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

D6.6 Connectors for Inter-Factory Interoperability and Logistics II
Date: 2019-06-29
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: D6.6 Connectors for Inter-Factory Interoperability and Logistics II_v1.0.docx
Document version: 1.0
Document owner: LINKS

Work package: WP6 – COMPOSITION Collaborative Ecosystem
Task: T6.3 - Connectors for Inter-Factory Interoperability and Logistics
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	Jure Rosso (LINKS)	2019-22-05	Initial content from D6.5
0.2	Alexandros Nizamis, Nikolaos Vakakis and Christos Ntinas (CERTH)	2019-06-06	Update Matchmaker section
0.3	Jure Rosso (LINKS)	2019-14-06	Updated APIs and schemas
0.4	Mathias Axling (CNET)	2019-18-06	Update of 6.3, 6.4
0.5	Jure Rosso (LINKS)	2019-20-06	Updated AMS-Matchmaker schema
0.6	Nacho Gonzáles (ATOS), Mathias Axling (CNET)	2019-20-06	Security section update
0.7	Jure Rosso (LINKS)	2019-21-06	Finalization for internal review
1.0	Jure Rosso (LINKS)	2019-27-06	Changes according to reviewers' suggestions

Internal review history:

Reviewed by	Date	Summary of comments
Jannis Warnat (FIT)	2019-06-24	The document is good.
Alexandros Nizamis and Vagia Rousopoulou (CERTH)	2019-06-25	The document is good. Some minor comments have been made regarding mostly the structure and format of the document.

Index:

1 Executive Summary

The present document named “D6.6 Connectors for Inter-factory Interoperability and Logistics II” is a public deliverable of the COMPOSITION project, co-funded by the European Union’s Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 723145. This deliverable presents the final results of the Task 6.3 “Connectors for Inter-Factory Interoperability and Logistics”. It aims to describe and analyse the final version of COMPOSITION Marketplace’s components.

COMPOSITION has two main goals:

1. The integration of data along the value chain from the inside of a factory into one integrated information management system (IIMS).
2. The creation of a (semi-)automatic ecosystem that extends the local IIMS concept to a holistic and collaborative system incorporating and interlinking both the supply and the value chains.

The purpose of this deliverable is to further describe the development process related to the generation of the (semi-)automatic ecosystem.

Particularly, the analysis will focus on:

- The design of the different connectors between the various marketplace components:
  - Marketplace Agents
  - Stakeholder Agents
  - Matchmaker Agent
  - Security framework
- The implementation of the connectors described above.
- A short description of every component.

The results of the analysis of all the aforementioned features have been implemented in the COMPOSITION Marketplace’s components.

The Agents have been developed in a full custom way to avoid constraints coming from frameworks usage. The existing technologies have been analysed as state of the art, then, due to the project needs, the COMPOSITION Marketplace has been developed without using any of these frameworks but taking inspiration from those ones.

The Agent Management Service is the core Marketplace Agent that enable Stakeholder Agents to register to the COMPOSITION Marketplace and to take part to the negotiation process.

Another Marketplace Agent developed is the Matchmaker. It supports semantic matching in terms of manufacturing capabilities, in order to find the best possible supplier to fulfill a request for a service with raw materials or products involved in the supply chain. Different decision criteria for supplier selection are considered by the Matchmaker according to several qualitative and quantitative factors.

The two Stakeholder Agents, requester and supplier work together to achieve the goal of creating new supply chains by negotiating in a semi-automatic way. They receive command through the UI from the owners exploiting a set of REST APIs developed during the project and reported in this document. The negotiation is enabled by COMPOSITION eXchange Language, a language derived from FIPA ACL standard. This language provide high flexibility thanks to a large set of action and the description of the message content by means of custom ontologies.

The Security Framework together with the Blockchain layer is strongly interlaced with the COMPOSITION ecosystem, providing all the necessary security features as authentication, authorization, message integrity and message traceability.

The COMPOSITION Marketplace brings a new and dynamic way in supply chain creation. It has shown an easy and successful way in doing negotiation in a semi-automatic manner outperforming the old mechanism based on a full manual process that usually takes much more time.

2 Terminology

The currently adopted domain-specific terminology used in the remainder of the document is presented in Table 1 below.

Table 1: Terminology

Term	Definition
AMS	Agent Management Service
AMQP	Advanced Message Queueing Protocol
API	Application Programming Interface
CRUD	Create Read Update Delete
CXL	COMPOSITION eXchange Language
DCO	Distributed Constraints Optimization
FIPA	Foundations for Intelligent Physical Agents
HTTP	HyperText Transfer Protocol
IIMS	Integrated Information Management System
JSON	JavaScript Object Notation
REST	REpresentational State Transfer
SPARQL	Simple Protocol and RDF Query Language
TLS	Transport Layer Security
UI	User Interface

3 Introduction

3.1 Purpose, Context and Scope of this Deliverable

This deliverable presents the actions performed on the integration regarding the different components involved in the COMPOSITION Marketplace. The work has been carried out mainly in Work Package 6 (WP6), “COMPOSITION Collaborative Ecosystem”. 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 6.3 “Connectors for Inter-Factory Interoperability and Logistics”
- Task 6.4 “Collaborative manufacturing services ontology and language”
- Task 6.5 “Brokering and Matchmaking for Efficient Management of Manufacturing Processes”

This deliverable is preceded by D6.5 “Connectors for Inter-Factory Interoperability and Logistics I”, which provided the first version of the development.

The communication design is tightly integrated with the Security Framework, reported in D4.2 “Design of the Security Framework II”. This report will include an overview of this integration.

3.2 Content and Structure of this Deliverable

The structure of this deliverable is as follows:

Section 4 – Provides an overview of the protocols used in the current implementation.

Section 5 – Describes the development status of the Agents, together with the interaction protocol.

Section 6 – Describes, from a high-level perspective, the connections between the different systems.

Section 7 – Presents a summary of the current development state with conclusions.

4 Background

4.1 Multi-Agent System

Multi-Agent Systems (MAS) have been widely investigated in research, their application domains range from Distributed Constraints Optimization (DCO) problems to coordination and delegation of computational tasks. While the adoption of agent systems in automatic negotiation, i.e., for DCO problems, is not new, as witnessed by the huge amount of literature available, the application of such techniques in real-industrial environments, in a fully decentralized set-up still presents some research challenge and offers possibilities for advancing the state of the art. The COMPOSITION marketplace can be seen as a particular variation of a Multi-Agent System.

4.2 Protocols

In this section, we will perform a high-level analysis of the most important protocols used for the communication between the different marketplace components.

4.2.1 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 several software vendors.

4.2.2 MQTT

MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium. For example, it has been used in sensors communicating to a broker via satellite link, over occasional dial-up connections with healthcare providers, and in a range of home automation and small device scenarios. It is also ideal for mobile applications because of its small size, low power usage, minimised data packets, and efficient distribution of information to one or many receivers.

4.2.3 RESTful APIs

REST (REpresentional State Transfer) is an architectural style for developing web services. It is neither an architecture nor a standard, but a set of guidelines allowing the realization of system architecture. Resources can be identified by URIs, that might allow CRUD (Create, Read, Update, Delete) operation on such resource. In fact, CRUD operations can be mapped on HTTP methods as shown in Table 2:

Table 2: Mapping between CRUD and HTTP methods

CRUD method	HTTP method
Create	POST
Read	GET
Update	PUT
Delete	DELETE

One of the main advantages in using RESTful APIs is that client and server can be completely independent, therefore allowing easy communication between systems developed on different platforms and written in different programming languages.

4.2.4 TLS/SSL

TLS/SSL are cryptographic protocols that provide communication security over a computer network. They provide a secure means of communication between two entities, preventing data tampering, data forgery and man-in-the-middle attacks.

5 Agents

The COMPOSITION marketplace is a fully distributed multi-agent system designed to support industry 4.0 exchanges between involved stakeholders. It is particularly aimed at supporting automatic supply chain formation and negotiation of goods/data exchanges. The COMPOSITION marketplace exploits a microservice architecture (based on Docker) and relies upon a scalable messaging infrastructure. Trust and security are granted in every negotiation step undertaken by automated agents on behalf of involved stakeholders.
The marketplace includes the following elements: (a) a Marketplace management portal; (b) an easy to deploy Marketplace core based on Docker; (c) a set of "default" agent implementations ready to be adopted by interested stakeholders.

The main building blocks of the Marketplace are displayed in Figure 1 below.

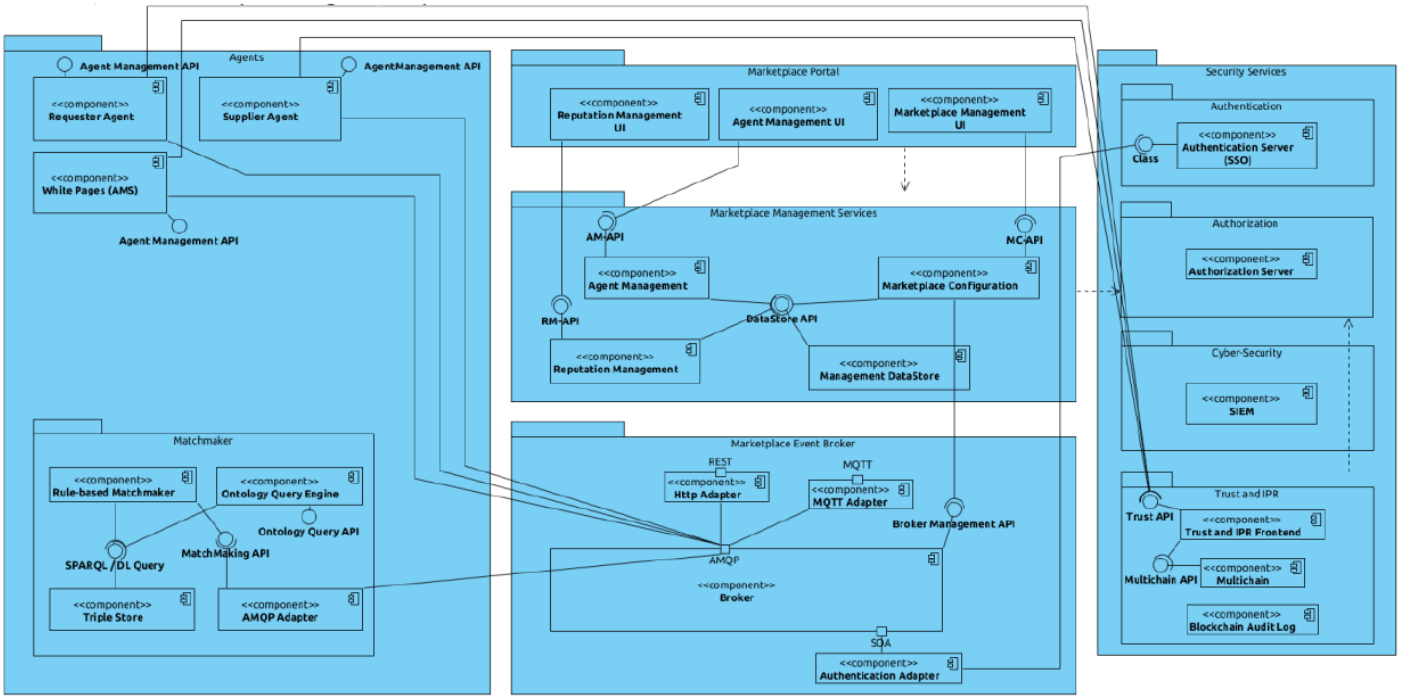

Figure 1: Marketplace Components

As stated in previous deliverable D2.3 there are two main categories of agents that can be defined a priori, depending on the kind of services provided:

- Marketplace agents
- Stakeholder agents

The former category groups all the agents providing services that are crucial for the marketplace to operate. The latter category, instead, groups agents developed and deployed by the marketplace stakeholders to take part in chain formation rounds.

Stakeholder agents can be divided in two different categories, Requester and Supplier. From an implementation point of view, they are very similar and share a large set of features, especially the communication protocols used for the interaction with stakeholders and other agents.

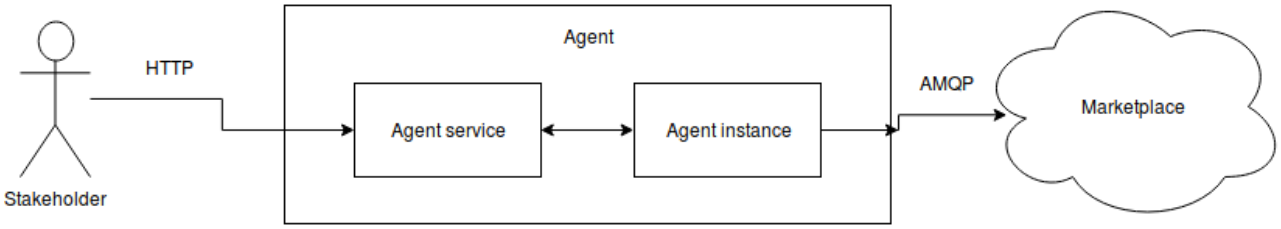

Figure 2: Simplified agents' communication logic

The communication protocols used by the agent for the interaction with the stakeholder on left side and with other agents on the marketplace on the right side are shown in Figure 2.
Any agent can be controlled by the stakeholder via RESTful APIs which will be listed and explained in the following sections. Agents on the marketplace can exchange messages using different connectivity protocol: it is built on AMQP by default, but it supports also MQTT protocol as messages exchange mechanism.

5.1 Requester Agent

The Requester Agent is the agent used by a factory to request the execution of an existing supply chain or to initiate a new supply chain. Due to the dynamics of exchanges pursued in COMPOSITION, there is no actual distinction between the two processes, i.e., for any supply need a new chain is formed and a new execution of the chain is triggered. The Requester agent may act according to several negotiation protocols, which can possibly be supported by only a subset of the agents active on a specific marketplace instance. The baseline protocol, which must be supported by any COMPOSITION agent, is the so-called CONTRACT-NET. In such a protocol, a Requester agent plays the role of “Initiator”.

The communication between UI and corresponding agent(s) on the marketplace happens via a set of RESTful APIs, described in the following section.

5.1.1 APIs

APIs has been updated with respect to the previous version of these deliverable “D6.5 Connectors for Inter-factory Interoperability and Logistics I” all the modified schemas and new endpoints are reported in the following sections.

5.1.1.1 Start bidding

This API is exposed on:

<Agent_IP_address>/agent/messages/iims/startbidding

It accepts the following methods: POST, PUT
This API is triggered by the IIMS whenever an installed sensor in the intra-factory system reaches a critical level, meaning that a new negotiation session (namely bidding process) should start.

The schema of the message accepted by the API is the following:

Table 3: Start Bidding Schema

```
{
 "description": "Trigger message sent by intra-factory toolchains to the requester agent",
 "type": "object",
 "properties": {
  "action": {
   "type": "string"
  },
  "quantity": {
   "type": "number"
  },
```

```
   "quantity_uom": {
    "type": "string"
   }
  },
  "additionalProperties": true
}
```

The fields ‘quantity’ and ‘quantity_uom’ are the only ones that have not been removed compared with the previous solution, because additional information are transmitted inside other messages and is not necessary to make this simple message bigger. The previous version included other information about the price, the time, the currency and details about the logistic. All these information are not necessary at start bidding time.

5.1.1.2 Start action

This API is exposed on:

<Agent_IP_address>/agent/messages/ui/action

It accepts the following methods: POST, PUT

This API is triggered whenever the stakeholder wants the agent to perform a certain action. This API supports several actions that are described below.

The updated schema of the message accepted by the API is the following:

Table 4: Action Schema

```
{
  "description":"An action request sent by a UI to the associated agent",
  "type":"object",
  "properties":{
    "session_id":{
      "type":"string"
    },
    "agent_role":{
      "type":"string",
      "enum":["requester","supplier"]
    },
    "action":{
      "type":"string",
      "enum":["withdraw",
              "selected_option",
              "confirmed",
              "rejected",
              "start_bid",
              "search",
              "rating_update"]
    },
    "payload":{
      "selected_option":{
        "type":"object",
        "properties":{
          "price":{
            "type":"number"
```

```
      },
      "currency":{
       "type":"string",
       "enum":["EUR","USD"]
      },
      "company":{
       "type":"string"
      },
      "rating":{
       "type":"number",
       "minimum":0,
       "maximum":5
      },
      "quantity":{
       "type":"number"
      },
      "quantity_uom":{
       "type":"string"
      },
      "good":{
       "type":"string"
      }
     }
    },
    "search_info":{
     "type":"object",
     "description":"Search for recommended solutions on the marketplace",
     "properties":{
      "service": {
       "type": "string",
       "description": "The service that needs to be searched for."
      },
      "quantity":{
       "type":"number"
      },
      "quantity_uom":{
       "type":"string"
      },
      "good":{
       "type":"string"
      },
      "currency":{
       "type":"string",
       "enum":["EUR","USD"]
      },
      "expiration":{
       "type": "string",
       "format": "date-time"
      }
     }
    },
```

```
    "rating":{
     "type": "object",
     "description": "Used for updating or retrieving the reputation of an agent.",
     "properties":{
      "update":{
       "type": "object",
       "description": "This will be used when updating another agent reputation.",
       "properties":{
        "agent_id":{
         "type": "string"
        },
        "rating":{
         "description": "An integer number [1,5] representative of the rating.",
         "type": "number",
         "minimum":1,
         "maximum":5
        }
       }
      }
     }
    }
   }

 },
 "additionalProperties": false
}
```


Several actions have been added compared with the previous version of this API described in the first deliverable. The two unchanged actions are the first two reported in the following list, and the new ones will be reported next:

- ***Selected_option****:* Action is required when, at the end of the negotiation protocol, the winning bid must be selected.
- ***Withdraw****:* Action is required when a bidding process must be stopped before reaching any agreement.
- ***Confirmed:*** Action used to confirm an offer
- ***Rejected:*** Action used to reject an offer
- ***Search:*** Action used to search a supplier for a defined service
- ***Rating_update:*** Action used to update the rating of the supplier offering for a good/service

5.1.1.3 Deregister

This API is exposed on:

<Agent_IP_address>/agent/messages/ui/deregister

It accepts the following methods: POST, PUT

This API is triggered whenever the stakeholder wants to deregister an agent from the current marketplace. The agent will, in turn, send an HTTP POST request to the AMS's API to be removed from the shared agents database residing on the AMS. (as described in 5.3.1.1.2).

5.1.1.4 Send Information

The API is exposed on:

<Agent_IP_address>/agent/messages/ui/info

It accepts the following methods: POST, PUT.

This API is triggered when an agent needs to communicate information to other agents (such as the fill-level of a container for a certain good) without requiring a protocol interaction.

The data format accepted are the following:

Table 5: Inform Schema

```
{
 "description": "Informative message sent by intra-factory toolchains to the agent",
 "type": "object",
 "properties": {
  "action": {
   "type": "string",
   "enum": ["inform"]
  },
  "quantity": {
   "type": "number"
  },
  "quantity_uom": {
   "type": "string"
  }
 },
 "additionalProperties": true
}
```

The schema above is used for messages coming from the IIMS.

Table 6: Notification Schema

```
{
 "description":"Notification sent by an agent to the corresponding UI, may be replied with a withdraw
action request",
 "type":"object",
 "properties":{
  "session_id":{
   "type":"string"
  },
  "agent_role":{
   "type":"string",
   "enum":["requester","supplier"]
  },
  "agent_owner":{
   "type":"string"
  },
  "sender_owner":{
   "type":"string"
  },
  "notification_type":{
```

```
  "type":"string",
  "enum":["withdrawable",
        "confirmable",
        "selection",
        "info",
        "ack"]
 },
 "status":{
  "type":"string"
 },
 "result":{
  "type":"string"
 },
 "payload":{
  "type":"object",
  "properties":{
   "offer_details":{
    "type":"object",
    "properties":{
     "good":{
      "type":"string"
     },
     "expiration":{
      "type":"string"
     },
     "price":{
      "type":"number"
     },
     "currency":{
      "type":"string",
      "enum":["EUR","USD"]
     },
     "participants":{
      "type":"number"
     },
     "quantity":{
      "type":"number"
     },
     "quantity_uom":{
        "type":"string"
     }
    }
   },
   "options":{
    "type":"array",
    "items": {
     "type":"object",
     "properties":{
      "price":{
       "type":"number"
      },
```

```
          "currency":{
             "type":"string",
             "enum":["EUR","USD"]
          },
          "company":{
           "type":"string"
          },
          "rating":{
           "type":"number",
           "minimum":0,
           "maximum":5
          },
          "quantity":{
           "type":"number"
          },
          "quantity_uom":{
           "type":"string"
          },
          "good":{
           "type":"string"
          }
         }
        }
       },
       "pickup_details":{
        "type":"object",
        "properties":{
         "date":{
          "type":"string",
          "format":"date-time"
         },
         "supplier":{
          "type":"string"
         }
        }
       }
      }
     }
    },
    "additionalProperties":false
}
```


This second schema is used for messages coming from the UI. It supports different type of informative messages as:

- Withdrawable
- Confirmable
- Selection
- Info
- Ack

This is done to support every possible kind of information coming from the user interface towards the agents.

5.1.1.5 Predict

This API is exposed on:

<Agent_IP_address>/agent/messages/ui/predictions/<id_good>/last
or
<Agent_IP_address>/agent/messages/ui/predictions/<id_good>/all

It accepts the following method: GET

This API is triggered to get the last or the whole set of price predictions for a certain good from the DLT COMPOSITION module.

<id_good> is a placeholder to be replaced with a supported type of goods. In this final implementation, the supported values are ["hdpe", "paper", "pet", "scrap_metal"].

5.1.1.6 Reputation Update

This API is exposed on:

<Agent_IP_address>/agent/messages/ui/reputation/update

It accepts the following method: POST

This API is triggered by the UI to update an agent reputation.

5.1.1.7 Reputation

This API is exposed on:

<Agent_IP_address>/agent/messages/ui/reputation/<agent_id>

It accepts the following method: GET

This API is triggered to get a specific agent reputation.

It accepts the same schema shown in 5.1.1.2 with specified action “rating_update” and a “rating” value between 0 and 5 in the “payload” field.

5.1.1.8 Update Configuration

This API is exposed on:

<Agent_IP_address>/agent/messages/ui/configUpdate

It accepts the following method: POST, PUT

This API is triggered to modify some of the configuration properties.

The data format accepted is the following:

Table 7: Configuration Schema

```
{
 "description":"Schema for updating agent configuration",
 "type":"object",
 "properties": {
  "section": {
   "description": "Identifier for the configuration file section",
   "type": "string",
```

```
    "example": "network"
   },
   "subsection": {
    "description": "The identifier of the agent which has to be removed from the marketplace.",
    "type": "string",
    "example": "ui_endpoint"
   },
   "value": {
    "description": "Identifier for the configuration file subsection",
    "type": "string",
    "example": "http://130.192.85.226:32836/"
   }
  },
  "additionalProperties": false
}
```

5.2 Supplier Agent

The Supplier agent is the counterpart of the Requester agent on the COMPOSITION Marketplace. It is usually adopted by actual suppliers to respond to supply requests coming from other stakeholders in the marketplace. Factories transforming goods typically employ at least one Requester agent, to get prime goods and one Supplier agent to sell intermediate products to other factories.

The communication between UI and corresponding agent(s) on the Marketplace happens via a set of RESTful APIs, described in the following section.

5.2.1 APIs

5.2.1.1 Start action

This API is exposed on:

<Agent_IP_address>/agent/messages/ui/action

It accepts the following methods: POST, PUT

This API is triggered whenever the stakeholder wants the agent to perform a certain action. This API supports several actions that are descripted below.

The schema of the message accepted by the API is the following:

Table 8: Action Schema

```
{
  "description":"An action request sent by a UI to the associated agent",
  "type":"object",
  "properties":{
    "session_id":{
      "type":"string"
    },
    "agent_role":{
      "type":"string",
      "enum":["requester","supplier"]
    },
    "action":{
      "type":"string",
```

```
  "enum":["withdraw",
        "selected_option",
        "confirmed",
        "rejected",
        "start_bid",
        "search",
        "rating_update"]
 },
 "payload":{
  "selected_option":{
   "type":"object",
   "properties":{
    "price":{
     "type":"number"
    },
    "currency":{
     "type":"string",
     "enum":["EUR","USD"]
    },
    "company":{
     "type":"string"
    },
    "rating":{
     "type":"number",
     "minimum":0,
     "maximum":5
    },
    "quantity":{
     "type":"number"
    },
    "quantity_uom":{
     "type":"string"
    },
    "good":{
     "type":"string"
    }
   }
  },
  "search_info":{
   "type":"object",
   "description":"Search for recommended solutions on the marketplace",
   "properties":{
    "service": {
     "type": "string",
     "description": "The service that needs to be searched for."
    },
    "quantity":{
     "type":"number"
    },
    "quantity_uom":{
     "type":"string"
```

```
        },
        "good":{
          "type":"string"
        },
        "currency":{
          "type":"string",
          "enum":["EUR","USD"]
        },
        "expiration":{
          "type": "string",
          "format": "date-time"
        }
      }
    },
    "rating":{
      "type": "object",
      "description": "Used for updating or retrieving the reputation of an agent.",
      "properties":{
        "update":{
          "type": "object",
          "description": "This will be used when updating another agent reputation.",
          "properties":{
            "agent_id":{
              "type": "string"
            },
            "rating":{
              "description": "An integer number [1,5] representative of the rating.",
              "type": "number",
              "minimum":1,
              "maximum":5
            }
          }
        }
      }
    }
  }

 },
 "additionalProperties": false
}
```

Six different actions are supported:

- ***Selected_option****:* Action is required when, at the end of the negotiation protocol, the winning bid must be selected.
- ***Withdraw****:* Action is required when a bidding process must be stopped before reaching any agreement.
- ***Confirmed:*** Action used to confirm an offer
- ***Rejected:*** Action used to reject an offer
- ***Search:*** Action used to search a supplier for a defined service

- ***Rating_update:*** Action used to update the rating of the supplier offering for a good/service

5.2.1.2 Deregister

This API is exposed on:

<Agent_IP_address>/agent/messages/ui/deregister

It accepts the following methods: POST, PUT

This API is triggered whenever the stakeholder wants to deregister an agent from the current marketplace. The agent will, in turn, send a POST request to the AMS's API for being deregistered (as described in 5.3.1.1.2).

5.2.1.3 Prediction update

This API is exposed on:

<Agent_IP_address>/agent/messages/ui/predictions/good
or
<Agent_IP_address>/agent/messages/ui/predictions/value

It accepts the following methods: POST, PUT

This API is triggered to send information to the DLT for updating values of a certain good with the aim of improving DLT's predictions.

The data format accepted is the following:

Table 9: Prediction Schema

```
{
 "description": "The JSON syntax specification of the message used for updating the price in the DLT",
 "type": "object",
 "properties": {
  "id_good": {
   "type": "string",
   "enum": ["hdpe", "paper", "pet", "scrap_metal"]
  },
  "price": {
   "description": "The price to be updated, in floating point",
   "type": "number",
   "examples": [
    393.68
   ]
  },
  "quantity": {
   "description": "The quantity to be updated, in floating point, [in tons]",
   "type": "number",
   "examples": [
    95.00
   ]
  },
  "start_date": {
   "description": "The start date quantity and price refer to, in epoch time (GMT-2)",
```

```
    "type": "integer",
    "examples": [
      1524233245
    ]
   },
   "end_date": {
    "description": "The end date quantity and price refer to, in epoch time (GMT-2)",
    "type": "integer",
    "examples": [
      1524233250
    ]
   }
  },
  "additionalProperties": false
}
```


5.2.1.4 Predict

This API is the same one descripted for the Requester Agent. Refer to 5.1.1.5.

5.2.1.5 Reputation Update

This API is the same one descripted for the Requester Agent. Refer to 5.1.1.6.

5.2.1.6 Reputation

This API is the same one descripted for the Requester Agent 5.1.1.7.

5.2.1.7 Update Configuration

This API is the same one descripted for the Requester Agent 5.1.1.8.

5.3 Agent Management Service

According to FIPA specifications,[1] an Agent Management System (AMS) is a mandatory component of every agent platform, and only one AMS should exist in every platform. It offers the white pages service to other agents on the platform by maintaining a directory of the agent identifiers currently active on the platform. Prior to any operation, every agent should register to the AMS to get a valid agent identifier, which will be unique within the agent platform.

In COMPOSITION the marketplace is the platform where agents operate.

5.3.1 White Pages Service

A White Pages service is a mandatory component of any MAS system. It is required to locate and name agents on the system, making it possible for one agent to connect with one another. In the current implementation of the Agent Management Service, the agent identifiers are stored in a MySQL Database. MySQL has been chosen because it offers relevant features for the project such as on-demand scalability, high availability and reliability. Other agent platforms, like SPADE[2], use MySQL as well for offering the White Pages service.

The table storing the agents has the following schema:

Table 10: Table schema

Agent_id	Agent_owner	Agent_role

[1] http://www.fipa.org/specs/fipa00023/SC00023K.pdf
[2] https://pypi.python.org/pypi/SPADE

They are, respectively:

- agent_id: the unique identifier for the agent. It is the primary key for the table, since it has the constraint of being unique.
- agent_owner: The name of the company owning the agent.
- agent_role: The role of the agent on the Marketplace, can be either ‘requester’ or ‘supplier’.

Since the directory service is offered by the AMS, it is the only agent allowed to directly interact with the database. When the AMS is executed, it needs the following configurations:

- IP address of the host running the MySQL instance.
- Username and password for CRUD operations on the database.
- Name of the database to operate on, since the same instance may run different databases.

5.3.1.1 APIs

The AMS exposes the White Pages service with a set of RESTful APIs.

In order to exploit current standards provided by the state of the art, COMPOSITION adopts the Foundation for Intelligent Physical Agents (FIPA) definition of AMS and the corresponding set of offered services, summarized in Table 11.

Table 11: AMS services

Service	Description
Register	Allows registering an agent, it will be considered as "active" on the marketplace.
Deregister	Removes an agent from the list of agents currently active on the market.
Modify	Modifies the registration information about a certain agent.
Search	Allows searching for active agents, given a set of search constraints.

The current implementation of these services is detailed in the following sections.

5.3.1.1.1 Register a new agent

This API is exposed on:

<AMS_IP_address>/agents/register

It accepts the following methods: POST

This API is called by the agent willing to join a certain marketplace, upon its first activation. It is, in fact, required to obtain the unique agent identifier within the marketplace.

The schema of the message accepted by the API is the following:

Table 12: Register Schema

```
{
  "description" : "A message used by user to register an agent",
  "type" : "object",
  "properties" : {
    "agent_id" : {
      "description" : "The unique identifier of the agent, empty",
      "type" : "string"
    },
    "agent_owner" : {
      "description" : "Identifier for agent's owner",
      "type" : "string"
```

```
  },
  "agent_role" : {
   "description" : "An agent can be either requester or supplier",
   "type" : "string",
   "enum" : ["requester", "supplier"]
  }
 }
}
```

Since the agent does not have an identifier when the call is performed, the field is left empty.

It will be filled in by the AMS, with an identifier that will be unique within the marketplace and sent back to the agent requiring registration.

5.3.1.1.2 Deregister an agent

This API is exposed on:

<AMS_IP_address>/agents/deregister

It accepts the following methods: POST, PUT

This API is called when an agent asks to be deregistered from the marketplace.
The schema of the message accepted by the API is the following:

Table 13: Deregister Schema

```
{
 "description" : "A message used by user to deregister an agent",
 "type" : "object",
 "properties" : {
  "agent_id" : {
   "description" : "The unique identifier of the agent",
   "type" : "string"
  }
 }
}
```

The AMS removes the entry associated with the ‘agent_id’ from the database storing all the registered agents.

5.3.1.1.3 Deregister all the agents

This API is exposed on:

<AMS_IP_address>/agents/deregister/all

It accepts the following methods: POST, PUT

The API is called in case all the agents have to be removed from the marketplace. Since this operation is very critical, it can only be called by the AMS itself.

5.3.1.1.4 Update agent

This API is exposed on:

<AMS_IP_address>/agents/update

It accepts the following methods: POST, PUT

This API is triggered when an agent asks for the update of its details within the marketplace. The schema of the message accepted by the API is the following:

Table 14: Agent Update Schema

```
{
 "description" : "A message used by user to register an agent",
 "type" : "object",
 "properties" : {
  "agent_id" : {
   "description" : "The unique identifier of the agent",
   "type" : "string"
  },
  "agent_owner" : {
   "description" : "Identifier for agent's owner",
   "type" : "string"
  },
  "agent_role" : {
   "description" : "An agent can be either requester or supplier",
   "type" : "string",
   "enum" : ["requester", "supplier"]
  }
 }
}
```

The AMS updates the fields that need to be changed. The field ‘agent_id’ cannot be changed.

5.3.1.1.5 Get registered agents

This API is exposed on:

<AMS_IP_address>/agents
or
<AMS_IP_address>/agents/<agent_type>

It accepts the following methods: GET

This API returns the list of all the agents currently registered (active) on the marketplace. Specifying the “agent_type” will return a list of the agents belonging to the selected category.

5.3.1.1.6 Perform matching on agents

This API is exposed on:

<AMS_IP_address>/matchmaker/performMatching

It accepts the following methods: POST, PUT

The schema of the message accepted by the API is the following:

Table 15: Matching Schema

```
{
 "conversation_id": "the conversation id",
 "sender_id":"agent id",
 "agent_owner": "the agent owner company id",
 "type":"type of agent: CFP ",
 "service":"type of service: Waste_management / Provide_raw_materials / Software_solutions",
```

```
 "offer_details":{
  "good":"the name of the requested good",
  "expiration":"expiration date",
  "currency":"the currency",
  "quantity":"the requested quantity",
  "quantity_uom":"the units of measurement of the good"
 },
 "offers": "empty array []",
 "preferences": "empty"
}
```

This API receives a request from a Requester agent willing to obtain the list of matching Supplier agents for a certain offer. The AMS forwards the request to the Matchmaker, which will return the list of potential Supplier agents in a compatible data-format understandable by the AMS and return it to the Requester agent.

5.3.1.1.7 Evaluate offer

This API is exposed on:

<AMS_IP_address>/matchmaker/offerEval

It accepts the following methods: POST, PUT

Table 16: Offer Evaluation Schema

```
{
 "conversation_id": "the conversation id",
 "sender_id":"agent id",
 "agent_owner": "the agent owner company id",
 "type":"type of agent: OFFER ",
 "service":"type of service: Waste_management / Provide_raw_materials / Software_solutions",
 "offer_details":{
  "good":"the name of the requested good",
  "expiration":"expiration date",
  "currency":"the currency",
  "quantity":"the requested quantity",
  "quantity_uom":"the units of measurement of the good"
 },
 "preferences": {
                "priority_1": "the first priority rule for Waste_management offers",
                "priority_2": "the second priority rule for Waste_management offers",
                "priority_3": "the third priority rule for Waste_management offers"
        },
 "offers": [
 {
        "offer_details":{
                "sender_id":"agent offer id",
                "agent_owner": "the company that makes an offer",
           "good":"the name of the offered good",
           "delivery":
                {
                "time": "delivery time",
                "methods": ["array of available delivery methods"]
```

```
                },
            "payment":
                {
                "methods": ["array of available payment methods"],
                "terms": "payment terms in days",
                "currency": "payment currency"
                },
            "price":
                {
                "transportation": "transportation price",
                "insurance": "insurance price",
                "service": "service price"
                }
        }
  },
  {
        "offer_details":{
                "sender_id":"agent offer id",
                "agent_owner": "the company that makes an offer",
            "good":"the name of the offered good",
            "delivery":
                {
                "time": "delivery time",
                "methods": ["array of available delivery methods"]
                },
            "payment":
                {
                "methods": ["array of available payment methods"],
                "terms": "payment terms in days",
                "currency": "payment currency"
                },
            "price":
                {
                "transportation": "transportation price",
                "insurance": "insurance price",
                "service": "service price"
                }
        }
  }

  ]
}
```


This API receives a request from a Requester agent willing to evaluate the received offers through the matchmaker. The AMS forwards the request to the Matchmaker and returns the selected offer(s) to the Requester agent.

5.3.1.1.8 Search Solution

This API is exposed on:

<AMS_IP_address>/matchmaker/searchSolutions

It accepts the following methods: POST, PUT

The schema used here is the same one accepted by 5.3.1.1.6 API.

This API has been implemented to support the use case UC-ATL-3 (COMPOSITION D2.1) that expect that an agent should be able to find Suppliers that provide a specific service.

5.4 Matchmaker Agent

The COMPOSITION Matchmaker is designed to be the core component of the COMPOSITION Broker. It supports semantic matching in terms of manufacturing capabilities, in order to find the best possible supplier to fulfill a request for a service with raw materials or products involved in the supply chain. Different decision criteria for supplier selection are considered by the Matchmaker according to several qualitative and quantitative factors.

The Matchmaker component acts as a kind of special agent in the Marketplace. The matchmaking component is connected with the rest agents using REST protocol. The Matchmaker receives requests from the Agents, infers new knowledge by applying semantic rules to Collaborative Manufacturing Services Ontology and then responses back to agents.

The matchmaking module which is described at D2.4 - The COMPOSITION architecture specification II, comprises a complete semantic framework which offers to the Marketplace agents a high-performance ontology store with querying capabilities and a matchmaking engine which provides efficient matching in both agent and offer level.

The agents can access the ontology store using the Ontology API and its querying interfaces. The API provides to marketplace participants a catalogue of services for data access and management using SPARQL queries. This component is able to query the whole dataset with a satisfactory level of efficiency. Every marketplace agent is able to send a request for a service using HTTP protocol and the agent exchange language from the marketplace, which is described in JSON format. Every service translates the request into a SPARQL query and applies it to the ontology model in the store. The requests' types are GET and POST for retrieve, store or delete data.

The matchmaking services are available to Marketplace agents by the Rule-Based Matchmaker component. It is developed in Java language and it is offered through RESTful web services. The Rule-Based Matchmaker will be used by Marketplace's agents in order to match requests and evaluate offers between the agents. The Matchmaker's functionalities are exclusively depended on the Collaborative Manufacturing Ontology. The Matchmaker package is deployed as a Docker container to the COMPOSITION production server and offers to the agents the following functionalities:

- *Semantic Matchmaking between Marketplace agents – Agent Level matchmaking*: The engine provides the matching of an agent who sends a request for a service to the agents who provide services related to the request. The matchmaking is performed by using a generic terms dictionary. which matches every vendor specific process and raw material to common terms in the ontology. Therefore, the Matchmaker is able to match agents that offer different types of services (waste management, sell raw materials, provide software solutions) in the Marketplace with the agents that request them. Furthermore, the matchmaker is able to match an agent who offers e.g. raw materials with possible customers by applying reasoning at the customers' manufacturing services and perform matching to resources or material level, which were using these services.
- *Offers Evaluation – Offer Level Matchmaking:* A marketplace requester agent is able to send a list of provided offers by the possible service providers to the matchmaker for evaluation. The Rule-based Matchmaking Engine applies set of rules in order to match the request with the best provided offer. Based on some predefined criteria, i.e. the requester prefers the cheapest solution over the quickest one, the matchmaker reorganize the list of applied rules and offers different results for each request. This kind of offers evaluation fits perfects to the needs of simple evaluations for scenarios such as KLE-4. In the case of KLE-7 and the evaluation of raw materials' offers the rule-based approach is enhanced with weighted-average algorithms in order to cover the needs of a more complex evaluation.

The figure below presents the information flow between the agents of the collaborative manufacturing eco-system and the Matchmaker:

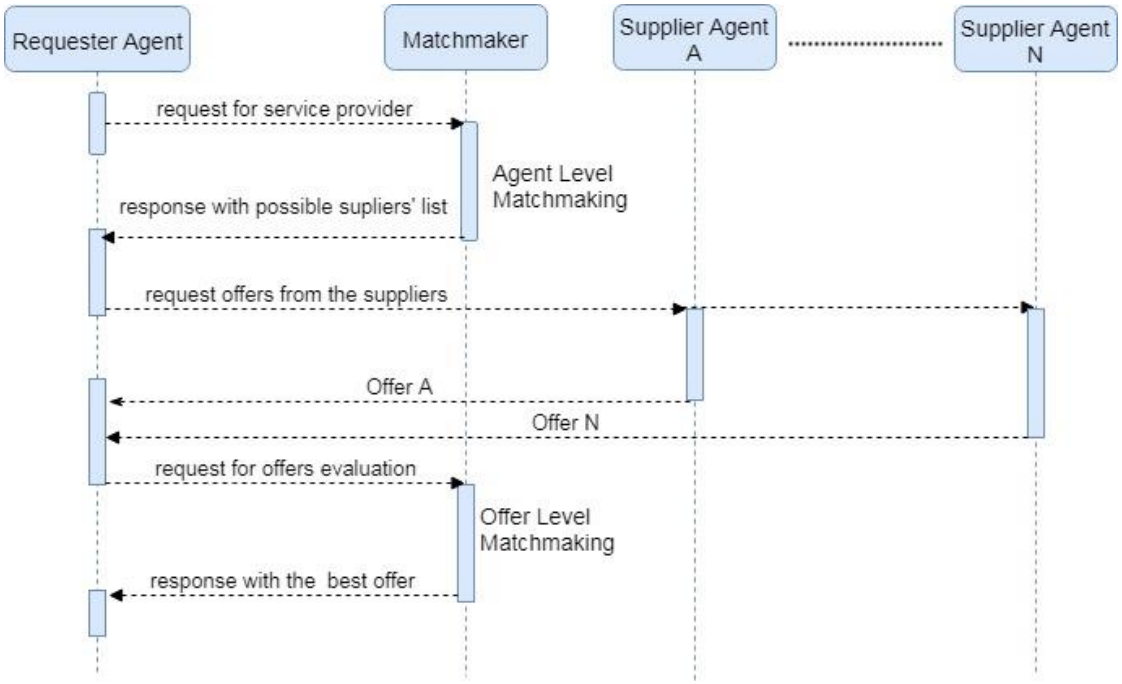

Figure 3: High Level Information Model of Matchmaker and Agents Collaboration

5.5 Sample Interaction Protocol

In order to provide a better and clear explanation about how the different components interact with each other, a brief explanation of a sample protocol (CONTRACT-NET) interaction is provided in this section. Technical details about the schema of the exchanged messages between the components is not provided here, since it is already covered by the remaining sections of this deliverable.

In Figure 4 a sample protocol interaction is shown as an example.

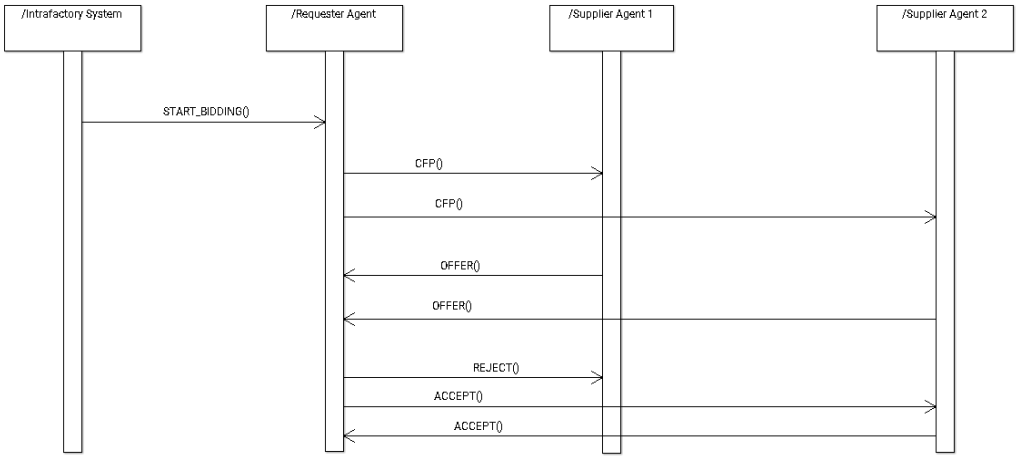

Figure 4: Successful CONTRACT-NET protocol interaction

The first mandatory step allowing an agent to operate on the Marketplace is the registration to the AMS, which provides the agent with a unique identifier. After the registration has taken place the agents remain silent waiting for a protocol interaction to start, which happens when a Requester agent has to introduce a good on the market.

The IIMS sends a notification to its connected Requester agent when a certain good is available to be sold, and the stakeholder is in turn notified via UI. Autonomously, the Requester agent prepares the call for proposal to be forwarded to potential suppliers.

The IIMS sends the “start bid” command to the Requester agent which:

1. Asks the Matchmaker for the list of all the Supplier agents capable and interested in replying to the proposal;
2. Forwards the proposal to all the Supplier agents retrieved from the Matchmaker.

Supplier agents receive the offer, evaluate the received proposal and send back an offer before the expiration. In case a supplier is not interested in a certain negotiation it can just avoid responding with an offer.

When expiration time is reached, Requester agent forwards the offers to the Matchmaker for receiving an evaluation. Matchmaker selects the best three offers, sending them back to the Requester agent which will in turn forward them to the UI, so that the stakeholder can choose the winning one. In the meantime, reject messages are automatically sent to the other Supplier agents participating on the bid.
When the Requester agent receives the instruction from the stakeholder via UI about the offer that has to be accepted, it sends the remaining reject and accept messages.

At this point the winner Supplier agent notifies its UI and waits for the confirmation of acceptance, before sending the final accept message to the Requester.

As described by Shoham in (Shoham Y., 1993), any agent can pass through a different set of states, the state of an agent consisting of components such as beliefs, decisions, capabilities and obligations. In CONTRACT-NET protocol Requester and Supplier agents go through different sets of states, performing different actions upon receipt of a message (either from the UI or from other agents) according to their current state. In the following figures (Figure 5 and Figure 6), the states for Requester and Supplier agents are shown with arrows indicating the allowed transitions between one state and another.

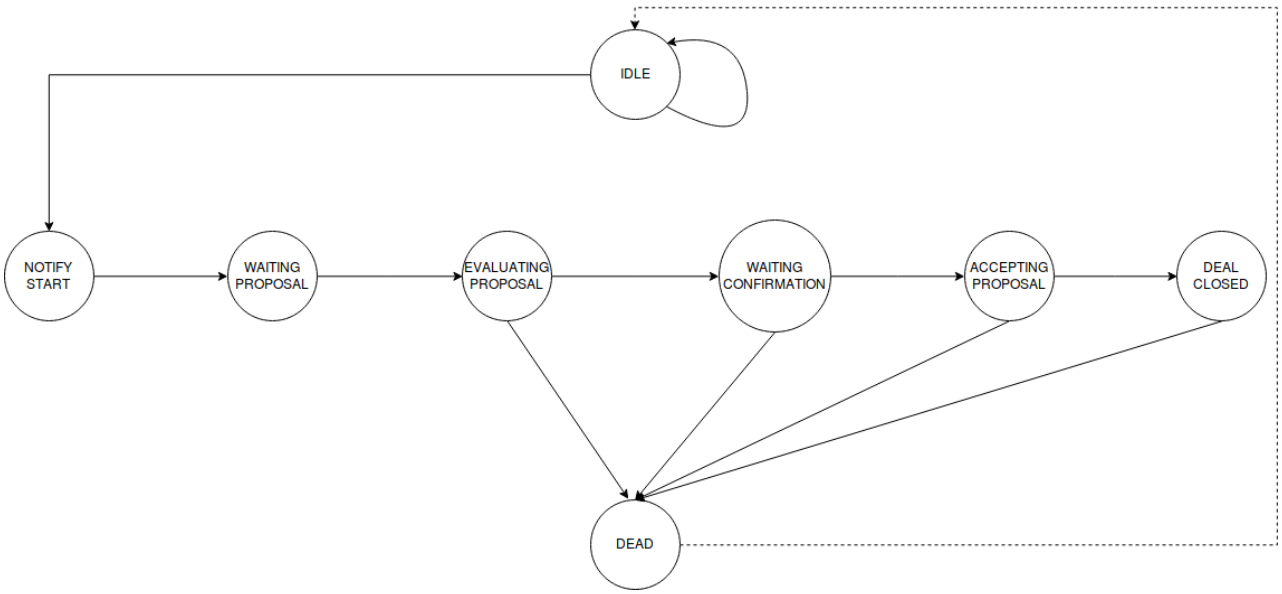

Figure 5: Requester states

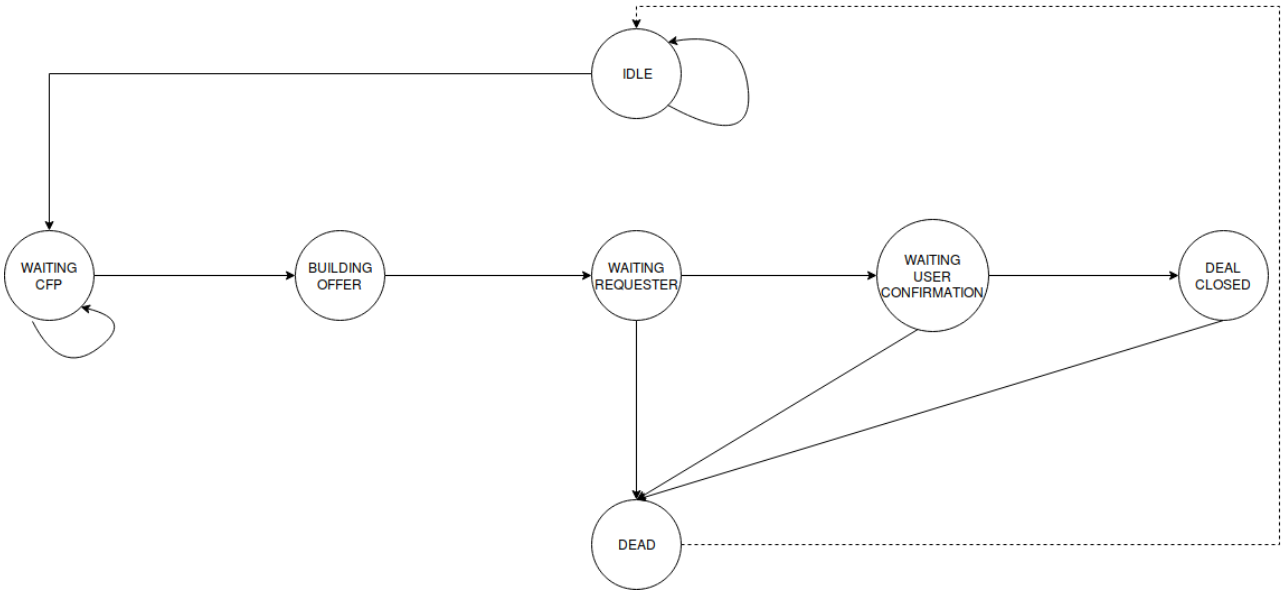

Figure 6: Supplier states

5.6 Informative Messages

Alongside with agents' interaction within the Contract-NET protocol, shortly described in Section 5.5, agents can exchange informative messages with each other. As a first implementation for this feature, the fill-level is propagated from a Requester agent to all the agents registered on the marketplace. In future development, such information will be made available only to a set of authorized agents, through a tight integration with the security framework.

6 Connectors

6.1 COMPOSITION eXchange Language

Agents communicate through messages encoded in a dedicated language named COMPOSITION eXchange Language (CXL). Rather than defining yet another agent communication language, the consortium decided to stick to existing standards and to extend them wherever needed. CXL has therefore been designed as a dialect of the well-known FIPA ACL language specification[3], with a dedicated syntax ("codec" in the FIPA jargon) and with reference to a well-defined set of ontologies for representing the message payload data.

The CXL schema has been changed a bit compared to the one defined in the previous deliverable D2.3 by adding a "message-id" field to keep track of the single message inside a conversation. It is, however, listed here for clarification purposes.

Table 17: CXL

```
{
  "description": "The JSON syntax specification of the COMPOSITION CXL language, mainly focus on the
message envelope",
  "type": "object",
  "properties": {
    "act": {
      "type": "string",
      "enum": [
        "accept-proposal","agree","cancel","cfp","confirm","disconfirm","failure","inform","inform-if","inform-
ref","not-understood","propagate","propose","proxy","query-if","query-ref","refuse","reject-
proposal","request","request-when","request-whenever","subscribe"
      ]
    },
    "sender": {
      "type": "object",
      "description": "the message originator",
      "properties": {
        "name": {
          "type": "string"
        },
        "addresses": {
          "type": "array",
          "items": {
            "type": "object"
          }
        },
        "user-defined": {
          "type": "object"
        }
      }
    },
    "receiver": {
      "type": "array",
      "description": "The set of recipients for this message",
      "items": {
```

[3] http://www.fipa.org/specs/fipa00061/SC00061G.pdf

```
    "type": "object",
    "description": "the message recipient",
    "properties": {
     "name": {
      "type": "string"
     },
     "addresses": {
      "type": "array",
      "items": {
       "type": "object"
      }
     },
     "user-defined": {
      "type": "object"
     }
    }
   }
  },
  "reply-to": {
   "type": "object",
   "description": "The agent to which replies for this message shall be sent",
   "properties": {
    "name": {
     "type": "string"
    },
    "addresses": {
     "type": "array",
     "items": {
      "type": "object"
     }
    },
    "user-defined": {
     "type": "object"
    }
   }
  },
  "language": {
   "type": "string",
   "description": "The language used for encoding the message content"
  },
  "encoding": {
   "type": "string",
   "description": "The specific encoding used for language expressions, typically a mime type"
  },
  "ontology": {
   "type": "array",
   "description": "The set of ontologies defining the primitives that are valid within the message content",
   "items": {
    "type": "string",
    "format": "url"
   }
```

```
    },
    "protocol": {
      "type": "string",
      "description": "Identifies the agent communication protocol to which the message adheres"
    },
    "content": {
      "type": "object",
      "description": "The actual payload of the message"
    },
    "conversation-id": {
      "type": "string",
      "description": "Provides an identifier for the sequence of communicative acts (messages) that together
form a conversation"
    },
    "message-id": {
      "type": "string",
      "description": "Provides an identifier for the single message, to be used in conjunction with message-id
to have a way to uniquely address a certain message."
    },
    "reply-with": {
      "type": "string",
      "description": "Provides an expression that the message recipient shall include in the answer, exploiting
the in-reply-to field. This allows following a conversation when multiple dialogues occur simultaneously."
    },
    "in-reply-to": {
      "type": "string",
      "description": "Denotes an expression that references and earlier action to which this message is a
reply"
    },
    "reply-by": {
      "type": "string",
      "format": "date-time"
    }
  },
  "additionalProperties": false
}
```

6.2 IIMS to Marketplace

The connection between IIMS and Requester Agent on the marketplace is the only point of contact between the Intra-Factory and Inter-Factory systems. The IIMS sends a notification to the Requester Agent whenever a new bidding process should start because certain conditions have been met. A high-level overview of such interaction is shown in Figure 7.

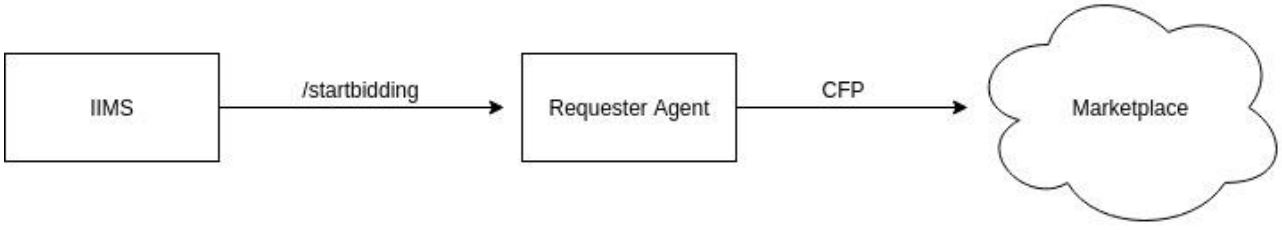

Figure 7: Interaction between IIMS and Requester Agent

The IIMS propagates the message through the Learning Agent, using a format described in 5.1.1.1, to the Requester agent on the marketplace by using the Requester agent's API located at:

<Agent_IP_address>/agent/messages/ui/startbidding.

6.3 Cloud Service APIs

All inter-factory interaction and data exchange takes place in the cloud, through CXL messages exchanged between agents (via the Message Broker) or the Marketplace Management Services APIs: Agent Management API, Reputation Management API and Marketplace Configuration API. The Marketplace Portal builds on the full set of Marketplace Management Services and provides a user interface to these. However, the services may also be exposed for access by stakeholder systems for automation of tasks such as receiving agent credentials and configuration parameters. The Security Framework manages authentication, authorization and access control for the entire marketplace including cloud services and the Message Broker.

6.3.1 Marketplace Data Sharing

The COMPOSITION mechanisms for configurable data sharing from the intra-factory IIMS with other stakeholders in the marketplace is designed and currently in development. A factory may choose to share certain data with partners across the supply chain on a permanent basis or a single interaction. The Big Data Analytics (BDA) component may be set up by the agent system to route certain data streams for delivery to the Marketplace Broker. These could be e.g. sensor data, inventory alerts or analysis results from Simulation and Forecasting Toolkit (SFT) or Deep Learning Toolkit (DLT). The BDA may also filter messages for sensitive data, transform them to another format (CXL INFORM messages) or annotate them for easier integration on the recipient side. The negotiation between agents using CXL to set up the data exchange is described in *D2.3 COMPOSITION architecture specification I*. The data owner is fully in control of what data is shared and can control the flow.

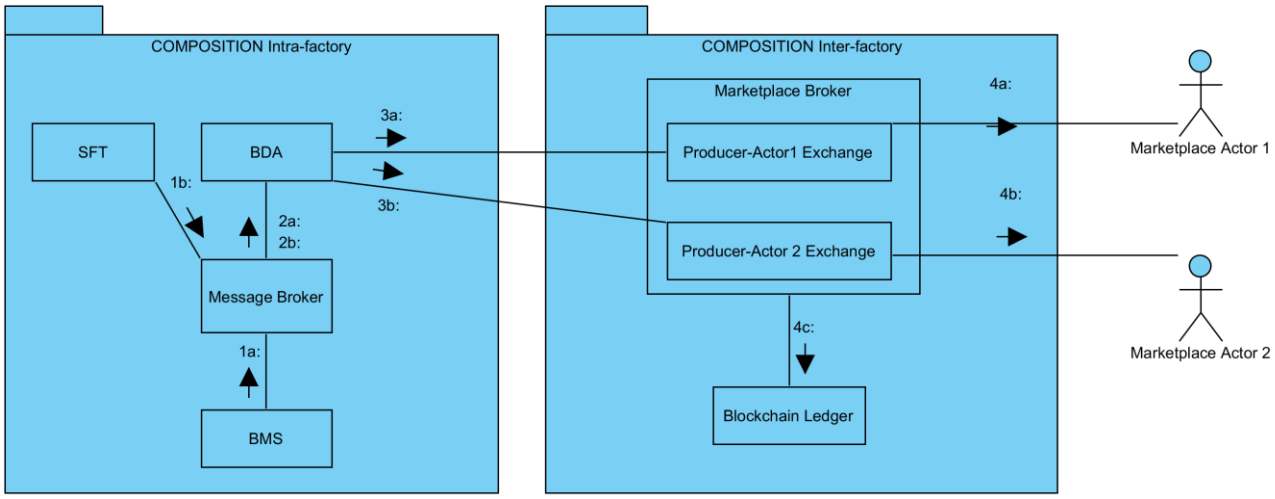

Figure 8: Simplified model of the marketplace data exchange

The integration of Message Broker access control with the Security Framework identity provider makes it possible to set up an exclusive message exchange for a business partner at the Marketplace Broker. Exchanges are lightweight entities; messages are only copied to the queues created for listening clients. These queues reside on only one node in a broker cluster, or on a separate broker provided by the sender or recipient actor, keeping this design scalable. The Marketplace management Services and the Marketplace Portal can be used to set up the agreement to exchange data, or the dynamic approach using CXL SUBCRIBE messages between agents may be applied. All marketplace actors are identified and only the approved actors in the marketplace may publish and/or read data from the exchange. 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. If the messages need to be encrypted, the blockchain can be used to share symmetric keys employing the technique described in section 6.7 in *D4.3 The COMPOSITION Blockchain*.

6.3.2 Marketplace Portal

The current version of the Marketplace Portal is mainly centred around two use cases described in D2.1: *INTRA-Factory-3 Material Management* and *INTER-Factory-1 Scrap Metal Management*. This section will describe the graphical user interface (GUI) developed to handle these use cases. The GUI is built in Angular following the Material Design guidelines and the micro frontend architecture outlined in D2.4. As mentioned in the previous section, the Marketplace portal interacts with the Marketplace Management Services to retrieve information (e.g. container details), get notifications (e.g. bidding status) and execute actions (e.g. withdraw from bidding, select contractor).

Starting from the INTRA-Factory-3 use case, Figure 9 shows the Material Management Containers view where the users can see their containers, current prices and historic container emptying. Clicking the down-arrow on a container component reveals detail information on the container such as filling prediction, location, model and battery status (see Figure 10).

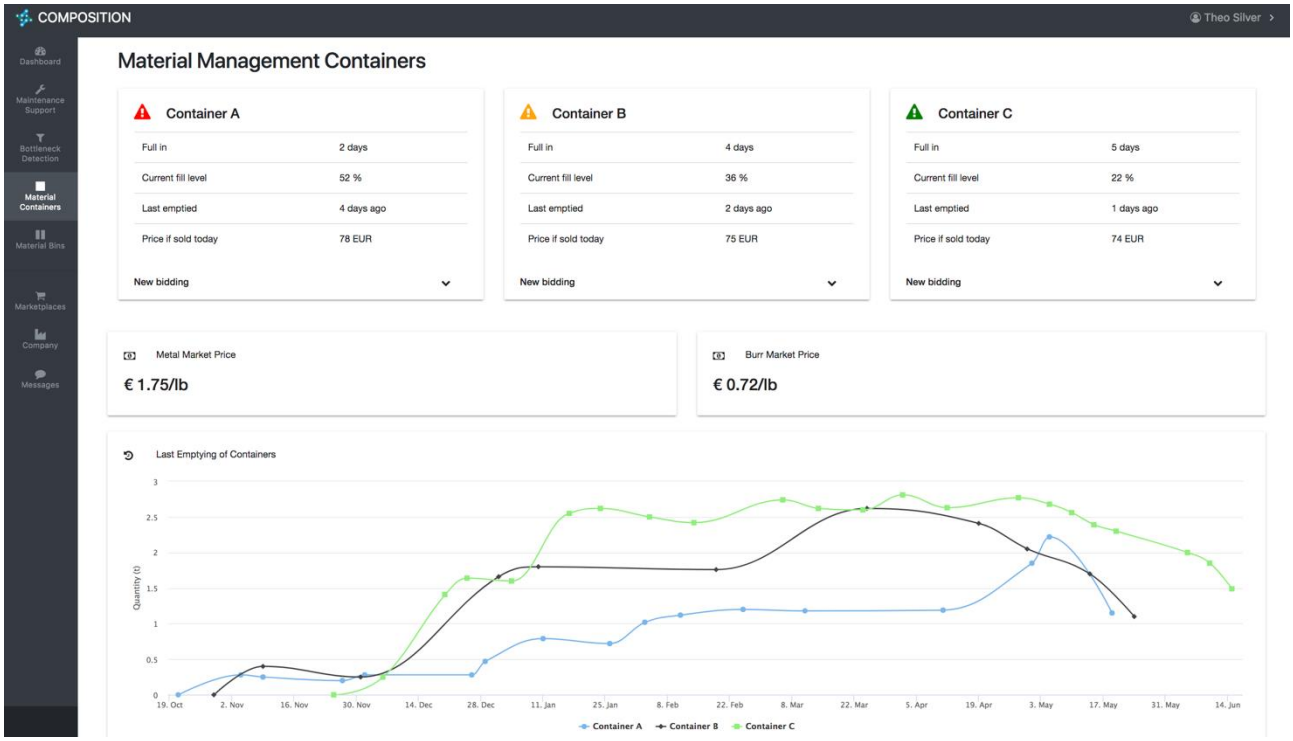

Figure 9: Intra - Material Management Containers

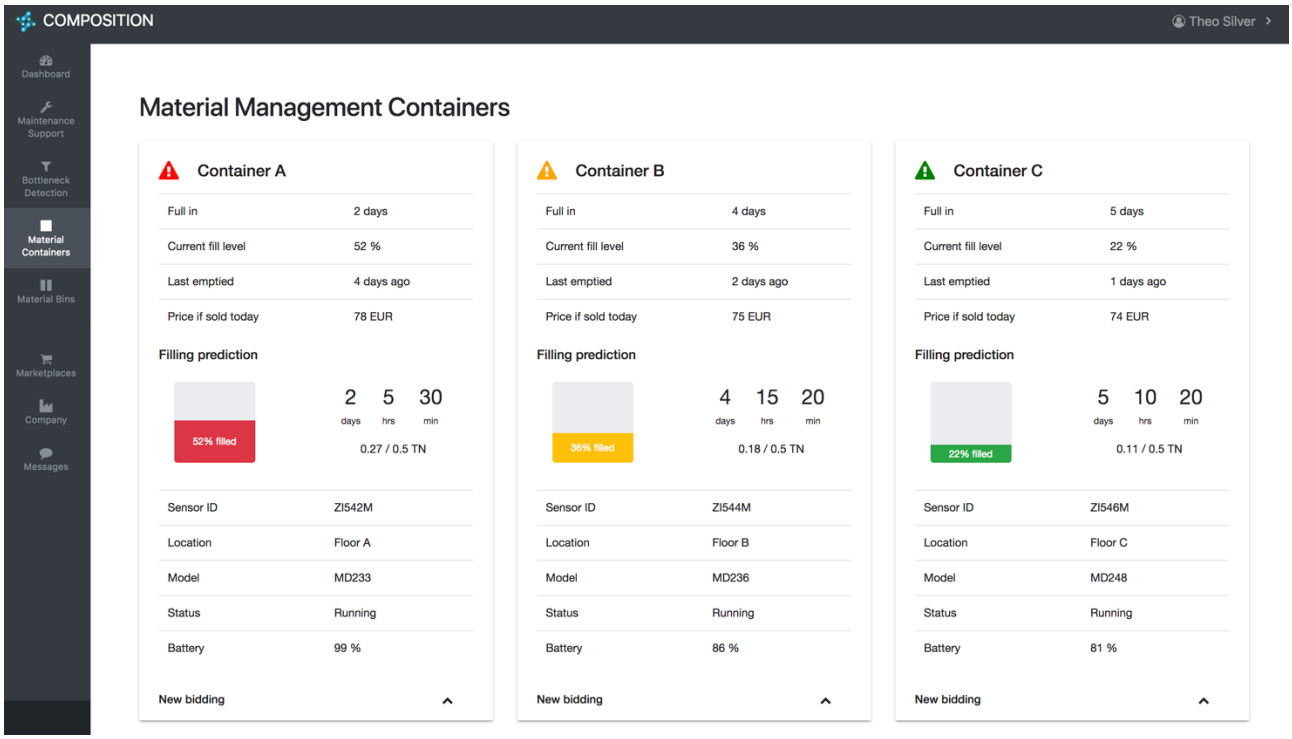

Figure 10: Intra - Material container details

Continuing to the INTER-factory-1 use case, Figure 11 shows the Bidding Process Management view where the users can see current bidding processes including status, contractor, quantity, final price offer, number of participants in bidding and history of the process. Depending on the status of the process, the users might be requested to perform an action in order to proceed or cancel the bidding process e.g. select a contractor, withdraw, confirm deal or reject deal. These actions are visualized as buttons in the *Action* column together with the details button (grey with 'i' symbol) where the users can get more detailed information on the bid and where the action buttons are accessible as well (see Figure 13). Once the user selects an action e.g. withdraw (blue button) a confirmation window appears (see Figure 12). The same goes if the withdraw button in the information window is clicked.

If the user is requested to select a contractor, a dropdown list is shown with the best option pre-selected and at most the top three best options with ratings available to select (see Figure 14).

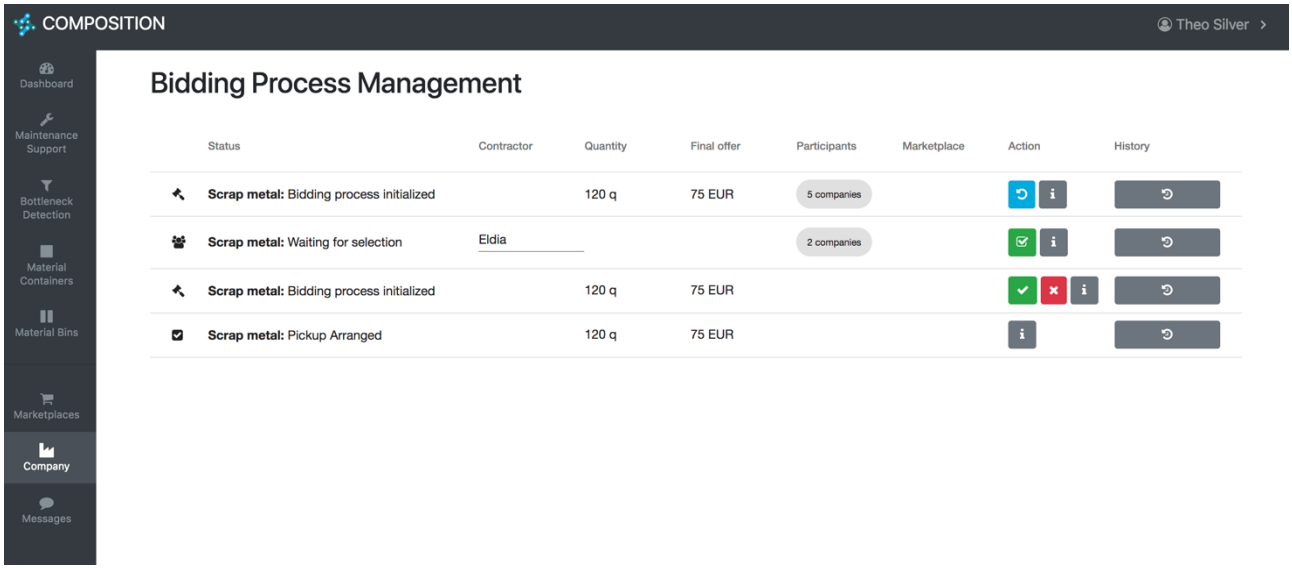

Figure 11: Inter - Bidding Process Management

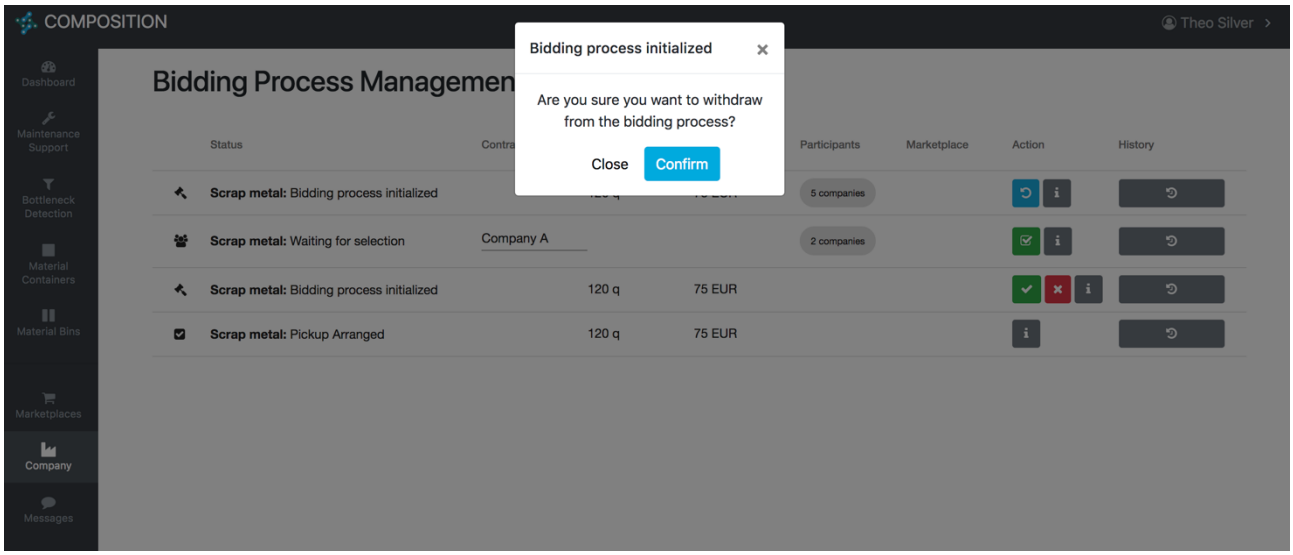

Figure 12: Bidding process withdraw confirmation

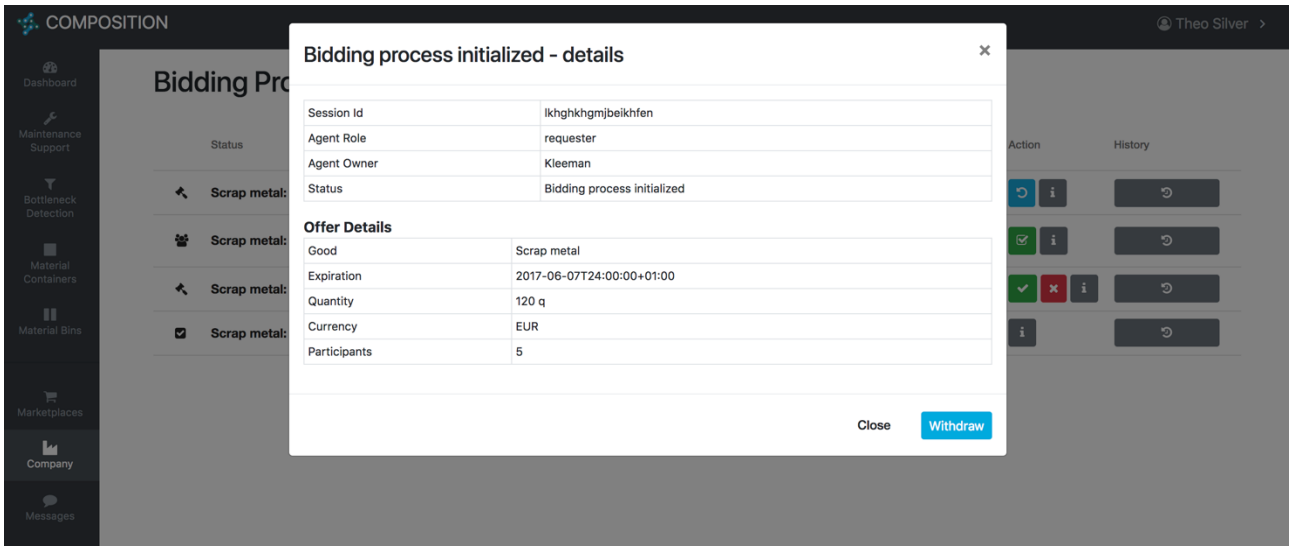

Figure 13: Bidding process details

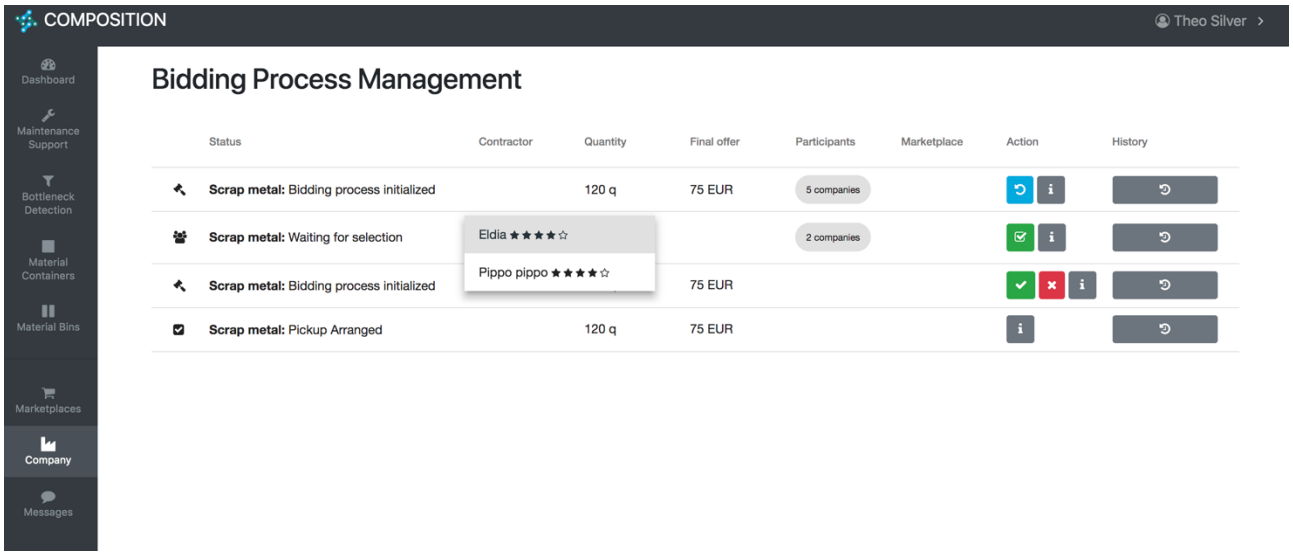

Figure 14: Bidding process contractor selection

Currently in development are functionalities for more configuration of the bidding process and matchmaking preferences.

6.4 End-to-End Security

This section will cover security aspects of both Marketplace and RabbitMQ; Mainly authentication and communication encryption, as well as the digital signing of messages transmitted through RabbitMQ.

6.4.1 Marketplace

6.4.1.1 Authentication

User authentication in COMPOSITION Marketplace is done through COMPOSITION Security Framework Authentication service (Keycloak[4]), which among others also is responsible for user storage and management. In the micro frontend architecture, login is done once through Open ID Connect (OIDC), and the received authentication token is shared among the micro frontends where the authorization calls are made as necessary.

The following steps need to be done to secure marketplace UI:

1. Create and configure client on Keycloak

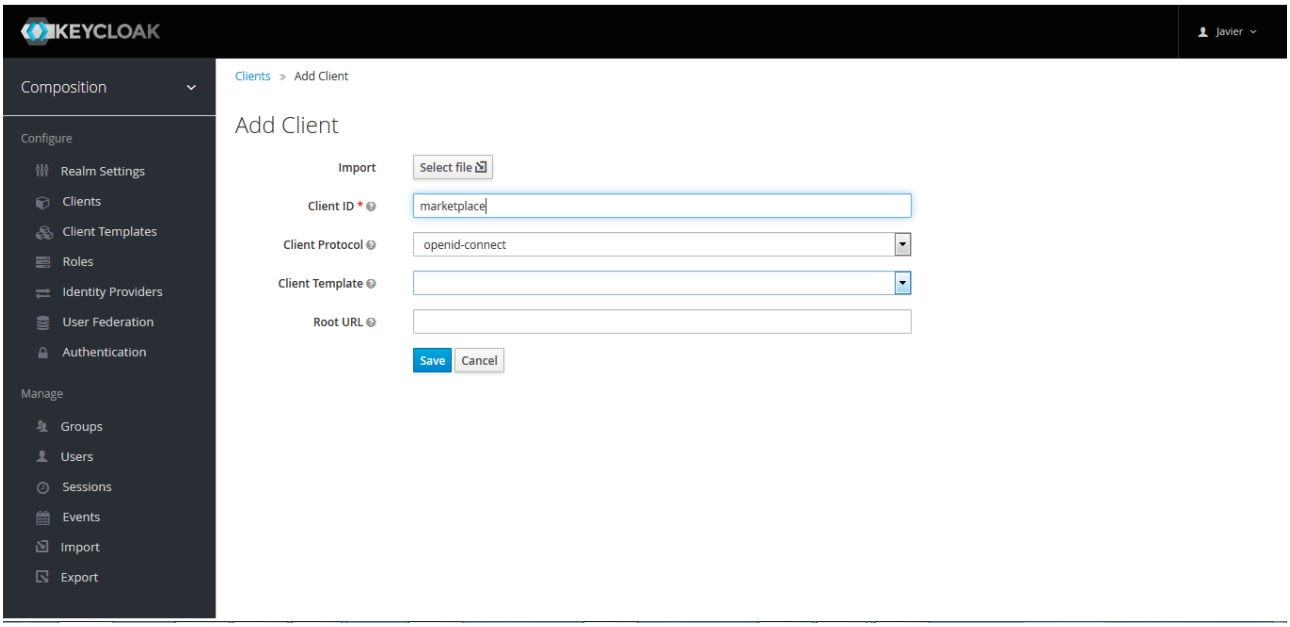

Figure 15: Keycloak client creation interface

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

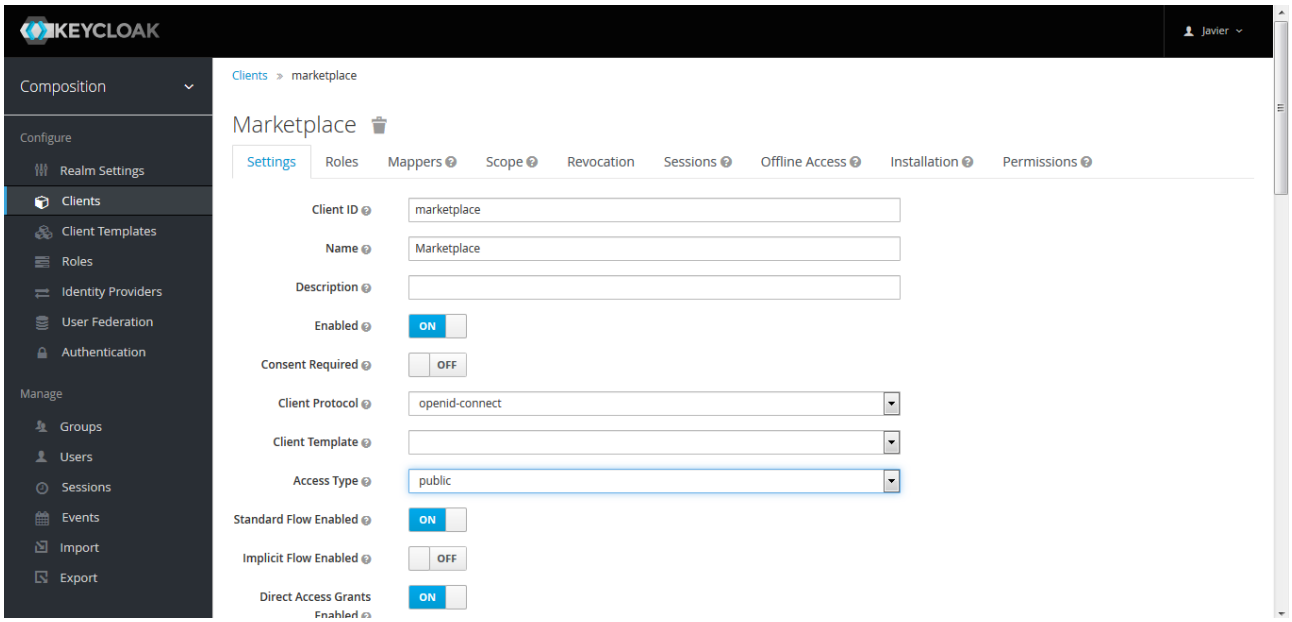

Figure 16: Keycloak client configuration interface

2. Create needed role(s) and users
3. Configure the Marketplace UI component to make use of Keycloak as its login provider. This step will depend on the chosen target platform of the respective Marketplace UI component. Keycloak has a wide range of adapters:
    a. Java adapters (for Tomcat, Jetty, JBOSS web servers)
    b. Javascript adapters (for HTML5/Javascript apps)
    c. Node.js (for Node.js web applications using Express)

    If no adapter is available, the generic OpenID Connect Resource Provider (RP) library can be used.

Once everything is configured, users accessing Marketplace will be redirected to a Keycloak login page where upon entering user credentials, these will be validated. If credentials supplied are valid users they will be redirected back to Marketplace where they will be allowed to enter, otherwise they will remain on the login page.

Very similar is the process to secure a RESTful API. In this case no login page or redirections are involved. Users need to obtain a token using their credentials by making a request to the following endpoint:

- /auth/realms/composition/protocol/openid-connect/token

The token obtained is then used to access the RESTful API.

6.4.1.2 Authorization

Once an authentication token is received, the COMPOSITION Marketplace (and all other micro frontend GUIs) use the GUI authorization Service (GaS) component to evaluate access control policies using EPICA (COMPOSITION D4.5).

It expects to receive the Keycloak token in the request together with the requested resource (URL) and HTTP method (GET, POST, PUT, …). It will the validate the received token, extract the relevant information and forward this to EPICA for evaluation. The response is a JSON structure with a Boolean value indicating if the request is authorized or not.

Table 18: Authorization Example

POST https://auth.composition-ecosystem.eu/gas/gas-rest/authorizeGUI Headers: Authorization: Bearer <keycloak_token> Body:

```
{
      "resource": "https://composition-ecosystem.eu/marketplace ",
      "method": "GET",
}
RESPONSE
{
   "permission": true/false
}
```

6.4.1.3 Communication Encryption

As the rest of COMPOSITION services, Marketplace is configured to use TLS[5] (Transport Layer Security) cryptographic protocol to secure all communication with it. Depending on the deployment options in for a specific marketplace, an external reverse proxy (e.g. Nginx[6]) may also be deployed and configured to provide TLS support.

6.4.2 RabbitMQ

6.4.2.1 Authentication

In order to be able to use COMPOSITION Security Framework authentication and authorization services an http service, RAAS (COMPOSITION D4.5), has been developed and two RabbitMQ plugins, *rabbitmq-auth-backend-http* and *rabbitmq_auth_backend_cache* (to reduce the load on RAAS and improve response times), need to be deployed and configured.

The following figure (Figure 17) depicts a high-level view of the RabbitMQ and the COMPOSITION Security framework architecture:

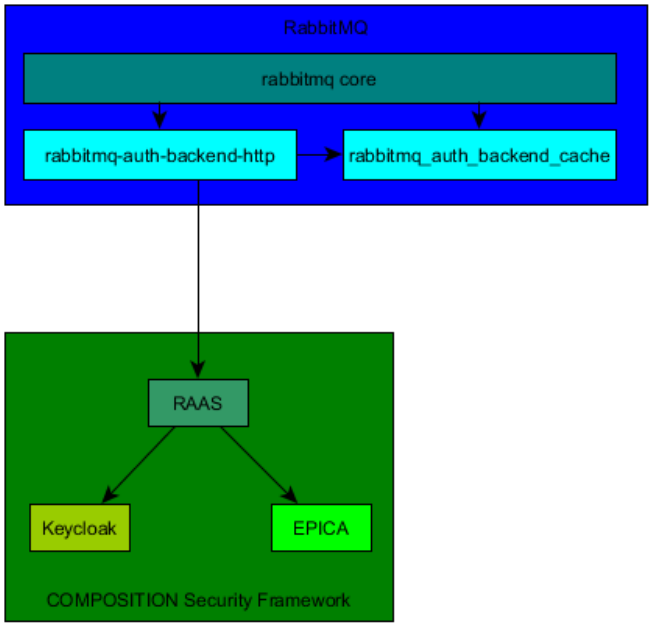

Figure 17: RabbitMQ - COMPOSITION Security Framework architecture overview

6.4.2.1.1 RabbitMQ Authentication and Authorization Service - RAAS

RAAS enables the use of the Authentication (Keycloak) and Authorization (EPICA) services as replacement for the built-in identities and access control in RabbitMQ. More information on these services is provided D4.2 Design of the Security Framework II.

[5] https://tools.ietf.org/html/rfc5246
[6] https://nginx.org/en/

This http service exposes the following endpoints needed by the *rabbitmq-auth-backend-http* (see section 6.4.2.1.2) RabbitMQ plugin:

- /auth/user: user authentication
- /auth/vhost: virtual host authorization
- /auth/resource: resource authorization
- /auth/topic: topic authorization

RAAS supports two ways of dealing with user credentials:

1. Clients authenticate in RabbitMQ with username and password. In this case RAAS is responsible to request and manage tokens from COMPOSITION Security Framework Authentication service (Keycloak) and to perform authorization request to COMPOSITION Security Framework Authorization service (EPICA) with the obtained tokens.
2. Clients authenticate in RabbitMQ with a COMPOSITION Security Framework Authentication service (Keycloak) token. In this mode RAAS is responsible to verify the validity of tokens received and performs authorization requests to COMPOSITION Security Framework Authorization service (EPICA) with the provided tokens. The clients are responsible to obtain and manage the authentication tokens and provide them to RAAS. In this mode no password is passed to RAAS.

6.4.2.1.2 rabbitmq-auth-backend-http

This plugin provides the ability to RabbitMQ to perform authentication and authorisation by making requests to an external http service, in our case RAAS (see Section 6.4.2.1.1). The plugin needs to be configured with the following URIs:

- user_path: user authentication
- vhost_path: virtual host authorization
- resource_path: resource authorization
- topic_path: topic authorization

The plugin expects from the external http service the values *allow* or *deny* as an answer to a request.

The following snippet shows an example of the plugin configuration:

```
[
  {rabbit,[{auth_backends, [rabbit_auth_backend_http]}]},
  {rabbitmq_auth_backend_http,
   [{http_method,   post},
    {user_path,     "http(s)://server:port/auth/user"},
    {vhost_path,    "http(s)://server:port/auth/vhost"},
    {resource_path, "http(s)://server:port/auth/resource"},
    {topic_path,    "http(s)://server:port/auth/topic"}]}
].
```

6.4.2.1.3 rabbitmq_auth_backend_cache

This RabbitMQ plugin provides the ability to cache authentication and authorization results obtained from an external authentication and/or authorization service for a configurable amount of time reducing the amount of load on the external server providing authentication and/or authorization. The following snippet shows how to configure message broker to use cache plugin. In this case authentication and authorization backend results are cached for 1 minute:

```
[{rabbitmq_auth_backend_cache,
      [{cached_backend, rabbit_auth_backend_http}, {cache_ttl, 60000}]
```

```
}].
```

6.4.2.2 Communication Encryption

RabbitMQ has been configured to make use of TLS encryption protocol on both AMPQ[7] and MQTT[8] communication protocols and non-secured communications over these protocols have been disabled. Access to RabbitMQ management UI is also configured to use TLS only.

Below is exposed the MQTT configuration snippet allowing TLS connections only:

```
{rabbitmq_mqtt, [
                {allow_anonymous,  false},
                {vhost,            <<"/">>},
                {exchange,         <<"amq.topic">>},
                {subscription_ttl, 1800000},
                {prefetch,         10},
                {ssl_listeners,    [65183]},
                {tcp_listeners,    []},
                {tcp_listen_options, [{backlog,   128},
                                      {nodelay,   true}]}
]}
```

And below the configuration for AMPQ TLS connections only:

```
{ rabbit, [
                { tcp_listeners, [ ] },
                { ssl_listeners, [ 65182 ] },
                {ssl_options, [{cacertfile,"/etc/rabbitmq/certs/testca/cacert.pem"},
                               {certfile,"/etc/rabbitmq/certs/server/cert.pem"},
                               {keyfile," /etc/rabbitmq/certs/server/key.pem"}]},
] }
```

6.4.2.3 Message Signature

All messages transmitted over RabbitMQ should be signed using JWS[9] (JSON Web Signature) standard, which represents signed content using JSON data structures and base64-url-encoding. JWS representation consists of three parts:

- Header: describes the signature method and parameters employed.
- Payload: message content to be secured.
- Signature: ensures the integrity of both the Header and the Payload.

All three parts are base64-url-encoded for transmission and these are typically represented by the concatenation of the encoded strings in that order, with the three strings separated by period ('.') characters (see Figure 18)

[7] https://www.amqp.org/
[8] https://mqtt.org/
[9] https://tools.ietf.org/html/rfc7515

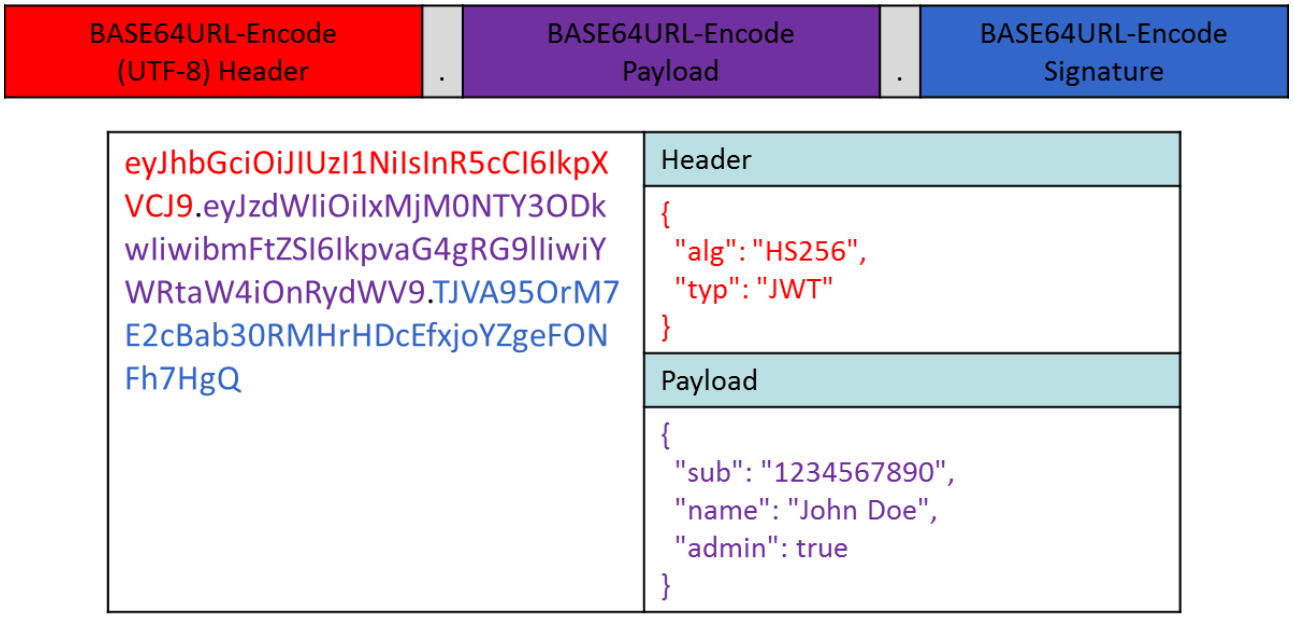

Figure 18: JWS representation

7 Conclusions

The work that has been carried out in Task 6.3: “Connectors for Inter-Factory Interoperability and Logistics” has been extensively described in this document. The activities that have taken place have been divided between the implementation of the different components and the actual interconnections between them.

Several achievements have been obtained. The ones worth mentioning are:

- Interconnection between the AMS, Requester and Supplier agent with the security framework.
- Integrated the support of the Matchmaker in the bidding process, being exploited by AMS and Requester agent.

During the months that have been passed from the first iteration of this deliverable the Marketplace has been continuously improved and tested by adding feature and enhancing software robustness. The development process has not been changed with respect to the initial guidelines defined and described in the first iteration of this deliverable. All the decisions made in the first part of the project have proved to be successful.

The integration between all the components has been carried out in conjunction with the work performed in Task 6.2.

The work has particularly focused on the format of the data exchanged between the different components, a first but very important step for any integration process. In fact, it has undergone a heavy phase of design prior to any integration and testing. The communication models between the different agents (AMS, Requester, Supplier, Matchmaker) have been thoroughly investigated in this deliverable, showing how the integration of the different actors composing the COMPOSITION Marketplace has been performed.
The integration with the security framework has been very important to secure every communication, allowing only authenticated users to participate in the marketplace operations, while keeping every message encrypted.

8 List of Figures and Tables

8.1 Figures

8.2 Tables

9 References

(COMPOSITION D2.1) D2.1 Industrial Use Cases for an Integrated Information Management System

(COMPOSITION D4.5) D4.5 Prototype of the Security Framework II

(Shoham Y., 1993) Shoham Y., 1993. Agent-oriented programming, Artificial intelligence.