D2.3 The COMPOSITION Architecture Specification I“. COMPOSITION Consortium.

consortium, C. (2017). D2.3 “The COMPOSITION Architecture Specification I“. COMPOSITION.

Fernandes, J. L. (2013). Performance evaluation of RESTful web services and AMQP protocol. Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on. IEEE.

Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison Wesley.

Hohpe, G., & Woolf, B. (2003). Enterprise Integration Patterns. Addison-Wesley Professional.

Homer, A., Sharp, J., Brader, L. N., & Swanson, T. (2014). Cloud Design Patterns. Microsoft patterns & practices.

IEC. (2013). IEC 62890: IEC Project: Life Cycle Management for Systems and Products used in Industrial-Process Measurement, Control, and Automation. IEC.

IEC62264. (2013). IEC 62264-1: Enterprise-control system integration Part 1: Models and Terminology. IEC.

IEEE. (2000). IEEE 1471 Recommended Practice for Architectural Description for Software Intensive Systems. IEEE.

ISO/IEC/IEEE42010. (2011). ISO/IEC 42010: Systems Engineering – Architecture description. ISO/IEC/IEEE.

ISO19156. (2011). Geographic information -- Observations and measurements. ISO.

Kruchten, P. (2004). The Rational Unified Process: An Introduction. Addison-Wesley Professional.

Lehrig, S., et al (2015). Scalability, Elasticity, and Efficiency in Cloud Computing: a Systematic Literature Review of Definitions and Metrics. Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA '15), Montreal, QC, Canada, May 4–7.

Maciej, R., et al (2014). Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ. Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on. IEEE.

Milagro, F. A. (2008). SOAP tunnel through a P2P network of physical devices. Internet of Things Workshop. Sophia Antopolis: Internet of Things Workshop, Sophia Antopolis.

Rozanski, N., Woods, E. (2012). Software Systems Architecture,: working with stakeholders using viewpoints and perspectives. Addison-Wesley.

(Zwei 2015). Status Report Reference Architecture Model Industrie 4.0 (RAMI4.0). Düsseldorf: VDI e.V.

Wilder, B. (2012). Cloud Architecture Patterns. O'Reilly.

Y.2060, I.-T. (2012). ITU-T Y.2060 : Overview of the Internet of things. ITU.