Grant Agreement Number: 957204 (H2020-ICT-38-2020)
Project Acronym: MAS4AI
Project Start Date: 1st October 2020
Project Full Title: Multi-Agent Systems for Pervasive Artificial Intelligence for assisting Humans in Modular Production

D2.1 – User Manuals on Accessing and Using the MAS

Dissemination level:	PU
Date:	2022-04-14
Deliverable leader:	DFKI
Contributors:	DFKI
Reviewers:	AIMEN
Type:	R
WP / Task responsible:	DFKI
Keywords:	Keywords: Janus SARL Multi-Agent System, BaSyx-Middleware, Asset Administration Shell, Kafka, Deployment of AI agents

Executive Summary

This document comprises the deliverable D2.1 that describes the setup on the Multi-Agent System based on the Janus SARL runtime as MAS, BaSyx as a middleware for hosting AASs and Kafka as an event strimming middleware. The goal is to separate the agent internal technologies from the agent framework in order enable separate development of AI components which will be efficiently deployed in the MAS. The system will provide the technology for the subsequent integration of various kinds of agents from the work packages WP2 – WP5.

First, the general concepts of the Janus SARL MAS as well as its setup are described. Second, an overview, installation steps and setup of the BaSyx-Middleware, for example for AAS usage, are given. After that, the Kafka middleware installation and setup steps as well as code snippets for creating simple publisher/subscriber are presented. Different possibilities for integrating of agents' programs into the MAS framework are show. In the end the setup of knowledge-based interactions inside the MAS4AI framework is discussed.

Document History			
Version	**Date**	**Contributors**	**Description**
V01	2022-01-18	DFKI	Initial Version of the document
V02	2022-03-30	DFKI	Version for the review
V03	2022-04-11	DFKI	Final Version

Table of Contents

Table of Figures

1 Introduction

The main objective of this document is to describe the setup of the MAS4AI framework, which is based on the holonic Multi Agent System (MAS) framework Janus and the message-based middleware solutions of BaSyx and Apache Kafka, providing a system for integration of various kinds of software agents from other MAS4AI work packages. The integration of Cyber-Physical Production Modules (CPPM) with resource agents as well as for Industrial AI software agents are modelled and described. The integration of several technologies inside of an agent thus will be enabled by a related interface description as provided by the Asset Administration Shell (AAS) and the Janus skill pattern, for template-based concept with the SARL-language for separation of implemented functionalities without changing the agents itself. The implemented framework setup follows the approach, to integrate available microservices of production environments which can be connected to agent types.

The MAS4AI framework in general follows a modular approach, which is independent of the selected technological implementation. Nevertheless, the reference setup of the framework elements is provided with related MAS frameworks as well as middleware solutions, but the main scope is to enable an approach to provide integration possibilities for several technologies, if they can implement the concepts of this framework. The framework related system elements, like the used MAS, can be deployed as own Docker environments and the related open-source framework parts like BaSyx also provides encapsulated functions like registry or AAS server functions inside of predefined and deployed docker containers.

The specification and integration of information models and interfaces for agents in the framework thus follows this general approach, which is provided and integrated in the MAS4AI framework by using the AAS for agent usage. The usage of the AAS as well as registration and discovery, which is needed in the MAS4AI framework, thus follows the concept of assets with virtualized representation with the AAS.

The sequence model how (holonic) agents can interact with the information models of agents and assets as well the integration of the knowledge base for agent interactions are described. The integration of the knowledge base as well as the usage of stored data can also be integrated with the same microservice integration to the agent system.

2 MAS4AI framework overview

The initial MAS4AI framework of the proposal considers an approach of several components, which are related to several work packages of the project. Inside this document, the setup of the related components is described, as well as approaches for interface setup towards several agents.

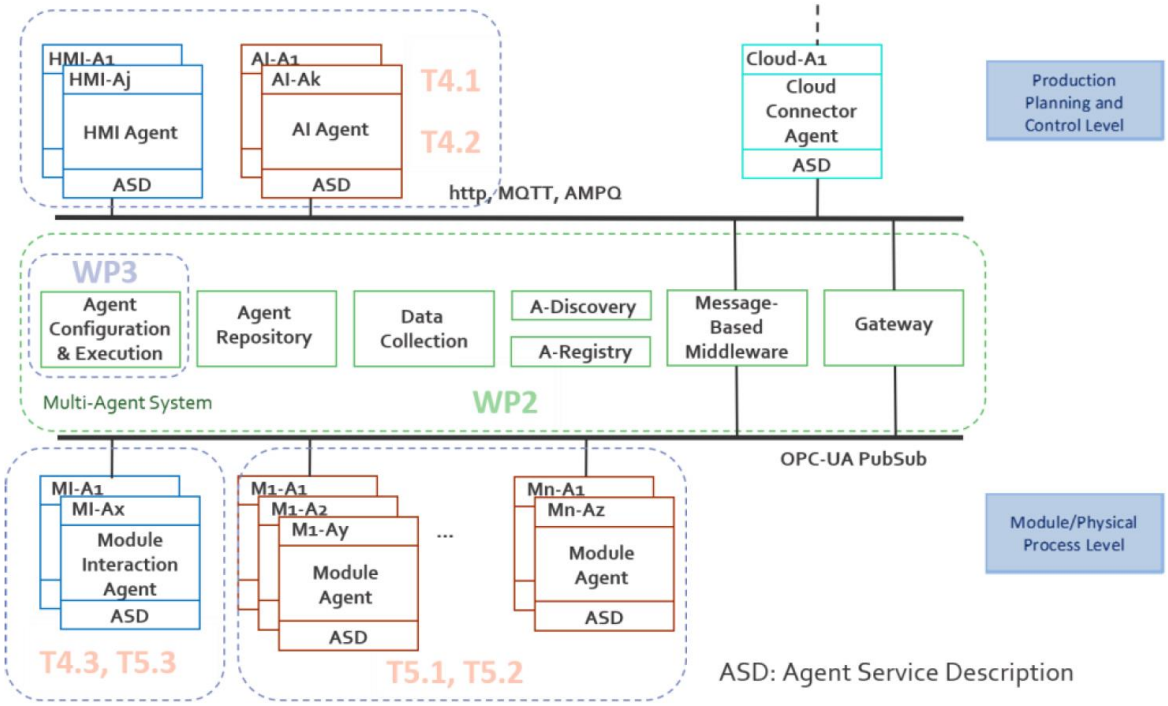

Figure 1: Initial MAS4AI approach with work package relation

The MAS4AI framework consists of several system elements to interact with a manufacturing environment. The setup for these elements and their integration will be described in the scope of this document with a related implementation reference. It is important to note, that for each framework elements also alternative solutions could be used and integrated, if they can follow the requirements of the MAS4AI setup and communication interfaces. The following elements are presented in Figure 2.

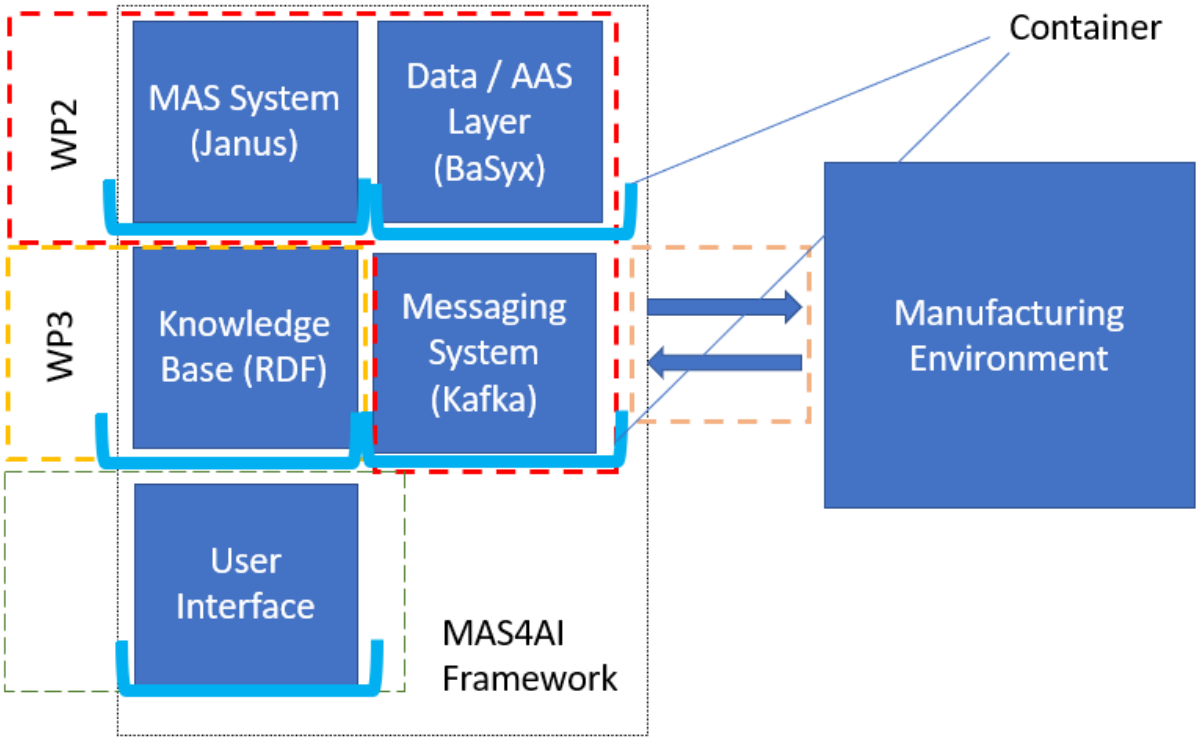

Figure 2: Proposed MAS4AI Framework elements

The MAS4AI framework elements enables a distributed setup, in which each of these elements could also be deployed with different solutions on cloud or edge-side. The general role of these framework elements is described in this document as follows:

- MAS System component

  The MAS system components consist of a MAS framework which support the structure of holonic agents and adaptable software patterns. In scope of the presented approach, the Janus SARL framework has been evaluated due to the possibility to use the agent-oriented programming language SARL. Furthermore, it is possible to use other MAS, if they can provide similar agent patterns, for example JADE. It must be possible to host many instances of the MAS in a distributed environment (or also many different MAS, if they communicate by using standardized interfaces with the agents AAS).

- Data / AAS Layer

  The Data / AAS Layer is used in the framework for the virtual representation of the manufacturing environment and responsible for the setup of the AAS for physical and software assets as well as for the agents itself, representing the functionality of the

agent service description. The Data / AAS environment also consist of functionalities to register and discover available AAS. Several parts of this element could be distributed as well as the MAS System element of the framework. In the scope of the MAS4AI system setup, the BaSyx-Middleware is prepared with the related components of AAS hosting asset integration and data storage.

- Knowledge Base

  The knowledge base stores information about the system configuration and about the manufacturing information in a semantic way. This also is of interest of the interface setup for the AAS for assets and agents and extends the possibilities for discovering and reasoning in the system. In the MAS4AI framework, the integration of an RDF-Store with Dydra as knowledge base is used.

- Messaging System

  The messaging system acts as common communication channel for MAS agents in a distributed environment. In the framework setup, for this element Apache Kafka is explored. The integration of Kafka furthermore enables the possibility to collect data from the Data / AAS Layer, which for example could be used for machine learning AI agents. For agents inside a holonic agent it is also possible to use the communication mechanisms of such a MAS framework, for example Janus.

- User Interface

  The user interface allows to participate with the general framework setup and allows the interaction and monitoring during runtime. The interface therefore should also consider tools for support of the MAS framework configuration.

The proposed method of MAS4AI to build up the framework thus contains the introduced framework elements and the necessity to build up the Data / AAS layer and to combine them with the predefined agent patterns by using the integrated existing manufacturing environment with the AAS as common interface. Nevertheless, the setup and the related configuration and integration of system elements like the Messaging System and the Knowledge Base takes place during this process, building up the complete MAS4AI framework.

a. Digital twin preparation and integration with the AI agents for early assessment,
b. pilot setup that involves integration of existing legacy systems,
c. execution and validation of the individual agents and
d. execution and validation with the whole MAS4AI system

Based on the holonic agent concept of the MAS4AI architecture with static and dynamic holons, the elements interact from a holonic agent point of view with the framework elements in a predefined way.

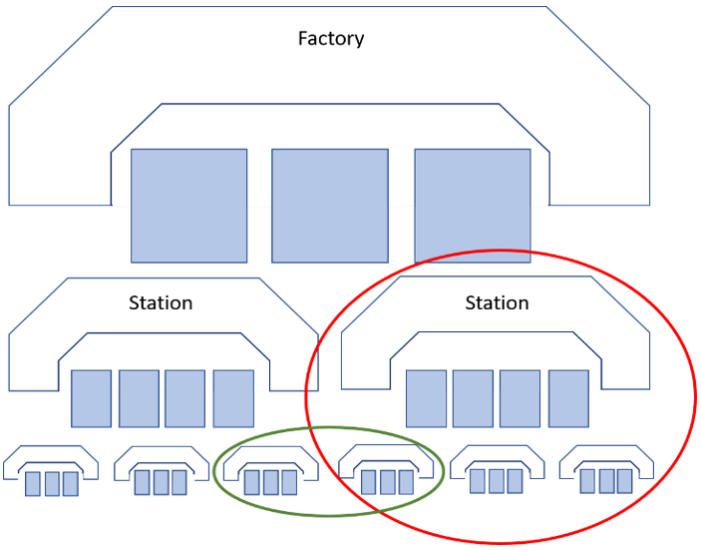

Figure 3: Holonic Approach of MAS4AI

Figure 4 describes the integration possibilities of the introduced MAS4AI framework elements inside of an holonic agent.

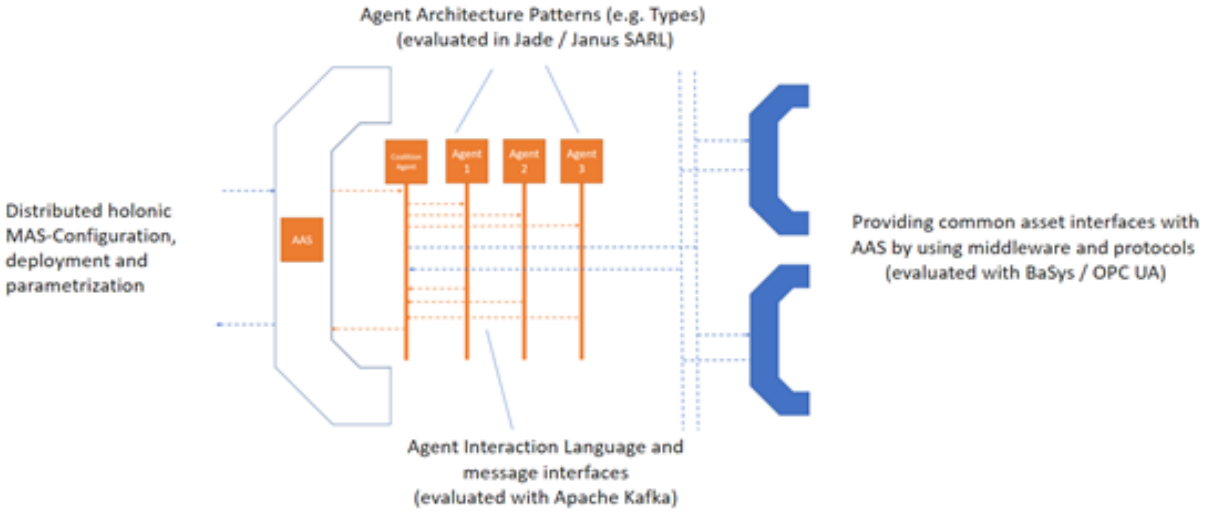

Figure 4: MAS4AI framework elements inside of an holonic agent

A holonic agent in this point of view is an agent, which can be deployed in a MAS framework like Janus and could be described with an AAS that, for example, is provided by using the BaSyx-Middleware. The AAS stores information about the interface of this holonic agent, considering input and output parameters, and can be used to setup the related agent types of this holonic agent. These agent types could be deployed in a MAS like the Janus framework and initialize their own AAS as interface description, but it is also possible to use different MAS.

The search of suitable (holonic) agents in this context could be done by using the BaSyx-Middleware and the knowledge base, which stores semantic information with relation to the agents AAS. In case of a distributed agents or usage of different MAS inside a holonic agent, the communication must be enabled by using a common message channel like Apache Kafka. If a communication with the manufacturing environment is necessary, the agents must access the AAS layer of related assets like physical machines or software components.

3 MAS-Setup

3.1 Janus SARL MAS Setup and Installation

For the MAS setup the Janus SARL environment [1] has been proposed in the MAS4AI project. Due to the requirements, it is also possible to use other, suitable MAS frameworks or MAS related software concepts if they can follow the concepts of MAS4AI.

The setup of the Janus framework is described with own tutorials for installation [2] and guidelines for holonic agent deployment of the MAS4AI Janus demonstration project package, which is based on the generalized Smart Factory testbed implementation. It is very important, that the used SARL Development Environment Version is used with the recommended Java Runtime Environment to avoid installation problems. The Janus SARL Version 0.12.0 (stable) should be implemented with a Java Runtime Environment 8. The Janus MAS framework setup has been tested with a free pre-built JRE from Adoptium [3] (former OpenJDK) and used to setup the basics for the Janus setup.

The MAS development and configuration environment of Janus SARL can be started by using the prebuilt “eclipse-sarl” application. The environment comes also with a Janus command line launcher [4] and Maven plugin for the SARL compiler [5], to generate executable Java Jar-Files based on the domain-specific modelling of SARL.

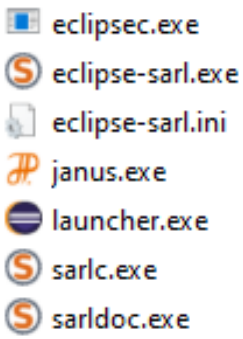

Figure 5: Janus SARL Package [1]

It is recommended to use the pre-configured Eclipse with SARL SDK to work and develop with the predefined environment. The setup of Maven should also be set for Janus projects inside of the SARL IDE.

[1] Janus SARL components for agent development

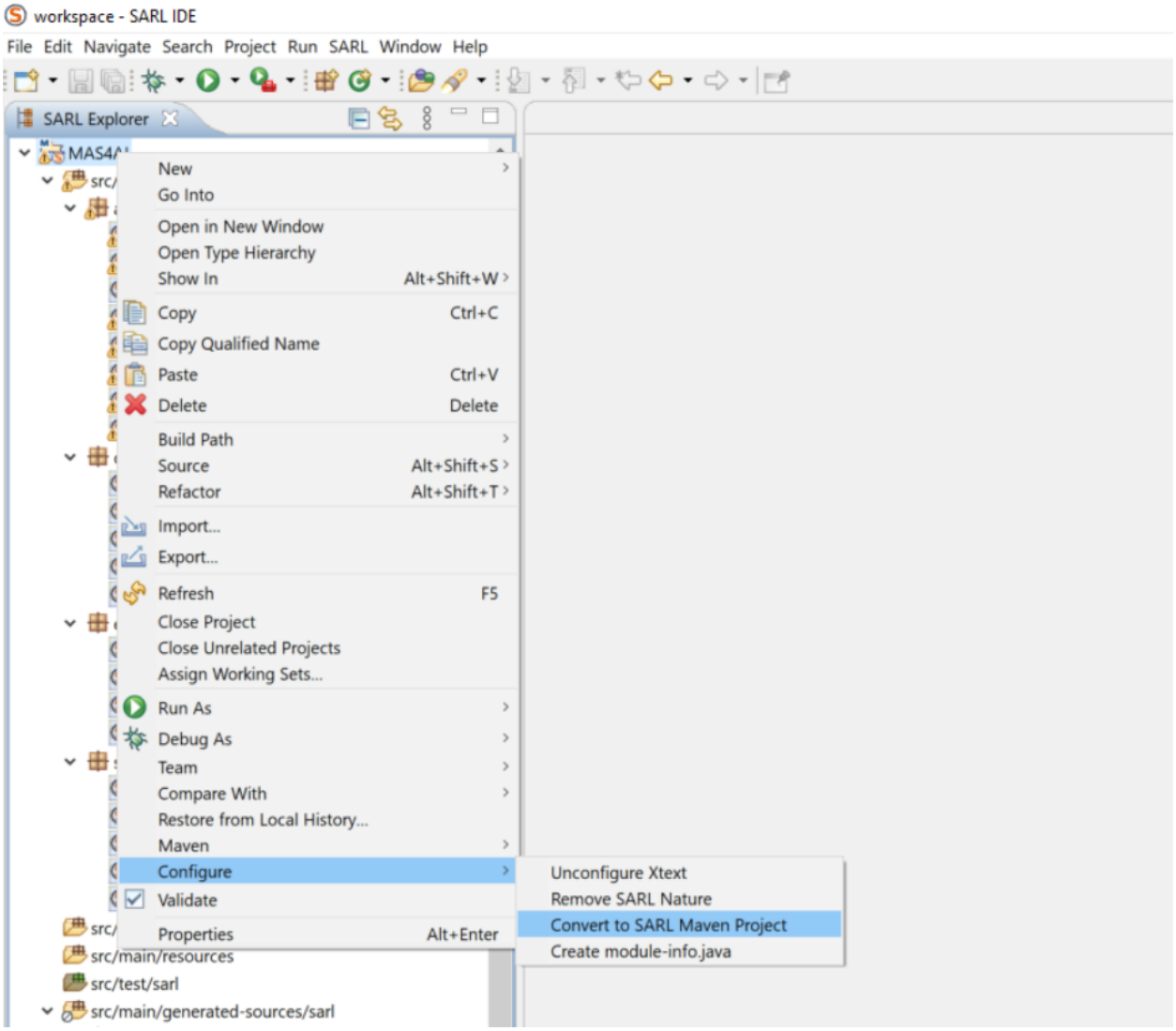

Figure 6: Conversion Janus SARL-Project into SARL Maven-Project [2]

To enable the possibility of a Maven project deployment, the project POM-File must be upgraded with dependencies with are described for actual versions in the Janus SARL installer environment [6], to work with the Maven Build Process.

If a project with related agents is available, the environment can be started with a SARL or Java Launch Configuration [7]. If the project is started, it is possible to use the setup holonic agent for booting purpose.

[2] SARL IDE (Setup as Maven Project)

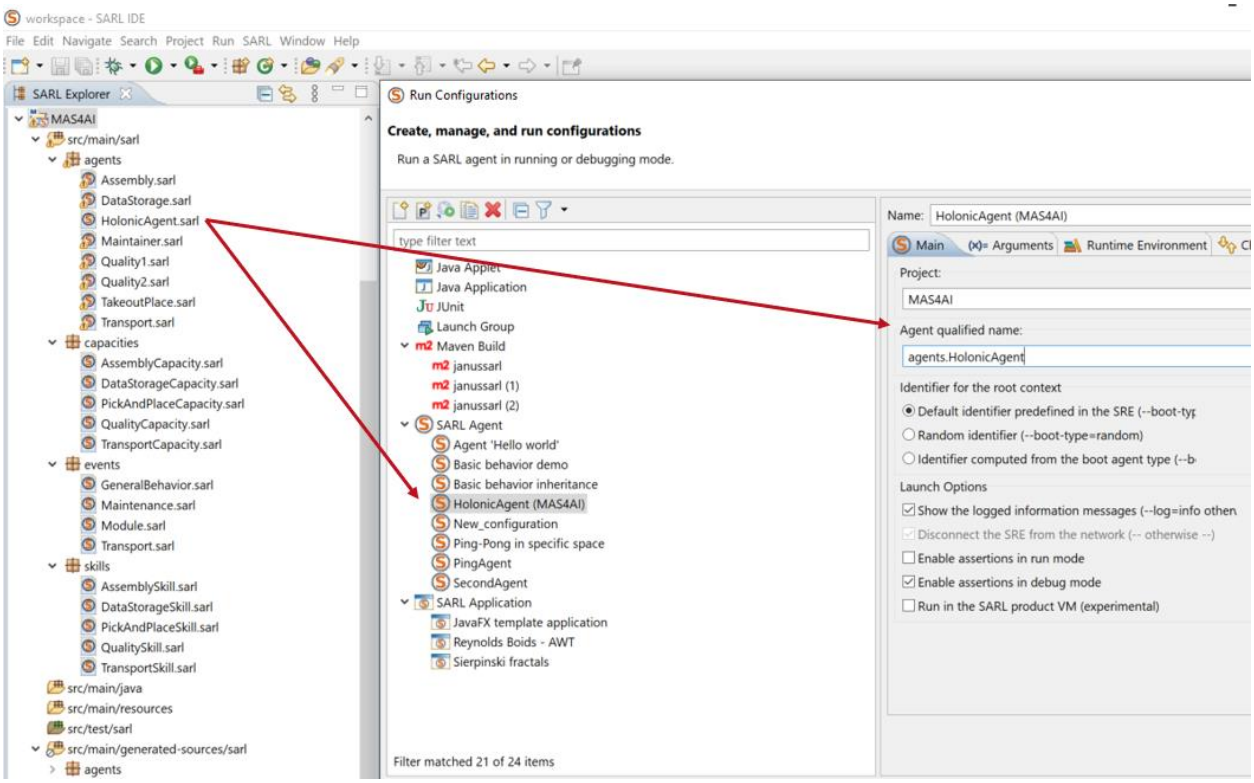

Figure 7: Janus SARL Launch Configuration [3]

The launch of agents can be processed by using a given Java or SARL launch configuration. The SARL launch configuration need the Janus command line tool from the framework, which can start and control several agents in the environment. The Java launch configuration just enables an execution of an holonic agent by using the default Java runtime environment and it is possible to interact with it with several input arguments during the booting procedure.

[3] SARL IDE (Janus Launch Configuration)

3.2 Setup of SARL elements

SARL is an agent-oriented programming language where the SARL metamodel consists of several elements, which are based on the following four main concepts [8] and presented in Figure 8:

- Agent
- Capacity
- Space
- Skill

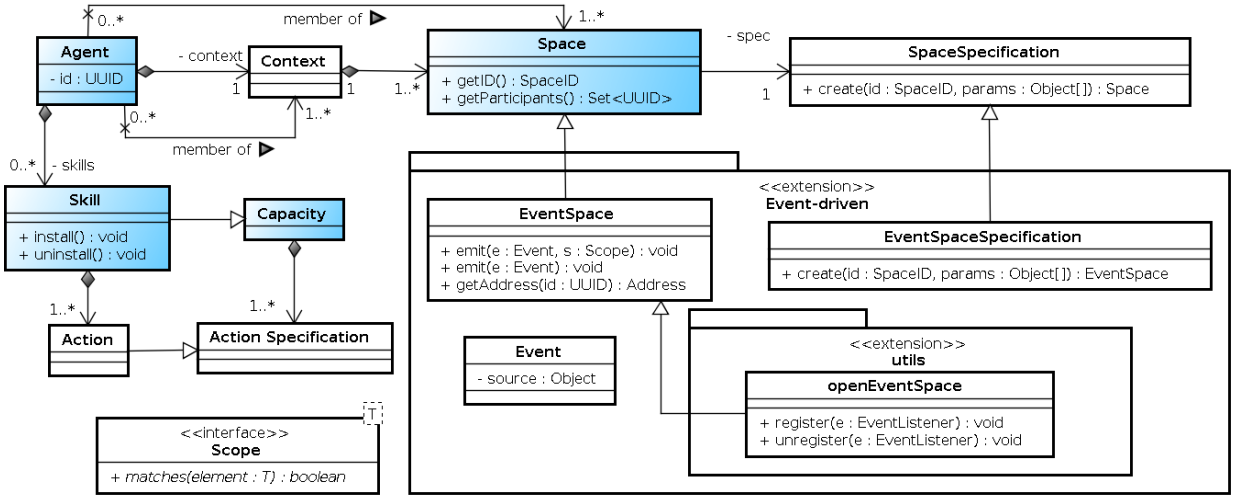

Figure 8: Janus SARL metamodel, from [9]

An agent in the framework can join several communication spaces, which can be specified for an event-based communication. Furthermore, an agent implements skills, which are described as abstract interfaces in terms of capacities. Related to the Janus SARL environment, more objects can be predefined by using the agent-based programming language as be build up as templates.

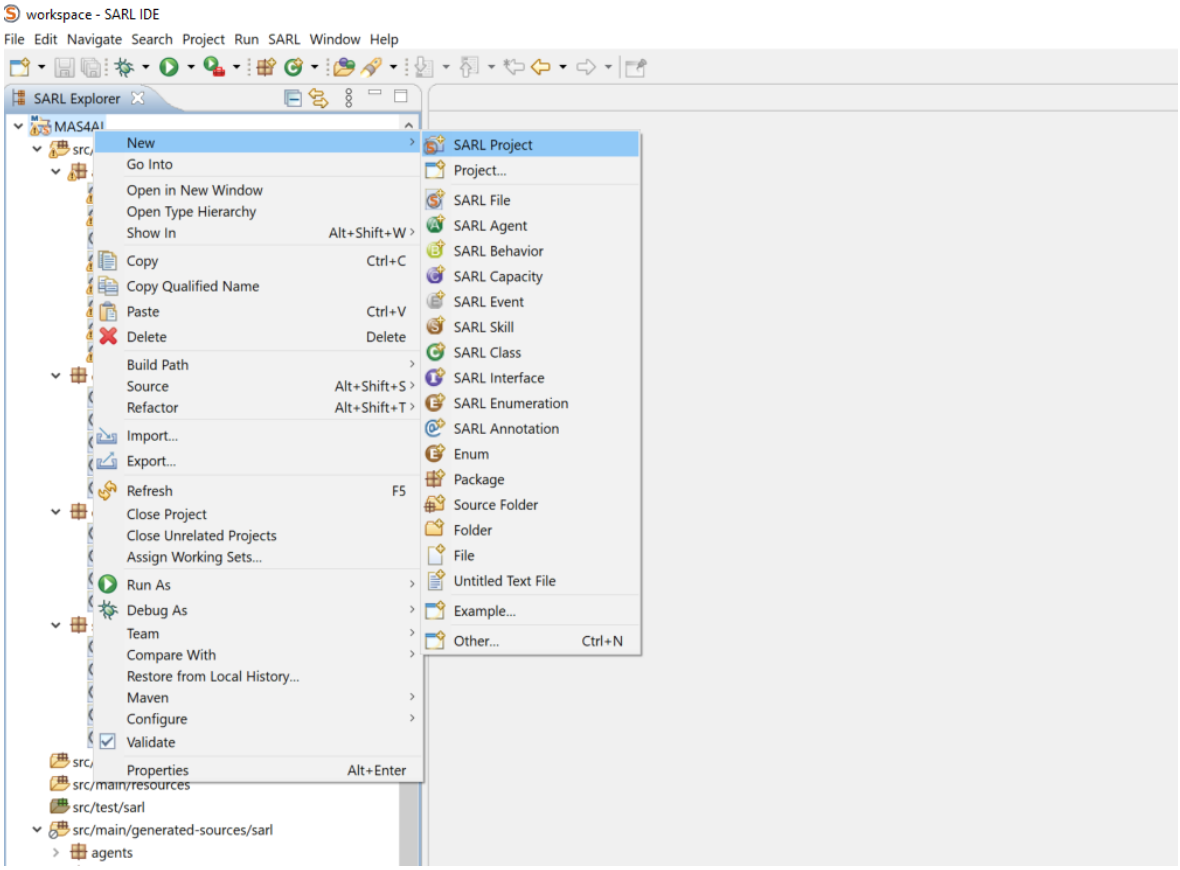

Figure 9: SARL Objects [4]

For the MAS4AI system setup, the related SARL elements of the framework environment can be considered. In the MAS4AI system setup, the usage of these elements for a MAS setup has been validated on the SmartFactory demonstration testbed, considering a template-specific build-up of these elements in a reusable manner. The concept of capabilities and skills thus are used, to build up re-useable agent patterns, which are exchangeable and furthermore adaptable in a respective implementation.

The communication aspects to let the MAS interact with the MAS4AI data / AAS layer, are also setup and encapsulated in the respective skills. This allows to define the agent's behaviour in the MAS4AI environment with similar patterns, regardless the communication aspects of an asset. It is also possible to use the skills to interact with an AAS or an AI agent. In Table 1, all used and evaluated Janus Objects are described. In the next section, an implementation example, related to the Smart Factory testbed environment is presented.

[4] SARL IDE (SARL Objects)

Janus Object	Creatable Object	MAS4AI-Setup
Agent	x	Agent patterns that are defined on basis of the defined requirements. It is possible to define agents for resources as templates and to extend them with the concept of object-oriented inheritance. This allows to provide re-useable SARL templates.
Behaviour	x	The behaviour which has been initialized on each agent can be defined for agents and implements one or many skills, that uses capacities.
Capacity	x	A capacity consists of one or many actions and is used as abstract description for an implementation in skills. A capacity could be for example “Transport” for a resource agent. The concept of capacities allows to setup several and re-useable capabilities of agent patterns, without definition of implementation-relevant details.
Skill	x	A skill implements a capacity and can be used in agents in related behaviours and is used as concept in the MAS4AI framework to communicate with physical or AI agents. Therefore, the skill implements related functions to communicate for example with OPC UA or HTTP REST and to interact with corresponding Asset Administration Shells. It is possible, that agents can implement one or many skills. The concept of the skills enables the possibility to exchange or modify implementations by using own or adapted skills without changing the agent’s behaviour or the template agent itself.
Events	x	Events are defined in own patterns which are useable for agent-related communication in a distributed framework but also for the communication inside of an agent if an internal behaviour should be started or changed.
Space		Default communication mechanism if all related agents are deployed in one Janus Framework environment. The possibility to integrate Kafka as communication channel can be implemented and configured in the related AAS of the given agents.

Table 1: Janus Objects for agent modelling and setup

3.3 Setup Application on MAS4AI example

In this section, various examples of the Janus system setup will be shown and how a framework like Janus can be used to fulfil the required MAS4AI concepts.

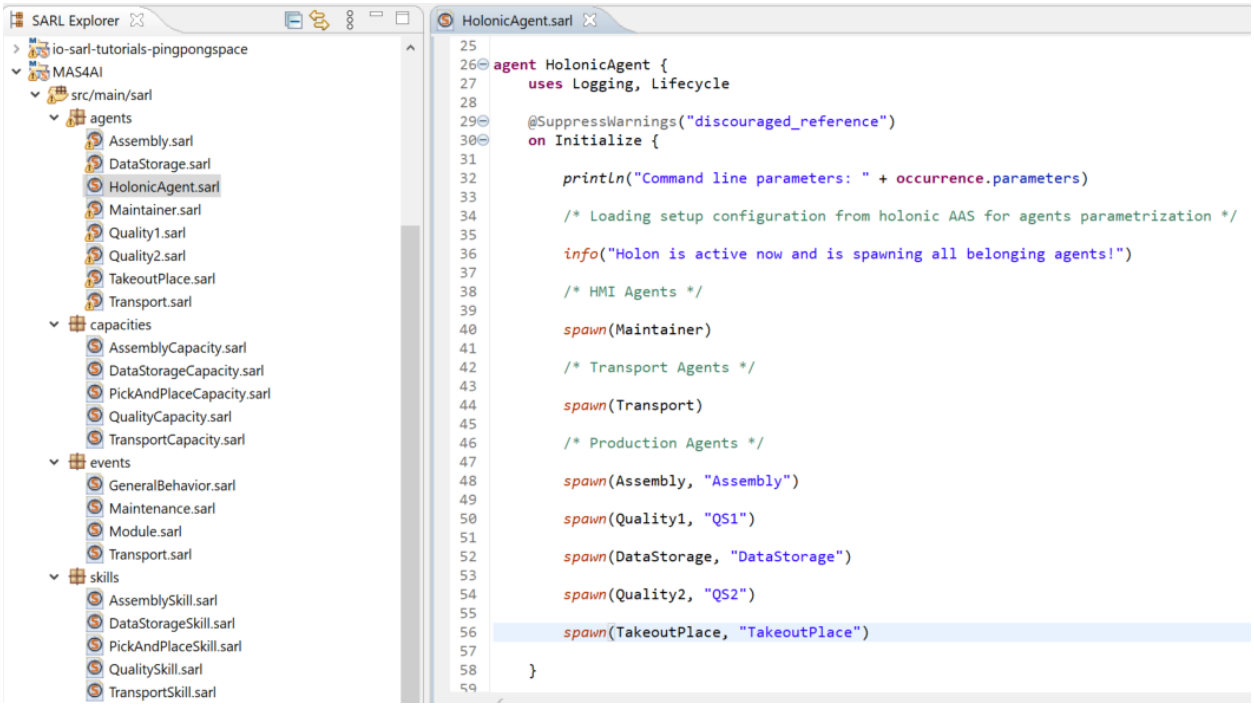

Figure 10: Holonic Agent with initial spawn procedure [5]

Inside of the MAS4AI example, an holonic agent (which is also used for the boot configuration) can spawn predefined agents. In the MAS4AI example implementation of the Smart Factory testbed environment, several resource agents (for manufacturing or transportation) as well as an HMI agent will be spawned during the initialization.

It is possible to setup a configuration for several points in time, for example during initialization, to define which agents should be available and which configuration should be used. Therefore, it is possible to start the holonic agent with several input parameters. For the setup of this agent, the configuration could also be stored inside the related AAS of this holonic agent, loading the configuration and the parametrization of the agents to spawn.

It is possible, if the AAS is used for the configuration, that during runtime of an holonic instance, several changes could be done, to enable a flexible system setup of related agents. Each agent implements a behaviour where related skills are considered can composed during the

[5] SARL IDE (Project Setup and Holonic Agent)

runtime. In Figure 11, an example for a resource agent which is responsible for an assembly module is presented. The resource agent can be requested to start manufacturing by using skills for “Pick and Place” and “Assembly” in a predefined manner, creating a behaviour as an own pattern.

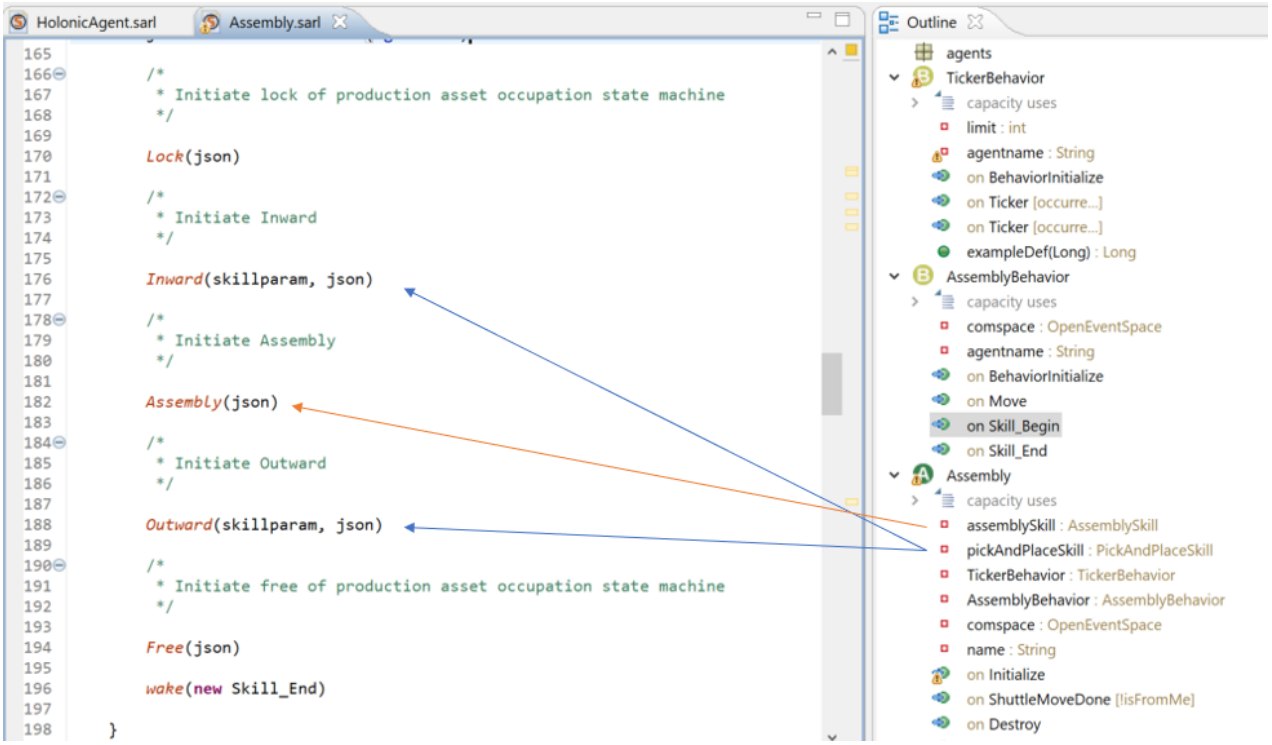

Figure 11: Skill implementation inside behaviour of an Assembly Agent [6]

To use the possibility to implement skills inside of an agent or behaviour, at first the related capacity must be defined, as presented in Figure 12.

[6] SARL IDE (Skill usage inside of agent behavior)

```
capacity AssemblyCapacity {

    /* Build JSON Argument Call String */

    def BuildJSONCommand(asset : String) : String

    /* Get Control-Lock of Production Asset State Machine */

    def Lock(json : String) : Integer

    /* Invoke Assembly */

    def Assembly(json : String) : Integer

    /* Remove Control-Lock of Production Asset State Machine */

    def Free(json : String) : Integer

}
```

Figure 12: Example Capacity object for AAS-related submodel methods of an Assembly Module [7]

The predefined methods of the capacity are matching to AAS submodel operations, which must be available in the related data / AAS layer, for example by using the BaSyx-Middleware or OPC UA. The skill thus also represents the communication related implementation to interact with an AAS. The implementation is based on a predefined AAS submodel structure with operations and properties, which represents a physical or software asset.

HolonicAgent.sarl | Assembly.sarl | AssemblySkill.sarl

```
82      }
83
84      override Assembly(json : String) : Integer {
85
86          /*
87           * AAS-Example-Command
88           */
89
90          var url_command : String = "" // Insert AAS-URL here
91          var response : int
92
93          if (url_command.equals("")) {
94              info("Access of AAS-Function Assembly - TBD.")
95              return response
96          }
97
98          var url : URL = new URL(url_command)
99          var connection : HttpURLConnection = url.openConnection() as HttpURLConnection
100
101         connection.setDoOutput(true)
102         connection.setInstanceFollowRedirects(false)
103
104         connection.setRequestMethod("POST")
105         connection.setRequestProperty("Content-Type", "application/json")
106
107         connection.getOutputStream().write(json.getBytes("UTF-8"))
108
109         response = connection.getResponseCode()
110         info("Response code for Assemble is: " + response)
111
112         connection.disconnect();
113
114         return response
```

Figure 13: Communication pattern inside the skill [8]

[7] SARL IDE (Capacity Example)

[8] SARL IDE (Implemented Skill Example)

The definition of several skills patterns, for example if the request of an AAS is done, can help to support a standardized AAS interface description, which can be flexibly adapted by the agents. For future setup mechanisms, this approach could be combined with a dynamic search in the related RDF Store, as knowledge base, and an AAS registry, to search for related assets, their capabilities, and their communication possibilities. Related to the search results, it should be investigated, if the selection for the suitable skill template in Janus could be parametrized by an agents AAS or be selected dynamically during the runtime behaviour of the MAS.

4 AAS BaSyx-Setup

4.1 Overview of BaSyx-Middleware Elements

The BaSyx-Middleware consist of several components, interactions, and interfaces, which are intended to use on four layers, which are described as follows [10]:

- Plant Layer

  The plant layer consists of higher-level components, which can manage and monitor the production. Furthermore, optimization approaches can also be seen on this level.

- Middleware Layer

  The middleware layer provides reusable Industrie 4.0-Components which implements capabilities for production lines. This layer also provides components for Registry and Discovery, protocol gateways and software to provide the Asset Administration Shell.

- Device Layer

  Based on automation devices with an interface to integrate into BaSys. This layer can also provide the BaSys-conformant interface for field layer components without this interface.

- Field Layer

  Based on automation devices, sensors, and actuators without interface to BaSys.

Related to the requirement of the MAS4AI framework for virtualized representation with defined interfaces of the manufacturing environment, the BaSyx-Middleware provides the possibility to integrate field level devices for data collection and representation by using the AAS.

Related to the intended setup of the MAS, Agent Service Descriptions are necessary, which can also be provided with the AAS. The necessity of an Agent Repository and Agent Discovery Service can thus be based on the same mechanisms of AAS for physical or virtualized assets. The gateway function of BaSys thus enables the possibility to integrate various components into a single point for data collection and usage, for example by using an AAS.

The MAS system element of the proposed MAS4AI framework can thus be seen as additional element on the Plant Layer of BaSyx, which is responsible for control and monitor of production

line, but it is also possible to let agents act on several levels, like for example the device layer. Furthermore, the collected data from the environment could be used for further optimization by AI agents. For the MAS4AI framework approach, the overview of the BaSyx components is enhanced with several elements, as presented in Figure 14:

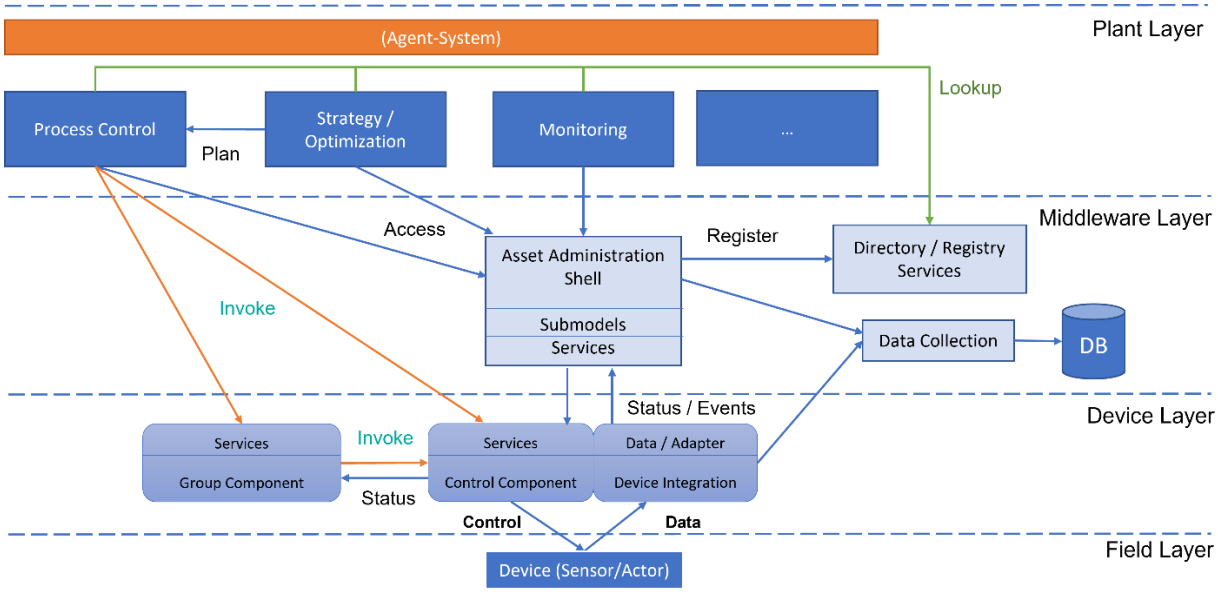

Figure 14: BaSyx-Components according to [10], extended with agent-based component

The (holonic) agents of the MAS framework components interact with the AAS of integrated assets of the BaSyx system. On the Device Layer, the concepts of the Group Component, the Control Component as well as de Device Integration plays an important role. The device integration object which is used in the MAS4AI AAS / Data Layer setup thus consists of protocol adapters which translates data from and towards the Field Layer in a BaSys-conformant way and stores the information in a related AAS.

For control purpose, the concepts of the Control Components are considered for agent-related process control. The Group Component could be used as aggregated concept to interact with many control components and could be used for holonic agents as well as for resource agents. The control component also provides the possibility as predefined interface in the AAS of the asset to let agents interact with the related submodel by using agent-related concepts, like the Janus SARL skill pattern. The executed skill of the agent therefore uses the AAS with the Control Component submodel and places the request with the predefined structure in a suitable call pattern. If the AAS is hosted in HTTP or OPC UA, skills for these implementations could be used.

BaSyx thus provides several elements that are categorized into Plattform I4.0 compliant components and BaSys components, with the following main components [11]:

- Asset Administration Shell with Submodels

  The AAS provides a digital representation of I4.0 assets, where submodels describes one logic aspect of the represented asset, which are accessible by using HTTP REST or OPC UA.

- Registry

  The registry allows to register of new AAS and the search by using the identifier.

- Control Components

  The concept of the control component provides a unified possibility to access services on devices, for example by using related state machines like PackML.

Figure 15 visualizes the components of the BaSyx-Middleware, which are described in the BaSys documentation [11]. Each of these components provides own documentations of their integration and setup.

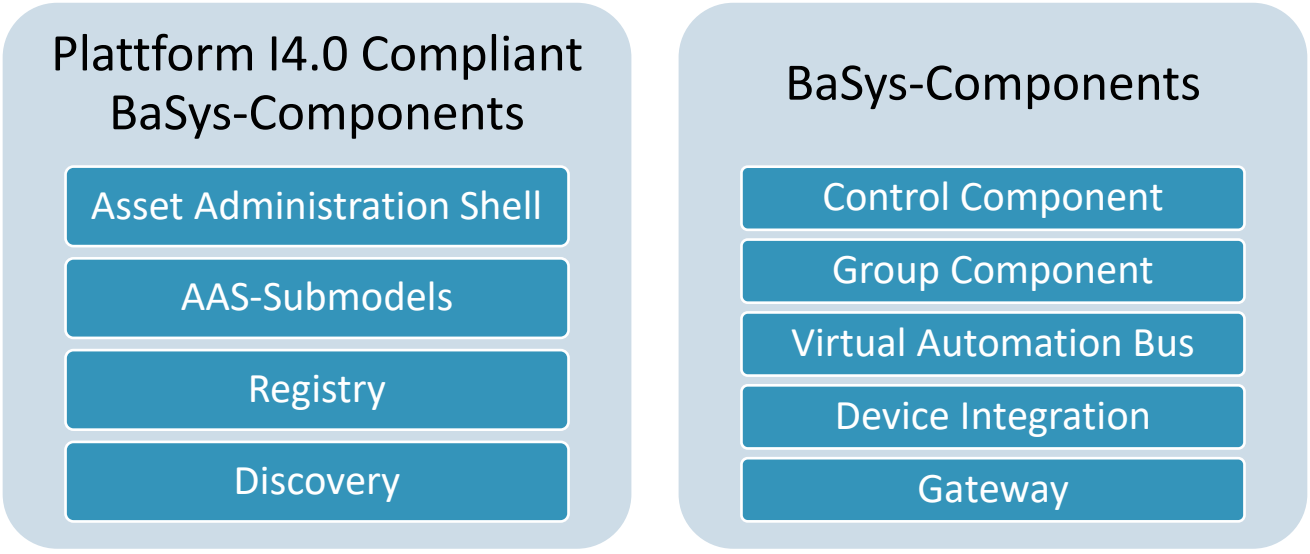

Figure 15: Platform I4.0 compliant and additional BaSys-components, according to [11]

4.2 BaSyx-Installation and setup of related concepts

The BaSyx-Middleware provides implementations with different programming languages, which are provided as own development stacks [12]. The development stacks which provide the largest functional scope are the implementation in Java, .Net and C++. The implementations in RUST and Python thus are additional stacks, which are provided. In the scope of the MAS4AI framework setup, the BaSyx-Middleware which is implemented with the Java SDK has been used. It is also possible to use another development stack, if they can provide similar function scope as provided already in the Java development stack.

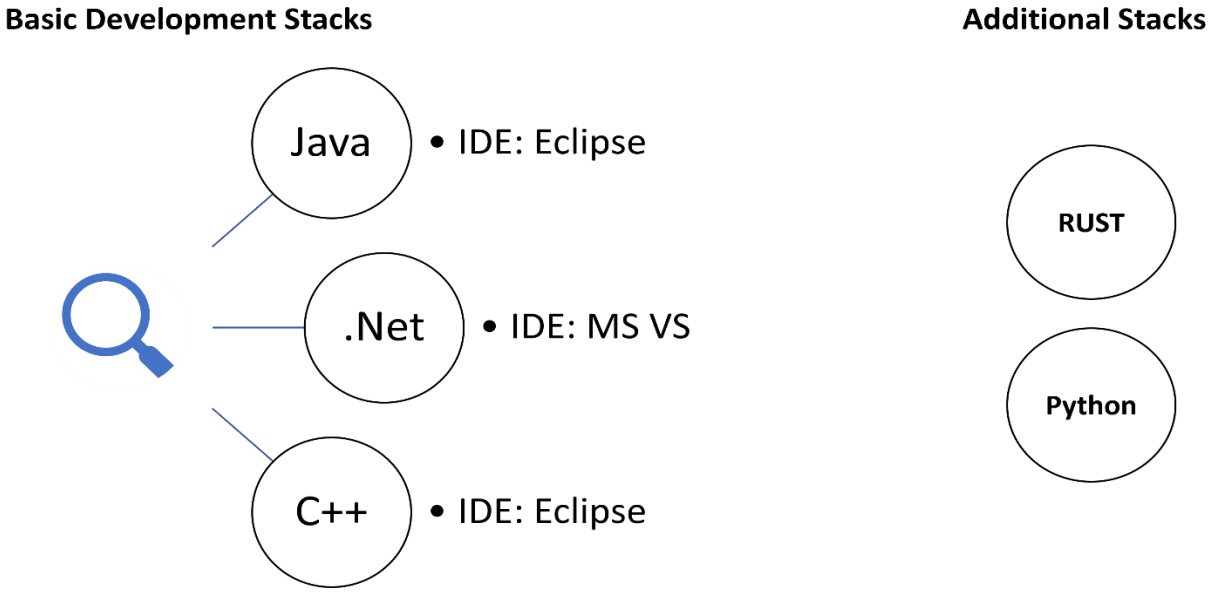

Figure 16: BaSyx Implementations, according to [12]

The BaSyx Java SDK have installation requirements which are described for related setup in the related documentation [13] and can be provided by using GIT. During the installation of the BaSyx Java SDK, the projects with dependencies must be imported (BaSys.sdk, BaSys.components and BaSys.examples). The implemented projects can be built by using the functionalities of Maven.

The content of the several packages consists of Java files. The core elements and basic functions like the AAS and submodel classes can be found inside the BaSys.sdk, and additional components, for example database connectors and registry components are in the BaSys.components project.

The structure with the mentioned projects and dependencies are presented in Figure 17:

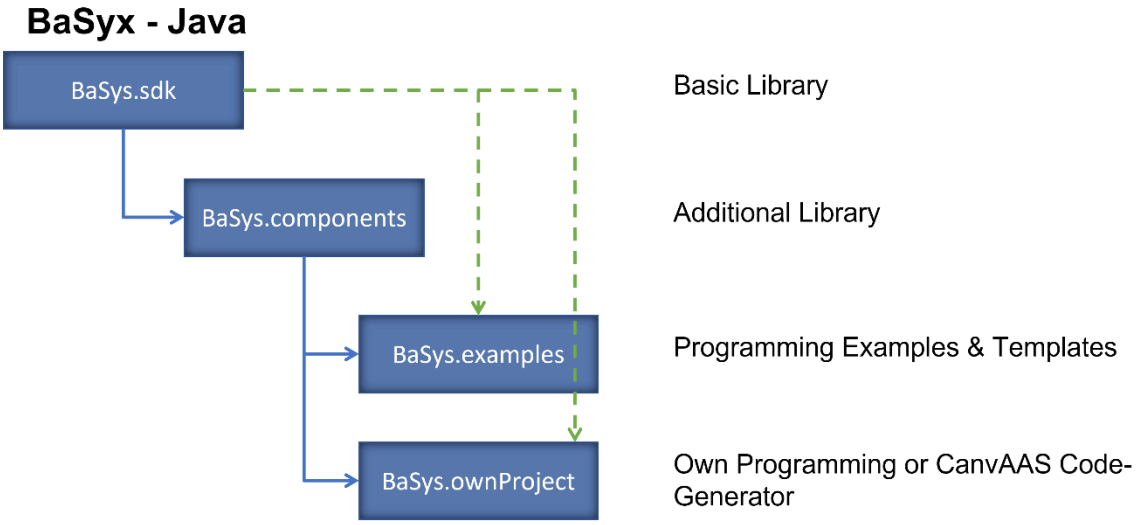

Figure 17: BaSys Java SDK projects and dependencies, according to [13]

The BaSys.sdk consist of core functionalities, as presented in Figure 18. This project has all classes, to build up the AAS, the AAS Submodels with all corresponding elements and the components for the Virtual Automation Bus, which is needed to integrate several protocols.

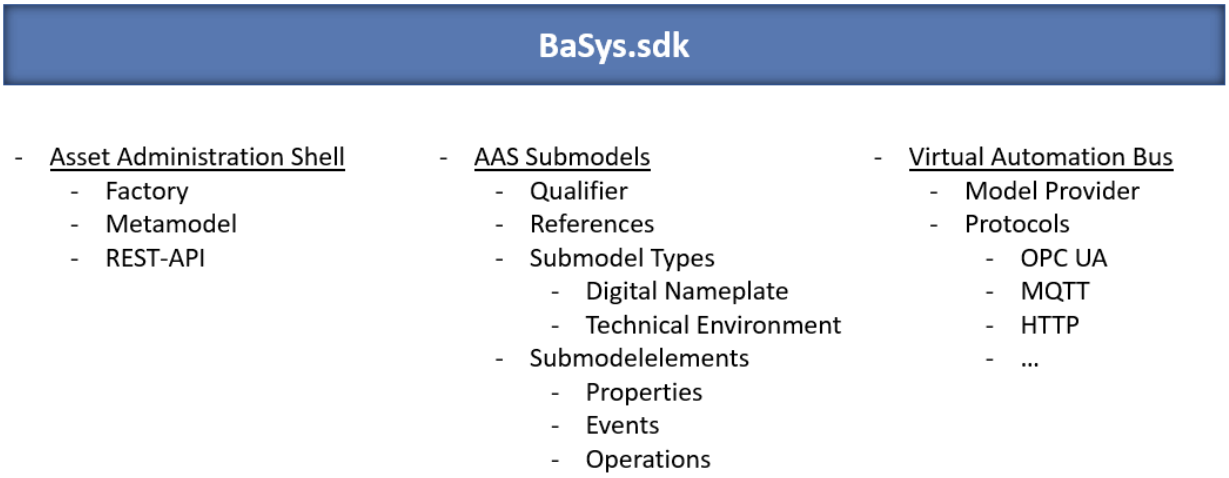

Figure 18: BaSys Java SDK with content of BaSys.sdk [9]

[9] BaSyx-IDE (BaSys.sdk)

The BaSys.components project consist of a library part, where the components are available as additional Java classes, and a docker-related part. In Figure 19, the content of this project is presented:

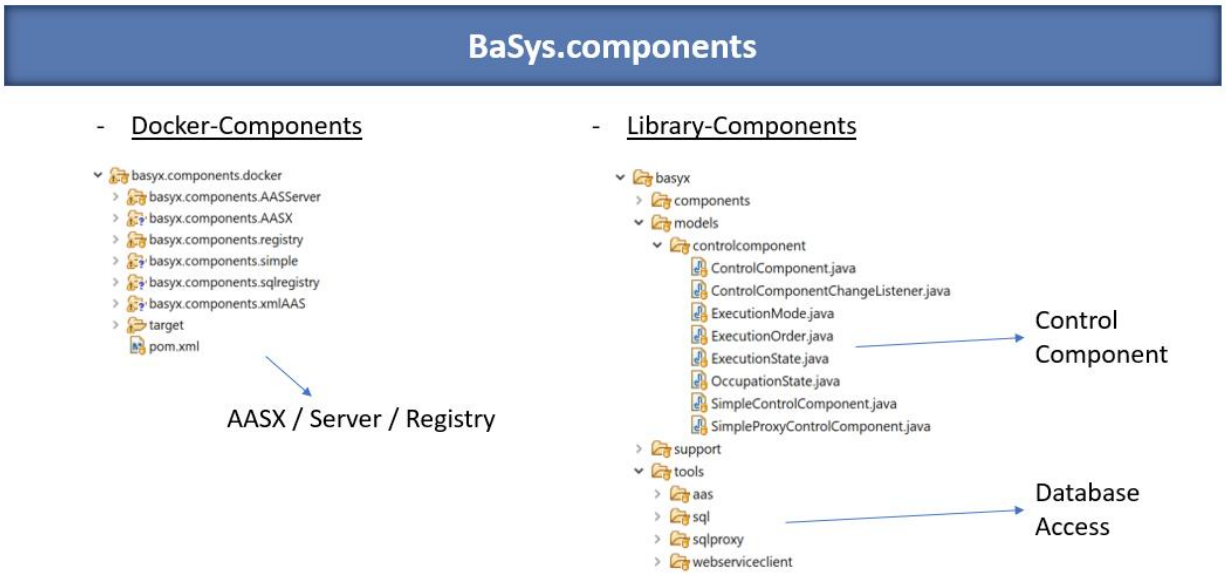

Figure 19: BaSys Java SDK with content of BaSys.components [10]

The docker-related components provide the possibility to set up an AAS deployment for several elements, which are necessary for the MAS4AI framework. Docker files with documentation can be found for the following components:

- AAS Server [14]

  Component to host several AAS with submodels for assets or MAS4AI agents.

- AAS Registry [15]

  Component to host registered AAS for assets or MAS4AI agents

- Updater Component [16]

  Component for asynchronous communication, providing data from several data sources like Apache Kafka to related AAS.

The BaSys.examples project provides several programming patterns and examples, how the elements of the BaSyx Java SDK can be used, for example to create and update an AAS with

[10] BaSyx-IDE (BaSys.components)

related submodels. It is also shown how predefined AASX models, which are modelled in the AASX Package Explorer [17], can be hosted by using the BaSyx-Middleware. This part is also relevant for the MAS4AI framework, to integrate and setup the modelled AAS for several agent types with the proposed functionality.

The examples which can be applied to the scope of the MAS4AI framework is related to the AAS and AAS Submodels examples and the AASX Hosting example, with consideration of the BaSyx Registry and AAS Server component. The content is presented in Figure 20:

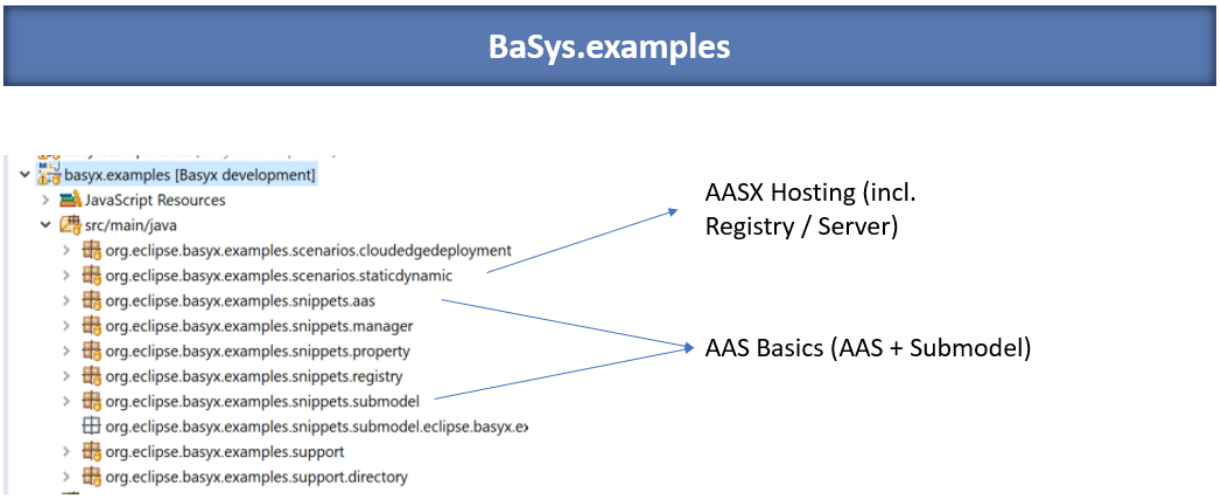

Figure 20: BaSys Java SDK with content of BaSys.examples [11]

An own project can thus be developed on basis of the predefined examples or by using Code-Generators like Eclipse CanvAAS [18]. The structure of an own implementation furthermore follows the same dependencies as the BaSys.examples project.

[11] BaSyx-IDE (BaSys.examples)

4.3 Setup of AAS with AASX Files for Agent Service Descriptions

The setup of the BaSyx components to build up AAS Server and registry with the AASX is required for the MAS4AI (holonic) agent's setup. Therefore, the modelled AASX information models can be integrated in BaSyx for several agent types and then instantiated. In Figure 21, a modelled example of a resource agent in the AASX Package Explorer is shown.

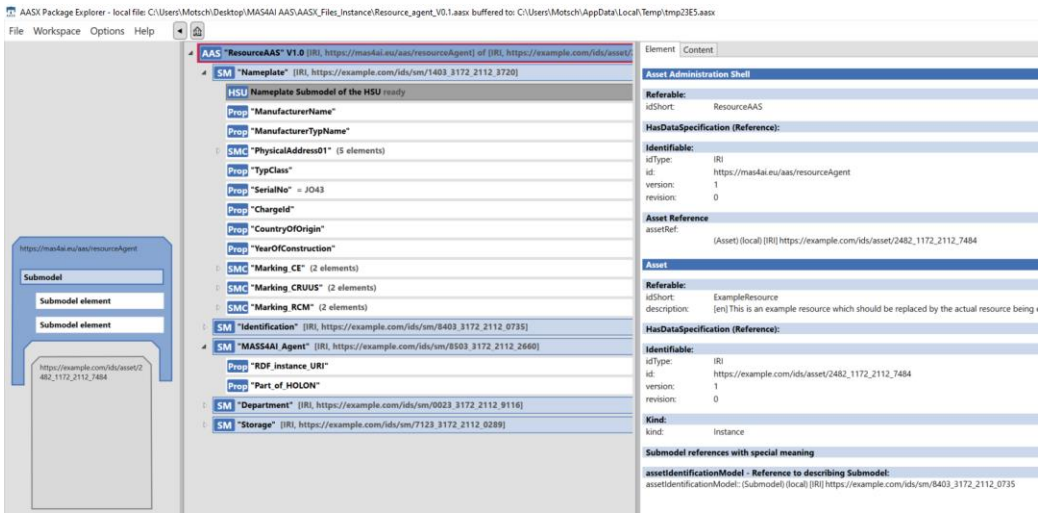

Figure 21: AAS Model for Resource agent [12]

To integrate the AASX files for usage in BaSyx, they must be integrated as resource files into the BaSyx project.

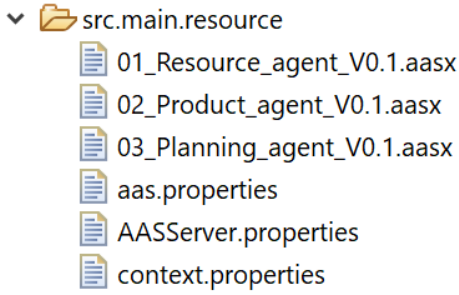

Figure 22: Integration of AASX Files into the BaSyx project [13]

Inside BaSyx, an own AAS Server must be created, and the AASX file of the resource folder can be integrated and registered on the started registry component.

[12] AASX Package Explorer (with Resource agent AAS model from WP3)

[13] BaSyx-IDE (Resource-Package of MAS4AI implementation for AAS Hosting)

ResourceAgentAASServer.java

```
53      ¤¶
54      private List<IComponent> startedComponents = new ArrayList<>();¤¶
55      ¤¶
56      public static void main(String[] args) throws InvalidFormatException, IOException, ParserConfigurationExc
57          new ResourceAgentAASServer();¤¶
58      }¤¶
59  ¤¶
60      public ResourceAgentAASServer() throws InvalidFormatException, IOException, ParserConfigurationException,
61  ¤¶
62          // Startup the registry server¤¶
63          startRegistry();¤¶
64          ¤¶
65          // Startup the server¤¶
66          startAASServer();¤¶
67          ¤¶
68          // Load Bundles from .aasx file¤¶
69          AASXPackageManager packageManager = new AASXPackageManager("01_Resource_agent_V0.1.aasx");¤¶
70          Set<AASBundle> bundles = packageManager.retrieveAASBundles();¤¶
71          ¤¶
72          // Load the new Bundles to the Server¤¶
73          AASBundleHelper.integrate(new AASAggregatorProxy(SERVER_URL), bundles);¤¶
74          ¤¶
75          // Get a RegistryProxy and register all Objects contained in the Bundles¤¶
76          AASRegistryProxy proxy = new AASRegistryProxy(REGISTRY_URL);¤¶
77          AASBundleHelper.register(proxy, bundles, SERVER_URL);¤¶
78              ¤¶
```

Figure 23: Parsing and Hosting the AASX Model inside BaSyx [14]

Related to the program example in the BaSys.examples, the Registry and AAS Server can be started by using the following program line.

Registry

```
/**¤¶
 * Starts an empty registry at "http://localhost:4000"¤¶
 */¤¶
private void startRegistry() {¤¶
    // Load a registry context configuration using a .properties file¤¶
    BaSyxContextConfiguration contextConfig = new BaSyxContextConfiguration();¤¶
    contextConfig.loadFromResource("RegistryContext.properties");¤¶
    BaSyxRegistryConfiguration registryConfig = new BaSyxRegistryConfiguration(RegistryBackend.INMEMORY);¤¶
    RegistryComponent registry = new RegistryComponent(contextConfig, registryConfig);¤¶
    registry.startComponent();¤¶
    startedComponents.add(registry);¤¶
}¤¶
```

Server

```
/**¤¶
 * Startup an empty server at "http://localhost:4001/aasx/"¤¶
 */¤¶
private void startAASServer() {¤¶
    // Create a server at port 4001 with the endpoint "/aasx"¤¶
    BaSyxContextConfiguration contextConfig = new BaSyxContextConfiguration(4001, "/aasx");¤¶
    AASServerComponent aasServer = new AASServerComponent(contextConfig);¤¶
    ¤¶
    // Start the created server¤¶
    aasServer.startComponent();¤¶
    startedComponents.add(aasServer);¤¶
}¤¶
¤¶
```

Figure 24: Program part to start AAS Server and Registry [15]

If the registry component is available, the content of all registered AAS is reachable with HTTP URL, getting all registered AAS with submodels.

[14] BaSyx-IDE (Code Example for AAS Server for the Resource Agent)

[15] BaSyx-IDE (Code Example for Start of AAS Server and AAS Registry)

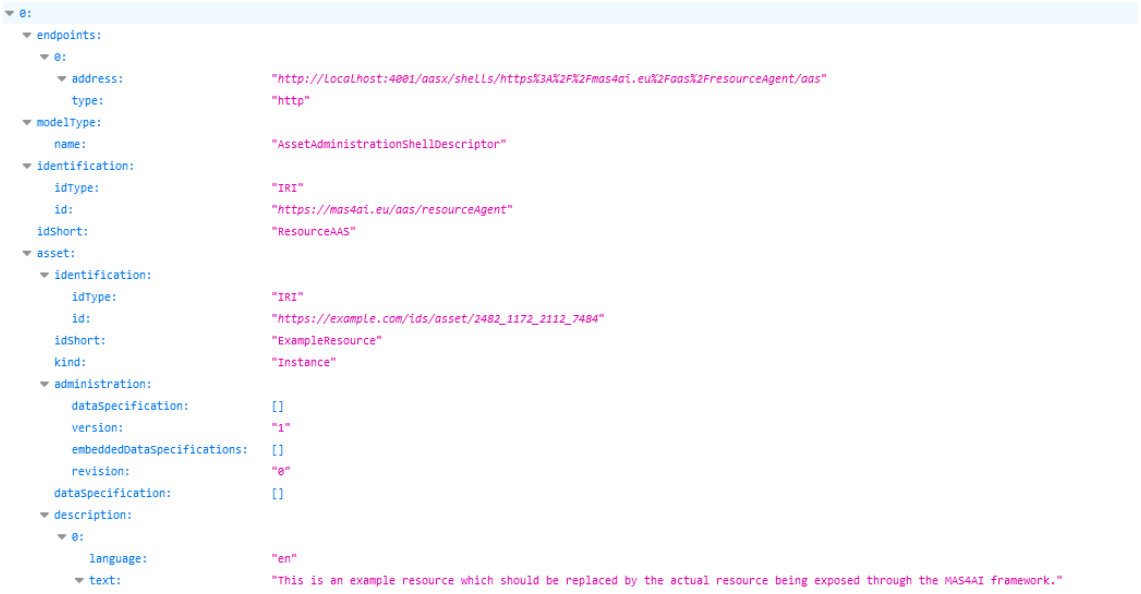

Figure 25: Registry example with registered AAS of Resource Agent [16]

The registered Resource Agent AAS can be reached by using the endpoint information, which is stored in the related Registry. The AAS and all related submodels are then available for usage with the MAS system element inside the MAS4AI framework.

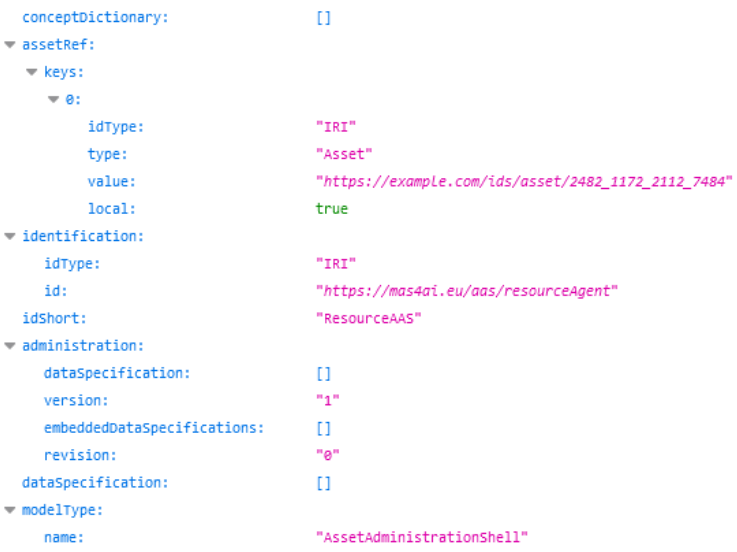

Figure 26: AAS server of the Resource Agent [17]

[16] Asset Administration Shell JSON-Serialization in Web-Browser

[17] Asset Administration Shell JSON-Serialization in Web-Browser

4.4 Data Collection Services

Inside the BaSyx-Middleware, there are several possibilities to store data from integrated devices into databases. Therefore, the BaSys.components project provides functionalities with docker and database connectors in the libraries part, for example with the Postgres-Integration as described in the related documentation [19]. The related connectors to collect data from the AAS can be found in the component. The related classes are shown in Figure 27:

Figure 27: BaSyx-Middleware - SQL Integration [18]

To collect data from the AAS, classes as shown in Figure 28 can be used to collect data from an AAS to a related database.

Figure 28: BaSyx-Middleware - AAS data collection with SQL [19]

[18] BaSyx-IDE (Java Class with SQL Driver)

[19] BaSyx-IDE (Java Class to prepare AAS properties for storage in the database)

Furthermore, it is possible to use the BaSyx Updater Component to represent data from several data sources like MQTT, Apache Kafka, Apache Active MQ or Eclipse Paho and to synchronize data in an asynchronous way to represent them into the AAS [16]. In the implementation during BaSys 4.0, there are also examples for integration of Kafka into the BaSyx-Middleware and to use communication channels for the data stream [20].

To setup a data collection service inside of an own BaSyx-Implementation, an update program, which actualizes selected AAS properties can be used to collect related data. Therefore, several data storage adapters can be implemented. In Figure 29, an example is given, where the actual values of an AAS property is stored in a connected database as well as provided as Kafka Stream.

```
public void CollectDataWithSQL(String value) {
    DBAdapter.writeDataToDatabase(AssetIDShort, prop.getIdShort().toString(), value);
}

public void CollectDataWithKafka(String value) {
    KafkaAdapter.writeDataToKafka(AssetIDShort, prop.getIdShort().toString(), value);
}
```

Figure 29: BaSyx-Middleware: Setup of Data Collection Services [20]

These possibilities for data collection could be used for AAS of related assets as well as for AAS which are setup for agents. Furthermore, it is possible to use the mechanism to collect data and to store it by using a Kafka data stream also directly in a MAS, for example if an agent observes values which are updated in AAS properties for physical assets, as well if an AI agent wants to collect selected data, for example for machine learning purpose.

[20] BaSyx-IDE (Code Example of data collection program with interface for storage in SQL-Database and Apache Kafka)

5 Kafka Setup

Agents in the MAS need to communicate with each other to effectively solve the problems and reach the goals. That is why the message transport system (MTS) the mandatory part of every MAS. In the setup with just one MAS runtime all the agents communicate using the mechanism provided by this MAS. In the multi-runtime setup some external middleware, external MTS, is needed to provide the reliable messaging services. There are several requirements to the external MTS:

- It should be based on open technologies and independent from the MAS. To establish communication between different MASs the external MTS needs to be MAS-agnostic.
- Agents in the different MASs should not notice that they are interacting the agents from the other MAS.
- The middleware should be robust and scalable solution.
- The middleware can also be used not only as an MTS, but also for the data gathering solution. It is especially valuable for the ML agents.
- The middleware should have extensive modelling capabilities to provide support for complex data types transmission.

Kafka is the middleware that can satisfy all the defined requirements. Kafka is an event streaming platform and combines three key capabilities:

1. To publish and subscribe to streams of events, including continuous import/export of data from other systems.
2. To store streams of events durably and reliably.
3. To process streams of events as they occur or retrospectively.

And all this functionality is provided in a distributed, highly scalable, elastic, fault-tolerant, and secure manner. Kafka can be deployed on bare-metal hardware, virtual machines, and containers, and on-premises as well as in the cloud. Kafka is a distributed system consisting of servers and clients that communicate via a high-performance TCP network protocol.

Kafka runs as a cluster on one or more servers. Some of these servers form the storage layer, called the brokers. Other servers run Kafka Connect to continuously import and export data as event streams to integrate Kafka with the existing systems. This can be especially useful for digesting the data from the factory flow to provide it to the ML agents. Kafka clients allow to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or

machine failures. Clients are available for Java and Scala including the higher-level Kafka Streams library, for Go, Python, C/C++, and many other programming languages as well as REST APIs.

5.1 Kafka main concepts and terminology in relation to agents.

In this section some of the main concepts and terminology used in Kafka are described. Figure 30 shows the relationship between the Kafka producer, consumer, and the topic.

Figure 30: Relationship between the event's producer, consumer and the topic, according to [21]

An event represents a fact that something has happened in the world of interest. Conceptually, an event in Kafka has a key, value, timestamp, and optional metadata headers.

Events as the high-level abstract concept can be easily used to implement the communication between the agents. When an agent wants to send a message to one or more other agents it generates an event of a particular type. The interested agents normally listen to the event types. The event-based communication middleware ensures that the generated event will be delivered to the listeners. The concrete delivery mechanisms depend on the middleware. The types of events can be arbitrary complex and represent the messages defined in ACL from the FIPA standard or the messages from the I4.0 Language. To implement complex event's types in Kafka we can use the metadate headers.

Producers are those client applications that publish events to Kafka, and **consumers** are those that subscribe to these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability. For example, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once. This information subscription pattern is useful when the agents consume data. To enable agents to interact with each other we need to implement

different interaction protocols on top on the Kafka communication layer. These protocols are defined in I4.0 Language.

Events in Kafka are organized in **topics, which** are always multi-producer and multi-subscriber in Kafka. A topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events. First agent-based systems were normally heterarchical where every agent spoke to every other one. That was not efficient form the point of data transfer. Such systems completely lacked structure, were unpredictable in many cases and were very hard to maintain. The holonic architectures use flexible hierarchies that can be build and dissolved in response to the changing conditions. To implement such flexible structures the Kafka topics can be used to constrain the producers and the consumers to the topics.

5.2 Kafka setup for MAS Framework.

Figure 31 shows the block diagram of how possibly two MAS runtime can communicate with the help of the Kafka middleware. Each MAS has the local agent management system (AMS) for the agents' supervisory control, the local directory facilitator (DF) or the yellow pages service for registering agents' services or skills, the message transport system (MTS), and the communication holon, which serves as a communication portal to the other MASs. For the global setup we will also need the global AMS and the global DF. The role of global MTS will play Kafka. Each local communication Holon will have at list one Kafka producer and one Kafka consumer and will ensure the seamless and transparent communication between the agents of different MASs.

Kafka provides several API that can be used in the communication holon:

- The ***Admin API*** to manage and inspect topics, brokers, and other Kafka objects.
- The ***Producer API*** to publish streams of events to one or more Kafka topics.
- The ***Consumer API*** to subscribe to one or more topics and to process the stream of events produced to them.

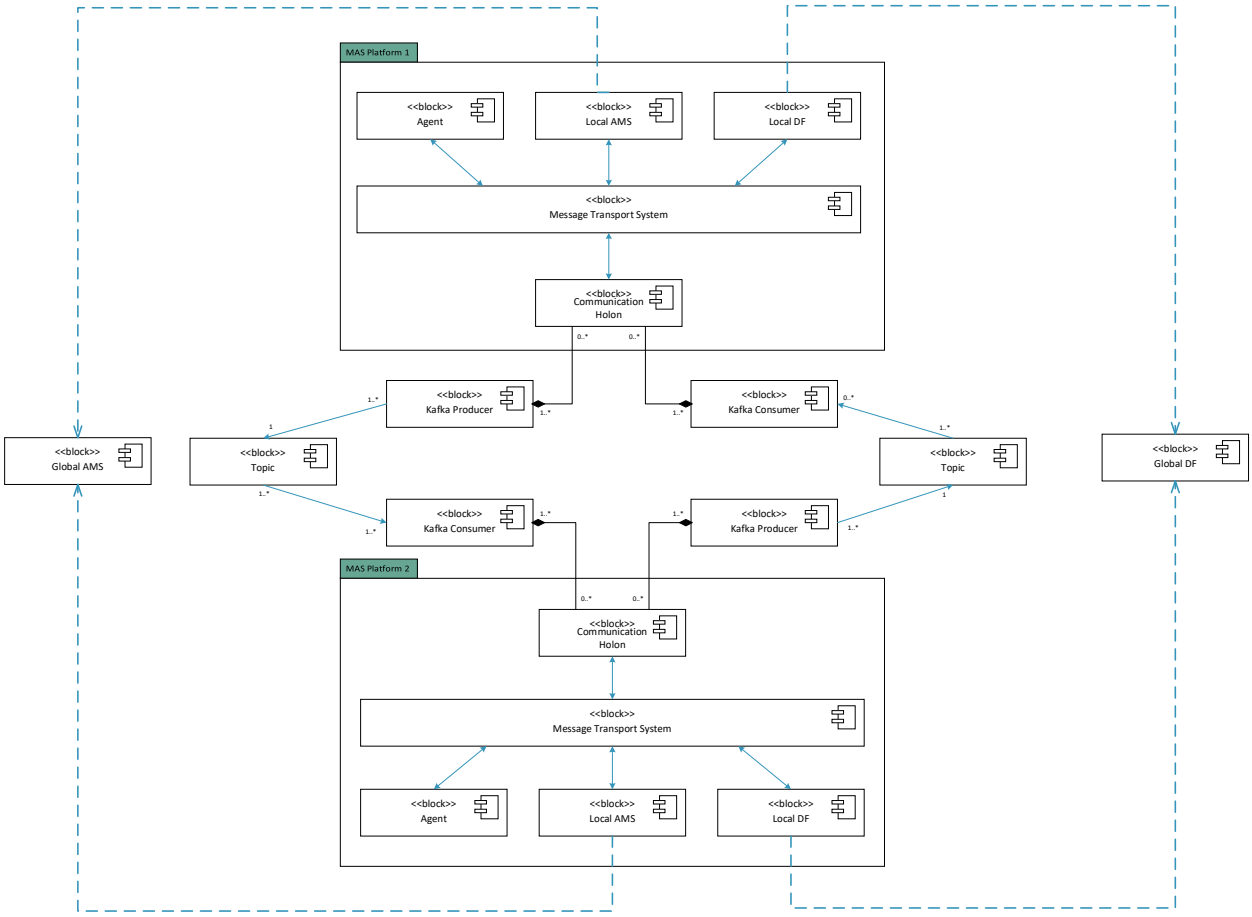

Figure 31: Two MAS runtimes communicates through Kafka

Next, we describe the minimal setup for the typical Kafka cluster.

For the setup of Kafka, we used an official quick start guide [22] from the Kafka website and [23].

Environment's setup.

1. Installing Java.

Before installing Kafka or ZooKeeper we need to install Java and get in running. Kafka and ZooKeeper work well with all OpenJDK-based Java implementations, including Oracle JDK. Though ZooKeeper and Kafka will work with a runtime edition of Java, it is recommended when developing tools and applications to have the full Java Development Kit (JDK).

2. Installing ZooKeeper.

Apache Kafka uses Apache ZooKeeper to store metadata about the Kafka cluster, as well as consumer client details. It is a centralized service for maintaining configuration information, naming, providing distributed synchronization and group services. The use of ZooKeeper in Kafka is shown on Figure 32.

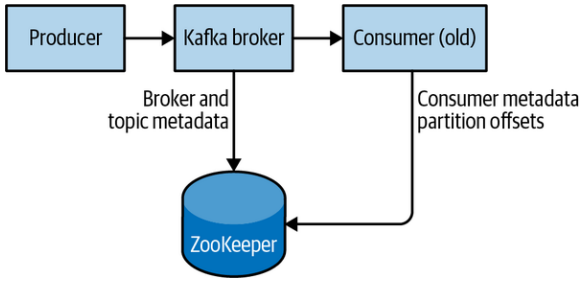

Figure 32: Kafka ZooKeeper, from [23]

3. Installing Kafka

After Java and Zookeeper are installed, Apache Kafka can also be installed. The current version can always be taken the Kafka website. The docker image can also be used.

Installing Kafka in Docker.

Probably the best way to install and setup the default Kafka is by using docker images and docker compose. For the demo purposes we used the docker images from https://hub.docker.com/u/aimvector.

Next, we show some snippets of the docker-compose file to install the ZooKeeper and the Kafka instances. The snippets are taken from the GitHub [24].

```
FROM openjdk:11.0.10-jre-buster

RUN apt-get update && \
    apt-get install -y curl

ENV KAFKA_VERSION 2.7.0
ENV SCALA_VERSION 2.13

RUN  mkdir /tmp/kafka && \
    curl "https://archive.apache.org/dist/kafka/${KAFKA_VERSION}/kafka_${SCALA_VERSION}-${KAFKA_VERSION}.tgz" \
    -o /tmp/kafka/kafka.tgz && \
    mkdir /kafka && cd /kafka && \
    tar -xvzf /tmp/kafka/kafka.tgz --strip 1

COPY start-kafka.sh  /usr/bin
RUN chmod +x  /usr/bin/start-kafka.sh

CMD ["start-kafka.sh"]
```

Figure 33: Dockerfile for installing Kafka

```
FROM openjdk:11.0.10-jre-buster

ENV KAFKA_VERSION 2.7.0
ENV SCALA_VERSION 2.13
RUN mkdir /tmp/kafka && \
    apt-get update && \
    apt-get install -y curl

RUN curl "https://archive.apache.org/dist/kafka/${KAFKA_VERSION}/kafka_${SCALA_VERSION}-${KAFKA_VERSION}.tgz" \
    -o /tmp/kafka/kafka.tgz && \
    mkdir /kafka && cd /kafka && \
    tar -xvzf /tmp/kafka/kafka.tgz --strip 1

COPY start-zookeeper.sh  /usr/bin
RUN chmod +x  /usr/bin/start-zookeeper.sh

CMD ["start-zookeeper.sh"]
```

Figure 34: Dockerfile for installing Zookeeper

1) Building Kafka from the dockerfile:

```
cd .\messaging\kafka\
docker build . -t aimvector/kafka:2.7.0
```

2) Building Zookeeper:

```
cd ./zookeeper
docker build . -t aimvector/zookeeper:2.7.0
cd ..
```

3) Starting Kafka:

```
docker run --rm --name kafka -it aimvector/kafka:2.7.0 bash
```

4) To tune some settings for the Kafka and the Zookeeper with copy the settings files out from the containers:

```
docker cp kafka:/kafka/config/server.properties ./server.properties
docker cp kafka:/kafka/config/zookeeper.properties ./zookeeper.properties
```

5) Create a Kafka network and run 1 zookeeper:

```
docker network create kafka
docker run -d `
--rm `
--name zookeeper-1 `
--net kafka `
-v ${PWD}/config/zookeeper-1/zookeeper.properties:/kafka/config/zookeeper.properties `
aimvector/zookeeper:2.7.0

docker logs zookeeper-1
```

6) Run Kafka:

```
docker run -d `
--rm `
--name kafka-1 `
--net kafka `
-v ${PWD}/config/kafka-1/server.properties:/kafka/config/server.properties `
aimvector/kafka:2.7.0

docker logs kafka-1
```

7) We can also start everything from one docker-compose file, which example is shown on Figure 35. Here we start one zookeeper, one Kafka server, one producer and one subscriber.

```yaml
version: "3.8"
services:
  zookeeper-1:
    container_name: zookeeper-1
    image: alsi/zookeeper:2.7.0
    build:
      context: ./zookeeper
    volumes:
    - ./config/zookeeper-1/zookeeper.properties:/kafka/config/zookeeper.properties
    - ./data/zookeeper-1/:/tmp/zookeeper/
    networks:
    - kafka
  kafka-1:
    container_name: kafka-1
    image: alsi/kafka:2.7.0
    build:
      context: .
    volumes:
    - ./config/kafka-1/server.properties:/kafka/config/server.properties
    - ./data/kafka-1/:/tmp/kafka-logs/
    networks:
    - kafka

  kafka-producer:
    container_name: kafka-producer
    image: alsi/kafka:2.7.0
    build:
      context: .
    working_dir: /kafka
    entrypoint: /bin/bash
    stdin_open: true
    tty: true
    networks:
    - kafka
  kafka-consumer:
    container_name: kafka-consumer
    image: alsi/kafka:2.7.0
    build:
      context: .
    working_dir: /kafka
    entrypoint: /bin/bash
    stdin_open: true
    tty: true
    networks:
    - kafka

networks:
  kafka:
    name: kafka
```

Figure 35: An example of the docker-compose file

5.2.1 Creating Kafka Producer

In this section we describe how to create a simple producer that writes to some Kafka topic. This producer will be a part of an agent. Figure 36 shows a high-level view on the Kafka components. To start publishing messages, we need to create a record with a Topic and a Value as the mandatory fields. The producer takes the record, serializes it, and sends to the network. There are also some handshaking procedures taking place at the background as it is shown on Figure 36.

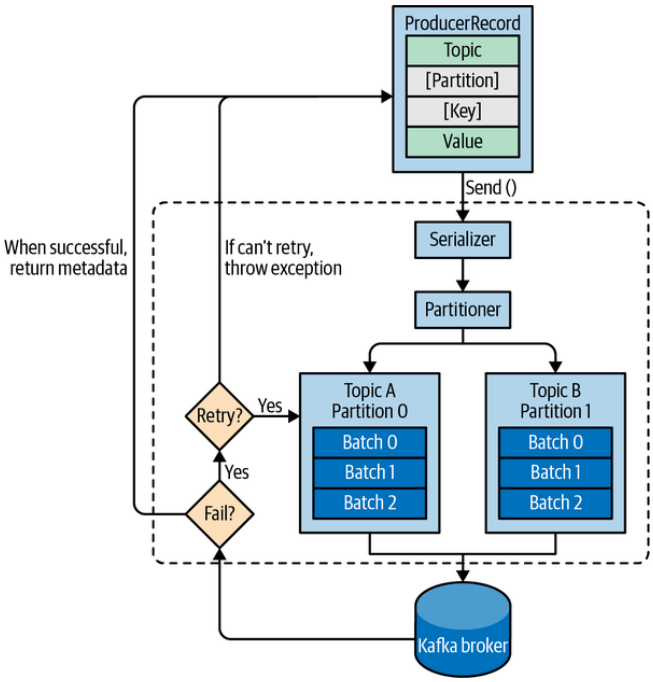

Figure 36: Kafka producer components, from [23]

The code snippet on Figure 37 shows the simplest producer with the default settings. First, we create a properties object. In this example we use strings for the messages, that is why we can use here a standard string serializer. In step 3 we create a producer with the proper key and value types and passing the properties object to it.

```
Properties kafkaProps = new Properties(); ❶
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
    "org.apache.kafka.common.serialization.StringSerializer"); ❷
kafkaProps.put("value.serializer",
    "org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps); ❸
```

Figure 37: Simple producer with the default settings

The next code snippet on Figure 38 shows how we can send a simplest message to Kafka. We start by creating a ProducerRecord object *agent_message* that is accepted by the producer. The record needs 3 parameters: the name of the topic where we send the message, key, and value.

```
ProducerRecord<String, String> agent_message =
    new ProducerRecord<>("Topic", "Key", "Value");
try {
    producer.send(agent_message)
}catch (Exception e) {
    e.printStackTrace();
}
```

Figure 38: Simplest way to send a message to Kafka

5.2.2 Serializing messages using Apache Avro

Apache Avro is a language-neutral date serialization format and can be used to create messages of custom types. Avro data is described in a language-independent schema that is usually defined as JSON document. Avro is especially interesting for the MAS4AI framework because it enables us to exchange messages in FIPA ACL or I4.0Language formats.

A simplified example of the Avro schema is shown on Figure 39. The official documentation can be found on the official website [25]. The complete schema of the I4.0Language standard is out of scope of this deliverable and will be developed during the project.

```
{"namespace": "i40Language.avro",
    "type": "record",
    "name": "AgentMessage",
    "fields": [
        {"name": "id", "type": "int"},
        {"name": "name",  "type": "string"},
        {"name": "value", "type": ["null", "string"], "default": "null"}
    ]
}
```

Figure 39: Simplified example of Avro schema

Avro requires the entire schema to be present when reading a record, so the common pattern is to use a Schema Registry, which is shown on Figure 40. The idea is to store all the schemas in the registry and add an identifier for a schema used to serialize the record in the record itself. The consumer can use the schema id to pull the required schema from the registry and deserialize the message.

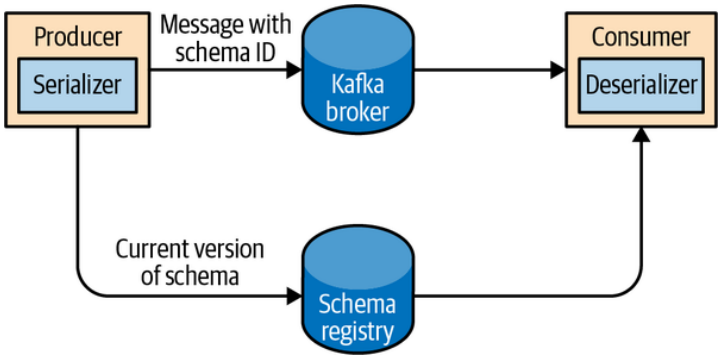

Figure 40: Serialization and deserialization of Avro messages, from [23]

5.2.3 Creating Kafka Consumer

Creating Kafka consumer is very similar to creating Kafka producer. First, we create a Properties-Object, set the required mandatory fields, and give it consumer constructor. Figure 41 shows a code snippet of simple Kafka consumer. To subscribe to a topic the subscribe method should be used as is showed on Figure 41.

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "AgentsMessages");
props.put("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer =
    new KafkaConsumer<String, String>(props);


consumer.subscribe(Collections.singletonList("Topic"));
```

Figure 41: Simple Kafka Consumer

6 Setup for Integration and deployment of Agent Types

6.1 Integration of Cyber-Physical Production Modules into Resource Agents

The integration of Cyber-Physical Production Modules within the MAS4AI framework requires the setup of the predefined framework elements like a MAS system and a middleware solution with represents the manufacturing environments with the AAS. Following the proposed method, that the assets provide a digital representation for integration in the MAS4AI framework, this approach has been evaluated within the SmartFactory demonstration testbed.

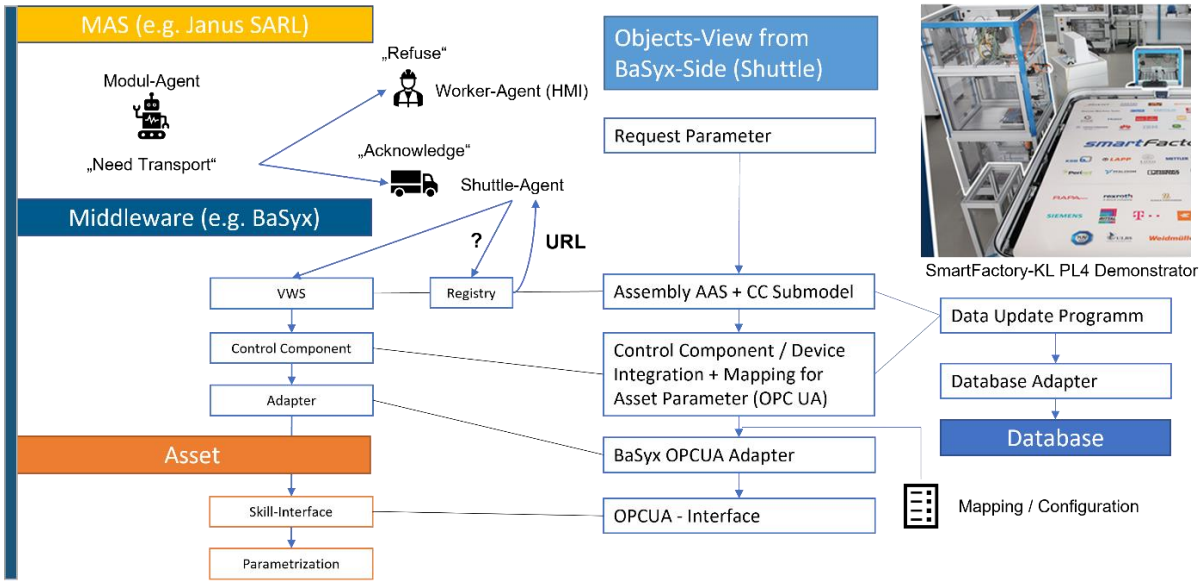

Figure 42: BaSyx Example Setup of Smart Factory testbed

The current state of implementation for CPPM follows the setup of the BaSyx-Middleware, to integrate the assets, which provides a OPC UA interface with skill-based information model, that encapsulates the CPPM capabilities in a predefined way. The integration is provided by using the OPC UA adapter in BaSyx, which is combined with the concept of the BaSyx Control Component. The BaSyx Control Component thus provides a predefined interface with the AAS Submodel for Control Component interaction which follows the specification [26].

The implementation thus must have a Mapping or Configuration File, to connect AAS submodel elements with respective native protocols like OPC UA. The data collection can be executed on the level of the control component by using a data update program and related database adapter.

Each CPPM of the Smart Factory testbed demonstrator thus provides an own AAS with Submodel, which is hosted on a respective AAS Server in BaSyx. The AAS is thus registered in the BaSyx Registry component, as presented in the BaSyx setup.

The related resource agents, which are considered as own MAS4AI agent type with subtypes for resource agents for manufacturing or transportation, can also discover and interact with the registered AAS of the CPPM setup. Therefore, an agent can request the registry to get the related endpoint.

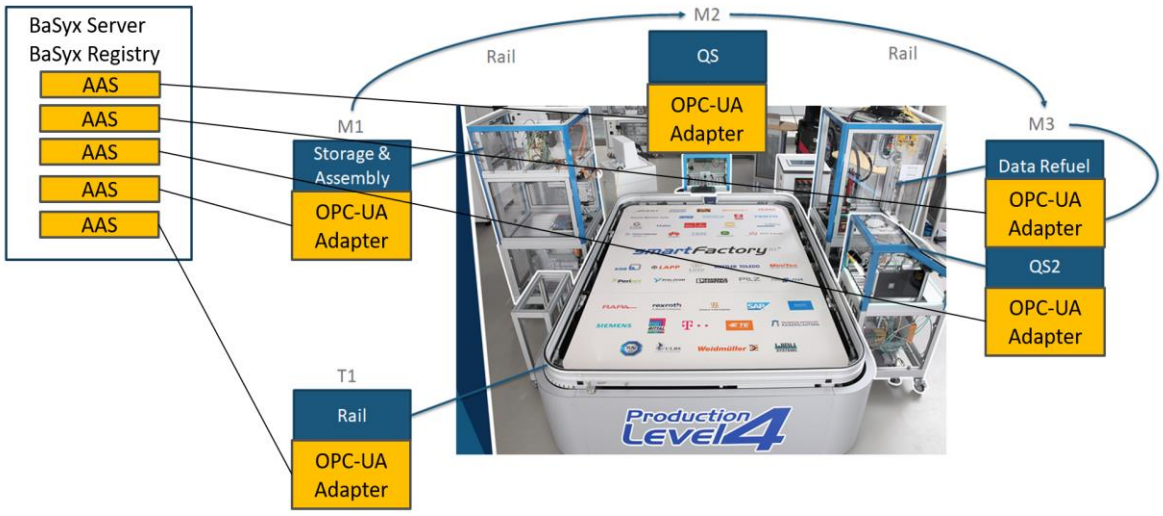

Figure 43: Setup and usage of BaSyx-AAS Server and Registry for CPPM

For the described example of the SmartFactory testbed environment, a resource agent contains a respective behaviour and, in terms of usage of the Janus MAS, implemented skills to interact with the AAS Submodels with the corresponding communication interface. If a more dynamic behaviour should be realized, it is also possible to use different parametrizations on side of the resource agents, which can be stored in a knowledge base or be optimized by using AI agents. The setup, that resource agents could select different parametrizations as input parameters of an AAS Submodel for the Control Component has been evaluated on basis of the transport shuttle agent of the demonstrator, which sets the destination dynamically to the AAS if the agent has confirmed a transportation request of a manufacturing resource agent, which represents an CPPM in the MAS setup.

6.2 Integration and deployment of AI Agents

We refer to the agent's function as an algorithm that maps the agent's perceptions to its actions. An agent's program is an internal implementation of the agent's function, which runs on a specific computational platform. Agent's function is an abstract mathematical description and can be a part of its AAS and agent's program is a piece of code that needs to be deployed. In this chapter, we are talking about the agents' programs and how they can be deployed and used by the agents.

The block diagram on Figure 44 shows the possible ways to integrate and deploy different agents and their programs.

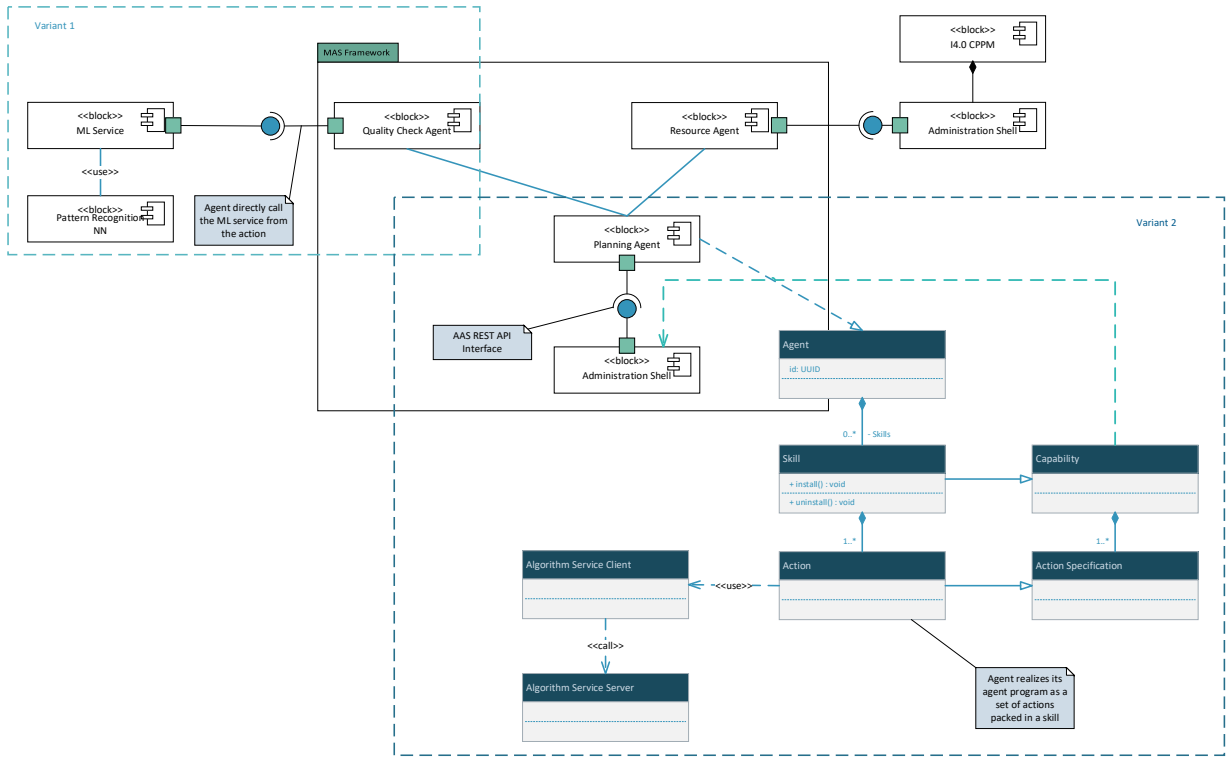

Figure 44 Integrating and deploying different agents and their programs

There are three main possibilities of how we can deploy and use the agents' programs:

1. The agent's program is implemented and deployed externally on some computational platform and the agents uses its functionality through a set of services.
2. The agent's program is implemented using MAS framework abstractions and language, in our case SARL.
3. The mix of the previous two variants.

6.3 Agent's Program as an external Service

This variant is presented on the diagram as variant 1. As an example, we have a pattern recognition algorithm, for example a ConvNet that is running on some TPU. This program provides a web-service that can be used via the REST API. A Quality Check agent in the MAS in its actions makes direct calls to that service to query the algorithm. This variant has an advantage that it completely separates the agent's program development and deployment from the actual agent in the MAS. The agent's program implementation can be freely changed, also in runtime, if the interface stays the same. It is important to hold the interface fixed, so we don't need to change the agent's action that calls this service.

The use of the skill-engineering principals can help here. Following skill-based approach we abstract some functionality as a skill with the standardised interface and clear usage semantics. In that case the external ML service, which is provided by the pattern recognition system, is implemented as a skill with the standard skill model and the agent always knows how to use it. This enables us to lower the integration efforts.

6.4 Agent's Program as an internal Agent's skill

This variant is presented on the diagram as variant 2. Here we exemplary show a Planning Agent that realises its algorithm as a set of Actions that are composed to a Skill. In that case we need to follow the SARL skill and capacity metamodel, as presented in the MAS system element setup.

Each agent in SARL has a set of Capacities that define what an agent can do. From the SARL specification "a Capacity is the specification of a collection of Actions. Consequently, only Action signatures can be defined inside a Capacity: no attribute or field is allowed, and nobody (code) for the Action may be present". Skill is the actual implementation of a Capacity and comprises of a set of actions. The actions can be executed as the reaction to the external and internal event, which are received by the agent. The combination of actions and events that trigger these actions builds the actual agent's program.

The SARL model of Skills and Capacities are very similar to the model of Skills and Capabilities from the skill-based engineering domain. We can easily combine them and use together. Skills and Capacities enable code reuse and modularity. If we pack the agent's program as the Skill that we can easily change it to another Skill if they both implement the same Capacity. This will be used in the MAS4AI project to share our Agent's Programs as Skills and hold them in Skills repositories.

7 Setup of Knowledge-Based Interactions inside MAS4AI framework

The setup of the integration of the AAS as predefined interface, the search for registered AAS of agents and assets as well as the usage of the Knowledge Base is important for the general system communication and requires the setup of the previous presented MAS4AI framework elements like Janus and BaSyx. The requests for the RDF-Store and the interactions with the AAS Servers as well as the Registry component of BaSyx are implemented in Janus with predefined skill templates, similar as they are used to communicate with the AAS Submodel of CPPM. The RDF-Store, which is provided by the Dydra solution [27], also enables the communication by using HTTP REST. The modelled integration is presented in Figure 45.

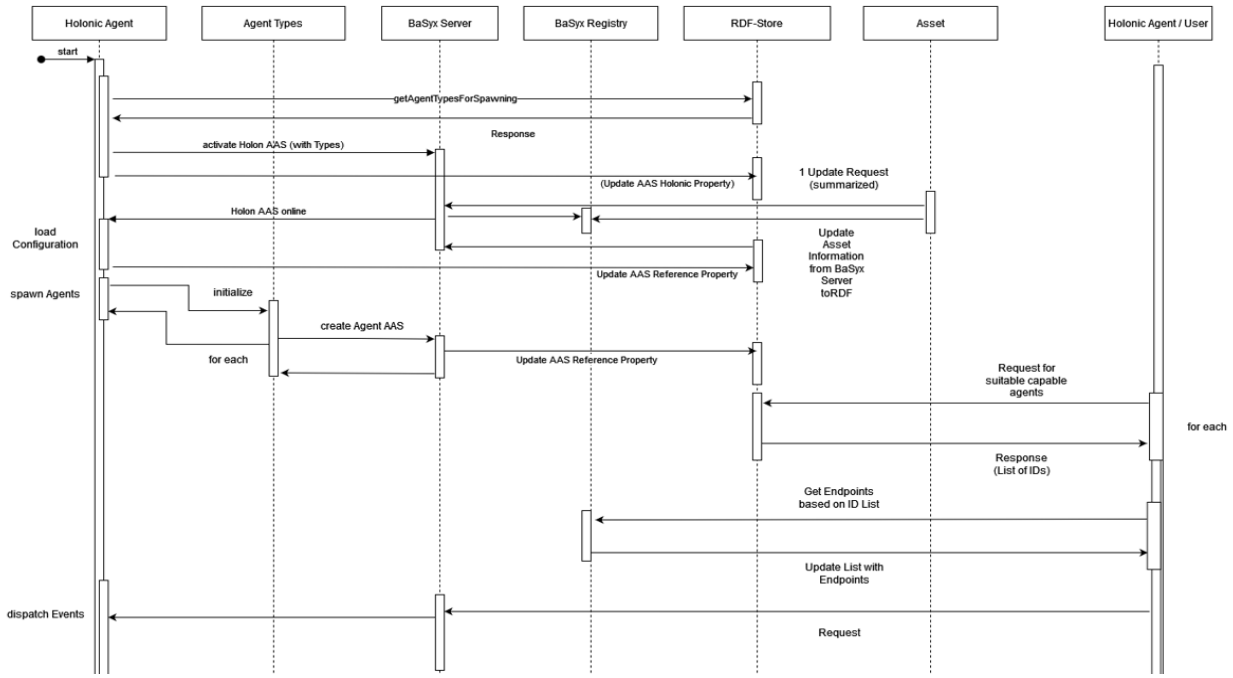

Figure 45: MAS interaction model with framework elements during start of an holonic agent [21]

At first the holonic agent starts and requests, by using a related Janus skill with a preconfigured HTTP-Address and request which should also be setup during the skill template, the RDF-Store as knowledge base. The result is a list from the RDF-Store, which agents should be used during the initial start behaviour. The holonic agent then starts its AAS by using the AAS

[21] Sequence-Diagram (Draw-IO Model from Appendix – Part 1)

Server of BaSyx and the predefined AAS Model of the holonic agent, for example if it's an holonic resource agent, then the AAS Model for Resource Agents is used. Related to the start of the agents AAS, the AAS is registered, and the RDF Store will be updated with the property, that the related AAS is holonic.

The same mechanism for built up an AAS and registration is done by assets like CPPM, so that if an asset is available, the AAS Server will be started in the BaSyx-Middleware and then registered, and the RDF Store is updated with the related AAS Server information. After the AAS of the holonic agent has been loaded, the configuration of the holonic agent is loaded and the reference of the AAS is updated also in the RDF-Store, so that the holonic agent could be found by other (holonic) agents in the MAS4AI framework. For each agent which will be spawned on basis of the loaded configuration, an own AAS that is related to the agent type will be started in the BaSyx-Middleware and the information of the AAS reference is then updated in the RDF Store. The registration into the Registry component of the BaSyx-Middleware is necessary for each spawned agent type inside of an holonic agent, if at least one agent is deployed in another environment or MAS framework.

If another, already active, (holonic) agent in the MAS4AI framework searches for the capabilities of the already spawned agents, they can be found by requesting the RDF-Store, with response of the related unique IDs of suitable agents AAS. The requesting agent then can get the endpoint information of the hosted agent AAS from the BaSyx registry, by using the list of IDs from the RDF-Store, to publish the request towards their related AAS.

The holonic agent thus gets events from the AAS, which should be dispatched and executed inside the related MAS of the holonic agent, for example a (distributed) setup in Janus. The general procedure is visualized in Figure 46.

If an asset is required for execution of the request, for example if the holonic agent is a resource agent which interacts with a CPPM, or an AI agent which interacts with a planning or machine learning algorithm, the holonic agent requests the RDF-Store to get the AAS IDs of suitable assets. The holonic agent then deploys the request to the already spawned agents inside of the holon with a related planning. The plan will then be scheduled with the required agents and the agents then gets the corresponding endpoint information from the Registry component of the BaSyx-Middleware for the required assets. If additional agents are necessary to fulfil a request, then these agents could be spawned as described in the initial start sequence of the holonic agent, leading to another agent setup.

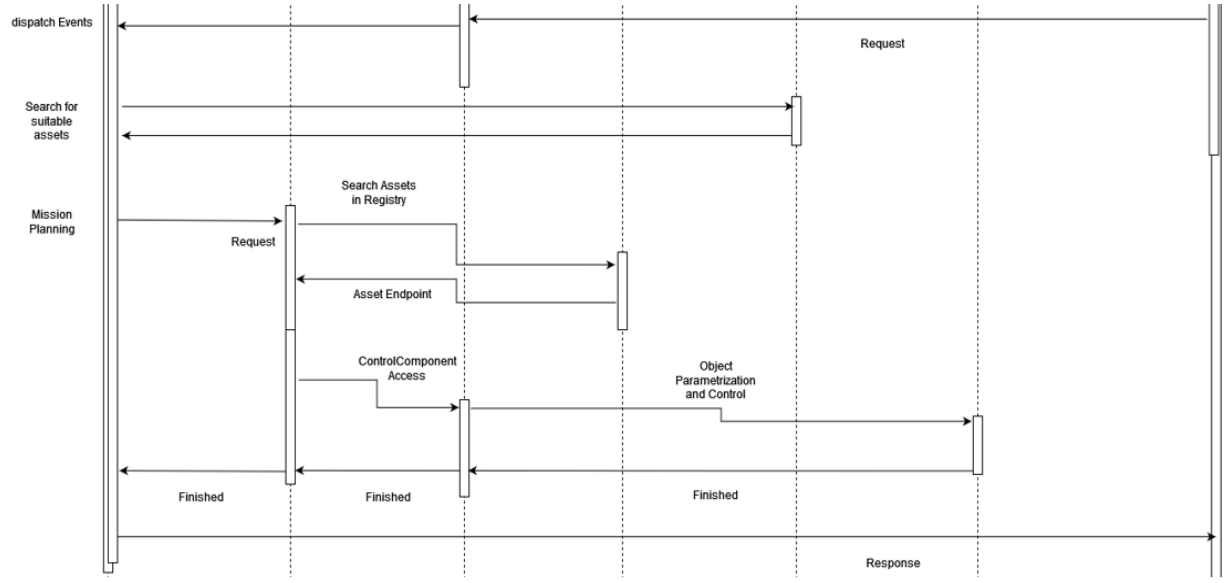

Figure 46: MAS interaction model with framework elements for agent request event processing [22]

In case of an CPPM, the resource agent will connect with a Janus skill towards the AAS of the asset, which is representing a Control Component and can use information for parametrization of this request. Furthermore, if the parametrization is part of stored information, for example in a related knowledge base, it is possible that an agent can send a request to the RDF-Store to get the necessary information. If the request towards an asset has been successfully communicated by using its AAS and the related action has been processed, then the response will be considered by the related agent. If all agents of the holonic agent have processed their related part of the scheduled plan successfully, the holonic agent communicates a response to the agent which have started the request.

In case of failure during the execution of a previous setup planning, for example if an error occurs on the asset side, an holonic agent can try initiate an alternative behaviour. Therefore, an agent type which is responsible for deviations, must be available and feasible to react to the occurred failure. If no possibility can be found to solve the problem inside the holonic setup, the requesting agent must be informed.

[22] Sequence-Diagram (Draw-IO Model from Appendix – Part 2)

8 Conclusion

The setup of the MAS4AI framework, which is based on several elements, has been described in the scope of this document with a general approach. The setup of selected framework elements has been presented with Janus as MAS and the message-based middleware solutions of BaSyx and Apache Kafka. The virtual representation and interface description of the manufacturing environment with AAS is important for the agent types inside the MAS and their interface setup, to enable an interaction between resource agents and Cyber-Physical Production Modules and to implement Industrial AI software agents. The setup of Kafka thus enabled the integration of message channels for distributed agents.

The setup to provide an AAS as self-description and interface for assets, with related components and services for registration and discovery, thus could also be used and validated as approach for the setup of an Agent Service Description, so that the components of the Data / AAS layer of the MAS4AI framework can be used as same basis. The interaction of MAS4AI framework elements during an initial start sequence of an holonic agent as well as the interactions and communication for executing a request has been modelled, as well as the importance of a knowledge base, for example in form of an RDF-Store. Due to the possibility to integrate semantic references inside an AAS, it is possible to connect to an RDF-Store or to send requests to the knowledge base. With the RDF-Store, it is also possible to search for suitable agents and assets, due to the functionality of the Registry component, which focuses more of the endpoint information of all registered AAS.

The encapsulation of agent interfaces to communicate with the AAS, the RDF-Store, with Kafka APIs or in general with available microservices could be setup and modelled by using the skill concept of the Janus MAS. The modelling and implementation of this concept allows the setup and integration of exchangeable communication patterns, which can be used inside an agent's behaviour, and allows in general to integrate already separated functionalities without the need to change the implemented agents or their behaviour. The concepts thus allow an integration of different technological implementations, if they can implement the described and modelled aspects of the framework, using the concepts of the AAS as main interface.

In further deployments of the presented setup, the described MAS4AI framework elements can be provided as own Docker containers. The distributed setup of the framework can be seen on the framework level, so that MAS4AI framework elements are deployed in own docker containers on cloud- or edge-level, but also in the possibility to deploy the framework elements itself also in a distributed way. Examples for this can be seen, if many different MAS with agents are deployed for a factory environment, or AAS Server and Registry or the knowledge base is distributed.

To realize a setup such a distributed realization of the MAS4AI framework, the importance of interface descriptions with the AAS and the usage of Apache Kafka as communication channel can be seen. Furthermore, usage and evaluation of the skill concept for agents, that encapsulates the interaction with other system elements, will be considered on the proposed MAS4AI setup. This concept also includes the possibility to integrate further connectors as microservices, for example for cloud level AI applications and interface concepts of distributed production, like IDS connectors.

9 Literature

[1] Janus Agent and Holonic Platform (2022). Available: http://www.sarl.io/runtime/janus/, Accessed: 2022-04-13.

[2] Download Options and Installation Instructions (2022). Available: http://www.sarl.io/download/index.html, Accessed: 2022-04-13.

[3] OpenJDK (2022). Available: https://adoptium.net/?variant=openjdk8&jvmVariant=hotspot, Accessed: 2022-04-13.

[4] Janus Command-line Launcher (2022). Available: http://www.sarl.io//docs/official/tools/Janus.html, Accessed: 2022-04-13.

[5] Maven Plugin for the SARL Compiler (2022). Available: http://www.sarl.io//docs/official/tools/MavenSarlPlugin.html, Accessed: 2022-04-13.

[6] Create your First Project (2022). Available: http://www.sarl.io//docs/official/gettingstarted/CreateFirstProject.html, Accessed: 2022-04-13.

[7] Create a SARL Launch Configuration (2022). Available: http://www.sarl.io//docs/official/gettingstarted/RunSARLAgentEclipse.html#1-create-a-sarl-launch-configuration, Accessed: 2022-04-13.

[8] Rodriguez, Sebastian, Nicolas Gaud, and Stéphane Galland. "SARL: a general-purpose agent-oriented programming language." 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT). Vol. 3. IEEE, 2014.

[9] SARL Language Concepts (2022). Available: https://en.wikipedia.org/wiki/SARL_language#/media/File:SARLLanguageconcepts.png, Accessed: 2022-04-13.

[10] BaSyx explained in 10 minutes (2022). Available: https://wiki.eclipse.org/BaSyx_/_WhatIsBasyx, Accessed: 2022-04-13.

[11] BaSyx Concepts (2022). Available: https://wiki.eclipse.org/BaSyx_/_Concepts, Accessed: 2022-04-13.

[12] Downloading BaSyx (2022). Available: https://wiki.eclipse.org/BaSyx_/_Download, Accessed: 2022-04-13.

[13] How to build the BaSyx Java SDK (2022). Available: https://wiki.eclipse.org/BaSyx_/_Download_/_Java_Setup, Accessed: 2022-04-13.

[14] AAS Server Component (2022). Available: https://wiki.eclipse.org/BaSyx_/_Documentation_/_Components_/_AAS_Server, Accessed: 2022-04-13.

[15] Registry Component (2022). Available: https://wiki.eclipse.org/BaSyx_/_Documentation_/_Components_/_Registry, Accessed: 2022-04-13.

[16] Components - Updater Component (2022). Available: https://wiki.eclipse.org/BaSyx_/_Documentation_/_Components#Updater_Component, Accessed: 2022-04-13.

[17] AASX Package Explorer (2022). Available: https://github.com/admin-shell-io/aasx-package-explorer, Accessed: 2022-04-13.

[18] CanvAAS – bridging a fragmented industry landscape (2022). Available: https://www.eitmanufacturing.eu/what-we-do/eit-manufacturing-case-studies/case-study-canvaas-bridging-a-fragmented-industry-landscape/, Accessed: 2022-04-13.

[19] Components - SQL (2022). Available: https://wiki.eclipse.org/BaSyx_/_Documentation_/_Components_/_SQL, Accessed: 2022-04-13.

[20] Kafka Communication Provider (2022). Available: https://github.com/BaSys-PC1/platform/blob/master/runtime/de.dfki.cos.basys.runtime.communication.provider.kafka/src/main/java/de/dfki/cos/basys/platform/runtime/communication/provider/KafkaCommunicationProvider.java, Accessed: 2022-04-13.

[21] Exploring the Apache Kafka “Castle” Part A: Architecture and Semantics (2022). Available: https://www.instaclustr.com/blog/exploring-apache-kafka-castle-architecture-semantics/, Accessed: 2022-04-13.

[22] Kafka QuickStart (2022). Available: https://kafka.apache.org/documentation/#quickstart, Accessed: 2022-04-13.

[23] Shapira, Gwen, et al. Kafka: the definitive guide. " O'Reilly Media, Inc.", 2021.

[24] Docker Development Kafka Readme (2022). Available: https://github.com/marcel-dempers/docker-development-youtube-series/blob/master/messaging/kafka/README.md, Accessed: 2022-04-13.

[25] Welcome to Apache Avro! (2022). Available: https://avro.apache.org/, Accessed: 2022-04-13.

[26] BaSys 4.0 control and group components (2022). Available: https://wiki.eclipse.org/BaSyx_/_Documentation_/_ControlComponent, Accessed: 2022-04-13.

[27] Getting started – What is Dydra? (2022). Available: http://docs.dydra.com/dydra, Accessed: 2022-04-13.

10 Appendix

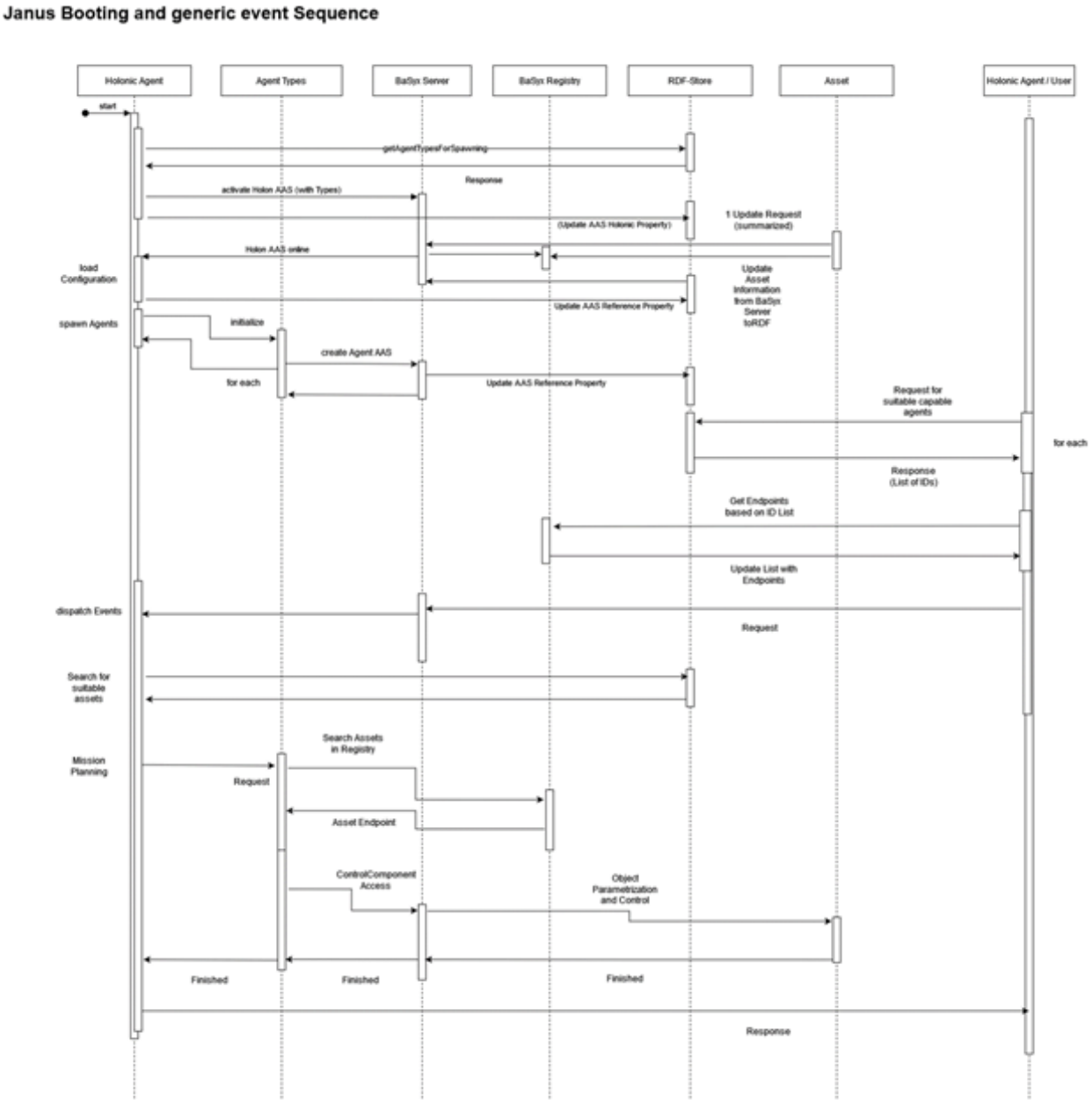

Figure 47: MAS4AI Elements interaction diagram [23]

[23] Sequence-Diagram (Draw-IO Model)