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

D6.1 Real-time event broker I

Date: 2017-10-30

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: Document
Document version: 1.0
Document owner: COMPOSITION Consortium

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)	2017-10-01	TOC, initial content
0.2	Paolo Vergori (ISMB)	2017-10-24	Added contents to sections 6.9 & 7.2
0.3	Matteo Pardi, Gianluca Insolvibile (NXW)	2017-10-26	Intra-factory contributions
0.4	Giuseppe Pacelli (ISMB)	2017-10-24	Inter-factory sections
0.5	Javier Romero (ATOS)	2017-10-26	Security, deployment
0.6	Mathias Axling (CNET)	2017-10-27	Merged contributions
0.7	Mathias Axling (CNET)	2017-10-27	Additional content
0.8	Mathias Axling (CNET)	2017-10-30	Restructuring and summary
0.9	Mathias Axling (CNET)	2017-10-30	Ready for peer review
1.0	Mathias Axling (CNET)	2017-01-02	Integrated internal review suggestions, ready for submission

Internal review history:

Reviewed by	Date	Summary of comments
Alexandros Nizamis (CERTH)	2017-10-31	Wrong enumeration of some figures and wrong format of tables. Suggestion for better explanation of this deliverable's outcome and conclusions. Minor syntax suggestions.
Tracy Brennan (BSL)	2017-10-31	Minor comments

Index:

1 Executive Summary

This deliverable presents the current state of the design of the real-time event broker infrastructure of the COMPOSITION system. During the first year of the project, evaluation of alternative implementation mechanisms, choice of protocols and investigation into scalability concerns have been performed. The design of the real-time event broker integration with other components has been performed.

The broker is a principal component in realizing both of COMPOSITIONs 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 crates 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. A first version of the real-time event broker has been deployed as a set of Docker images at the COMPOSITION intra-factory test server.

[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.
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 will be followed by D6.2 “Real-time event broker II”, which will provide an updated description at M26 of the project.

Preliminary results in this deliverable has been reported in D2.3 “The COMPOSITION Architecture Specification I“ (COMPOSITION D2.3, 2017). 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”. This report will include an overview of this integration.

3.2 Content and structure of this deliverable

The structure of this deliverable is aligned with that of the architecture description deliverable (COMPOSITION D2.3, 2017). 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. 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.

Section 7 – Describes the work on information models relevant to the real-time event broker; the information view. Much of this is still future work, however.

Section 8 – Provides an overview of the deployment view.

Section 9 – Addresses the scalability perspective.

Section 10 - Addresses the security perspective.

Section 11 - Provides an overview on how future design will proceed.

Section 12 - Presents a summary of the current state of development with conclusions.

4 Background

4.1 Message broker

The term message broker is used interchangeably with real-time event broker throughout this report. A message brokers decouples the destination of a message from the sender and maintains central control over the flow of messages. Messages can be received from multiple destinations and routed to the correct destination while implementing a multitude of messaging patterns (Hohpe & Woolf, 2003).

The architecture inception phase reported in the COMPOSITION project specification (COMPOSITION, 2016) identifies the message broker as a principal component in the IIMS and marketplace design. 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, Amazon AWS IoT Message Broker). However, there are some significant additions and integrations in performed COMPOSITION that contribute to added value from the component.

4.2 RAMI4.0 Communication Layer

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

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.

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[6]. There is both a binary and HTTP protocol specified for 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://opcfoundation.org/developer-tools/specifications-unified-architecture/part-14-pubsub

4.3.4 XMPP

Extensible Messaging and Presence Protocol (XMPP) is an open XML-based messaging technology standardized by IETF XMPP working group. Originally developed as Jabber for instant messaging and still predominantly used in this area, there are now applications for general messaging and IoT based on its support for federation, publish/subscribe messaging, and client security. XMPP has been considered for marketplace agent communication in COMPOSITION.

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 integration with existing systems are concerns expressed in these requirements.

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).

6 Design of the Real-time Event 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 Real-time Event Broker, or 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 to this data without direct addressing between components. This will be built using standard message broker components with extensions for security, multi-protocol and multi-format support.

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 components in the intra-factory IIMS that already implement MQTT support.

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.

6.2 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. Repeated from (COMPOSITION D2.3, 2017) for readability, the main messaging scenarios that the Real Time Multi-Protocol Event Broker will support are the following:

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

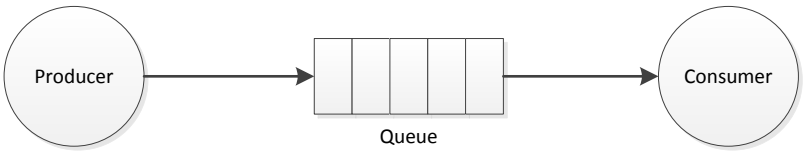

Figure 1: Simple queuing.

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.

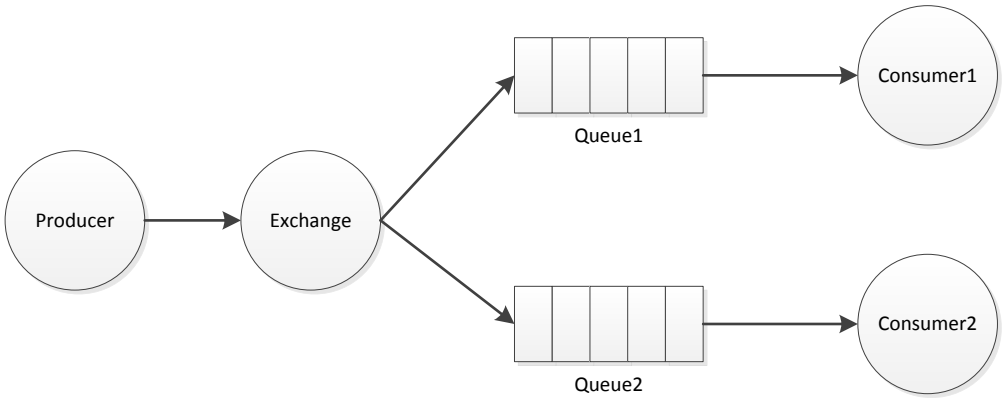

Figure 2: Publish-subscribe

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.

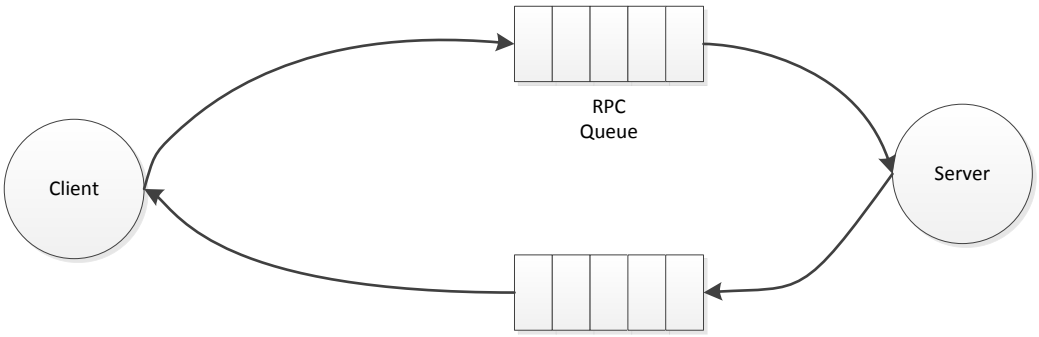

Figure 3: Remote Procedure Call

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.

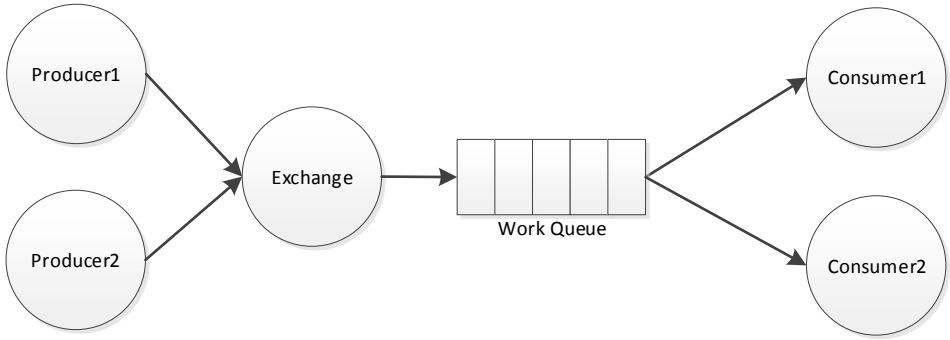

Figure 4: Competing consumers

6.3 Considered implementations

While RabbitMQ was the chosen message broker implementation in the architecture inception phase (COMPOSITION, 2016), other candidates have been evaluated as complements or substitutes for the sensor and agent platform broker. This section provides a brief overview of these.

6.3.1 Mosquitto

Eclipse Mosquitto is an open source, Eclipse licensed (EDL/EPL[7]) 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.

6.3.2 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[8] 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).

6.3.3 ZeroMQ

ZeroMQ [9](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.

6.3.4 ActiveMQ

Apache ActiveMQ[10] 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[11], 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 on.

6.4 RabbitMQ

The COMPOSITION project has selected RabbitMQ[12] as the implementation mechanism for the message broker as suggested from the inception phase documented in the project description (COMPOSITION, 2016). RabbitMQ is an open source component supplied under the Mozilla Public License. RabbitMQ is a widely used open source message broker[13] 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. 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.

[7] https://www.eclipse.org/org/documents/epl-v10.php
[8] http://www.apache.org/licenses/LICENSE-2.0
[9] http://zeromq.org/
[10] http://activemq.apache.org/
[11] https://en.wikipedia.org/wiki/Java_Message_Service
[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

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 can all reside on different brokers. The following section describes these basic concepts of AMQP 0.9.1 in the context of the RabbitMQ implementation.

6.4.1 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.

6.4.2 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.

6.4.3 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.

6.4.3.1 Direct exchange

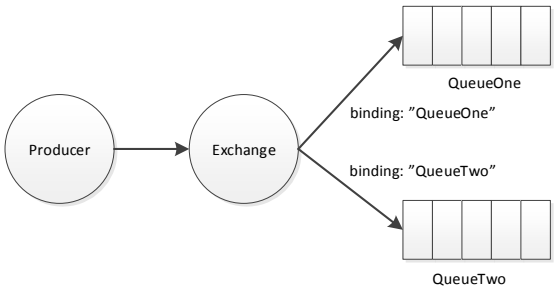

Figure 5: Direct exchange

A direct exchange delivers messages to queues based on the message routing key. 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.

6.4.3.2 Fanout exchange

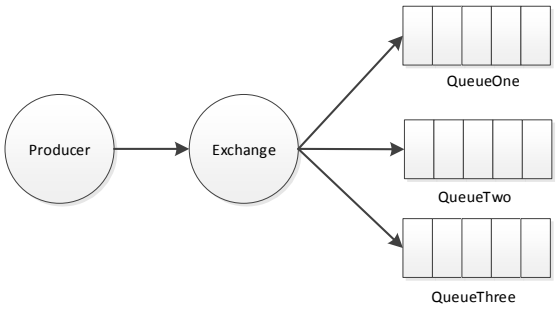

Figure 6: Fanout exchange

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.

6.4.3.3 Topic exchange

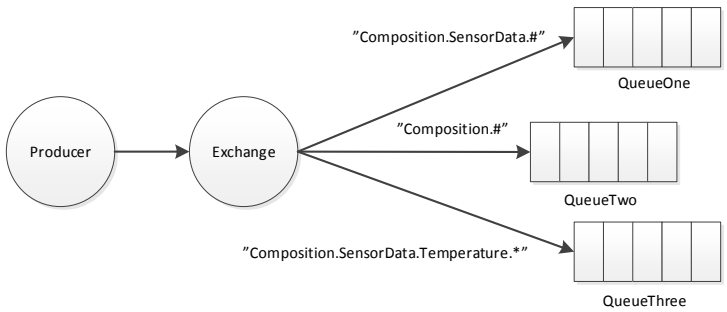

Figure 7: Topic exchange

The topic exchange 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.

6.4.3.4 Headers exchange

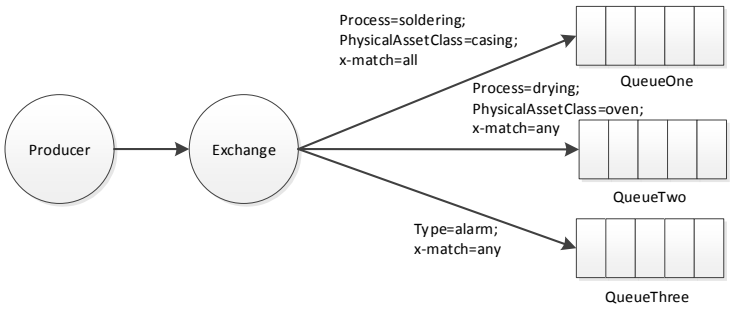

Figure 8: Headers exchange

The headers exchange 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.

6.4.4 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.

6.4.5 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.

6.4.6 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.

6.5 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[15].

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[16] 3.1 protocol.
rabbitmq_stomp	Provides STOMP[17] 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[18].

- 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[19] 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

6.6 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.

6.7 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[20].

6.8 Licensing

RabbitMQ is distributed under the Mozilla Public License (MPL)[21], 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.

6.8.1 FITMAN

The SEM[22] (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

[15] https://www.rabbitmq.com/plugins.html
[16] http://mqtt.org/
[17] https://stomp.github.io/
[18] https://www.rabbitmq.com/community-plugins.html
[19] https://www.erlang.org/
[20] https://content.pivotal.io/blog/rabbitmq-hits-one-million-messages-per-second-on-google-compute-engine
[21] https://www.mozilla.org/en-US/MPL/
[22] http://www.fiware4industry.com/?portfolio=secure-event-management-sem

RabbitMQ with blockchain technology and Keycloak [23]identity and access management. COMPOSITION will also provide micro services for real-time message and protocol translation under high loads, where necessary.

6.9 Intra-factory Real-time Multi-Protocol Event Broker

The COMPOSITION infrastructure leverage on a broker based protocol for handling communications within the intra-factory components. This agreement has been reached between the two involved work packages in which this interoperability is required to be outputted, namely work package 5 “Key enabling technologies for Intra and Inter-factory interoperability and data analysis” and work package 6 “COMPOSITION collaborative ecosystem”. In specific task 5.5 “Adaptation layer for intra-factory interoperability” and task 6.1 “Real-time event brokering for intra-factory interoperability”. The latter is represented in this deliverable and has specifically tested multiple broker based system in heterogeneous scenarios, making a final choice in an MQTT protocol supporting software, namely RabbitMQ.

The aforementioned task 5.5, has drawn the component connections and has set the requirements for the intra-factory interoperability layer, whereas task 6.1 has produced the broker based connections. Figure 9 depicts the components connected by the intra-factory and used as an input to this document.

[23] http://www.keycloak.org/

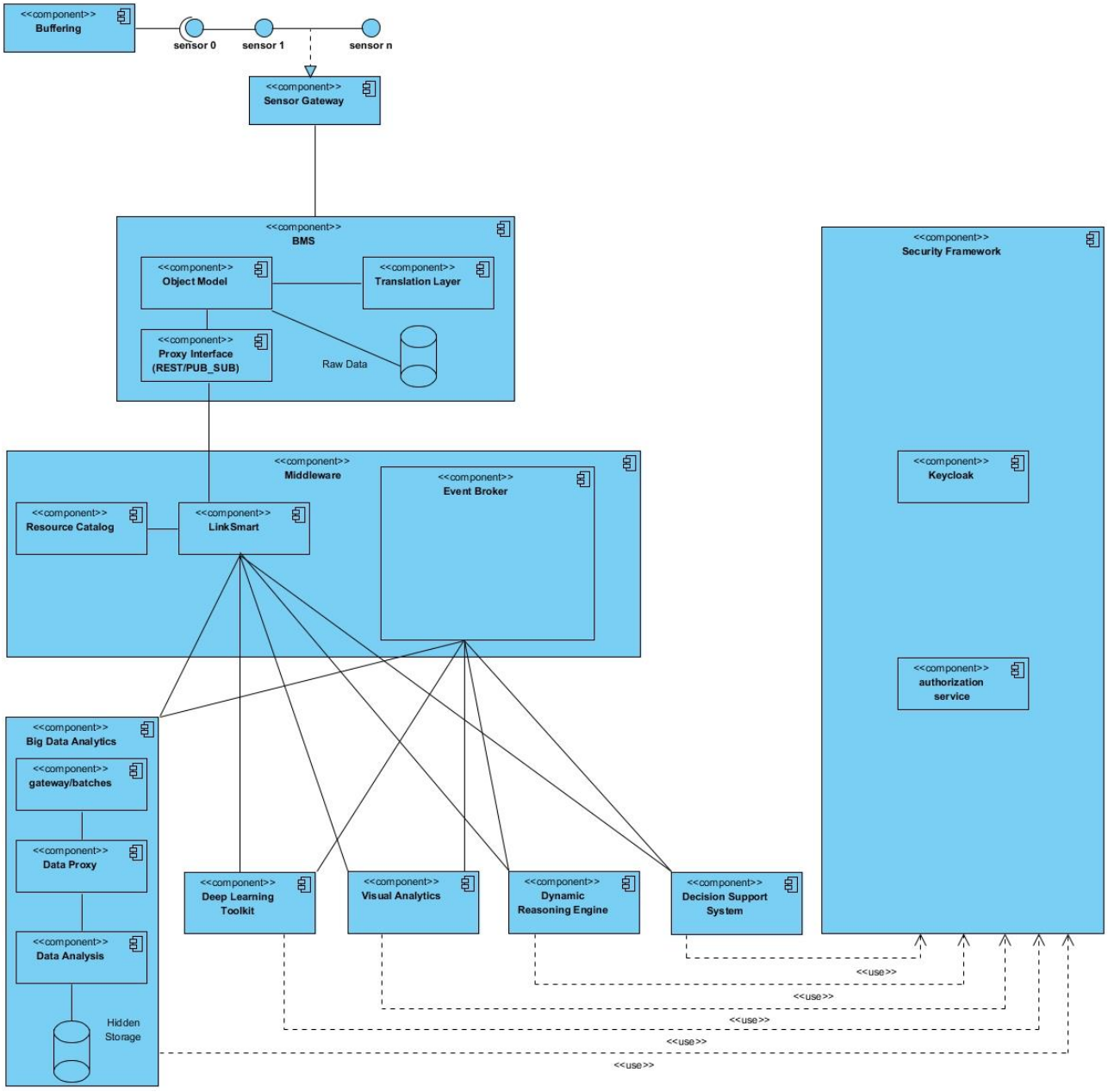

Figure 9: Intra-factory components

In task 6.1 the broker based system has been characterized based on the design concerns described in section 5. It has also been tested and deployed in a first draft of the Docker container that will encapsulate it in its final form. The brokering system will be the main distribution point for the intra-factory scenario. In this way, there will be a single point 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 9, 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.
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. Every message that passes to the event broker is 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.
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.

In this overall picture, a common data format is required for having data consumed by components in coherent manner. Hence, data are dispatched in an OGC form, in order to be available to every component in a common manner. From the OGC SensorThings API specification:

> *The OGC SensorThings API provides an open, geospatial-enabled and unified way to interconnect the Internet of Things (IoT) devices, data, and applications over the Web. At a high level the OGC SensorThings API provides two main functionalities and each function is handled by a part.*

Despite the data standard being already scouted and confirmed, the data format is still under discussion. There are some proposals on using a subset of existing standards that are available in an XML format and translate them in a JSON OGC compatible data structure. This information will be consolidated in the next reports and will be made available.

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.

The Building Management System, as shown in Figure 9, 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 intra-factory interoperability layer both provides a model for interconnecting the COMPOSITION ecosystem in the intra-factory scenario and ensures the conformity between communications among interconnected components. The Building Management System (BMS), provided by a project development stakeholder (NXW), will be the translation layer providing shop floor connectivity from sensors to the COMPOSITION system. In this way, information is pervasively collected from any connected systems in order to support the management operators in making decisions and to take direct control for automation tasks. Since the BMS enables the connection with all the major automation standards (such as BACnet, Konnex, Modbus, etc.) allowing to seamlessly inter-connect them altogether, it will act as a bridge between the cyber-physical systems (sensors, gateways, etc.) and the other IIMS components.

For this reason, the BMS, once it has gathered the raw data coming from the field, must organise it into a uniform Data Model. This model provides a representation of sensor and actuator data which is independent of the physical type of underlying devices. 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. The OGC SensorThings Data Model is used for system-generated data, i.e. data in the IIMS that has passed through the BMS and is exposed in inter-component communication will use the OGC SensorThings Data Model, with links to the DFM types and instances.

The BMS Hardware Abstraction Layer (HAL) and COMPOSITION Object Mapper expose a virtualized version of the underlying physical objects from which information can be read and actuations can be performed, thus providing the equivalent of an Administration Shell in the RAMI architecture. Unlike what has been stated in deliverable (COMPOSITION D2.3, 2017) regarding the IIMS architecture, LinkSmart won't be responsible anymore for forwarding information towards the other IIMS components and it will be used only as a Resource Catalogue: the real-time and historical data connectors will be provided by the BMS as well.

All this data will be made available for the upper component (the message broker) using a message-oriented architecture such as MQTT, attaching to this some extra information though additional metadata (described with RDF-A or JSON-LD format). As an intermediary, decoupling system components, the Message Broker also provides the means to manage scalability in a consistent manner.

6.10 Inter-factory Market Event Broker

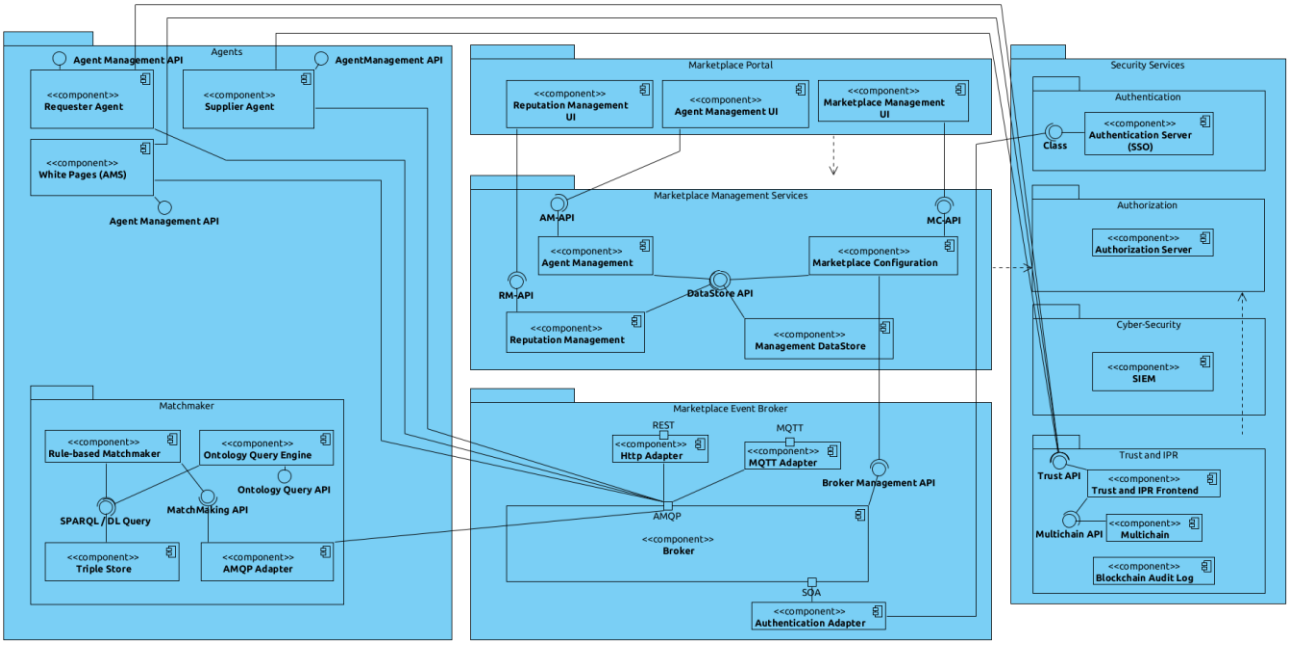

Figure 10: Marketplace components.

The message broker instance used in the COMPOSITION collaborative marketplace 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. Future updates of the Real-time event broker may introduce features e.g. to validate the correctness of sender and receiver identifies, as well as to perform checks on message correctness and

compliance against the formats of the COMPOSITION eXchange Language (CXL). At the moment, such features have not been yet planned, because of the expectation that the computational load at agents' side is not a major issue, while such checks may drastically decrease the scalability on broker's side (what if messages are encrypted?).

As previously stated in deliverable (COMPOSITION D2.3, 2017), agents only communicate through CXL, which has been designed in order to be FIPA-ACL compliant. Messages not compliant with CXL language should never be exchanged and they would be dropped at agents' side.

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 furtherly implemented as a direct evolution of the open one.

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 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 must be 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. For the time being, this is work in progress and the definitive results will be published in the next iteration of this report at M26 in the project.

7.1 Inter-factory Market Event Broker

The agent communication in the marketplace uses AMQP. A pre-defined set of topics and formats for message exchanges will be specified as part of the COMPOSITION specifications to support communication between agents. Only a set of predefined schemas for messages should be accepted. 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. This is part of the specification of a COMPOSITION Marketplace compliant agent.

7.2 Intra-factory Real-time Event Broker

MQTT is used for the distribution of sensor data and processed information in the factory IIMS. In the intra-factory scenario there is the need to create a hierarchy that defines the topics structure for the event broker. The COMPOSITION project sets a background that will be common among all components that will be identified by:

- a topic root that will use the “Composition” tag as identifier;
- the discriminative dichotomy identifier of the intra or inter factory scenario;
- the component name that is in charge of generating the data;
- the scope of the data produced.

For instance, we can provide a simple example for this nomenclature that will be used by the Deep Learning Toolkit component that is developed in task 5.2. This component dispatches the latest available prediction, in the format of an event, providing an update whenever a new prediction is available. So, if we take this scope as an example it will use the topic with the following nomenclature:

```
Composition/IntraFactory/DeepLearningToolkit/LatestPrediction
```

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.

It is expected in the next iteration of this document, as the integration among intra-factory components progresses and will be more prominent, the definitive topics list to be published alongside its end-to-end subscriptions’ list for each publishing entity with a scenario of 1-N cardinality.

8 Deployment view

As described in (COMPOSITION D2.3, 2017), Docker will be used for deployment in COMPOSITION. While using a single host for development and testing, configurations for wide-area, heterogenous deployment can be tested there and later deployed on separate nodes. The process of preparing the COMPOSITION components for Docker deployment is in final phase and some components have already been deployed for test.

Intra-factory real-time event broker has been already deployed as a Docker container in the COMPOSITION Test Environment. An overview of current deployment of and the related security framework components is shown in Figure 11.

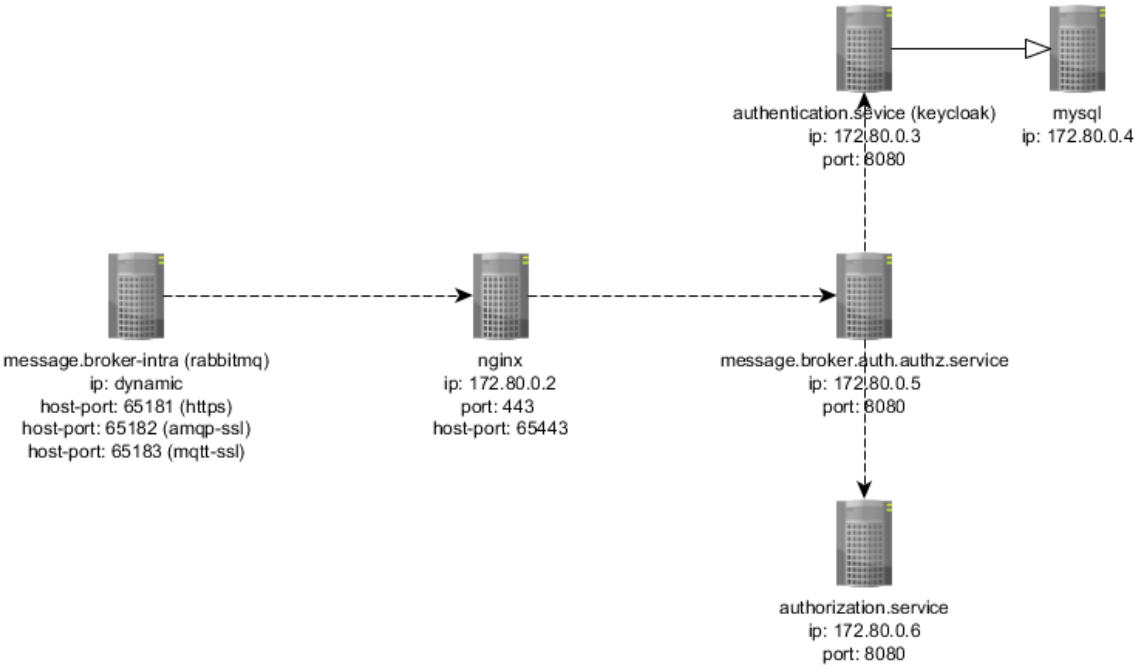

Figure 11: Intra-factory real-time event broker Docker deployment

The deployment of the Inter-factory Market Event Broker will have a similar topology as the one described for the Intra-factory real-time event broker, with the exception of the ports exposed. An overview of the deployment is presented in Figure 12.

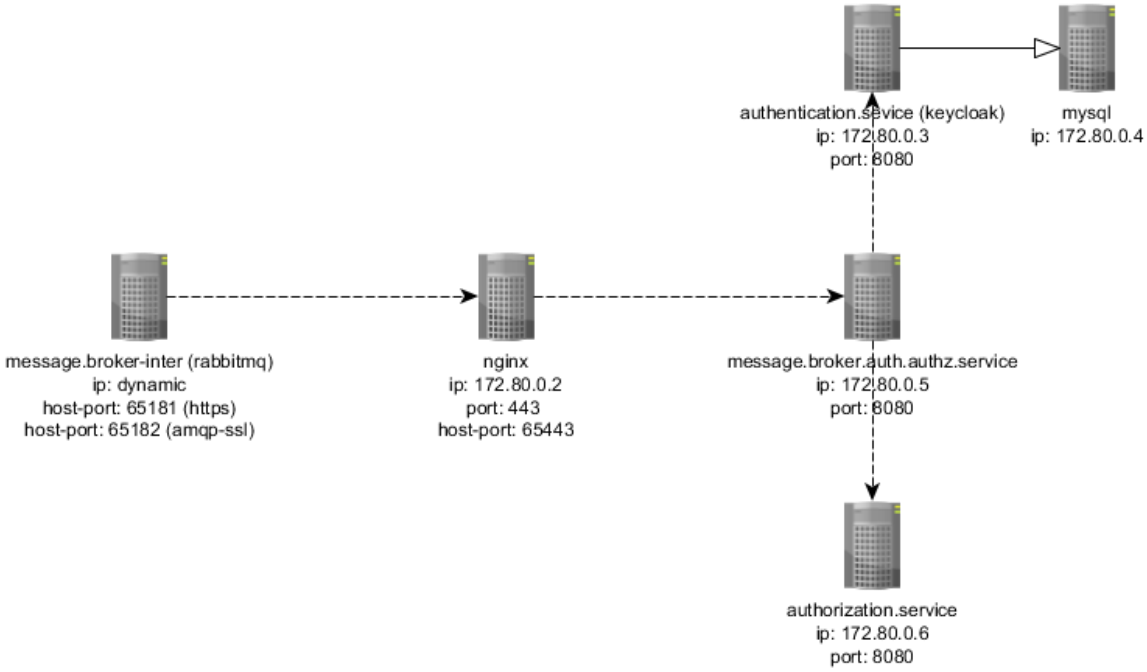

Figure 12: Inter-factory market event broker Docker deployment

9 Scalability Perspective

The massage 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 performance of a component is the capability to handle a specific workload, given a specific set of resources, e.g. CPU cycles, memory and disk space. 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 by adding capacity within the existing nodes – scaling up or vertical scaling – and by adding more nodes – scaling out or horizontal scaling. (COMPOSITION architecture deliverable (COMPOSITION D2.3, 2017) has a more thorough section on this subject.)

Based on prior experience (ALMANAC project, PICASO project, FITMAN SEM) and published performance figures[24][25] (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 real-world scenarios. 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 D2.3, 2017), the centralized approach to 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 is likely to applicable in COMPOSITION. RabbitMQ is also available as highly scalable cloud services[26].

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).

9.1 RabbitMQ scalability

This section will describe the techniques 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.

9.1.1 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[27] 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. 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.

[24] https://www.rabbitmq.com/blog/tag/performance/
[25] http://underthehood.meltwater.com/blog/2016/09/01/rabbitmq-performance/
[26] https://www.cloudamqp.com/
[27] And other entities, e.g. virtual hosts, users, permissions, runtime parameters, et c.

.

9.1.2 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 exchanges are connected 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.

Federated queues are also connected one-to-one, in one direction. 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.

9.1.3 “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 COMPOSITION extensions

The COMPOSITION integration with Keycloak, 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 above, as we can manage users for all brokers from one Keycloak system, whether in a federation or connected with ”the shovel”.

9.3 Message broker uses for scalability

The versatile and programmable design of the message broker allows for implementing common scalability design patterns (COMPOSITION D2.3, 2017) (Wilder, 2012) for the connected COMPOSITION components, e.g. load balancing, queue based load levelling and competing consumers.

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 is being developed. With the use of rabbitmq-auth-backend-http[28] community plugin RabbitMQ built-in authentication and authorization can be overridden and managed from outside with other components as already said.
The adapter in development is a web-service developed in Node.js and exposes the following endpoints required by the plugin:

- https://server:port/auth/user: Used to authenticate a user providing username and password.
- https://server:port/auth/vhost: Used to authorize access to a virtual host.
- https://server:port/auth/resource: Used to authorize access to a resource.
- https://server:port/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. All tokens are stored in an in-memory database for fast access and they are never replicated to the filesystem for security reasons.
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[29] (JWS) standard.

An adapter for the blockchain distributed trust mechanism will be built to allow 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 following sections 10.1 and 10.2 will present the differences between Inter-factory and Intra-factory event brokers' configuration and related security components.

10.1 Inter-factory Market Event Broker

RabbitMQ will only support default AMPQ[30] protocol over TLS in Inter-factory deployment.

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 13 depicts the architecture and the relation between the message broker and the security components.

[28] https://github.com/rabbitmq/rabbitmq-auth-backend-http
[29] https://tools.ietf.org/html/rfc7515
[30] https://www.amqp.org/

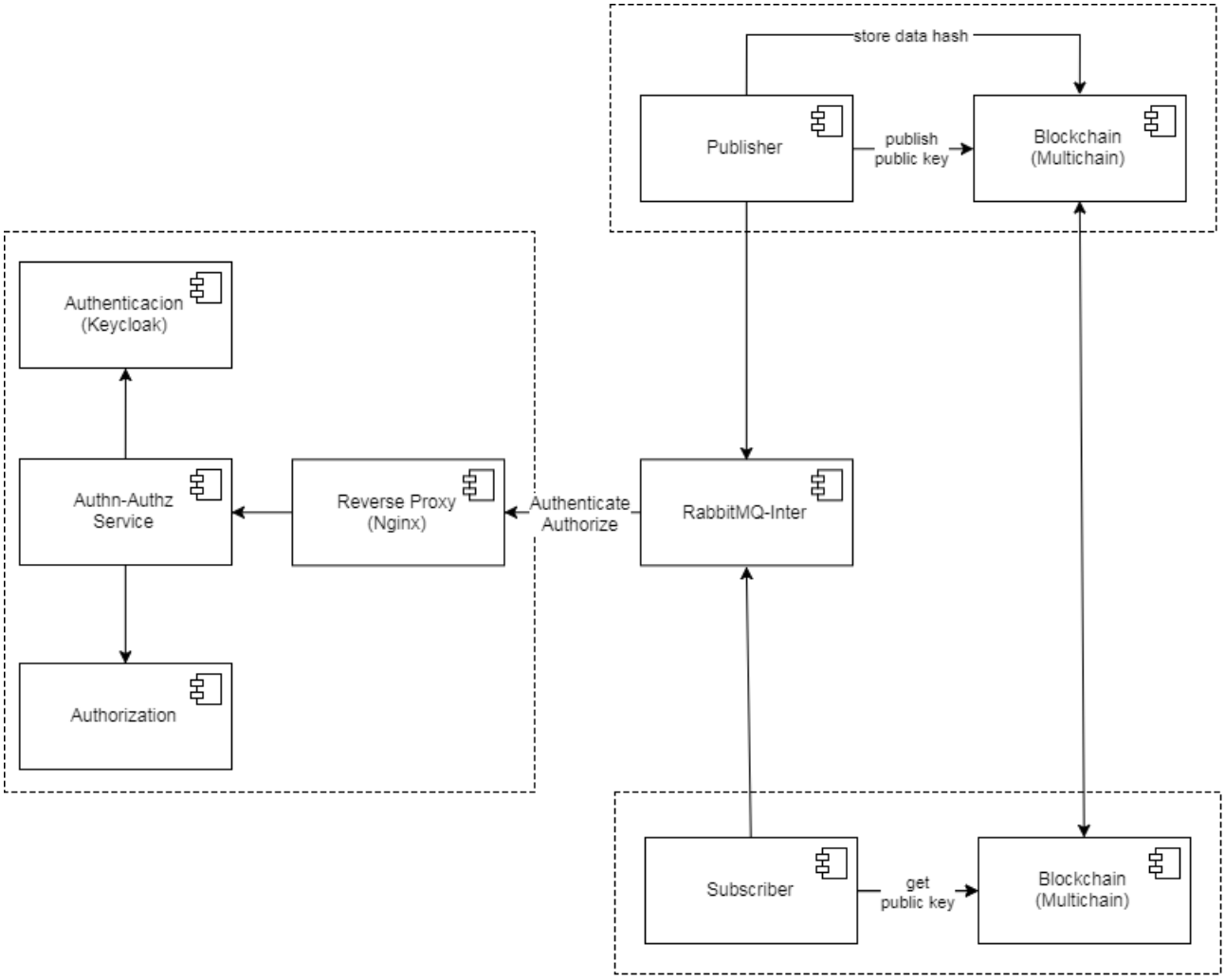

Figure 13: Inter-factory Market Event Broker security architecture

10.2 Intra-factory Real-time Event Broker

In this case, RabbitMQ will support two different messaging protocols; default AMPQ protocol and MQTT[31] 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 14: Intra-factory Real-time Event Broker security architectureFigure 14 gives an overview of the architecture and the relation between the message broker and the security components.

[31] http://mqtt.org/

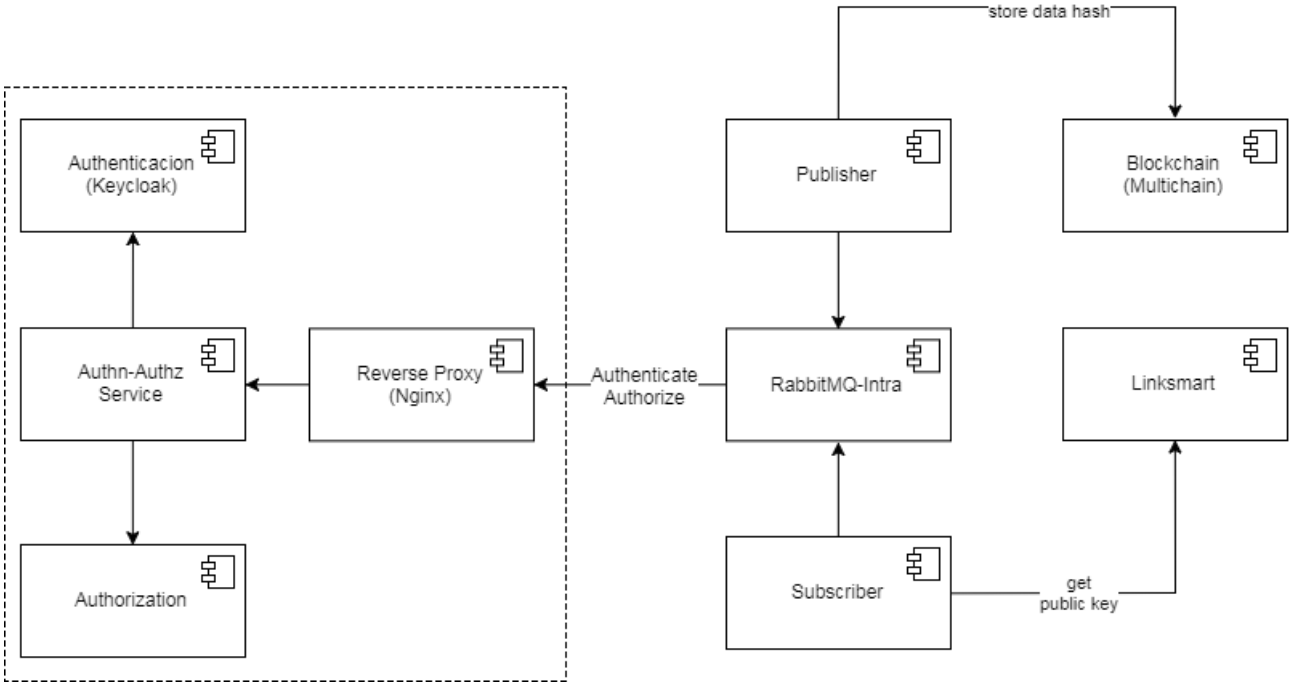

Figure 14: Intra-factory Real-time Event Broker security architecture

11 Future work

11.1 Information view

As mentioned in section 7, there is ongoing work on specifying the topic schemas for publishing and subscribing to messages from the broker. There will be multiple schemas, depending on the type of communication. Tightly coupled components, e.g. Big Data Analytics and Deep Learning Toolkit, may use a specific schema most efficient for the specific purpose. General sensor and factory data for consumption by loosely coupled components use another schema, and the marketplace use a schema based on the marketplace ontology. Moreover, the work queue configurations and operational data will need different schemas.

Technical scenarios for scalability will have to be developed to verify the findings of the evaluation of scalability designs for the message broker. These scenarios will be used to choose the combination of scalability design mechanisms used and design the concrete implementations.

11.2 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 propose 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 will provide decoupling of services, logical addressing of services, discovery and an integrated security solution for HTTP, MQTT and AMQP communication. RabbitMQ already provides support for RPC style request-response messaging, including facilities for sending responses directly to the client channel without a client queue[32]. A non-generic proof-of-concept has been developed in COMPOSITION, and there are other publicly available projects[33].

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.

[32] https://www.rabbitmq.com/direct-reply-to.html
[33] https://github.com/dsyer/http-amqp-tunnel

12 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. During design, MQTT was selected for factory sensor data and AMQP was selected for internet communication between heterogenous systems in the COMPOSITION Marketplace.

The project evaluated different broker implementations. This resulted in the candidate from the architecture inception phase, RabbitMQ, being confirmed. While RabbitMQ is not the fastest message broker, 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 will 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 alleviated the concerns for bottlenecks and a centralized point of failure. However, these findings will have to be evaluated against a set if concrete scalability scenarios.

Significant extensions to RabbitMQ undertaken in COMPOSITION include integration of end-to-end security with blockchain, REST-tunnelling, and to some extent the use of AMQP for platform-independent agent communication. A message broker with security framework integration has been deployed at the COMPOSITION intra-factory Docker host test environment. The design of topic schemas for the intra- and inter-factory message brokers has started and will be prioritized in the near future. The result of ongoing and future work, including concrete implementations of topic schemas for MQTT/AMQP, finalization of security integration and REST-tunnelling, will be presented in the follow-up report D6.2 “Real-time event broker II” at M26.

13 List of Figures and Tables

13.1 Figures

13.2 Tables

14 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.