A MQTT-based solution as an enabler of interoperability between MAS frameworks to manage a Self-organized Conveyor System

Bruno Patto Graciano Natal

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Electrical and Computer Engineering.

Work oriented by:

Prof. Dr. Paulo Jorge Pinto Leitão

Prof. Dr. Leandro Resende Mattioli

Bragança

2024

A MQTT-based solution as an enabler of interoperability between MAS frameworks to manage a Self-organized Conveyor System

Bruno Patto Graciano Natal

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Electrical and Computer Engineering.

Work oriented by:

Prof. Dr. Paulo Jorge Pinto Leitão

Prof. Dr. Leandro Resende Mattioli

Bragança

2024

Dedication

I would like to dedicate this work to my family, who have helped and supported me on this journey.

Acknowledgment

Here I would like to thank everyone who took part and helped me succeed at this stage of my life.

First of all, I would like to thank my supervisor at the Polytechnic Institute of Bragança (IPB), Professor Dr. Paulo Leitão, and my co-supervisor at the Federal Center for Technological Education of Minas Gerais (CEFET-MG), Professor Dr. Leandro Mattioli, for all the knowledge shared, all the advice and for always being available to discuss the various issues that arose during the development of this project.

I would also like to thank the great friends I made in the laboratory, Gustavo Funchal and Victória Melo, for the knowledge shared and for always being available to assist and guide me in an assertive way towards the path I needed to follow to achieve the expected results. I leave here my big hug, hoping that you achieve everything you aim for in life. You deserve it.

How can I not thank the family I created here in Bragança? Here's a special thank you to all the friends and brothers I gained in the Bairro Nobre republic: Melo (the Tupperware inspector), Edilson (why don't you like me?), Vini (my cart was made of clothespins, it had nothing to do with yours), Léo (the best roommate anyone could have), Guigaz (the best Masterchef), Arthur (you have to eat better) and Júlia(s). Thank you for all the company and all the happy moments we shared. You'll always be in my heart.

I want to thank my father, Bruno Natal, for I know that from where you are, you have always watched over and prayed for your family. To you, Dad, I am grateful for the integrity you've instilled in me and for always setting the example of how a person of

character should behave in society. Thank you for all the love, care, dedication, for the examples, and for the wonderful childhood you provided me. I know that wherever I go, you will always be by my side and in my heart.

I want to thank my mother, Gininha, for all the support, care, dedication, and love you have given me throughout my life. I know how hard it was for you to raise your two children, and I am aware of all the dreams and wishes you sacrificed to provide us with a better future. I thank you from the bottom of my heart for the education and the values you have instilled in me; without your support, none of this would make sense. Thank you, mom, I love you immensely.

Here is my thanks to one of the most important women in my life, my sister Julianna. To you, Jurubeba, I am grateful for all the support, brotherhood, care, companionship, and for being an example to follow. Without your support, I could not have finished this project. Thank you for being the best sister in the world, I love you very much.

Finally, I would like to thank the woman of my life, my fiancée Táisa Leite (Craudia). It's impossible to express my gratitude in words only, because you have been the fundamental pillar that has transformed me into the person I am today. Thank you for sharing each of your dreams and for being part of all of mine, always supporting and encouraging me to pursue and fight for my goals. Thank you for always believing in me, supporting me in difficult times and for the long phone calls that calmed me down and gave me the strength to continue with the project. You were, are and always will be the main reason why I strive to improve, being my eternal inspiration. We were two dreamers when we took our walks in Barreiro, always talking and envisioning a future together, and with each trip, each conversation, and our dedicated commitment, we made our dreams come true. Without you, I wouldn't be at this stage in my life, thank you for putting up with the periods when we were apart. I will love you today, tomorrow, and forever.

Abstract

In the era of Industry 4.0, the interconnection of devices through the Internet of Things (IoT) elevates Cyber Physical Systems (CPS) to a central position, i.e. integrated systems of computational algorithms and physical components, whose efficient operation can be significantly improved by the implementation of Multi-Agent Systems (MAS), which facilitate the integration of intelligence in a distributed and decentralized manner. However, the diversity of frameworks available on the market to implement the logic present in MAS presents a difficulty in terms of interoperability between these frameworks, since each one has its own characteristics, with its own protocols, languages and structures. Thus, this study aims to implement an interoperability mechanism that combines a middleware with dedicated interface agents to enable the seamless communication between different MAS frameworks. The middleware plays a fundamental role by routing the exchanged messages to the appropriate interface agents, guaranteeing not only the interoperability, security and authentication of the interconnected components, but also the integrity of the message delivery process. The interface agents, in turn, are tasked with the essential functions of translating incoming messages into their respective frameworks and managing the publication of messages sent back to the middleware. The proposed approach was validated in a case study consisting of a cyber-physical conveyor system, where the main characteristic is the flexibility of implementing self-organizing logics to adapt to a changing environment. Based on the results obtained, it was possible to conclude that, for the case study applied, the adopted approach allowed the system to operate effectively with different frameworks, achieving the expected interoperability between MAS based systems.

Keywords: Multi-agent Systems, Interoperability, Middleware, Cyber-Physical System.

Resumo

Na era da Indústria 4.0, a interconexão de dispositivos por meio da Internet das Coisas eleva os Sistemas ciber-físicos a uma posição central, ou seja, sistemas integrados de algoritmos computacionais e componentes físicos, cuja operação eficiente pode ser significativamente aprimorada pela implementação de Sistemas Multi-Agentes (MAS), os quais facilitam a integração de inteligência de maneira distribuída e descentralizada. Entretanto, a diversidade de frameworks disponíveis no mercado para implementar a lógica presente no MAS apresenta uma dificuldade em termos de interoperabilidade entre esses frameworks, uma vez que cada um possui características próprias, com seus próprios protocolos, linguagens e estruturas. Assim, este estudo tem como objetivo implementar um mecanismo de interoperabilidade que combina um middleware robusto com agentes de interface dedicados para permitir a comunicação contínua entre diferentes estruturas de MAS. O middleware é fundamental para esse sistema, que encaminha as mensagens para os agentes de interface apropriados, garantindo não apenas a segurança e a autenticação dos componentes interconectados, mas também a integridade do processo de entrega de mensagens. Os agentes de interface, por sua vez, são encarregados das funções essenciais de traduzir e enviar as mensagens recebidas para suas respectivas frameworks e gerenciar a publicação das mensagens enviadas de volta ao middleware. Para isso, a abordagem foi validada em um estudo de caso que consiste em um sistema de transporte ciberfísico, cuja principal característica é a flexibilidade de implementar lógicas de auto-organização para se adaptar a um ambiente em constante mudança. Com base nos resultados obtidos, foi possível concluir que, para o estudo de caso aplicado, a abordagem adotada permitiu que o sistema operasse efetivamente com diferentes estruturas, alcançando a interoperabilidade entre sistemas baseados em MAS.

Palavras-chave: Sistemas Multi-Agentes; Interoperabilidade; Middleware; Sistema Ciber-Físico.

Contents

List of Tables

List of Figures

Acronyms

ACC Agent Communication Channel.

AI Artificial intelligence.

AID Agent Identifier.

AMS Agent Management System.

CA Certificate Authority.

CPS Cyber Physical Systems.

DF Directory Facilitator.

DNS Domain Name System.

FIPA Foundation for Intelligent Physical Agents.

FIPA-ACL Foundation for Intelligent Physical Agents - Agent Communication Language.

IDE Integrated Development Environment.

IIoT Industrial Internet of things.

IoT Internet of Things.

JID JabberID.

JRE Java Runtime Environment.

JSON Java Script Object Notation.

M2M Machine to Machine.

MAS Multi-Agent Systems.

ML Machine Learning.

MQTT Message Queuing Telemetry Transport.

PLC Programmable Logic Controller.

QoS Quality of Service.

SPADE Smart Python Agent Development Environment.

SSL Secure Sockets Layer.

TLS Transport Layer Security.

XML Extensible Markup Language.

XMPP eXtensible Messaging and Presence Protocol.

List of Algorithms

Chapter 1

Introduction

This work is framed in the context of Industry 4.0, focusing on the concepts of "Interoperability", "Multi-agent System" and "Cyber-Physical Systems". These pivotal technological areas are at the forefront of fostering a transformative blend between the physical and computational realms. The shift towards distributed intelligence and system decentralization is precipitating notable enhancements in diverse sectors, encompassing manufacturing, energy, healthcare, and building automation, among others.

1.1 Motivation

The fourth industrial revolution, known as Industry 4.0, heralds a paradigm shift in the design and operation of production systems. In this new era, there is a pivotal integration of smart manufacturing systems with cutting-edge technologies such as Machine Learning (ML), the Internet of Things (IoT), cloud computing, and automated robotics. These advancements are revolutionizing industrial service capabilities, enabling the production of high-quality goods [1]. According to [2], intelligent manufacturing is the principal sector for expansion within the Industrial Internet of things (IIoT) market, with projections indicating a leap from 17.7 billion IIoT connections in 2020 to 36.8 billion by 2025, underscoring the scale and pace of this technological evolution.

In this scenario, the Cyber-Physical Systems (CPS) [3] can be considered the backbone

platform to implement the principles of Industry 4.0 since they are enabling the next generation of production systems. These systems are characterized by their ability to integrate a diverse array of processes, systems, and devices in a large-scale, networked manner. CPS combine cyber and physical elements possibiliting to collect accurate and reliable data from the physical world and process it at the Cyber level [4]. This leads to improvements in the performance, responsiveness, and reconfigurability of production systems. Such integration is pivotal in managing the inherent complexity and uncertainty of modern manufacturing environments. By integrating heterogeneous, interconnected, and intelligent components, CPS facilitate a symbiotic interaction between computational and physical elements [5].

In industries replete with CPS performing various roles within the process chain, Multi-Agent Systems (MAS) [6] have emerged as a viable solution to manage the complexities inherent in such environments. The MAS approach is well-suited for the realization of extensive and intricate industrial CPS due to its core attributes of modularity, autonomy, de-centralization, and the capability to self-adapt to emergent situations without the need for external intervention. MAS specifically decentralizes control functions across a network of distributed and intelligent software agents that operate on cloud and edge computing layers. These agents work in union to achieve collective system objectives, dynamically adjusting to changes and reconfiguration demands inherent in the industry's evolving landscape [5].

In this context, with many heterogeneous and autonomous processes exchanging data across the network, it is crucial to ensure interoperability among different systems [7]. The demand for interoperability extends from the operational intricacies of the factory floor to the strategic decision-making realms of enterprise and business systems [8]. Therefore, to guarantee the interoperability of different applications, it is necessary to create standards to specify how the information is exchanged, processed, and communicated among things [9]. As pointed out by [10], standardization plays a crucial role in Industry 4.0. It provides a universal language, ensuring that different devices and systems can communicate effectively, using a shared vocabulary, syntax, semantics, and protocols.

This standardization will significantly increase the interoperability, in both vertical and horizontal terms, among various devices. However, despite these standardization efforts, achieving interoperability between MAS different frameworks remains a challenge.

Having this in mind, this work aims to study and develop a mechanism that allows interoperability between MAS frameworks. To this end, a cyber-physical modular transfer system was used as a case study, in which the main characteristic is the flexibility to implement self-organizing logics to adapt to a changing environment. In this way, the aim is to implement and guarantee that different MAS-based solutions developed using different frameworks can exchange information to control and present self-organizing characteristics on the system's modules.

1.2 Objectives

The main objective of this work is to develop a mechanism that allows interoperability between different MAS frameworks. To achieve this primary objective, secondary objectives have been established:

- In-depth study and analysis of commonly used MAS frameworks;
- Examination of the existing modular and self-organized transport system for automatic conveyors FischerTechniks, operating under MAS;
- Implementation of a self-organizing solution within both frameworks;
- Design and development of the middleware's architecture to the system under study;
- Incorporation of security techniques in the middleware to safeguard the experimental case study;
- Development of a dashboard for real-time visualization and monitoring of the system;
- Analysis of the middleware performance.

1.3 Document Structure

This document is organized in 6 chapters, beginning with the present chapter where the proposal of the work, contextualization and the objectives were presented.

Chapter 2, entitled “Theoretical Background”, provides a comprehensive review of the fundamental concepts necessary for the development and understanding of the study.

Chapter 3, entitled “Middleware for Implementing MAS Frameworks”, describes a comprehensive overview of the middleware architecture developed to facilitate interoperability between the MAS frameworks.

Chapter 4, entitled "Development of the MAS Applications for the Case Study", discusses the practical implementation of self-organized behavior in the system using the JADE and SPADE frameworks. Therefore, that chapter presents the case study and the techniques employed to implement self-organization using the different frameworks.

Chapter 5, titled “Integration of the MAS Applications using the Proposed Middleware Approach”, outlines the design and functionality of the developed middleware, focusing on its role in facilitating interoperability across different MAS frameworks. It also introduces a graphical interface developed for system monitoring and visualization, enhancing user interaction and understanding of the system’s operations. Additionally, the chapter highlights the implementation of security measures within the middleware to protect the integrity and confidentiality of the message exchanges, thereby ensuring the system’s secure and efficient operation. Finally, the chapter presents the tests carried out on the middleware to analyze its performance.

The last chapter, entitled “Conclusions and Future Work”, summarizes the work with the conclusions and points out future work.

Chapter 2

Theoretical Background

This chapter presents a comprehensive literature review in the domain of intelligent and distributed systems, with a particular focus on CPS supported by MAS. It begins by elucidating the fundamental concepts of CPS, MAS, and Middleware, laying the groundwork for understanding their interaction and significance. Subsequently, the chapter navigates through various fundamental communication protocols in this context, offering a concise overview of their roles and functionalities. Finally, attention then turns to addressing security concerns within these systems, highlighting both challenges and strategies for ensuring secure operations.

2.1 Industry 4.0 and Cyber-Physical System

The fourth industrial revolution, also known as Industry 4.0, marks a significant transformation in the design and operation of production systems, signaling a new era in manufacturing. As delineated by [11], each industrial revolution brought with it a transformation of the manufacturing process. These transformations have not only reshaped the industry but have also had profound impacts on various other sectors. The main points of these transformative shifts are elucidated by [12] as (see Fig. 2.1):

- The First Industrial Revolution introduced mechanization and steam power;

- The Second Industrial Revolution brought mass production and electrical energy, with assembly lines enhancing efficiency and scale;
- The Third Industrial Revolution ushered in automation, computers, and robotics, significantly advancing production capabilities and enabling the rise of digital communication.

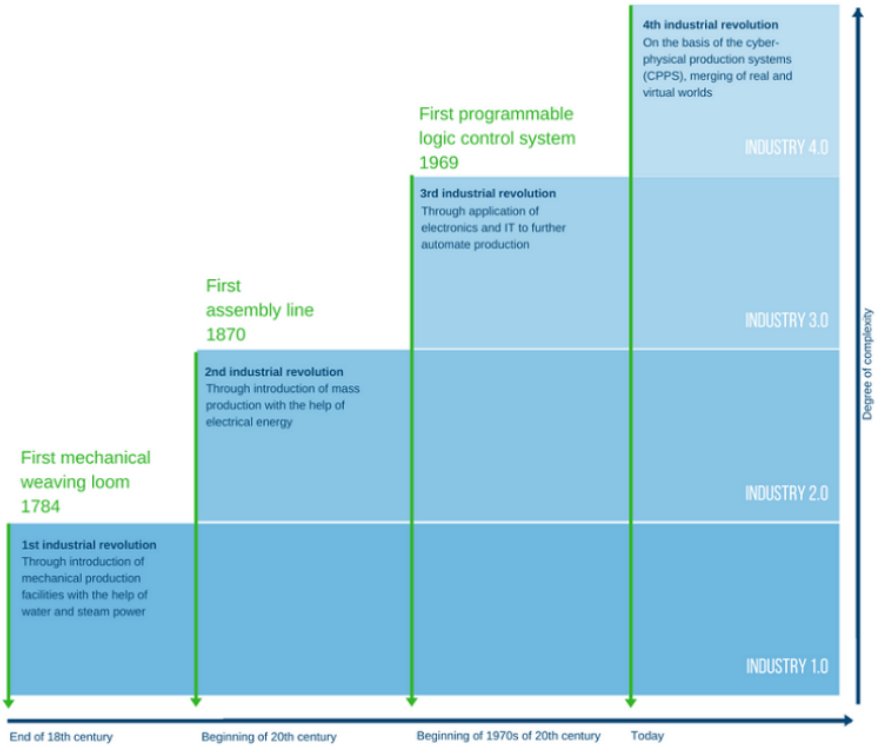

Figure 2.1: Evolution of industry [11].

Finally, the manufacturing system reaches the Fourth Industrial Revolution, which is distinguished by the integration of diverse devices and the deployment of robust technologies such as Artificial intelligence (AI), Robotics, IoT, and Cloud Computing. The objective of this revolution is to drive the digitization of industrial activities, transforming conventional manufacturing equipment and systems into interconnected cyber-physical systems [13].

CPS is a term that comes up a lot when talking about Industry 4.0. According to [4], [14], these systems are the core of Industry 4.0, as they embody the integration

of computational resources with physical processes. This integration is achieved through sophisticated networks of sensors and actuators, coordinated via advanced communication infrastructure. The CPS operates by continuously gathering data from the physical world, processing it at the Cyber level, and acting upon the process in real-time, resulting in enhanced operational efficiency and adaptability [4], [15].

The fundamental aspects of a CPS are based on three pillars: communication, control, and computing. Taking advantage of these elements, a CPS can perform critical functions, including detecting external stimuli through sensors, interacting with them, and manipulating their environment via actuators, and collecting and analyzing data. Additionally, it is capable of making informed decisions based on the accumulated data and it can interact with other CPS in a networked ecosystem [16]. As highlighted by [17], the application of CPS extends beyond industrial settings, i.e., they are utilized in medical devices for patient monitoring, environmental control, electric power systems, water resources management, and communication systems. A CPS offers autonomy, efficiency, and reliability, operating independently of human intervention.

2.2 Multi-Agent Systems

MAS [6] have emerged as a viable solution to manage the complexities inherent in a distributed environment. According to [18], MAS is considered a branch of Artificial Intelligence. The research work discussed in [6] highlights that the application of MAS introduces system characteristics such as modularity, decentralization, autonomy, scalability, and reusability. This is related to the ability of MAS to decentralize control functions across a network of distributed and intelligent software agents operating on both cloud and edge computing layers [5]. Therefore, as described by [19] and [20], what distinguishes an agent as a singular and intelligent entity are its characteristic features, which include:

- Autonomy: An autonomous agent governs its behavior and goals, functioning independently without human intervention. It self-regulates its internal state and is the sole arbiter of its actions;

- Responsiveness: Agents actively respond to environmental stimuli, processing sensory inputs and effectuating actions through actuators to meet their objectives promptly;
- Proactivity: Proactive agents not only react to their surroundings but also anticipate and adapt to potential changes, taking the initiative to shape future conditions to their advantage;
- Social ability: Agents are capable of communicating and cooperating with other agents and humans, considering others' abilities, reliability, and the nature of their relationships;
- Learning Capability: Agents can autonomously learn and adapt, enhancing their performance over time-based on experience and background knowledge.

Consequently, MAS consists of numerous autonomous agents that interact with each other and with their environment, aiming to achieve and fulfill individual and collective goals [21]. MAS agents communicate and coordinate their actions to solve problems that go beyond their individual capacities and achieve an overall goal. An example of an environment of MAS can be seen in Fig 2.2.

The effectiveness and versatility of MAS in various domains are due to their main advantages, such as greater efficiency, robustness against failures, and scalability to respond to different levels of demand. The flexibility and reusability of MAS allow for perfect adaptation and responsiveness to change, while their decentralized nature supports complex and distributed problem-solving processes, making them an integral part of applications that require dynamic and intelligent system behaviour [23].

2.3 MAS Frameworks

As [24] pointed out, a software framework comprises a structured set of libraries that offer essential functionalities and structures for the development of applications in a specific domain, thus facilitating the reuse of code and simplifying the development process.

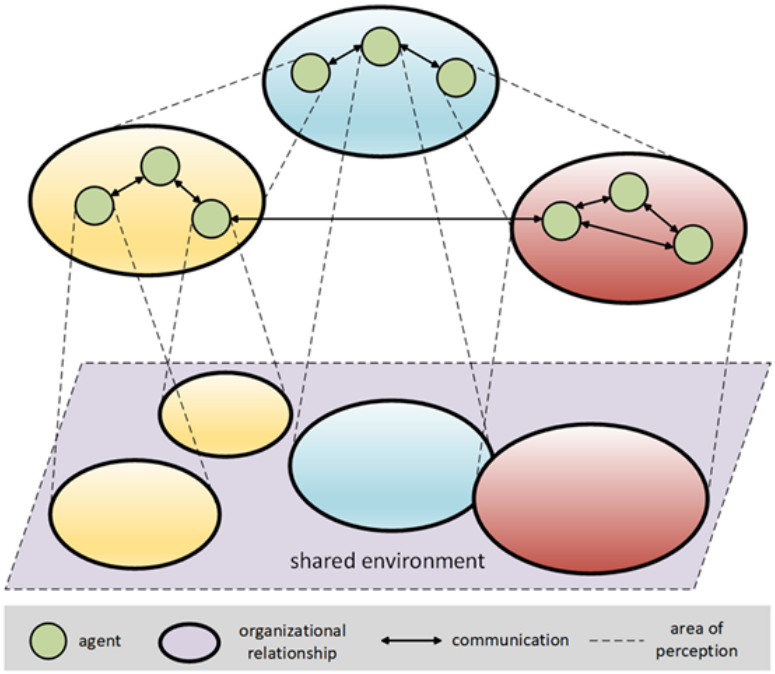

Figure 2.2: Structure of MAS [22].

Therefore, in the realm of computer science, especially for embedding the MAS principles, frameworks like SPADE and JADE are particularly notable. They provide an extensive collection of classes, methods, and attributes, enabling developers to efficiently create MAS application. These frameworks are adept at managing the complexities inherent in these systems, offering robust and flexible platforms that streamline the creation and implementation of sophisticated systems.

2.3.1 SPADE

According to [25], the Smart Python Agent Development Environment (SPADE) is a platform designed to support the new generation of multi-agent systems. It is a software project that has been publicly available for more than a decade and has recently been completely redesigned and re-implemented to create SPADE 3.

This framework was developed in Python [26] and is based on an asynchronous programming paradigm where it is possible to execute one or several behaviors, which are

independent tasks that perform the actions of the agent [25]. In this context, the framework is equipped to support five main types of behavior to develop the functionality of an agent:

- Cyclic: This type continuously repeats a specific action or set of actions. It's useful for tasks that require constant monitoring or ongoing execution without a predefined end;
- One-shot: As the name suggests, this behavior executes an action only once. It's ideal for tasks that need to be performed a single time, such as initialization procedures;
- Periodic: This behavior allows an action to be executed at regular intervals. It's suitable for tasks that need to happen periodically, like routine checks or updates;
- Time-Out: In this case, an action is performed after a specified time interval or delay. This behavior is beneficial for tasks that require a pause or delay before execution, such as timed responses or wait conditions;
- Finite State Machine: This behavior involves multiple states and transitions between these states. It's used for complex tasks that require different stages of execution, decision-making processes, or a sequence of varied actions.

Furthermore, SPADE's performance in multi-agent systems is greatly improved by its chosen communication protocol, which is essential for the functioning of intelligent, autonomous agents. Specifically, SPADE uses the eXtensible Messaging and Presence Protocol (XMPP), a comprehensive protocol that extends well beyond basic messaging. XMPP offers a unique presence notification mechanism, allowing agents to maintain a dynamic list of contacts and receive timely updates about any changes in their states, which is vital for real-time decision-making and coordination [25].

Finally, SPADE also accommodates the need for Foundation for Intelligent Physical Agents (FIPA) standard communication modes to ensure compliance with established

communication norms, especially when an agent requires information sharing. This is achieved by augmenting the XMPP protocol with a new message tag that adheres to FIPA message standards, enabling seamless and standardized communication among agents [27].

2.3.2 JADE

As highlighted by [28], [19], The Java Agent DEvelopment Framework, is a software framework designed for creating agent applications that align with the FIPA specifications for interoperable, intelligent multi-agent systems. Therefore, JADE provides a comprehensive suite of tools and libraries that aid in the development, deployment, and management of agent-based applications.

Developed using Java [29], JADE's architecture allows for the simultaneous operation of multiple Java Virtual Machines (VMs). Communication in JADE is facilitated by Java RMI (Remote Method Invocation) across different VMs, complemented by event signaling within the same VM [28], it also uses communication over TCP/IP. This inclusion of TCP/IP facilitates broader network interactions and connectivity, allowing JADE agents to communicate effectively over distributed networks, thereby enhancing the framework's versatility and scalability in diverse computing environments.

Furthermore, for the effective operation of an application using JADE, it is essential to run a set of specific agents that manage the platform, as emphasized by [19]. These key agents include:

- Agent Management System (AMS): This agent acts as the central authority of the JADE platform, managing the lifecycle of all agents. It registers new agents, deregisters agents, suspends or resumes agents, and handles agent migration between containers.

- Agent Communication Channel (ACC): The ACC is responsible for managing all incoming and outgoing messages, serving as a gateway for inter-platform communication. It ensures that messages are correctly routed to their respective recipients, both within and across different JADE platforms.

- Directory Facilitator (DF): The DF is similar to a Yellow Pages service for agents. It provides directory services where agents can advertise their services and capabilities, as well as search for other agents based on their services. This facilitates service discovery and dynamic agent collaboration.

Figure 2.3 shows the architecture of the JADE platform, illustrating how agents work in a Java-enabled runtime environment. The architecture is centered on the main container, which is fundamental for initializing and managing the platform. It is in this main container that the main system management agents, detailed above (AMS, DF, ACC), reside and operate. Other containers can be seamlessly incorporated into the system, establishing connections with the main container and taking advantage of the management services to maintain a coherent framework of various agents [19].

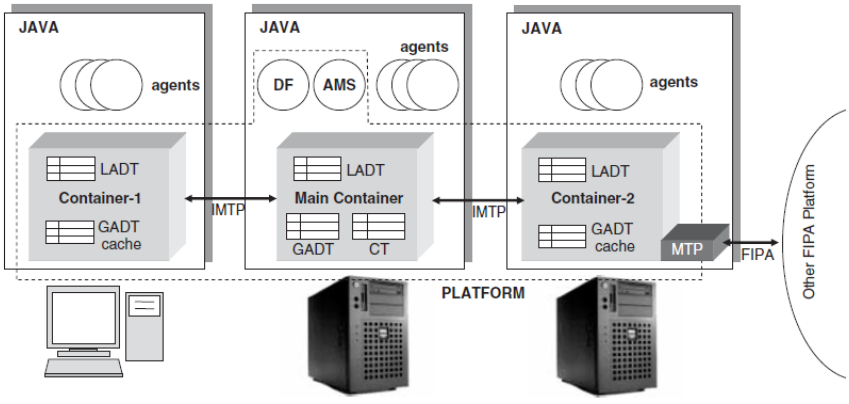

Figure 2.3: Architecture of the JADE platform elements [19].

In addition, agents developed with JADE also can perform various behaviors, in addition to those available in SPADE. JADE's versatile structure allows each agent to execute several behaviors simultaneously, an essential feature for creating sophisticated and reactive agents. In addition to the behaviors previously discussed in the context of SPADE, JADE offers several other behaviors [19], for example:

- ParallelBehaviour: This behavior allows an agent to perform multiple sub-behaviors in parallel. It is particularly useful in scenarios where an agent needs to multitask,

handling several operations concurrently without waiting for one to complete before starting another.

- SequentialBehaviour: As the name implies, this behavior ensures that an agent performs a series of sub-behaviors in a specific order, one after the other. It is ideal for tasks that require a particular sequence of actions, where each subsequent behavior depends on the completion of the previous one.

- CompositeBehaviour: This is a more flexible and generalized form of behavior that can include both parallel and sequential sub-behaviors. It offers agents the ability to handle complex tasks that require a combination of concurrent and sequential operations.

Therefore, by using JADE it is possible to perform various behaviors that will provide agents with capabilities inherent to MAS concepts, e.g. distributing intelligence in a heterogeneous scenario, autonomy to deal with changes in the scenario without human intervention, and proactivity to predict and adapt to changes applied to the system.

2.4 Communication

This section will cover the protocols used to communicate and standardize the messages exchanged in the different frameworks used to control the CPS. Therefore, an overview will be given of the Foundation for Intelligent Physical Agents - Agent Communication Language (FIPA-ACL) standards used to standardize the messages exchanged by agents, the XMPP protocol, used for real-time instant messaging, and the Message Queuing Telemetry Transport (MQTT) protocol, due to its large-scale application in IoT systems.

2.4.1 FIPA-ACL

According to [30], FIPA is an international organization formed by companies and universities, dedicated to developing specifications that support interoperability between agents

and agent-based applications. In this sense, all agents have to interact in a system with the same language, assigning identical meanings to the concepts under discussion, in order to be able to understand and be understood by other agents. The main way to ensure the interoperability is by defining a standard to to exchange information. In this sense, FIPA-ACL presents itself as a structure to be followed to ensure the standardization of messages exchanged between agents, thus guaranteeing the interoperability of agents developed on different platforms.

Therefore, any structure that intends to adhere to the standards established by FIPA-ACL must satisfy certain key parameters. They are the performative act of the message, the identification of the sender, the intended recipient, and the content of the message itself. Although there are several additional parameters in FIPA-ACL, their application and relevance vary depending on the specific type of agent-based application being developed [19],[30]. Table 2.1 shows the parameters present in the FIPA-ACL message.

Table 2.1: Parameters of the ACL message [19].

Parameter	**Description**
Performative	Type of the communicative act of the message
Sender	Identity of the sender of the message
Receiver	Identity of the intended recipients of the message
Reply-to	Which agent to direct subsequent messages to within a conversation thread
Content	Content of the message
Language	Language in which the content parameter is expressed
Encoding	Specific encoding of the message content
Ontology	Reference to an ontology to give meaning to symbols in the message content
Protocol	Interaction protocol used to structure a conversation
Conversation-id	Unique identity of a conversation thread
Reply-with	An expression to be used by a responding agent to identify the message
In-reply-to	Reference to an earlier action to which the message is a reply
Reply-by	A time/date indicating by when a reply should be received

In this sense, all agents interacting in a system with the the same message structure can

attribute identical meanings to the concepts under discussion, enabling it to understand and be understood by other agents.

2.4.2 XMPP

As [31] mentioned, XMPP is an open protocol used mainly for real-time communication, which works by using Extensible Markup Language (XML) to encode messages and facilitate their exchange between clients and servers, providing a standardized way for clients to not only exchange messages but also to share presence information and manage contact lists. Typically, XMPP is implemented through a distributed client-server architecture, requiring clients to connect with a server for information exchange.

The versatility of XMPP extends to a wide range of applications beyond basic messaging. It is well-suited for multi-party conversations, voice and video calls, collaborative efforts, functioning as a lightweight middleware, content syndication, and generalized XML data routing [32]. According to [33], XMPP provides different features that can be used for any given application, some of these features are:

- Channel encryption - It provides encryption of the connection between a client and a server;
- Presence - A notification mechanism by which clients can receive or set availability information;
- Contact list - A "friend list" where entities that are known are stored;
- Notification - Mechanism to generate a notification and deliver it to multiple contacts.

As XMPP communication takes place over a network, each entity using this protocol is assigned a unique identifier, known as a JabberID (JID). The JID is based on the Domain Name System (DNS) to provide its address structure, assigning each client an identifier that resembles an email address, following the format 'user@domain.tld'. This

JID is then used to route messages between clients and servers and to identify the sender and recipient of each message [33]. Fig. 2.4 illustrates this process, depicting how two clients, each with their own JID, send messages to a server that recognizes all client JIDs and, consequently, routes messages to the correct recipient.

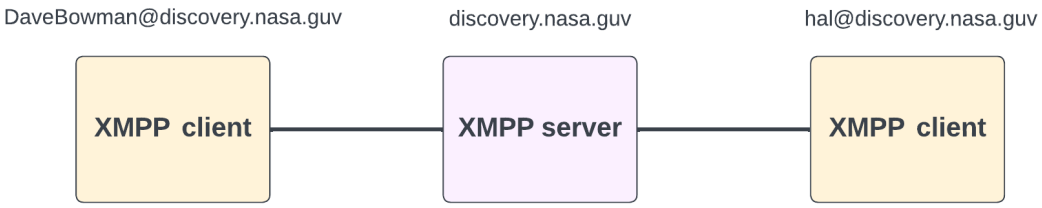

Figure 2.4: XMPP Architecture (adapted from [34]).

2.4.3 MQTT

MQTT [35] is considered a lightweight publish/subscribe messaging transport, especially suitable for applications with limited resources, such as those with limited bandwidth or high latency environments. In addition, its efficiency makes it widely used in the Internet of Things (IoT) for Machine to Machine (M2M) communication, where low-power devices, such as microcontrollers, can transmit and receive information seamlessly with other machines in a process.

This protocol fundamentally operates with two entities: the broker and the clients [36]. The broker is the message dispatcher, responsible for receiving all messages from the clients, determining which topics the messages belong to, and then distributing them accordingly to clients subscribed to those topics. On the other hand, clients can be any device or application that communicates with the broker to send (publish) or receive (subscribe) messages, Fig. 2.5 depicts this interaction.

Another key feature of this protocol is the guaranteed reliability of message delivery, which is supported by three levels of Quality of Service (QoS) [17]. These QoS levels

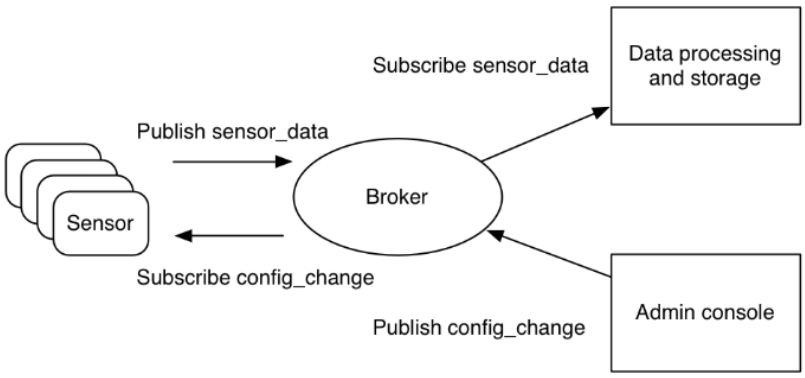

Figure 2.5: Publish-subscribe model using MQTT [36].

dictate the guaranteed delivery of messages via topics to their subscribed recipients. Each level offers a different degree of guarantee, ranging from delivery "at most once" to delivery "exactly once", responding to various message reliability requirements. The characteristics of each Quality of Service (QoS) are:

- QoS 0: This level attempts to deliver the message only once without acknowledging receipt, thus not guaranteeing delivery.
- QoS 1: Ensures the message is delivered at least once by waiting for an acknowledgment from the recipient. If no acknowledgment is received, the message may be sent again.
- QoS 2: Guarantees that the message is delivered exactly once by employing a four-step handshake process, which eliminates the possibility of message loss.

Consequently, to publish or subscribe to a topic within an MQTT broker, a client must specify some details. Initially, the client identifies the topic it wishes to publish. Subsequently, it selects the desired QoS level to establish the required reliability for message delivery. Importantly, when the client decides to publish a message, it has the option to provide a payload, which represents the content of the message. It is crucial to note that this payload can be furnished with specific content or left empty, according to the requirements of the publication [36].

2.5 Middleware

In industries replete of CPS that rely on different protocols to share data, ensuring interoperability is crucial. According to [7], interoperability is fundamental for the real-time collection of production data, ranging from low-level devices to enterprise-level applications. Furthermore, [37] emphasizes that interoperability is an essential enabler for the dynamic configuration of production lines, a key aspect of Industry 4.0. In this context, employing middleware emerges as an effective strategy to ensure interoperability among a network of heterogeneous devices with differing protocols [38].

Middleware is a type of software that functions as an intermediary, enabling seamless communication and information exchange between applications and devices with diverse functionalities. It acts as a bridge, facilitating the integration and interoperability of systems that may otherwise be incompatible due to their differing architectures or protocols [39].

Since middleware provides a software layer that abstracts the complexities of communications between different systems in an IoT application, it needs to have certain requirements to be effective, some of which are [38]:

- Scalability: The middleware must be scalable to accommodate the growth of the network and IoT applications;
- Reliability: It should maintain operational consistency throughout its usage;
- Availability: The middleware needs to be available all the time when it is supporting an IoT application;
- Security: It is crucial for the middleware to safeguard the data exchanged within applications;
- Interoperable: The middleware must facilitate data and service exchange between different applications.

Therefore, when middleware is applied to a heterogeneous system made up of several CPS, it is possible to provide the interoperability that is essential for integrating the functions of the various devices. However, it is essential to identify and use the right middleware for each situation. For example, MQTT [35] is specialized for IoT applications, promoting asynchronous communication in environments with multiple devices, ideal for facilitating real-time interaction. On the other hand, when it comes to web applications, Apache Tomcat [40] is an excellent choice due to its efficiency in hosting and managing Java applications such as JSP and Servlets. For systems integration, MuleSoft [41] is ideal due to its ability to easily connect different applications and services via APIs. These are just examples of the variety of middleware available, each designed to meet the specific needs of each type of application.

2.6 Security Issues

In a distributed environment composed of several heterogeneous CPS devices, the security of message exchanges between different applications is fundamental. The threat of unauthorized entities gaining access to data poses a substantial risk, potentially leading to significant system malfunctions. As [42] points out, the seriousness of such violations can be seen in, e.g., altering sensor data, incorrectly activating actuators, or even spying on the data to obtain confidential information. Thus, ensuring data integrity and controlling access of entities, is essential to guarantee the reliability of the environment.

Therefore, integrating a middleware layer to ensure interoperability between different CPS introduces critical security concerns for optimal system operation. The addition of this layer exposes the system to possible threats that can significantly damage its functionality. According to [43], the use of middleware in an IoT scenario can expose the system to threats such as denial of service (DoS), Identity Spoofing, Information Disclosure, Elevation of Privileges and Data Tampering. Such vulnerabilities can lead to severe consequences, including service disruptions, unauthorized access, privacy breaches, unauthorized control or alterations of system operations, and compromised data integrity.

Thus, it is essential to identify all the threats that middleware is subject to in order to carry out the necessary mitigations.

In order to assess the potential threats that the middleware could face, the STRIDE threat model is a good aproach. Introduced by Microsoft in 1999, this model is widely recognized for its effectiveness in identifying and categorizing six primary types of threats that a system might face. These threats are classified as follows [44]:

- Spoofing: This type of attack occurs when an attacker assumes the identity of a user and performs actions on their behalf.
- Tampering: Involves an attacker modifying or editing legitimate information.
- Repudiation: Repudiation threats are associated with users denying their actions without other parties being able to prove otherwise. For example, a user performs an illegal operation on a system that cannot track prohibited operations.
- Information Disclosure: Refers to the unauthorized exposure of confidential information, such as data leakage and unauthorized access to sensitive information.
- Denial of Service: This threat involves actions aimed at damaging the availability or performance of a system, often by overloading resources or exploiting vulnerabilities,e.g., by sending an abnormal amount of requests to the target service in a short period.
- Elevation of Privilege: In this type of attack, an unprivileged user gains privileged access, allowing them to compromise or destroy the entire system by performing unauthorized actions.

Therefore, by using STRIDE modeling, it is possible to identify the types of attacks and risks to which the system is subject, allowing mitigation measures to be adopted and applied to prevent such threats.

Chapter 3

Middleware Architecture to Integrate MAS Frameworks

This chapter provides a comprehensive overview of the middleware architecture developed to facilitate interoperability between the MAS frameworks. It details the structural design of the middleware, elucidating the function and meaning of each component within the system. This discussion aims to highlight how the architecture effectively bridges the communication gap between the frameworks, ensuring seamless interaction and coordination among agents.

3.1 Architecture Overview

The proposed architecture, designed to facilitate interoperability between MAS frameworks, is built upon a lightweight protocol optimized for efficiency and minimal resource utilization during message exchanges. This architecture possesses essential attributes that align with the principles of MAS systems in IoT applications, ensuring flexibility, efficiency, scalability, adaptability, and security. These characteristics are vital for reliable message delivery and the ability to adapt to various MAS frameworks.

As illustrated in Figure 3.1, the generic architecture consists of various distinct niches, each representing a separate MAS framework. Within each domain of these frameworks,

there are ecosystems containing agents, each with unique knowledge and characteristics. Through interaction and communication between these agents, it becomes possible to comprehend the environment they are part of and, consequently, to achieve local and global objectives pertinent to the ecosystem. Additionally, each ecosystem is capable of communicating with and obtaining information from other ecosystems, facilitating the accomplishment of holistic goals that reflect the decentralized nature inherent to a MAS. However, direct communication between different framework niches is not straightforward due to distinct characteristics, such as the use of different communication protocols or data structures. That is why, the interface agents depicted in this architecture are necessary to bridge the various frameworks.

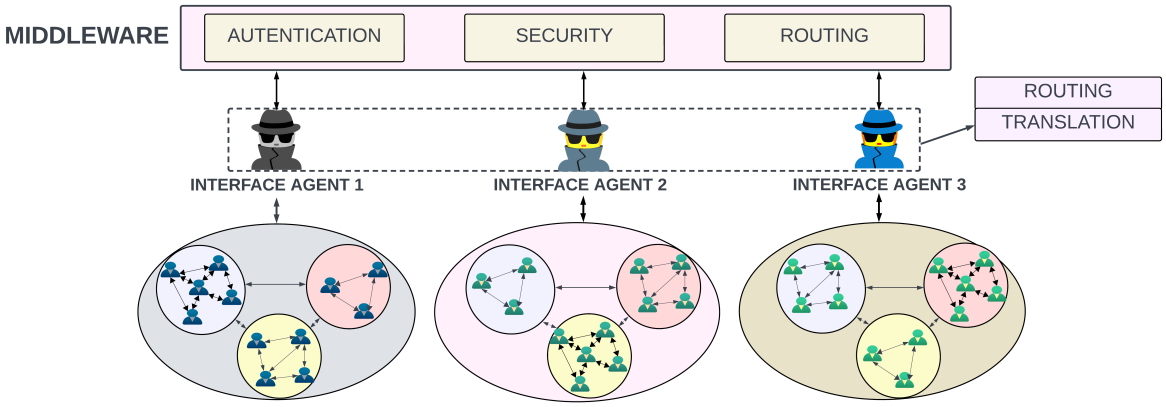

Figure 3.1: System architecture.

Interface agents serve a pivotal role as a component of the middleware, facilitating seamless communication between the various MAS structures. These agents ensure that the messages exchanged comply with the specific parameters and protocols of each structure. Thus, their responsibilities go beyond the mere transmission of messages; the interface agents also have the task of translating and routing the messages that pass through the system's various ecosystems.

To enable and secure the exchange of messages between distinct frameworks, the interface agents of the middleware employ the Publish-Subscribe model. This model ensures

that messages are disseminated in a robust and orderly manner. Furthermore, the middleware leverages the MQTT protocol for agent communication, which not only facilitates reliable message exchanges but also adheres to stringent security protocols. MQTT's contribution to security is significant, offering encryption of messages, authentication of clients, and checks for data integrity. Consequently, the middleware architecture proposed in this project is adept at authenticating and securing messages. Moreover, it provides essential services such as message routing and translation through interface agents, thus achieving interoperability among various MAS frameworks.

3.2 Interface Agent Behaviour

The middleware's functionality relies on the specialized behaviour of each interface agent, which is vital for guaranteeing interoperability between different frameworks. These agents have the task of translating, structuring and sending messages from the original framework to the appropriate broker topics. They also handle messages received from subscribed topics, adapting them to their own framework's protocol before forwarding them to the correct internal agents. Figure 3.2 visually represents this process using Petri nets [45], outlining the workflow of the interface agents within the middleware.

Each interface agent executes a cyclic behavior, actively monitoring for messages to publish or receive. At startup, T1, the agent configures itself by recognizing its framework's agents and subscribing to other frameworks' topics. This preparatory phase leads to a waiting loop, P3, where the agent anticipates two kinds of events: messages from its framework or the broker.

Upon receiving a framework message, T3, the agent converts it into a neutral format and publishes it in a topic in the middleware, T5. This action represents the message's origin and targets the respective framework. After publishing, the agent reverts to its waiting state, P3.

Alternatively, if a message arrives from a subscribed topic, T6, the agent decodes payload and creates a native message, T7, that conforms to its framework's protocols. The

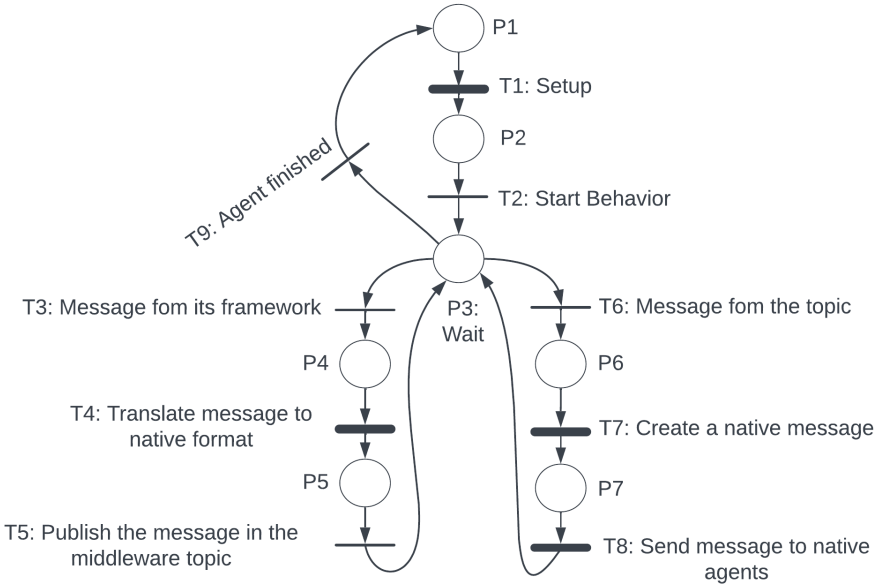

Figure 3.2: Logic control for a interface agent.

agent then locates the active agents within its framework and sends them the translated message, T8, before returning to the initial waiting state, P3.

3.2.1 Translation Mechanism to Publish the Message in Middleware.

As the interface agent must translate and standardize the messages exchanged with other interface agents via middleware, it must have the appropriate mechanism for dealing with messages received from its own framework. Therefore, as illustrated in Figure 3.3, when an interface agent receives a message from another agent belonging to its framework, it goes through specific procedures to ensure that the message complies with the standards defined for exchanging messages between interface agents via middleware. After receiving the message, each interface agent creates a dictionary (T4.1) in which all the keys represent parameters defined by the FIPA-ACL standard. The agent then maps the message received (T4.2) and fills the dictionary with the values passed in that correspond to each parameter (T4.3). In cases where the message does not have certain parameters from

the FIPA-ACL standard, the agent inserts null values for each missing parameter. When the dictionary is complete, the agent converts it into JSON (T4.4) and defines the topic and subtopic in which the message will be published (T4.5), which in this case is the framework in which the interface agent is operating and the identifier of the agent who sent the message, respectively.

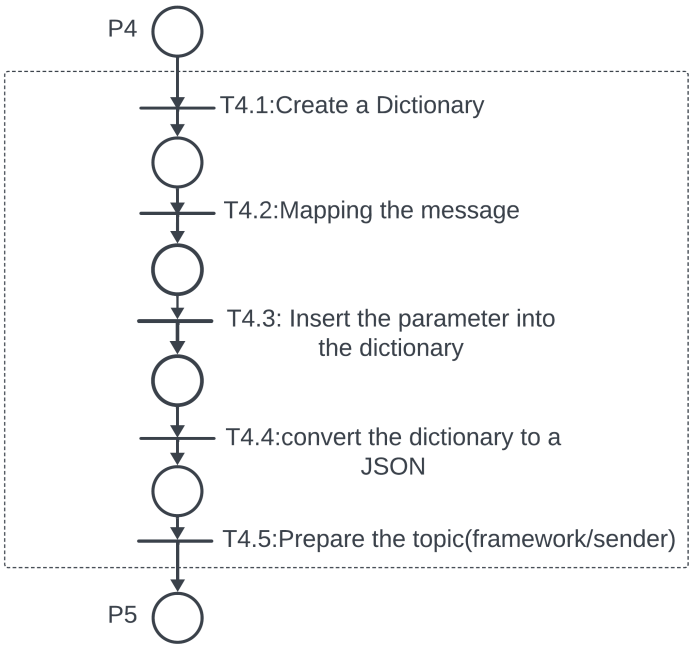

Figure 3.3: Mechanism for dealing with messages received from its framework.

3.2.2 Translation Mechanism for Messages Received by the Middleware.

On the other hand, as illustrated in Figure 3.4, when the agent receives a message from another interface agent via middleware, it goes through an alternative message analysis procedure. In this scenario, upon receiving the message, the agent decodes the payload (T7.1), maps the received JSON to identify all the FIPA-ACL parameters filled in the received file (T7.3) and then performs the translation and creates the message following the protocols and structure required to send it to the agents in its framework (T7.3). After

creating the message, the interface agent searches for the active contacts in its framework (T7.4) and prepares the message for sending.

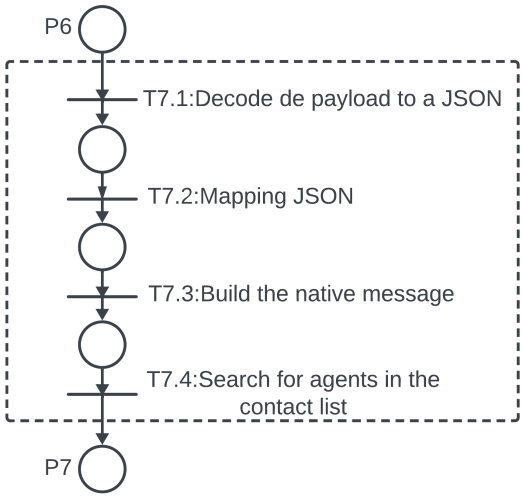

Figure 3.4: Mechanism for handling messages from other interface agent.

3.3 Middleware Features

This section presents the main features that the middleware uses to exchange messages between the interface agents in its catalog.

3.3.1 Communication by Message Topics

The middleware's adoption of the Publish-Subscribe model facilitates a well-organized exchange of messages, enabling precise identification of different frameworks within the system. This orderly structure is achieved through the delineation of topics and the message broker's role inherent in the Publish-Subscribe model. Consequently, in the proposed middleware, each interface agent functions dually: as a publisher, it disseminates messages across various topics, and as a subscriber, it receives messages dispatched by other agents. This dual capability ensures that agents are actively engaged in both broadcasting and receiving pertinent information, thus maintaining a dynamic and responsive

communication network.

For this reason, as shown in Figure 3.5, the interface agents utilize a structured topic hierarchy within the message broker. Each agent, acting on behalf of its respective framework, registers with the broker to publish messages originating from its framework and subscribes to topics associated with other frameworks. It is crucial to recognize that each primary topic represents a distinct framework, while sub-topics correspond to individual agents operating within those frameworks. Thus, an interface agent is responsible for creating sub-topics not for itself but for the agents that are part of its framework, ensuring a structured and systematic approach to message dissemination.

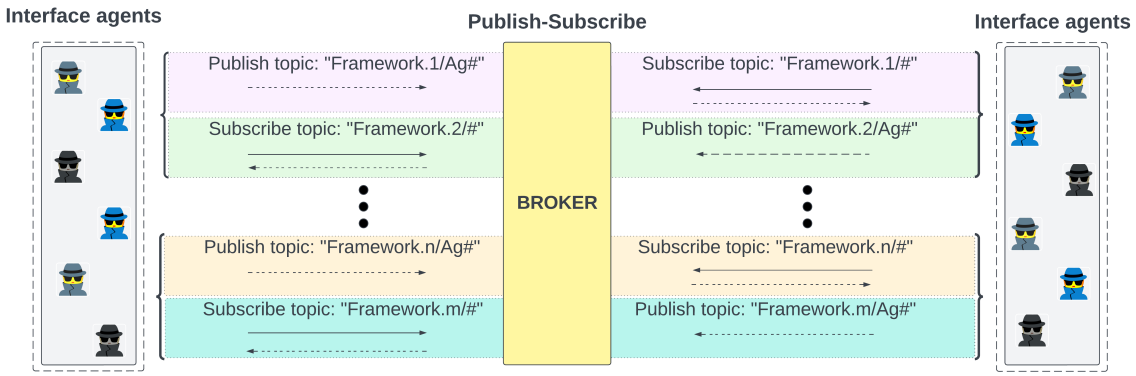

Figure 3.5: Scheme for exchanging messages by topics.

Therefore, this approach ensures a systematic and structured method of disseminating messages. In addition, it allows the middleware to have fundamental characteristics for operating in a multi-agent system scenario, since it allows the middleware to acquire the following characteristics:

- Scalability: The integration of a new framework into the middleware requires simply adding a new topic. This ease of expansion allows the system to grow and adapt as needed.
- Flexibility: The ability to enable or disable interface agents without disrupting the overall system enhances the middleware's adaptability to changing requirements.

- Reliability: the presence of a message broker guarantees the reliable delivery of publications to the subscribers of the respective topics, reinforcing the trustworthiness of the system.

Thus, once this approach has been adopted to exchange messages between interface agents, the only predefined topics are related to the frameworks that are already present in the middleware's catalog, meaning the interface agents that have been developed to communicate using the middleware. Specifically, when an interface agent for a framework is created, it establishes a topic pattern of 'framework/Ag#' for publication, where 'Ag#' represents any agent within that framework sending a message. In the event that a new MAS framework is integrated into the system, a corresponding new top-level topic is created for the framework. Existing interface agents then subscribe to this new framework's topic using the multi-level wildcard 'framework/#', enabling them to receive all messages disseminated within this new framework's domain.

3.3.2 Message Structure

Another key aspect of the proposed middleware is the method by which interface agents structure messages for exchange. Establishing a standardized format is crucial for ensuring interoperability among the various agents. To achieve this, the interface agents utilizes the Java Script Object Notation (JSON) [46] data format in conjunction with the FIPA-ACL standard for message structuring. This combination allows for efficient communication while adhering to established protocols.

The choice of JSON format for message structuring in the middleware was driven by its ease of interpretation by both developers and machines. Known for its lightweight and efficient nature, JSON facilitates the organization of complex data structures and can be implemented with relative ease. By utilizing JSON to structure messages in line with the FIPA-ACL standard, intuitive and organic structuring is achieved. JSON employs key-value pairs, where each key corresponds to a parameter outlined by FIPA-ACL, and the values represent the specific content of each aspect. This includes elements such as the

communicative act of the message, its actual content, the protocol used, and the language used in the message. An example is seen in Fig 3.6.

```
{
    "performative": "inform",
    "sender": ["masdeveloper",
      "jabbers.one",
      "fs6RB78BAE1t"],
    "receiver": "agentJADE",
    "reply-to": "null",
    "content": "1",
    "language": "null",
    "encoding": "null",
    "ontology": "null",
    "protocol":
    ↪  "tokenTransitionInput",
    "conversation-id": "null",
    "reply-with": "null",
    "in-reply-to": "null",
    "reply-by": "null"
}
```

Figure 3.6: Message structure.

3.3.3 Interaction Scheme

The sequence diagram shown in Figure 3.7 elucidates the structured communication flow between interface agents from different frameworks. This diagram illustrates a generic scenario in which an agent, involved in a specific process within its native framework, needs to request information from an agent in a separate framework. The interaction begins with the interface agents registering with the message broker, subscribing to topics relevant to the frameworks in question. This registration process is fundamental for the agents to participate in subsequent communicative exchanges.

An agent, known as the initiator, starts this exchange by sending a request for information to interface agent #1. Upon receipt of this request, interface agent #1 performs a translation of the content into a standardized format recognized by the system, ensuring interoperability. This translated message is then published in a topic dedicated to the agent who submitted the information request.

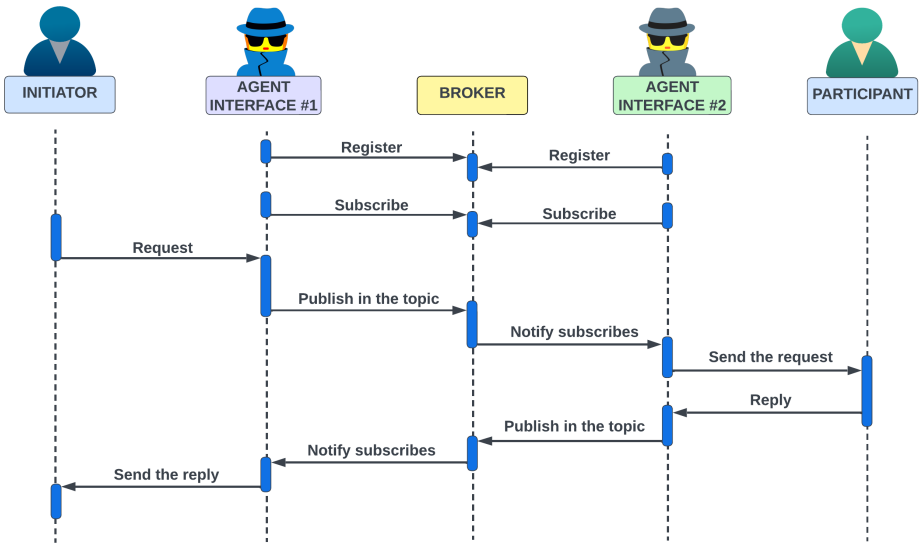

Figure 3.7: Sequence diagram for communication between frameworks.

The message broker, central to the process, receives the formatted message and forwards it to all the subscribing interface agents who have shown an interest in that particular topic. This ensures that the information reaches all the relevant parties efficiently.

When the interface agent #2 receives the notification from the broker, it processes the message received. It interprets and translates the payload into its native protocol and then forwards the message to the participant agent, which is in the same internal framework as the interface agent #2. The participant agent gets involved with the message, analyzing and processing the request in its domain of expertise and then preparing a response.

Once the reply is ready, interface agent #2 collects it and publishes it back in the topic associated with the participating agent. The broker, always alert, notifies interface agent #1 of the arrival of the new message, maintaining the flow of communication.

To complete the sequence, interface agent #1 retrieves the topic response, converts it into an intelligible format for the initiator and delivers the message. This final step concludes the communication loop, exemplifying a successful exchange of information between agents of different frameworks, as illustrated in the sequence diagram. This systematic approach highlights the importance of standardized message formats and a

reliable message broker to facilitate continuous communication between frameworks.

3.4 Middleware Security

The middleware leverages the MQTT protocol for message exchange, which provides a robust security mechanism. To safeguard communications between interface agents, the middleware provides several protective measures.

Firstly, it provides the Transport Layer Security (TLS)/Secure Sockets Layer (SSL) encryption protocol to secure message exchanges. This encryption ensures that messages transmitted between interface agents are obscured, thereby protecting against interception and interpretation by unauthorized third parties. Moreover, the utilization of digital certificates for authentication allows for each agent to be verified before connecting to the broker. This measure not only confirms the identity of each agent but also ensures the integrity of the data during transmission.

In addition, the middleware allows for improved system security through client authentication, requiring a username and password. This layer of security further reinforces the middleware's defenses, ensuring that only authorized clients can access the message brokering service.

Together, these security mechanisms fortify the middleware against a spectrum of cyber threats, ensuring a secure and reliable environment for the exchange of messages within the MAS frameworks.

Chapter 4

Development of the MAS Applications for the Case Study

This chapter presents the solutions adopted for the self-organization approach in the work. So, it is divided into two parts: the first deals with the case study in which both, the self-organizing logic and the middleware will be applied; the second presents how the system is implemented using each framework allowing the conveyors to be controlled to achieve the self-organization behavior.

4.1 Description of the Case Study

To simulate an Industry 4.0 environment, characterized by the presence of multiple CPS that demand dynamic and interactive components, it was employed the structure depicted in Figure 4.1. This structure comprises both, a physical and a cyber component.

The physical component consists of a group of modular conveyors developed by Fischetechnik. Each conveyor in this set shares an equal structure and provides the same functionalities, e.g, transporting objects from their initial positions to their respective endpoints. Therefore, each conveyor is equipped with a belt that is operated by a 24 V DC motor and two photoelectric sensors, which are responsible for detecting objects at the entry and exit points of the conveyors. For the cybernetic part of the system, a

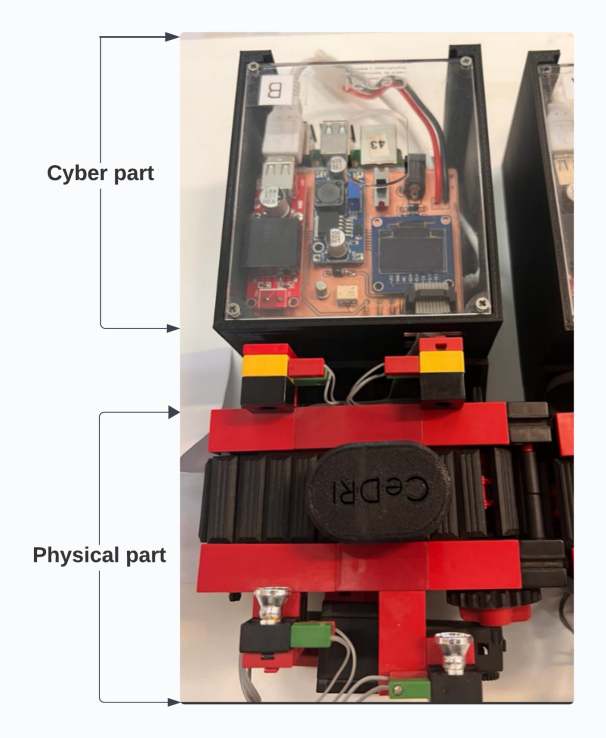

Figure 4.1: The modular conveyor transfer system.

Raspberry Pi single-board computer is integrated.

The proposed system aims to simulate a dynamic industrial environment in which each cyber-physical conveyor must handle the transport of parts from its conveyor belt to the next and must primarily adapt to changes that may occur in the environment, offering features that enable the process to be scalable, self-organizing and reconfigurable in an on-the-fly manner, which means that it does not need to be stopped, restarted or reprogrammed. Therefore, the control logic applied to the CPS must be suitable to ensure that the system has these characteristics.

As [47] pointed out, traditional control logics applied in industry, such as those based on an IEC 61131-3 program running on a Programmable Logic Controller (PLC), have limitations in terms of system scalability and reconfigurability. This is because this approach relies on centralized control logic, which means that if it becomes necessary to change the order of a conveyor, add, remove, or even perform isolated programming, it will be necessary to interrupt the process, resulting in a loss of time.

Therefore, the approach applied in this project, which enables the system to be flexible and reconfigurable on-the-fly, is based on MAS technology. Consequently, each cyber-physical device is integrated into a software agent responsible for representing the cyber aspect of each module, thereby facilitating control over the physical part of the system, as illustrated in Figure 4.2. To ensure that this process exhibits the characteristics of self-organization, the concepts demonstrated and explained by [47], are also employed. However, these concepts have been adapted and enhanced to address the specific requirements of the frameworks used within the system.

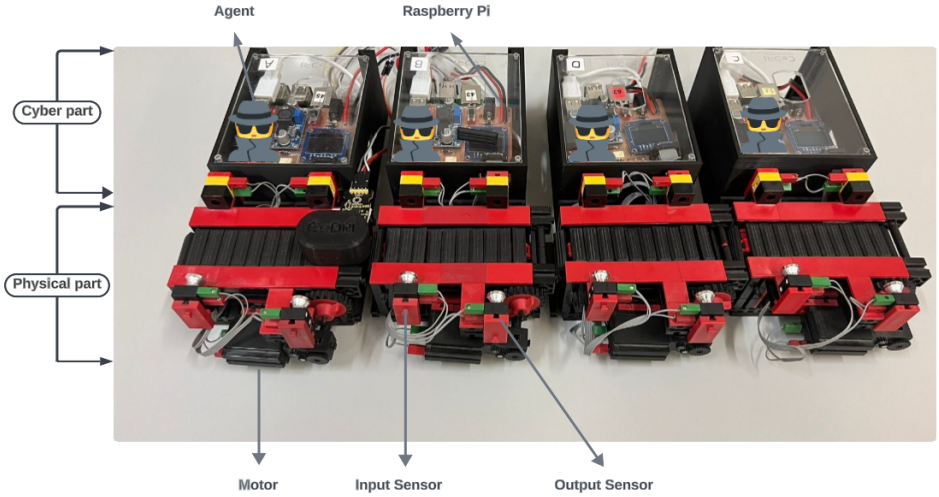

Figure 4.2: Cyber-physical conveyor system.

4.1.1 Self-Organization Capabilities

For the system to manifest the characteristics of self-organization, effective communication among agents is crucial. Structured messages are employed to convey pertinent information about the state of the modules. This ensures that each agent can identify its position in the system's transport order, enabling it to take action on its module as needed to achieve the system's primary goal: transporting parts from the first to the last module.

In the communication process, each agent assigns a specific interaction protocol based

on the message's purpose. The protocols used in this project are based on those presented by [47]: *tokenTransitionInput*, *tokenTransitionOutput*, *informAlive*, *thereIsSwap*, *swapTokenFound* and *leftDF*. Additionally, new protocols, including *firstTokenSwapped*, *conveyedAllPieces*, and *aliveToo*, were introduced. Through these protocols, agents can determine the appropriate mechanism to adapt to various system stages, such as initialization, module addition or removal, and swap positions.

4.1.2 Initialization

The initialization mechanism is elucidated using the system representation described in Fig 4.3.

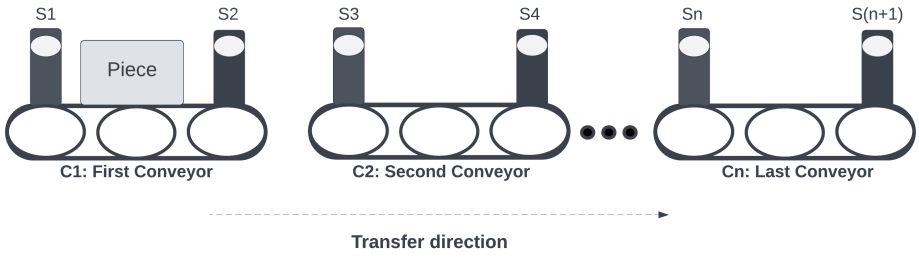

Figure 4.3: System workflow.

When the system is initialized, the agents are unaware of their positions in the transport sequence, which leads to the adoption of an initial learning process. As a part is introduced into the first module and detected by the S1 sensor, the corresponding agent assumes the *token* with the value 1, signifying the first position, and initiates the part transfer. Subsequently, when the part reaches the S2 output sensor, this agent dispatches a *tokenTransitionOutput* message containing the *token* value to all agents in the system. It then waits for confirmation that the part has arrived on the next conveyor.

On receiving the *tokenTransitionOutput* message, all the other agents start their motors and wait for the part to be detected on the input sensor. As the part enters the second conveyor belt, C2, and passes through sensor S3, the agent responsible for this

module assigns itself the value of the received *token* + 1 and broadcasts a *tokenTransitionInput* message to the agent of the previous module, C1, indicating that it can turn off its conveyor belt. This iterative process is repeated in all the modules until the last conveyor, Cn, is reached. Once the agents have acquired knowledge of their positions, they will turn their conveyors on or off only in response to messages from the previous or next agent.

4.1.3 Adding or Removing a Module

Another crucial mechanism for the self-organization process involves identifying all active agents, whether during system initialization or when a new module is added. As depicted in Figure 4.4, when an agent initializes, it sends an *informAlive* message to all agents on its contact list (T1) and awaits confirmations (T7). This approach is vital for recognizing agents developed in other frameworks if present, with the interface agent notifying the system of these confirmations.

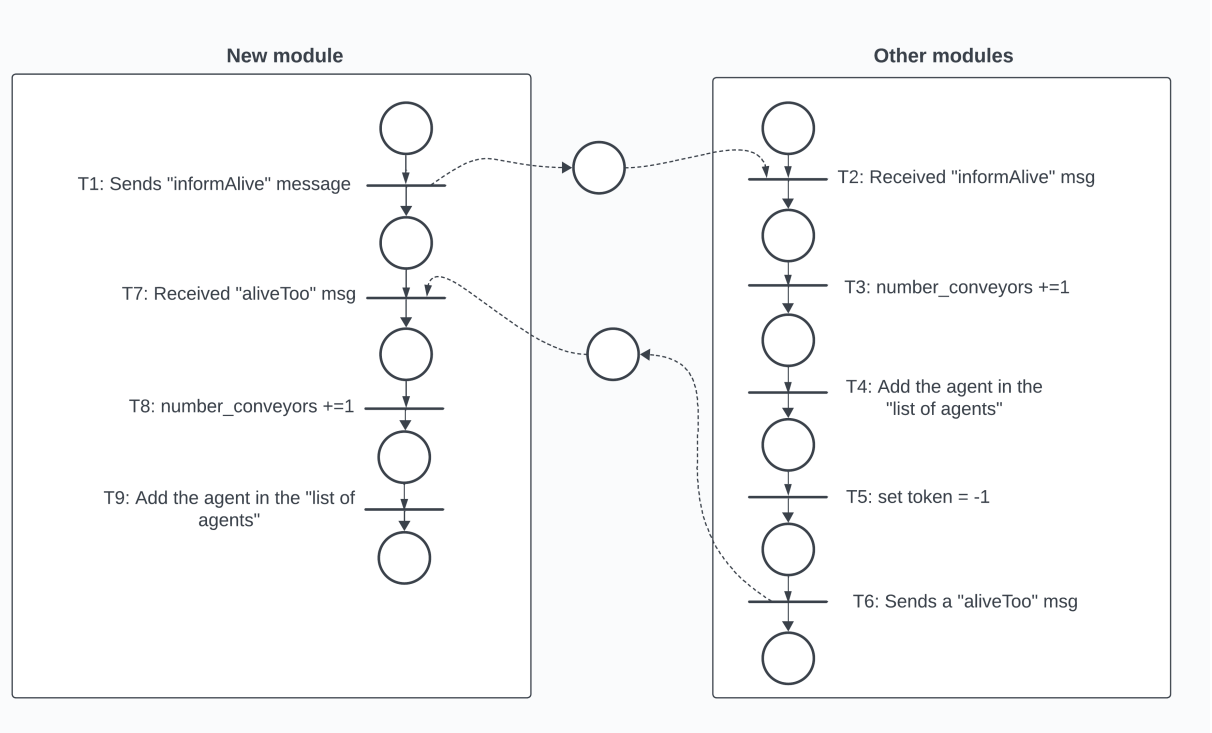

Figure 4.4: Mechanism for identifying agents

Upon receiving the *informAlive* message (T2), other agents update the count of active conveyors (T3) and add the sender agent to their list of agents (T4). Due to a lack of knowledge about the conveyor's current position and recent changes, agents enter a learning state (T5), resetting their tokens to the initial state. This learning process is the same as the one described above, in which the agents respond to stimulus to identify their tokens in the system's order.

After activating the learning process, the agents transmit an *aliveToo* message, indicating their active status in the system (and witch contains the agent's identifier). On receiving the confirmation message (T7), the initiating agent updates the number of conveyors (T8) and adds the agents that sent this message to its list of agents. Once the identification process is complete, the agents start the learning mechanism.

In the case of a conveyor being removed from the system, the terminating agent sends a *leftDF* message indicating its exit. When all other agents receive this message, they remove the agent sender from their agent lists, update the conveyor count and, if they have a token higher than that of the exiting agent, they decrease the value of their token by 1.

4.1.4 Swap Position

The mechanism that identifies the change of order of the conveyor modules requires a slightly different approach, since the count of conveyors is maintained and the new positions have to be discovered. In normal operation, when a part is at the output sensor, a *tokenTransitionOutput* message is transmitted to all the conveyor agents, and it is processed by the next conveyor agent and rejected by the others. When the conveyor agent detects that the part hasn't arrived on the next conveyor (via a timeout), it issues the message *thereIsSwap*, which warns of a possible change of order (and which contains the token where the change took place). As the message is received, all the agents that have a token higher than the one received, will turn on their motor. Afterward, when the part reaches one of the input sensors, the responsible agent updates its current token and

broadcasts a *swapTokenFound* message (with its previous and new token). The agent that was in the position of the swap, receives this message and updates its token.

The swapping mechanism described above works perfectly for all possible position changes of the conveyor agent, except when a conveyor agent is moved to the beginning of the sequence to assume the token equal to 1. Thus, in order to improve the performance of the system under study, a new mechanism was proposed which is responsible for analyzing the swapping of the first conveyor. In it, the first agent always raises a flag when a part is detected by its input sensor. As the part passes through the system and reaches the last module, it sends a *ConveyedAllPieces* message informing that the process has finished and the first conveyor can lower the flag. Once the flag is lowered, the system is able to change the first position, so if the first module is changed, and the agent who had the first token receives a *tokenTransitionInput* message, it evaluates its flag, and in case it is lowered, it changes its token with that of the agent who received the piece.

4.2 Implementation of the MAS System Using JADE

To ensure the development and implementation of the JADE agents, it was necessary to make some settings in the development environment. Firstly, it was necessary to have the Java Runtime Environment (JRE) installed, as JADE applications are based on Java. In addition, it was necessary to choose an Integrated Development Environment (IDE) to facilitate the development of JADE agents, and Eclipse was selected. To make it easier to manage the dependencies required for JADE programming, the MAVEN tool was utilized. Lastly, to launch the graphical user interface and the main JADE container, the 'jade.jar' file was required. Once these initial requirements had been satisfied, the JADE agents are ready to be developed.

To facilitate the development of agents within the JADE environment, the utilization of the Agent superclass was necessary. This superclass provided access to essential methods for initializing and finalizing settings, as well as for adding the behavior responsible for the system's self-organization characteristics. When creating an agent using the Agent

superclass, the implementation of two crucial methods, namely "setup" and "takeDown", became necessary.

The "setup" method plays a pivotal role in executing the agent's initialization configurations. This involves registering its services in the Directory Facilitator (DF) to offer information that aids other agents in comprehending the purpose and availability of the resources it provides. Such registration enhances the discovery and communication process among agents in a distributed environment. Each agent was registered under the service name "skill," with the offered service type being "conveyor." Additionally, during this configuration phase, each agent dispatches an "informAlive" message to all agents sharing the same service. This ensures that agents already operational in the system become aware of the newcomer's entry, prompting them to enter learning mode, as mentioned earlier. The final step in this method involves adding the behavior responsible for the self-organization logic to the agent, named "conveyorBehavior".

On the other hand, when the agent is killed, the "takeDown" method is called. In this method, the termination settings are executed. For JADE agents, when this method is called, two procedures are executed: The first is responsible for removing the agent's service registration in the DF, and the second is to inform the other agents that it is leaving the platform, allowing the other agents to reorganize with its departure. The "leftDF" message sent by this agent contains its Agent Identifier (AID) and the token it had. When this message is received, the agents update their conveyor count, list of agents, and, if necessary, adjust their tokens.

As for the behaviour responsible for the self-organization mechanisms, "SimpleBehaviour" was used. In this behavior three fundamental methods guarantee the system's functionality: 'done', "onEnd" and "action". The 'done' method is responsible for checking that the behavior has met the necessary conditions for conclusion. During each iteration of the main loop, this method is evaluated. If the conditions are fulfilled, it returns true and the behavior is completed. However, as the agents intend to continuously control and monitor their conveyors, the 'done' method always returns 'false', making the 'SimpleBehaviour' action cyclically.

Conversely, the "onEnd" method is invoked only when the behavior is terminated, e.g., when the agent is killed via the main-container. Upon invocation, it triggers the method responsible for finalization settings, the "takeDown".

Finally, the 'action' method encapsulates all the self-organization logic. To enable GPIO port control, the libraries from the PI4J Java API were used. These libraries allow the addition of listeners to the agent's behavior, responsible for detecting changes in the state of input and output photoelectric light sensors. Essentially, these listeners continuously monitor any alterations in the sensors, as well as the motor's state (on or off). This awareness empowers the agent to make real-time decisions for the system's operation. For instance, it can activate or deactivate the motor and dispatch 'tokenTranIn' or 'tokenTranOut' messages when a part enters or exits its conveyor belt. This capability ensures that agents promptly trigger the appropriate mechanisms for the system's smooth operation.

To facilitate self-organized behaviour, the "MessageTemplate" class was used to create a set of templates that validate incoming messages. When it receives a message, the agent compares it with the template, checking for the "Inform" performative act and any of the protocols listed in Table 4.1. After this comparison, the agent processes the message using the 'handleMessage' method. In this method, the agent analyzes the message and determines the appropriate action to take. This includes entering learning mode, activating the mechanisms related to any type of change of position, or managing the addition or removal of an agent (the script for implementing the JADE agents can be seen in Appendix C).

To enable the deployment of the agent on the Raspberry Pi in the modules, it is necessary to perform some initial preparations. First of all, it is necessary to ensure that the system has a Java Runtime Environment (JRE) to be able to interpret the agent's.jar file. In addition, to enable the agent to access the GPIO ports, it is essential to have a Java API installed, in this case the PI4J Java API. Last but not least, you need to upload the package (i.e. the .jar file containing the agent instantiation) into a system directory.

Protocols
tokenTransitionInput
tokenTransitionOutput
informAlive
thereIsSwap
swapTokenFound
leftDF
firstTokenSwapped
conveyedAllPieces
aliveToo

Table 4.1: List of protocols.

4.3 Implementation of the MAS System Using SPADE

In order for the agents developed with the SPADE framework to be able to control the cyber-physical conveyors, initial configurations were required in the development environment. As mentioned earlier, each agent must have a JID and a password to communicate via the XMPP protocol, which is the basis of the SPADE framework. To this end, the open-source ejabberd server was used to enable the exchange of instant messages. By using ejabberd, it is possible to acquire the fundamental characteristics of a distributed system, since it has the features of a server that is cross-platform, distributed, fault-tolerant, and based on open standards to achieve real-time communication [48].

To configure the server, an ejabberd container was created in Docker [49]. The necessary server settings were made using the 'ejabberd.yml' file. Key settings such as 'HOST,' 'PORT,' 'USER,' and 'PASSWORD' were defined in this file. The 'HOST' was set to IPv4, and the 'PORT' was set to the default port for XMPP client-server connections, which is 5222. To allow access as server administrator, the 'USER' and 'PASSWORD' were both set to 'admin' and 'raspberry' respectively. Once the initial settings had been made, it was possible to start the server container and register the JID clients on it. After this stage, the SPADE was configured.

To program the SPADE agents, the first step was to download the framework via a command in the control terminal. It was also essential to have Python version 3.8

or higher installed in the development environment. To simplify the development process, PyCharm was chosen as the integrated development environment, providing a user-friendly platform for writing, debugging, and maintaining the code. Finally, after these initial configurations, the development of SPADE agents became possible.

Since all the conveyor belts in the system share identical functionalities, it was only necessary to create one control logic for the agents. Thus, each agent will have the same behaviors to communicate, identify themselves, and control the conveyor modules. However, even though the agents share this common codebase, they each retain their individuality, allowing them to respond to stimuli from sensors or relevant messages sent by other agents. Consequently, the SPADE agents exhibit two primary behaviors: one is responsible for monitoring and updating the contact list, while the other controls, communicates, identifies, and organizes the CPS within the system.

The first behavior, illustrated in Figure 4.5, was implemented using a "CyclicBehaviour" and it serves two main purposes: registering the agent in the contact lists of other agents and monitoring subscription requests within its own list. When the agent is initialized and the behavior is invoked, it begins by searching a file named "Jid's.txt" (T1) where all the registered JIDs on the server are stored. Once this analysis is completed, the agent initiates subscription requests to be added to the contact lists of other agents (T2). Subsequently, the agent's behavior enters a standby mode to await potential subscription requests (P3). Upon receiving a subscription request (T4), the agent proceeds to authorize it (T5) and then returns to its waiting state.

The second behavior, which contains the primary self-organization logic, was also implemented using a 'CyclicBehaviour.' However, before initializing this behavior, it was necessary to define message templates in the agent's setup. Templates in SPADE are methods used to route incoming messages to the behaviors that are waiting for them [27]. In other words, the templates function as filters, allowing the agent to process a message only if its attributes match those defined in the template. Consequently, the templates used in this project were designed to adhere to the specifications outlined by FIPA-ACL. These templates included two types of parameters: the first one related to

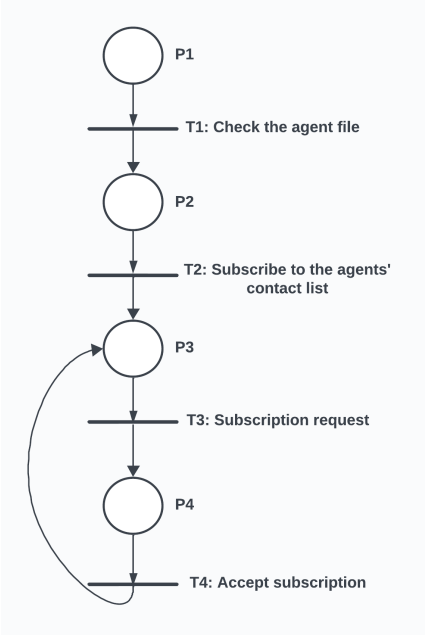

Figure 4.5: Logic control for register behavior.

the performative act of the message, which, in this approach, is *inform*. The second parameter is associated with the protocol used in the messages, which in this case are those shown in Table 4.1, the same as those presented previously.

After assigning the templates to the behavior, the agent adds and executes it. For a 'CyclicBehaviour' developed in SPADE, three methods are essential: 'on_start,' 'on_end,' and 'run.' The 'on_start' method is responsible for initializing the behavior, the 'on_end' method handles the behavior's termination, and the 'run' method contains the main code of the behavior. Therefore, when the 'on_start' coroutine is triggered, it initiates the following tasks: notify other agents that may already be active in the system using the *informAlive* protocol and setting the logic level of the port controlling the conveyor motor to 'LOW.' The control of the Raspberry Pi's GPIOs is achieved using the RPi.GPIO package [50].

On the other hand, when the behavior is killed, the "on_end" coroutine is triggered. As a finalization setting, the agent sends a message using the *leftDF* protocol and with the content consisting of its JID and the token it holds in the system. This message is intended to allow the other agents operating in the ecosystem to reorganize and update

themselves with its exit.

Finally, the method that contains the logical control of agent self-organization is the "run" method. This method is started after the initialization settings have been completed. When executed, the agent can perform the self-organization mechanisms mentioned earlier: initialization, adding or removing agents, swapping positions, and swapping the first conveyor.

However, for the agents to be able to execute these mechanisms effectively, it was necessary to improve the way they exchanged messages. In SPADE, a typical exchanes only three parameters: 'to' (the receiver of the message), 'sender' (the sender), and 'body' (content). To improve this structure, metadata was used to allow parameters to be included in the message content, similar to what happens in FIPA-ACL message communication. Thus, a 'create_message' function was implemented (Algorithm 1) which, based on the values provided, generates a message containing the parameters necessary for communication, including the performative act and the protocol.

Algorithm 1 Create Message

```
1: function createMessage(protocol, content)
2:     msg_template ← new Message()
3:     msg_template.sender ← jid
4:     msg_template.body ← content
5:     msg_template.set_metadata("performative", "inform")
6:     msg_template.set_metadata("protocol", protocol)
7:     return msg_template
8: end function
```

In addition, another fundamental aspect for SPADE agents to execute the mechanisms of self-organization involves the creation of a function to analyze the received messages. This function, named "handle_message", enables agents to determine the appropriate action within the system based on the protocol used in these messages. Actions may include starting or stopping the motor for part transportation, updating their token with the entry or exit of a module, or enabling an alert state in response to any swap conditions in the system's order(the script for implementing the SPADE agents can be seen in Appendix B).

To deploy the SPADE agents on the Raspberry Pi, it is necessary to have the phython interpreter updated to 3.8 or higher and to have installed SPADE in version 3.3.0. In addition, in order for the agents to have access to the Raspberry's GPIO ports, the RPi.GPIO package in version 0.7.1 is used.

4.4 Summary

In the initial part of the project, the aim was always to develop the various self-organization mechanisms in parts, first tested in JADE and then in SPADE. Once each framework had individually met the expectations of self-organization, the second part of the project began. In this part, the challenge was to integrate these two frameworks to achieve self-organization of the system, i.e. the implementation of the middleware, in which the interface agents would allow the agents controlling the modules to communicate independently of the framework in which they were developed.

Chapter 5

Integration of the MAS Applications using the Proposed Middleware Approach

This section covers the development and implementation of the middleware in the proposed system.

5.1 MQTT Middleware

One of the essential components for the development of the proposed middleware is the mechanism by which the interface agents exchange messages with each other. Following the publish-subscribe scheme, an open source broker implementing the MQTT protocol was adopted. Eclipse Mosquitto [51] was the chosen broker and was hosted on a Raspberry Pi connected to the local network where the transportation system was operating. As the Raspberry Pi was dedicated exclusively to running the broker, it required specific configurations to align with the system's needs.

To start the configuration, the Eclipse Mosquitto broker was installed on the Raspberry Pi via the command terminal. After installation, the broker's initial settings were defined using the "mosquitto.conf" file. In this file, the port for broker connections was initially set

to 1883, the default for unencrypted MQTT communication. This configuration served for the initial stages of the middleware's development, as there was no immediate need to protect communication in this environment. However, as the middleware became operational and the interface agents met the system's expectations, changes were made to this file. These changes will be detailed in the subsection dedicated to security.

5.2 Interface Agents

With the successful achievement of self-organization in both frameworks employed in the project, the subsequent implementation of the middleware became possible. At this stage, adherence to the predefined requirements was key to seamlessly integrating the middleware into the transportation system. Consequently, two interface agents were meticulously developed - one adapted for SPADE and the other for JADE. Although these agents could be implemented on any Raspberry Pi device used for self-organization, a strategic decision was made to allocate the interface agents to a dedicated computer. This choice aimed to imbue the system with intelligent distribution characteristics on different devices.

5.2.1 SPADE Interface Agent

Similar to the agents developed to ensure the self-organization of the modular conveyor system, the SPADE interface agent employs two cyclical behaviors to fulfill the objectives of the middleware. The first behavior constructs its contact list within the framework by reading the 'JIDs.txt' file, mirroring the approach used for conveyor agents, and subscribes to and creates its contact list comprised of the cyber-physical conveyor-controlling agents. The 'JIDs.txt' file, containing all agents registered with the XMPP server, necessitates the interface agent to exclude itself from the list and solicit presence subscription requests exclusively to the other agents, as delineated in Algorithm 2. Once the interface agent's contact list has been created, it monitors whether there are any new requests to be authorized if a new agent is added to the system.

Algorithm 2 Process JIDs

```
Open the file "Jids.txt" for reading
Read all lines of the file and store them
for each line in the file do
    Remove whitespace from the beginning and end of the line and store as jid
    if jid is different from the current agent's jid then
        try:
            Subscribe to the presence of jid
            Display "Agent - [AgentName] - Sent a message register to [jid]"
        catch exception (e):
            Display "Unexpected behavior, cause [e]"
    end if
end for
```

As for the main behavior of this agent, which is responsible for translating and sending messages both to the targets in its frameworks and to the other interface agents, the 'CyclicBehaviour' behavior provided by SPADE was used. In this behavior, it was necessary to connect to the MQTT broker. To do this, a function named "client_connection", as shown in Algorithm 3, was generated which is responsible for making the initial configurations of this connection when the "on_start" method is invoked. The function uses the Paho-mqtt library [52] to connect to the broker.

Algorithm 3 Client Connection Setup

```
function CLIENT_CONNECTION(self, broker_ip, port, client_name, user_name,
password, ca_file, certificate_file, key_file, keepalive=60)
    self._mqtt_client ← mqtt.Client(client_name)
    if user_name and password are not None then
        self._mqtt_client.username_pw_set(user_name, password)
    end if
    if ca_file and certificate_file and key_file are not None then
        self._mqtt_client.tls_set(ca_file, certificate_file, key_file)
    end if
    self._mqtt_client.on_connect ← self.on_connect
    self._mqtt_client.on_message ← self.on_message
    self._mqtt_client.on_subscribe ← self.on_subscribe
    self._mqtt_client.connect(broker_ip, port)
    self._mqtt_client.loop_start()
end function
```

In addition, an initial configuration script was created to improve the agent's functionality. Before activating the agent, the script provides all the information needed to connect to the XMPP server and the MQTT broker. This information includes the JID and password for accessing the XMPP server, as well as key details for connecting to the MQTT broker - including the host, port, topics, and user authentication credentials (username and password). The security settings, which are essential for TLS/SSL encryption, are also specified in this file, an example of this script is seen in Figure 5.1.

```
mqtt_broker_configs = {
    "HOST": "192.168.1.235",
    "PORT": 8884,
    "KEEPALIVE": 60,
    "TOPICS": ["JADE/#"],
    "USERNAME":
    ↪  "mqtt_server",
    "PASSWORD": "raspberry",
    "CA_CERTS": "",
    "CERT_FILE": "",
    "KEY_FILE": ""
}
xmpp_configs = {
    "JID":
    ↪  "agent3@jabbers.one",
    "PASSWORD": "99186901br*"
}
```

Figure 5.1: Script to provide initialization parameters.

Once the "client_connection" function has been activated and the consequent connection to the MQTT broker has been made, the "on_connect" callback is triggered. This function is programmed to subscribe the agent to the topics relevant to the frameworks cataloged by the middleware in the initialization script. In the scope of this project, the subscription initially focuses on the topics associated with the JADE framework. Once this subscription has been made, the SPADE interface agent is enabled to receive messages from the MQTT broker. Thus, these messages can then be interpreted by the interface agent so that it can forward them to the agents in its own framework, ensuring effective

communication and interaction within the multi-agent system ecosystem.

Once the main 'run' method of the cyclic behavior is invoked, the SPADE interface agent is ready to process the messages. Messages can originate either from agents in the same framework or from the MQTT broker, the latter being picked up by the 'on_message' callback. Therefore, as illustrated in Algorithm 4, when it receives a message from the broker, the agent performs several steps: first, it decodes the message payload and separates the relevant topics, identifying the framework and the sending agent. Subsequently, the message in JSON format is interpreted and translated using the 'handle_msg_interface_ag' function. The 'handle_msg_interface_ag' function is responsible for parsing the JSON, mapping it to a dictionary that encapsulates the message parameters. After successful conversion, the resulting dictionary is added to an internal message queue. This accumulation allows the agent to proceed with sending these messages in the agent's main execution loop.

Algorithm 4 Handle Incoming MQTT Message

1: $message\_received \leftarrow$ None
2: $mqtt\_message \leftarrow message.payload.decode()$
3: $topic \leftarrow message.topic.split("/")$
4: $mqtt\_json \leftarrow json.loads(mqtt\_message)$
5: $message\_received \leftarrow self.handle\_msg\_interface\_ag(mqtt\_json, topic)$
6: **if** $message\_received \neq$ None **then**
7: $\quad self.\_message\_to\_send.append(message\_received)$
8: **end if**

In the main loop, the SPADE interface agent is waiting for two types of stimulus: messages from its own framework and messages in the queue received via MQTT to be forwarded to the corresponding agents. When the message queue is not empty, indicating the presence of pending messages, the agent starts the sending process. This process begins by consulting its contact list using the get_contacts() method provided by SPADE. Once the contacts have been obtained, the message at the top of the queue is passed to the "send_message" function.

The "send_message" function is responsible for distributing the message to each agent in the contact list. Before sending, it converts the contents of the dictionary into a format

compatible with that adopted by SPADE, ensuring that the necessary metadata is present and correctly formatted. This step is essential so that the agents controlling the cyber-physical conveyors can interpret and act on the instructions received. This process is repeated until all the messages in the queue have been delivered.

On the other hand, if the agent receives a message from an agent in its framework, as seen in Algorithm 5, it does the opposite. Firstly, it passes this message as a parameter to the "handle_msg_interface_ag" function, which in turn maps the message sent by the agent in its framework and transforms it into a dictionary containing the parameters of the FIPA-ACL standard as the key and the content of the message sent by the agents controlling the conveyors as values. Once this is done, the interface agent prepares the topic in which the message will be published, converts the dictionary into JSON and publish it in the topic and subtopic related to the framewokr and agent that sent the message, respectively. (the code for implementing the SPADE interface agents can be found in Appendix D).

Algorithm 5 Process and Publish Received Message From its Framework.

1: $message\_received \leftarrow$ await $self.receive()$
2: **if** $message\_received \neq$ None **then**
3: $spade\_message \leftarrow self.handle\_msg\_interface\_ag$(spade_message=$message\_received$)
4: $topic \leftarrow$ "SPADE/" + str($spade\_message$[”$sender$”])
5: $message\_json \leftarrow json.dumps(spade\_message)$
6: $self.\_mqtt\_client.publish$(topic=$topic$, payload=$message\_json$, qos=0)
7: **end if**

5.2.2 JADE Interface Agent

To implement the interface agents in JADE, similar to the conveyor agents, a "Simple-Behaviour" was used to activate the mechanisms that guarantee interoperability. As the interface agent requires registration in the DF for searches and to allow the agents controlling the conveyors to locate it, two arguments (the name and type of the service it offers) must be passed when starting the agent in the container. To facilitate access to the messages exchanged by the conveyor agents, these two arguments have been defined

to be identical to those used by the conveyor agents, as shown in Figure 5.2.

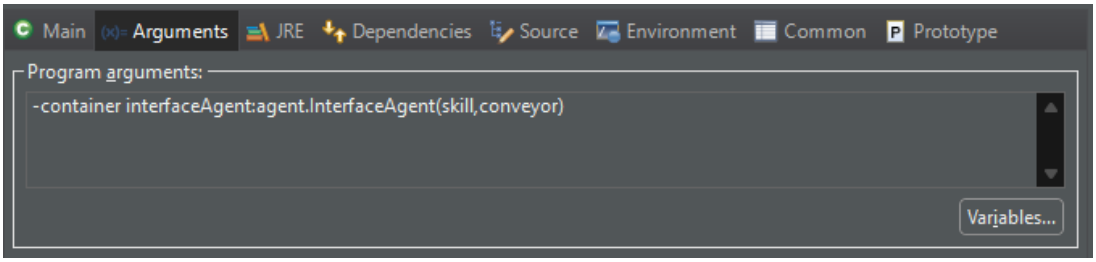

Figure 5.2: Arguments to start the interface agent.

Therefore, once the agent is started and registered in DF, it adds and executes the behavior responsible for monitoring messages and ensuring that messages are sent and received via the broker and intra-framework. Once the behavior had started, the interface agent needed to connect to the middleware. To do this, it used the Eclipse Paho-MQTT library [53] to make the connection and to set up the call-back responsible for handling the receipt of messages. Thus, once the "action" method of the behavior is started, the agent searches for the file containing the information pertinent to the broker and makes its connection, subscribing to the topics related to the frameoworks belonging to the middleware catalog and establishing the call-back for the messages received. Once this is done, the agent enters message waiting mode, i.e. waiting for messages from the intra-framework agents controlling the modules or for messages published by the SPADE interface agent regarding the SPADE agents controlling the other conveyor modules.

Once the message is received by the middleware, the call-back is triggered. Thus, according to algorithm 5, the payload is received and the JSON containing the parameters following the FIPA-ACL standard is mapped to the "handleSpadeToJade" function which is responsible for analyzing the messages received and translating them into a new message following the standards adopted by JADE. Once the message has been created, it is added to a queue of messages to be sent.

When it returns to the main loop, if there are any messages in the queue, the agent searches the DF for all active agents controlling the conveyors and sends the message. This is repeated for all the messages that have been added to the message queue.

Algorithm 6 Handle Arrival of MQTT Message

```
1: function MESSAGEARRIVED(topic, message)
2:     topics ← topic.split("/")
3:     aclMessage ← new ACLMessage(ACLMessage.INFORM)
4:     mqttMessage ← new String(message.getPayload())
5:     msg ← new JSONObject(mqttMessage)
6:     jsonMessage ← createAclJson(msg)
7:     aclMessage ← handleSpadeToJade(jsonMessage, topics)
8:     if aclMessage ≠ null then
9:         messageQueue.add(aclMessage)
10:    end if
11: end function
```

On the other hand, when the agent receives a message from its own framework, the JADE interface agent triggers the "handleJadeToSpade" function. In this function, the agent changes the message received from the agents controlling the conveyors into JSON format following the FIPA-ACL standard and publishes it in the broker following the (framework/agent) pattern where the framework is related to its own framework, and the agent is related to the agent controlling the conveyor. The code for the JADE interface agents is available in the Appendix E.

5.3 Security

As the proposed middleware depends on the MQTT protocol for interoperability between interface agents, it is essential to guarantee robust security and privacy. By taking these measures, it is possible to guarantee the confidentiality of the information exchanged between the interface agents, preventing unauthorized entities from gaining access to it and harming the middleware's functionality. To address these concerns effectively, a complete analysis of the system under study was carried out, with the aim of identifying possible vulnerabilities and weak points susceptible to security threats.

Therefore, utilizing the STRIDE methodology allowed for the identification of risks associated with the middleware as interface agents exchanged messages, subsequently facilitating the proposal of mitigations. Thus, for the **spoofing** threat, the following

considerations were made:

- **Attack**: Pretending to be an interface agent.
- **Risk**: The risk involves the creation of a fraudulent interface agent that imitates a legitimate interface agent. This fraudulent agent can then disseminate false information on the topics, leading other interface agents to receive and propagate the false message in their respective frameworks resulting in unintentional actions.
- **Mitigation**: This threat can be mitigated by adding username and password authentication to the broker so that only interface agents with this information can connect.

In terms of **tampering** threats, the following possibilities were made:

- **Attack**: An attacker tries to modify the messages in transit between the interface agents.
- **Risk**: By modifying the data sent from one interface agent to another, the attacker can produce false data that can affect the total functioning of the system in which the interface agents are inserted, e.g. by transmitting a message saying that a conveyor belt has been removed, while it is still active.
- **Mitigation**: For this attack, the mitigation adopted could be to encrypt the messages exchanged between the interface agents and the MQTT broker, making sure that they are not understood if accessed by an unauthorized entity.

Regarding the **repudiation** threats, it is possible to find the following possibilities:

- **Attack**: An interface agent denies having sent or received a message.
- **Risk**: If the interface agent denies receiving the message, it exhibits bad behavior that affects the system, as it does not broadcast the received message to the agents in its framework.

- **Mitigation**: Implement logging and auditing mechanisms to track transactions between interface agents.

With regard to threats to the **information disclosure**, the following possibilities can be analyzed:

- **Attack**: Sniffing topics.
- **Risk**: An unauthorized entity can access all the data exchanged by the interface agents.
- **Mitigation**: As mitigation, it is also possible to encrypt the messages exchanged between the interface agents.

Regarding the **DoS threats**, it is possible to find the following possibilities:

- **Attack**: Flooding topics.
- **Risk**: An attacker sends a large number of messages to a specific topic, overloading the interface agent and forcing it to process a huge number of messages, leaving no processing available for the agent to broadcast messages.
- **Mitigation**: Set up firewalls and access control lists to filter the traffic of messages posted in the topics.

In the context of the proposed middleware, all interface agents have the same privileges, so the **elevation of privileges** was not evaluated.

After analyzing the main risks that interface agents could suffer when exchanging messages using the broker, some measures were adopted to protect the integrity of the messages transmitted. Thus, two protection mechanisms were adopted, the user authentication and TLS/SSL protocols.

To allow user identification, it was necessary to modify the file "mosquitto.conf" from the broker to deny anonymous connections to the broker and register the allowed clients. Otherwise, for TLS/SSL certificates, it is common to use external Certificate Authority

(CA), which provide useful information to reliably identify the server (broker when it comes to MQTT) before communication begins. It's also possible to use a more economical solution: create self-signed keys and certificates for the interface agent and broker with a local CA using the open-source tool OpenSSL [54]. After the certificates were generated by OpenSSL, the paths for the broker certificate, the private key and the CA file were added to the initialization file 'mosquitto.conf'. Additionally, the default port for MQTT communication using TLS/SSL was changed from 8883 to 9100 to make the presence of the MQTT server less obvious.

These adopted approaches make it possible to mitigate several STRIDE modeling categories, however, some of them cannot be mitigated using these mechanisms only, but they will be addressed in future works.

5.4 Visualization of the Modular Conveyor System Using the Middleware

To better visualize how the system worked and to differentiate which agent was controlling the conveyors, a graphical interface was developed using the Node-RED platform [55]. Node-RED is a flow-based programming tool that allows building a control panel using a comprehensive set of nodes, which when used in combination simplifies the development of IoT applications. The Node-RED dashboard allows building a control panel capable of displaying graphs, presenting data at defined time intervals or in real-time, as well as allowing the manipulation of that data in a friendly interface [55]. Figure 5.3 illustrates the flows used to create the user interface used to identify the MAS frameworks and the agents that control the conveyors.

In this diagram, the input node (shown in purple) is key in establishing the connection between the messages published via MQTT and the Node-RED tool. On the other hand, the terminal nodes (highlighted in blue) are responsible for disseminating the messages on the dashboard. Meanwhile, the conveyor function nodes, identified as 1, 2, 3 and 4,

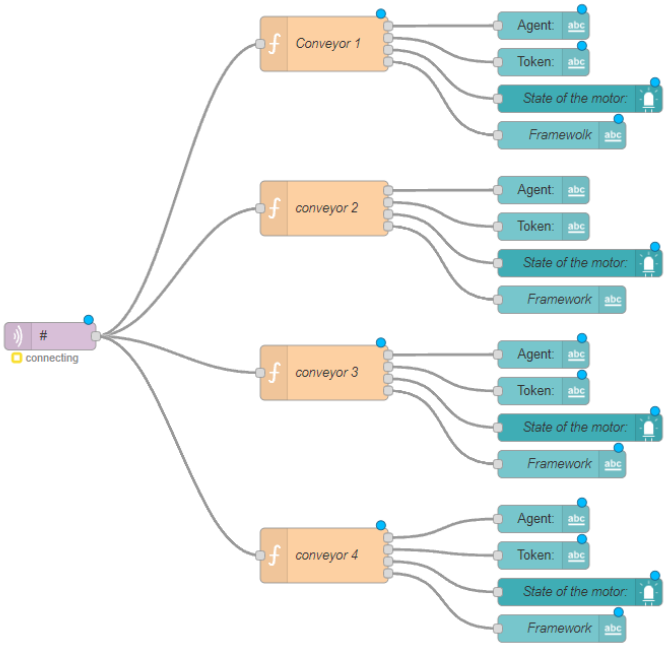

Figure 5.3: Node-RED flow for the development of the graphical interface.

have the task of processing the messages received by the MQTT. Their function involves identifying the content of the message and updating, if necessary, the nodes responsible for displaying the information to the user interface.

As the interface agents use the MQTT protocol to publish messages and share information, it was necessary to configure the node (purple) so that it could access all the data published by the interface agents. To do this, as shown in Figure 5.4, it was necessary to provide the IP where the broker was installed (10.20.38.38), the port used (9100) and the QoS used for the connection (QoS 0). With regard to the topics to be signed, the multilevel wildcard # was used, allowing Node-RED to access all the messages published in the broker.

Once the message is received, the conveyor function nodes analyze the content of the message. First of all, the nodes analyze the framework from which the message was sent. From this, they identify the agent sending the message and associate the content of its message (token) with the fixed value defined in each node. In addition, the nodes analyse which protocol was used in the message in order to turn on or off the

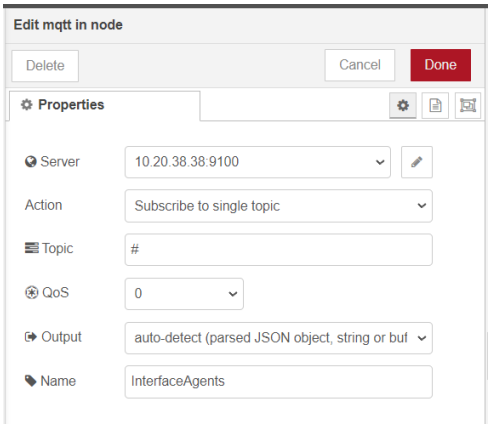

Figure 5.4: Connecting the node MQTT input to the MQTT broker.

LEDs responsible for symbolizing the conveyor motors. Therefore, as the interface agents publish the messages in the topics, the conveyor nodes are able to update the outputs so that the dashboard is updated in real-time as the part is transported between the conveyors. Another point to consider is that when the system is started, the agents have no position, so the dashboard displays the IDLE message on the node representing the token. An example of the dashboard in operation is shown in Figure 5.5.

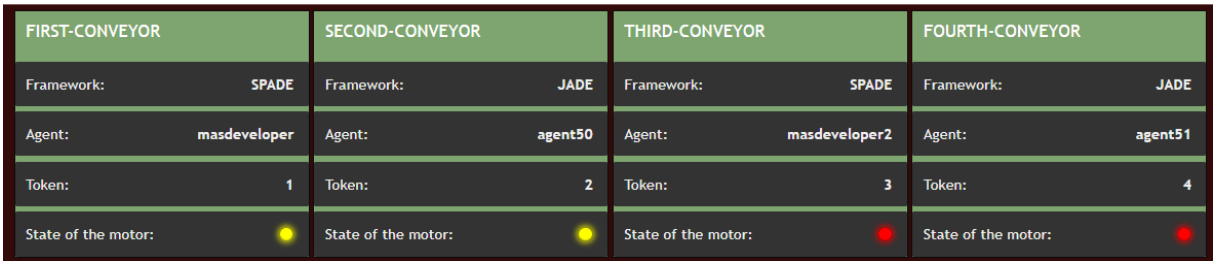

Figure 5.5: Graphical interface to monitor the system conditions.

As the proposed middleware uses TLS/SSL protocols to encrypt and authenticate clients and the MQTT server, this protection was also adopted on the Node-RED platform uploading the client certificate, private key and CA in the purple node settings. In addition, user authentication to the admin page was created to protect the graphical interface from a possible sniffer, as shown in Figure 5.6.

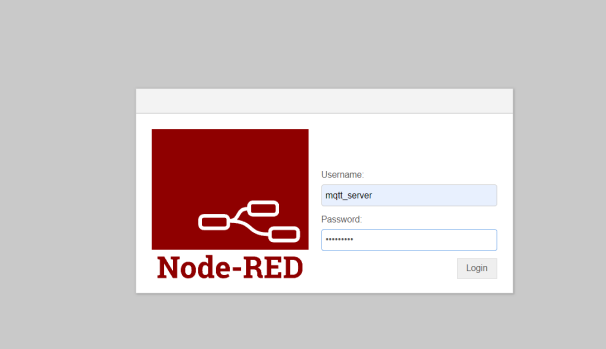

Figure 5.6: Login screen to access the Node-RED admin page.

5.5 Experimental Tests

In order to evaluate the performance of the middleware developed, which enables interoperability between frameworks such as SPADE and JADE, an analysis was carried out divided into three essential parts. The first focuses on the ability of the modules to self-organize as the two frameworks operate simultaneously on the system. The second is related to the latency overhead introduced by the middleware during communication between agents of the different frameworks, which is essential to understanding the impact of the middleware on the system's response time. Finally, the third part focuses on the system's scalability, with a specific analysis of the interface agents in different workload scenarios. The hardware used to carry out these tests is listed in Table 5.1.

Component	**Hardware**
Broker MQTT	Raspberry Pi model 3 Quad Core 1.2GHz 64bit CPU 1GB RAM
SPADE Interface agent	Acer Nitro 5 Laptop (Intel Core i7-8750H CPU @ 2.20GHz, 16 GB RAM)
JADE Interface agent	Acer Nitro 5 Laptop (Intel Core i7-8750H CPU @ 2.20GHz, 16 GB RAM)
agents SPADE	Raspberry Pi model 3 Quad Core 1.2GHz 64bit CPU 1GB RAM
agents JADE	Raspberry Pi model 3 Quad Core 1.2GHz 64bit CPU 1GB RAM

Table 5.1: Hardware used for the tests

5.5.1 Robustness and Self-Organization

As mentioned earlier, the JADE interface agent was registered with the DF using the same criteria applied by the agents supervising the conveyors. This approach was chosen to speed up the process of sending messages, eliminating the need for the module's agents to search the DF twice. Consequently, every time a message was exchanged, the interface agent was promptly informed of the content and efficiently transmitted it to the designated topic.

On the other hand, SPADE does not have a DF for agents to search for and find specific services. In this scenario, the interface agents rely on the contact list provided by the XMPP protocol, which is adopted for the exchange of messages in its framework. Consequently, the interface agent can create a contact list containing all the agents that control the conveyors and their states. This approach allows interface agents to receive and publish messages during the part exchange process on the relevant topics.

Therefore, once the middleware was running and the interface agents were in operation, it was possible to carry out the self-organization tests of the modular cyber-physical system operating with agents developed in two different frameworks. To do this, all the mechanisms responsible for enabling the system to self-organize were tested, the order learning mechanism when the system is initialized, the mechanism for the system to adapt when there is a plug-in or plug-out of a module, and finally the swapping mechanism when there is a change in the order of the modules. The tests were therefore conducted as follows:

- Initially, one SPADE and three JADE agents were used in the tests, as shown in Figure 5.7, and the tests mentioned earlier were carried out: the learning part, removal of a module and the swap of positions.

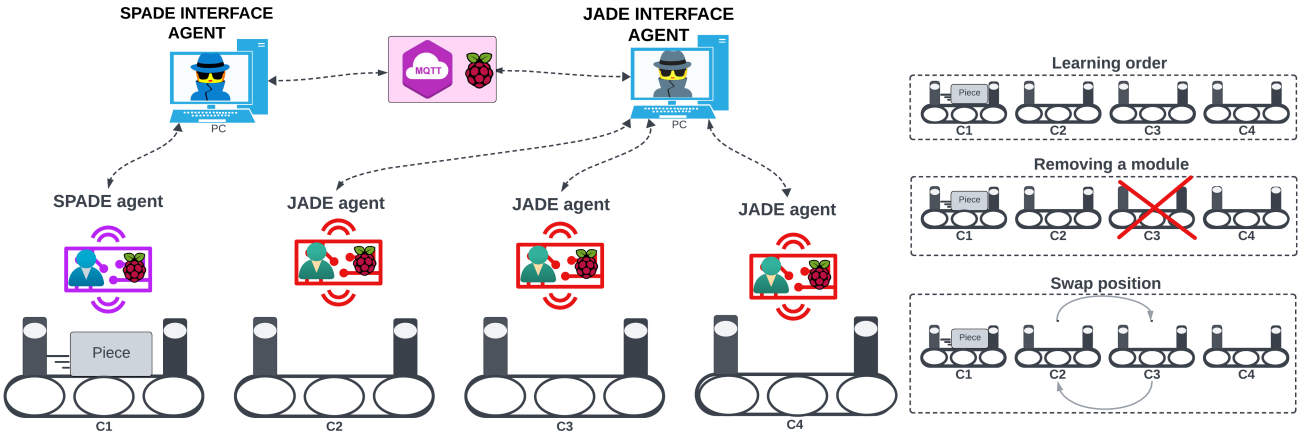

Figure 5.7: One SPADE agent and three JADE agents.

- Them, two agents in SPADE and two in JADE were used to control the modules and handle the self-organization mechanisms, as seen in Figure 5.8.

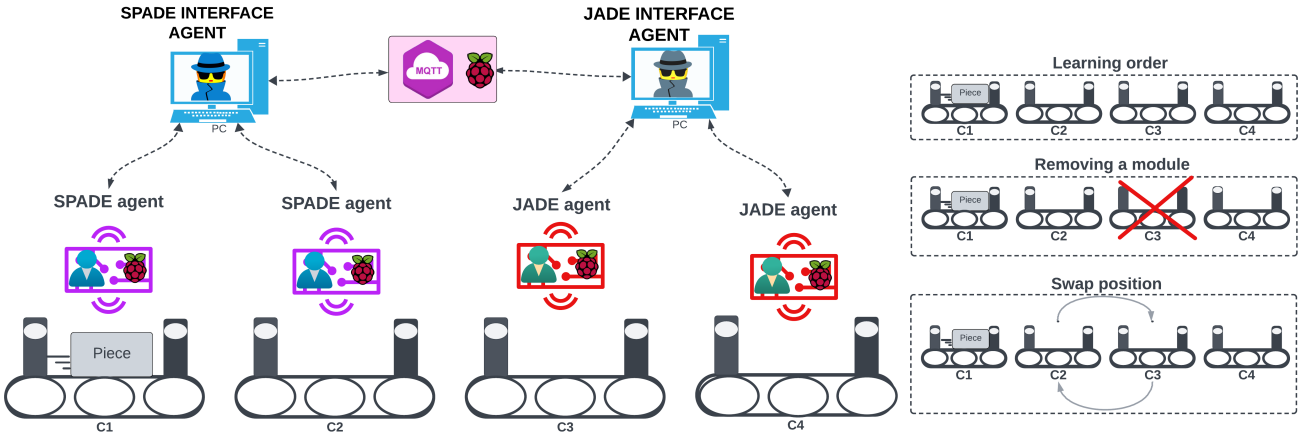

Figure 5.8: Two SPADE agents and two JADE agents.

- Finally, three agents in SPADE and one in JADE were used to handle the self-organization mechanisms, as seen in Figure 5.9.

The series of tests conducted served to validate the precise reception of messages by the conveyor agents through the middleware and interface agents. In these tests, the cyber-physical modules were intentionally removed, swapped, and reintegrated into the

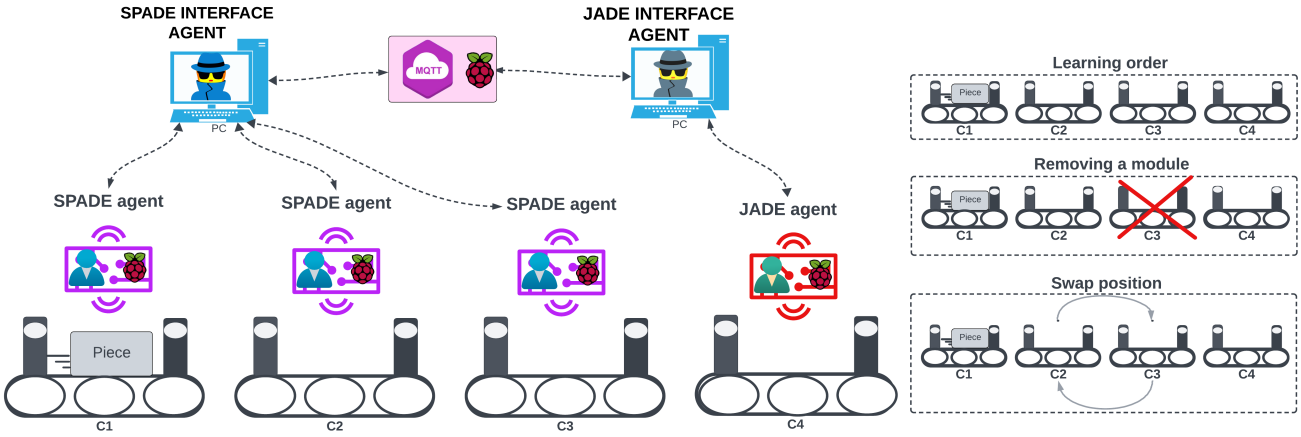

Figure 5.9: Three SPADE agents and one JADE agent.

system to challenge and assess the adaptability of both frameworks to dynamic changes. This evaluation aimed to confirm the efficacy of the proposed middleware in terms of accurate message routing, translation, and publication, which are critical for the seamless operation of the system.

After the tests were completed, the results were clear: the middleware, together with the interface agents, allowed the agents controlling the conveyors developed in SPADE and JADE to demonstrate the expected self-organization capabilities. Thus, the conveyor agents were able to recognize each other and organize themselves autonomously in the order of the conveyor modules. In addition, they were able to efficiently update their positions in response to the insertion or removal of modules and were also able to efficiently reconfigure themselves when the sequence of modules in the system changed. Therefore, it demonstrates the robustness of the proposed mechanism to achieve interoperability between systems once it was possible to operate both frameworks simultaneously.

5.5.2 Overhead Introduced When Using the Middleware

In order to carry out an analysis and identify possible bottlenecks as the middleware is inserted into the system, it was necessary to prepare test scenarios. To do this, this first test was divided into three parts: the first would be the latency associated with

communication between the agents developed in SPADE; the second would consist of calculating the latency in the exchange of messages between the agents developed in JADE; and the last part would be to check the latency associated with the insertion of the middleware so that a JADE agent can communicate with a SPADE agent.

Therefore, to carry out the first two tests, two Raspberry Pi embedded with the SPADE and JADE agents were used. To ensure the accuracy of the measurements, the Raspberry Pi clocks were synchronized using an NTP server. For the tests, a 300 bytes message was generated and sent 100 times to collect the time taken to send the message and the time taken to receive it, so that the latency involved in exchanging messages could be calculated. To calculate the sending and receiving times, timestamps were added to the agents' code to record the time at which the message was sent and received.

After conducting the first two tests, the middleware was added so that the latency overhead metrics introduced with the middleware could be collected. To do this, a dedicated Raspberry was used for the MQTT broker, while the interface agents were run on an Acer Nitro 5 Laptop. Similarly to the intra-framework tests, a message of approximately 300 bytes was generated and sent from a SPADE agent on one Raspberry to a JADE agent on the other Raspberry. It is important to note that, in this test, the overhead introduced by the middleware takes into account the processing time of each interface agent, as well as publishing and receiving using the MQTT broker. The results of these tests can be seen in Figure 5.10.

After analyzing the data, it was found that intra-framework communication in JADE is significantly faster compared to SPADE. The average latency for sending messages between JADE agents is around 4.82 milliseconds, while in SPADE, this time is around 70.23 milliseconds, making communication between JADE agents around 14 times faster than between SPADE agents. When evaluating the use of middleware, there is an average latency of approximately 87.45 milliseconds for transferring messages from a SPADE agent to a JADE agent. This process involves going through the SPADE interface agent, the MQTT broker and the JADE interface agent. The observed increase in latency is justifiable, considering the multiple processing stages that the message undergoes: it needs

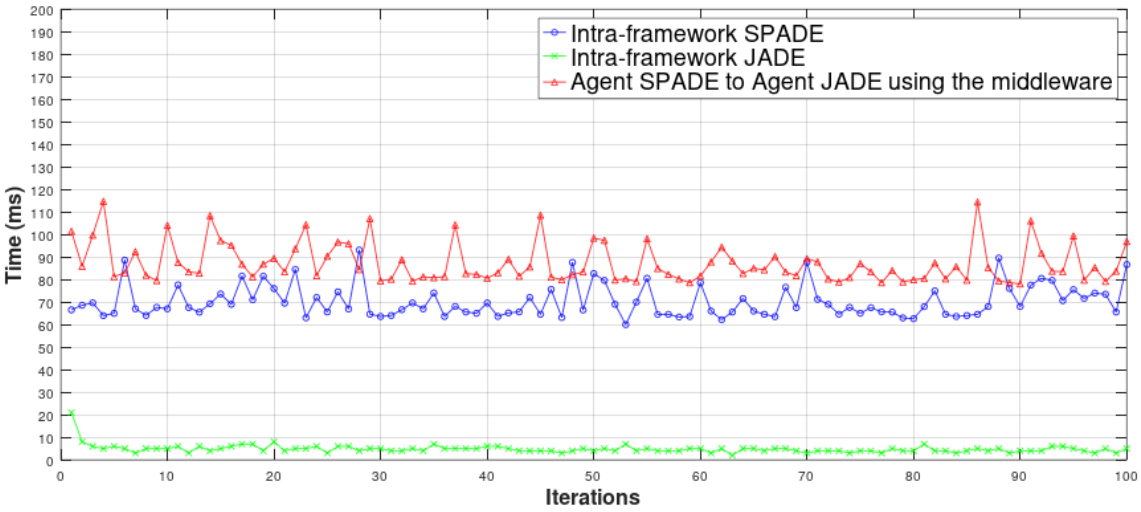

Figure 5.10: Overhead introduced into the system when using the middleware.

to be handled by the native protocols of each framework, transformed by the interface agents, forwarded by the MQTT broker and, finally, converted and delivered by the receiving interface agent. This detailed flow illustrates the complexity of the inter-framework communication process and explains the additional latency overhead.

Furthermore, the adoption of the MQTT protocol as part of the middleware proved to be satisfactory, since the overhead introduced into the system when using the middleware was approximately 24.52% higher than communication using only the XMPP protocol adopted by the SPADE agents. According to [56], this high latency presented by XMPP when compared to MQTT is due to the protocol reading and sending structured messages in the stanza XML format, which results in additional latency for the system, making it slower than other protocols. Therefore, the high latencies attributed to using middleware in this case study are mainly associated with the protocol adopted by SPADE.

5.5.3 Middleware Scalability

In this test, the focus was on the interface agents and how they behave as the middleware increases the number of messages, evaluating the time taken by the interface agents to process and deliver messages to the agents in their own framework. In addition, the

interface agents' ability to process and deliver messages as the number of agents in their own framework is increased will be evaluated. To carry out these tests, the Apache JMeter[57] tool was used to simulate an increase in the traffic of messages published in the MQTT broker, and to simulate an increase in the number of agents in the system, the test was carried out locally on the Acer Nitro 5 Laptop.

Thus, in order to publish to the MQTT broker, a test plan had to be set up in JMeter, as illustrated in Figure 5.11. The test plan was structured using three main elements: the 'Once Only Controller', which is configured to connect to the MQTT broker only once for each test execution thread; the 'Loop Controller', which controls the number of iterations of the publications to be made in the broker; and a 'Once Only Controller', designed to disconnect JMeter from the MQTT broker after the publications have been completed.

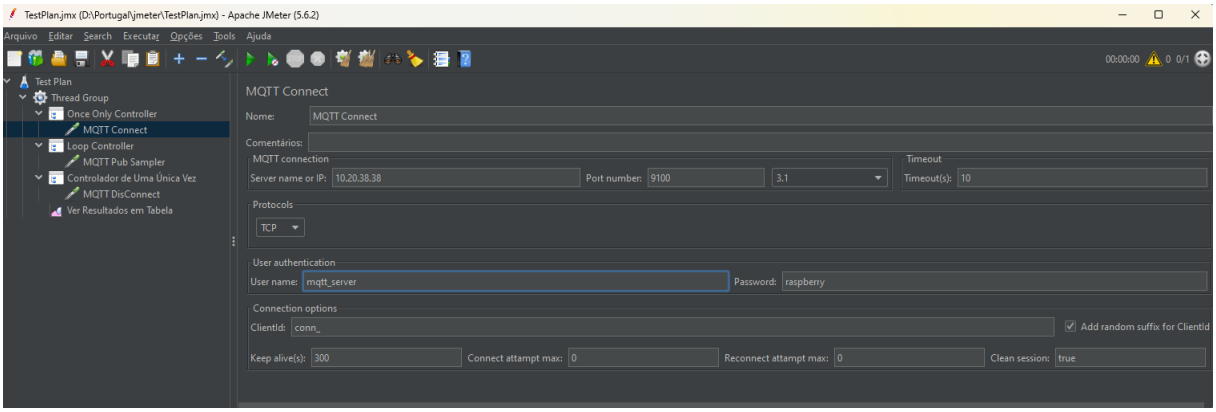

Figure 5.11: Setting up the test plan in JMeter to simulate publications in the MQTT broker.

After configuring JMeter, the test scenario for the interface agents was defined in order to assess their ability to process and deliver the messages received from the MQTT broker to the corresponding agents in the framework. As described in Table 5.2, the number of messages was progressively doubled in each test, starting at 10 and ending at 640. In addition, the Target Cycle Time was set to 'freewheeling' mode, i.e. configuring JMeter to transmit messages to the broker at the highest possible speed, avoiding any kind of programmed latency.

The tests were then run in three different scenarios to observe performance under

different loads. In the first scenario, the interface agents were tested with 10 target agents. At the end of this stage, the number of target agents was increased to 20 and then to 30, making it possible to evaluate the scalability and processing capacity of the interface agents as the workload increased. Furthermore, to ensure the reliability of the results, 10 simulations were carried out for each test configuration and the averages were calculated to mitigate any network-related variances.

Test	Number of Messages	Message Size (Bytes)	Target Cycle Time
1	10	300	freewheeling
2	20	300	freewheeling
3	40	300	freewheeling
4	80	300	freewheeling
5	160	300	freewheeling
6	320	300	freewheeling
7	640	300	freewheeling

Table 5.2: Test Parameters.

Therefore, once the scenarios had been defined and configured, it was possible to carry out the tests on the interface agents. Initially, tests were carried out on the JADE interface agent and then on the SPADE interface agent. Figure 5.12 and Figure 5.13 show the results obtained from the analysis of each interface agent.

Thus, once the data had been analyzed, it was possible to notice that the JADE interface agent was more efficient than the SPADE agent. For example, in the scenario in which there are 20 messages destined for 20 agents, the SPADE interface agent records an average latency of 1553 ms, approximately 9.6 times greater than the latency of the JADE interface agent, which maintains a latency of only 161.4 ms. This initial result already positions the JADE framework and the JADE interface agent as a considerably more efficient system in terms of response time.

Furthermore, as the scenario is scaled with a higher volume of messages, the SPADE interface agent's latency increases dramatically, suggesting an inability to scale effectively under load. For example, when analyzing performance with 80 messages sent to 10 agents, the SPADE interface agent shows a latency of 6451 ms, almost 18 times slower than the

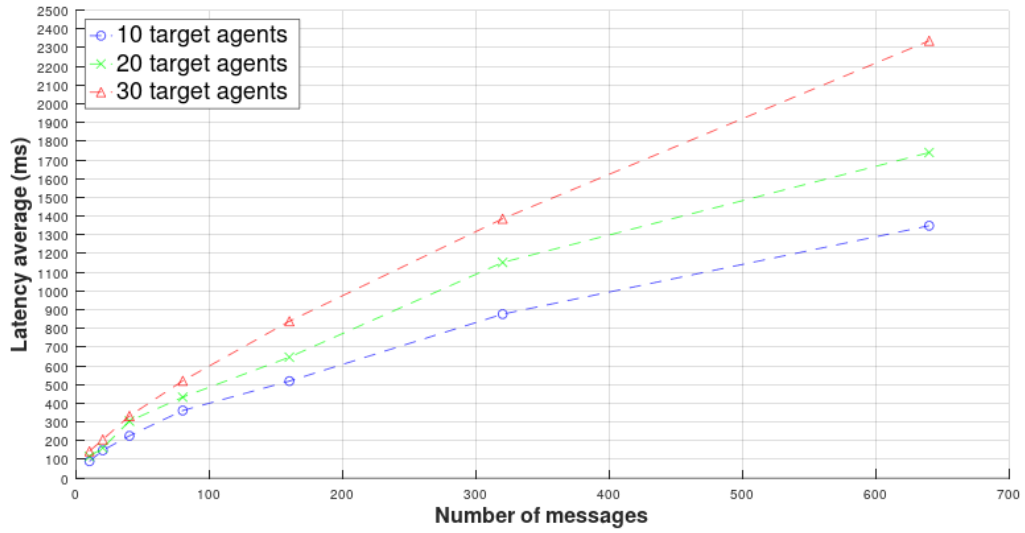

Figure 5.12: Response time for the transmission of data from the JADE interface agent to agents in its framework.

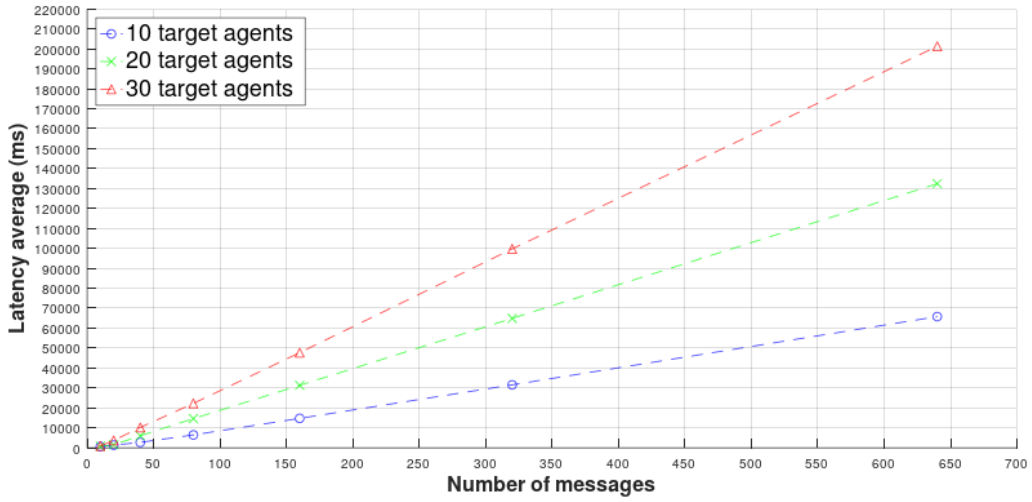

Figure 5.13: Response time for the transmission of data from the SPADE interface agent to agents in its framework.

JADE interface agent, which shows a latency of 361.1 ms. This tendency for the SPADE interface agent to exhibit significantly high latency indicates problems when it has to work under stress.

When analyzing 320 messages destined for 20 agents, the SPADE interface agent reaches a latency of 31301 ms, which is completely impractical for most applications, especially those that require quick responses. In contrast, the JADE interface agent has a latency of 646.2 ms, showing a throughput of around 494.50 messages per second and demonstrating its ability to handle high loads.

Finally, the most extreme scenario tested, with 640 messages for 30 agents, further highlights the superiority of the JADE interface agent. The SPADE interface agent records an average latency of 201299 ms (over 3 minutes), while the JADE interface agent maintains a latency of 2336.7 ms (approximately 2.3 seconds). In addition, the throughput in the most intense scenario for the JADE interface agent is around 273.89 messages per second, while the throughput of the SPADE interface agent is around 3.18 messages per second, making it unviable for a scenario with a high volume of messages over a very short period of time.

So, although the SPADE interface agent was sufficient for the scope of this project where the volume of messages is low. If it is necessary to use it for other applications with higher demands, one solution would be to implement and test horizontal scaling, i.e. adding more SPADE interface agents to distribute the message load more effectively.

Finally, it was possible to see that the JADE interface agent was able to cope with the increase in the number of messages and target agents, maintaining low latency even in the most extreme scenario tested.

Chapter 6

Conclusion and Future Work

In the era of Industry 4.0, the integration of revolutionary technologies such as the IoT, Cloud Computing and robotics is transforming production systems, promoting significant advances in quality and efficiency. Central to this transformation are CPS, which benefit immensely from MAS due to their ability to distribute intelligence and decentralize control, effectively handling the complexity and dynamism of modern production environments. However, a critical challenge arises when different CPS use different MAS frameworks in a process: the interoperability. This challenge is characterized by the way in which each MAS framework was created, since they may be designed with different programming languages, communication protocols and data structures. This means that communication and cooperation between different CPS in this scenario are limited due to the particularities of each MAS framework.

In this context, this work proposed developing a mechanism to enable interoperability and communication between different MAS frameworks, allowing a modular cyber-physical conveyor system to operate harmoniously with different frameworks to transport parts between the conveyor modules.

To achieve the desired interoperability between the SPADE and JADE frameworks, it was initially essential to implement self-organization logic, allowing modular cyber-physical systems to demonstrate advanced self-organization capabilities. This logic allowed the agents to initialize the system efficiently, identifying and learning the sequence

of operations as a token was transmitted between the modules. In addition, the agents were able to adapt to the insertion or removal of modules and successfully reorganized themselves when the configuration of the conveyors changed, thus demonstrating the effectiveness of the self-organization mechanisms implemented. Once self-organization had been established in each individual framework, it became possible to develop the proposed middleware, which was responsible for allowing the agents controlling the conveyors to operate the system independently of the framework used.

By applying the proposed middleware to the system, it was able to provide interoperability between the two frameworks in the self-organization process efficiently. The interface agents developed were able to translate, process and send messages, either from their framework or from the topics to which they were subscribed, enabling the system to operate correctly and achieve the characteristics of self-organization, without the need to stop, reprogram or even restart the system.

In terms of middleware security, after carrying out threat modeling based on the STRIDE methodology, it was possible to identify the main risks it was subjected to and it was possible to apply protection mechanisms, such as user authentication and the use of keys and certificates to encrypt communications based on the TLS/SSL protocol.

In relation to the tests carried out on the middleware, it was observed that the latency introduced into the system is mainly associated with the XMPP protocol used by SPADE, and not the MQTT protocol used by the middleware, as demonstrated by [56] when comparing these two protocols. Furthermore, individual analysis of the interface agents revealed that the JADE interface agent is robust and scalable, capable of supporting high transfer rates with low latency, which qualifies it as ideal for applications that require efficient real-time communication. On the other hand, although the SPADE interface agent is suitable for the current project, which has a low volume of messages, it has significant limitations that should be considered for future applications in scenarios with higher performance requirements.

Therefore, the middleware proposed in this study proved to be an effective solution to the challenge of interoperability between the SPADE and JADE frameworks, increasing

the ability of heterogeneous systems to manage a cyber-physical system in a self-organized way. The use of the MQTT protocol and the JSON format, in accordance with the FIPA-ACL messaging standards, has proved to be a robust and flexible approach to exchanging information between different agents and platforms. This approach not only benefits the integration between SPADE and JADE, but also extends to other frameworks, since it is only necessary to create another interface agent responsible for this new framework.

Finally, the creation of the graphical interface allowed for optimum visualization and monitoring of the conditions of the agents controlling the conveyors, which proved to be very efficient.

Future work will be devoted to:

- Develop more interface agents in other MAS frameworks to increase the catalog of interface agents offered by the middleware.
- Implement logging and auditing mechanisms to track transactions between interface agents to make the middleware more secure.
- Implement and test horizontal scaling in the SPADE interface agents to distribute the message load more effectively.

Bibliography

[1] R. Y. Zhong, X. Xu, E. Klotz, and S. T. Newman, "Intelligent manufacturing in the context of industry 4.0: A review," *Engineering*, vol. 3, no. 5, pp. 616–630, 2017.

[2] *Industrial IoT Connections to Reach 37 Billion Globally by 2025, as 'Smart Factory' Concept Realised*. [Online]. Available: `https://www.juniperresearch.com/press/press-releases/industrial-iot-iiot-connections-smart-factories` (visited on 11/30/2023).

[3] P. Leitão, A. W. Colombo, and S. Karnouskos, "Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges," *Computers in industry*, vol. 81, pp. 11–25, 2016.

[4] J. Lee, B. Bagheri, and H.-A. Kao, "A cyber-physical systems architecture for industry 4.0-based manufacturing systems," *Manufacturing letters*, vol. 3, pp. 18–23, 2015.

[5] P. Leitão, J. Barbosa, G. S. Funchal, and V. Melo, "Self-organized cyber-physical conveyor system using multi-agent systems," *International Journal of Artificial Intelligence*, 2020.

[6] P. Leitão, "Agent-based distributed manufacturing control: A state-of-the-art survey," *Engineering applications of artificial intelligence*, vol. 22, no. 7, pp. 979–991, 2009.

[7] T. Burns, J. Cosgrove, and F. Doyle, "A review of interoperability standards for industry 4.0.," *Procedia Manufacturing*, vol. 38, pp. 646–653, 2019.

[8] O. Givehchi, K. Landsdorf, P. Simoens, and A. W. Colombo, "Interoperability for industrial cyber-physical systems: An approach for legacy systems," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 6, pp. 3370–3378, 2017.

[9] L. Da Xu, W. He, and S. Li, "Internet of things in industries: A survey," *IEEE Transactions on industrial informatics*, vol. 10, no. 4, pp. 2233–2243, 2014.

[10] N. V. Villagrán, E. Estevez, P. Pesado, and J. D. J. Marquez, "Standardization: A key factor of industry 4.0," in *2019 Sixth International Conference on eDemocracy & eGovernment (ICEDEG)*, IEEE, 2019, pp. 350–354.

[11] *Industry 4.0: Definition, Design Principles, Challenges, and the Future of Employment*, https://cleverism.com/industry-4-0/, Jan. 2017. (visited on 11/27/2023).

[12] N. Jazdi, "Cyber physical systems in the context of industry 4.0," in *2014 IEEE International Conference on Automation, quality and testing, robotics*, IEEE, 2014, pp. 1–4.

[13] A. Issa, B. Hatiboglu, A. Bildstein, and T. Bauernhansl, "Industrie 4.0 roadmap: Framework for digital transformation based on the concepts of capability maturity and alignment," *Procedia Cirp*, vol. 72, pp. 973–978, 2018.

[14] J.-R. Jiang, "An improved cyber-physical systems architecture for industry 4.0 smart factories," *Advances in Mechanical Engineering*, vol. 10, no. 6, p. 1 687 814 018 784 192, 2018.

[15] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution," in *Proceedings of the 47th design automation conference*, 2010, pp. 731–736.

[16] C. K. Keerthi, M. Jabbar, and B. Seetharamulu, "Cyber physical systems (cps): Security issues, challenges and solutions," in *2017 IEEE international conference on computational intelligence and computing research (ICCIC)*, IEEE, 2017, pp. 1–4.

[17] E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*, IEEE, 2008, pp. 363–369.

[18] R. C. Cardoso and A. Ferrando, "A review of agent-based programming for multi-agent systems," *Computers*, vol. 10, no. 2, p. 16, 2021.

[19] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing multi-agent systems with JADE*. John Wiley & Sons, 2007.

[20] P. Leitão and S. Karnouskos, "Industrial agents: Emerging applications of software agents in industry," 2015.

[21] J.-H. Lee and C.-O. Kim, "Multi-agent systems applications in manufacturing systems and supply chain management: A review paper," *International Journal of Production Research*, vol. 46, no. 1, pp. 233–265, 2008.

[22] W. Lepuschitz, "Self-reconfigurable manufacturing control based on ontology-driven automation agents," Ph.D. dissertation, Wien, 2018.

[23] P. G. Balaji and D. Srinivasan, "An introduction to multi-agent systems," *Innovations in multi-agent systems and applications-1*, pp. 1–27, 2010.

[24] olawanletjoel, *What is a Framework? Software Frameworks Definition*, Sep. 2022. [Online]. Available: `https://www.freecodecamp.org/news/what-is-a-framework-software-frameworks-definition/` (visited on 11/30/2023).

[25] J. Palanca, A. Terrasa, V. Julian, and C. Carrascosa, "Spade 3: Supporting the new generation of multi-agent systems," *IEEE Access*, vol. 8, pp. 182 537–182 549, 2020.

[26] G. Van Rossum *et al.*, "Python programming language.," in *USENIX annual technical conference*, Santa Clara, CA, vol. 41, 2007, pp. 1–36.

[27] SPADE, *Spade - smart python multi-agent development environment*, Último acesso em: 21 de março de 2023, 2023. [Online]. Available: `https://spade-mas.readthedocs.io/en/latest/agents.html` (visited on 12/01/2023).

[28] F. Bellifemine, A. Poggi, and G. Rimassa, "Jade–a fipa-compliant agent framework," in *Proceedings of PAAM*, London, vol. 99, 1999, p. 33.

[29] J. Gosling, *The Java language specification*. Addison-Wesley Professional, 2000.

[30] *FIPA ACL Message Structure Specification*. [Online]. Available: `http://www.fipa.org/specs/fipa00061/SC00061G.html` (visited on 12/01/2023).

[31] P. Saint-Andre, K. Smith, and R. Tronçon, *XMPP: the definitive guide*. " O'Reilly Media, Inc.", 2009.

[32] *XMPP | An Overview of XMPP*. [Online]. Available: `https://xmpp.org/about/technology-overview/` (visited on 12/01/2023).

[33] P. Saint-Andre, "Extensible messaging and presence protocol (xmpp): Core," Tech. Rep., 2011.

[34] M. J. U. September 18, 2. |. P. September 14, and 2009, *Meet the Extensible Messaging and Presence Protocol (XMPP)*. [Online]. Available: `https://developer.ibm.com/tutorials/x-xmppintro/` (visited on 12/04/2023).

[35] S. Lee, H. Kim, D.-k. Hong, and H. Ju, "Correlation analysis of mqtt loss and delay according to qos level," in *The International Conference on Information Networking 2013 (ICOIN)*, IEEE, 2013, pp. 714–717.

[36] M. Y. U. December 9, 2. |. P. May 11, and 2017, *Getting to know MQTT*. [Online]. Available: `https://developer.ibm.com/articles/iot-mqtt-why-good-for-iot/` (visited on 12/04/2023).

[37] F. Lelli, "Interoperability of the time of industry 4.0 and the internet of things," *Future Internet*, vol. 11, no. 2, p. 36, 2019.

[38] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for internet of things: A survey," *IEEE Internet of things journal*, vol. 3, no. 1, pp. 70–95, 2015.

[39] *What is Middleware? - Middleware Explained - AWS*. [Online]. Available: `https://aws.amazon.com/what-is/middleware/` (visited on 12/04/2023).

[40] *Apache Tomcat® - Welcome!* https://tomcat.apache.org/. (visited on 02/18/2024).

[41] *MuleSoft | Automatize tudo. Empodere todos.* (visited on 02/18/2024).

[42] J. de las Morenas, C. M. da Silva, G. S. Funchal, V. Melo, M. Vallim, and P. Leitao, "Security experiences in iot based applications for building and factory automation," in *2020 IEEE International Conference on Industrial Technology (ICIT)*, IEEE, 2020, pp. 322–327.

[43] S. N. Firdous, Z. Baig, C. Valli, and A. Ibrahim, "Modelling and evaluation of malicious attacks against the iot mqtt protocol," in *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2017, pp. 748–755. DOI: 10.1109/iThings-GreenCom-CPSCom-SmartData.2017.115.

[44] R. Khan, K. McLaughlin, D. Laverty, and S. Sezer, "Stride-based threat modeling for cyber-physical systems," in *2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, 2017, pp. 1–6. DOI: 10.1109/ISGTEurope.2017.8260283.

[45] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[46] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *Proceedings of the 25th international conference on World Wide Web*, 2016, pp. 263–273.

[47] J. Barbosa, P. Leitão, and J. Teixeira, "Empowering a Cyber-Physical System for a Modular Conveyor System with Self-organization," in *Service Orientation in Holonic and Multi-Agent Manufacturing*, T. Borangiu, D. Trentesaux, A. Thomas, and O. Cardin, Eds., vol. 762, Springer International Publishing, 2018, pp. 157–170.

[48] G. M. D'silva, S. Thakare, S. More, and J. Kuriakose, "Real world smart chatbot for customer care using a software as a service (saas) architecture," in *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, 2017, pp. 658–664. DOI: 10.1109/I-SMAC.2017.8058261.

[49] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[50] B. Croston, *RPi.GPIO: A module to control Raspberry Pi GPIO channels*. (visited on 12/29/2023).

[51] *Eclipse Mosquitto*, https://mosquitto.org/, Jan. 2018. (visited on 01/05/2024).

[52] R. Light, *Paho-mqtt: MQTT version 5.0/3.1.1 client class*. (visited on 01/17/2024).

[53] I. Craggs, *Eclipse Paho | The Eclipse Foundation*. (visited on 01/17/2024).

[54] J. Viega, M. Messier, and P. Chandra, *Network security with openSSL: cryptography for secure communications*. " O'Reilly Media, Inc.", 2002.

[55] *Node-RED Programming Guide*, https://noderedguide.com/. (visited on 01/30/2024).

[56] B. H. Çorak, F. Y. Okay, M. Güzel, Ş. Murt, and S. Ozdemir, "Comparative analysis of iot communication protocols," in *2018 International symposium on networks, computers and communications (ISNCC)*, IEEE, 2018, pp. 1–6.

[57] E. H. Halili, *Apache JMeter*. 2008.

Appendix A

Proposta Original do Projeto

INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

Mestrado em Engenharia Eletrotéctica e de Computadores
Ano letivo de 2022/2023

An MQTT-based solution as an enabler of interoperability between MAS frameworks to manage a Self-organized Conveyor System

Aluno: Bruno Patto Graciano Natal

Orientador: Prof. Dr. Paulo Jorge Pinto Leitão

Coorientador: Prof. Dr. Leandro Resende Mattioli

1 Objetivo

O objetivo principal desse trabalho consiste no estudo e desenvolvimento de um mecanismo que permita a interoperabilidade entre frameworks de sistemas multiagentes (MAS). Para isso, será utilizado como caso de estudo um sistema de transferência modular ciber-físico, em que a principal característica é a flexibilidade de implementar logicas de auto-organização para se adaptar em um ambiente propicio a mudanças. Dessa forma, pretende-se implementar e garantir que diferentes soluções baseadas em MAS e desenvolvidas utilizando diferentes frameworks possam trocar informações para controlar e apresentar características de auto-organização sobre os módulos do sistema.

2 Detalhes

No contexto da Indústria 4.0, as tecnologias de Internet of Things (IoT), inteligência artificial (IA) e Sistema Ciber-Fısico (CPS) tem um papel essencial, não apenas para aumentar os níveis de automação, mas principalmente na criação de máquinas e ambientes inteligentes capazes de suportar a grande dinâmica dos mercados e atender às tendências emergentes de customização de produtos. Portanto, em uma indústria repleta de sistemas e componentes desempenhando diferentes funções, os MAS se apresentam como uma solução válida para lidar com essas complexidades. Visto que, eles possuem a capacidade de distribuir a inteligência entre os diversos dispositivos (agentes) que irão permitir que o sistema apresente características como descentralização, modularidade, proatividade e autonomia, presentes no contexto da Indústria 4.0.

Dentro do escopo do estudo proposto, a área de foco será a interoperabilidade entre agentes para o controle de um processo auto-organizável de transporte de peças. Para alcançar esse objetivo, será utilizado um sistema modular de transporte ciber-físico composto por esteiras FischerTechniks em conjunto com computadores de placa única Raspberry Pi. A programação dos agentes será garantida através do uso de frameworks, que oferecem um conjunto de ferramentas, bibliotecas e protocolos para criar agentes capazes de atender às necessidades do sistema. No entanto, ao desenvolver agentes em frameworks distintos, surge uma complexidade quando eles precisam se comunicar, devido às possíveis diferenças em linguagens de programação, protocolos de comunicação ou estruturação na troca de mensagens. Isso resulta na incompatibilidade dos diversos módulos do sistema.

Portanto, ao final deste projeto, a expectativa é desenvolver um mecanismo que permita a comunicação entre diferentes frameworks de sistemas MAS, proporcionando a interoperabilidade para alcançar as características de auto-organização nos módulos ciber-físicos.

3 Metodologia de trabalho

O desenvolvimento deste trabalho seguirá as seguintes etapas:

1. Estudo e análise aprofundados das frameworks SPADE e JADE (M1-M2).
2. Estudo e familiarização com o caso de estudo, que está relacionado à aplicação de lógicas de auto-organização no sistema modular de transporte ciber-físico composto por esteiras FischerTechniks e computadores de placa única RaspBerry Pi (M2).
3. Implementação da solução de auto-organização utilizando ambas frameworks (M2-M4).
4. Projeto e desenvolvimento da arquitetura do middleware baseada em MQTT para garantir a interoperabilidade (M5-M6).
5. Integração das soluções desenvolvidas com JADE e SPADE através da utilização do midlleware (M7-M8).
6. Testes e validação (M9-M10).
7. Escrita da dissertação e defesa final do trabalho (M10-M11).

Dimensão da equipa: 3 pessoas

Recursos necessários: Recursos de hardware disponibilizados nos laboratórios do Centro de Investigação em Digitalização e Robótica Inteligente (CeDRI) e recursos de software.

Appendix B

SPADE conveyor agent

B.1 Agent

```
1  import spade
2  from spade.agent import Agent
3  from conveyor_ciclic_behav import ConveyorClicBehaviour
4  from conveyor_register import ConveyorRegister
5  from spade.template import Template
6
7  """"This code is responsible to create and setup the Agent"""
8
9
10 class ConveyorAgent(Agent):
11     """"Class responsible to add behaviours and the setup configurations
       ↪  to the Agent"""
12     def __init__(self, jid: str, password: str):
13         super().__init__(jid, password)
14
15     async def setup(self) -> None:
```

```
16          """Create templates and behaviours to the Agent"""
17
18          jid = str(self.jid)
19          print(f' - The Agent [{jid.split("@")[0]}] is alive - ')
20
21          t1 = Template()
22          t2 = Template()
23          t3 = Template()
24          t4 = Template()
25          t5 = Template()
26          t6 = Template()
27          t7 = Template()
28          t8 = Template()
29          t9 = Template()
30          t10 = Template()
31
32          t1.set_metadata("performative", "inform")
33          t1.set_metadata("protocol", "tokenTransitionInput")
34          t2.set_metadata("performative", "inform")
35          t2.set_metadata("protocol", "tokenTransitionOutput")
36          t3.set_metadata("performative", "inform")
37          t3.set_metadata('protocol', "thereIsSwap")
38          t4.set_metadata("performative", "inform")
39          t4.set_metadata('protocol', "swapTokenFound")
40          t5.set_metadata("performative", "inform")
41          t5.set_metadata("protocol", "firstTokenSwapped")
42          t6.set_metadata("performative", "inform")
43          t6.set_metadata("protocol", "conveyedAllPieces")
44          t7.set_metadata("performative", "inform")
```

```
45          t7.set_metadata("protocol", "piecesToConveyor")
46          t8.set_metadata("performative", "inform")
47          t8.set_metadata("protocol", "informAlive")
48          t9.set_metadata("performative", "inform")
49          t9.set_metadata("protocol", "aliveToo")
50          t10.set_metadata("performative", "inform")
51          t10.set_metadata("protocol", "leftDF")
52
53          conveyor_behavior = ConveyorClicBehaviour(self.jid, self)
54          conveyor_register = ConveyorRegister(self.jid, self)
55          self.add_behaviour(conveyor_register)
56          self.add_behaviour(conveyor_behavior, t1 | t2 | t3 | t4 | t5 | t6
            ↪  | t7 | t8 | t9 | t10)
57
58
59  async def main():
60      conveyor_agent = ConveyorAgent("JID", "PASSWORD")
61      await conveyor_agent.start(auto_register=True)
62      await spade.wait_until_finished(conveyor_agent)
63
64
65  if __name__ == "__main__":
66      spade.run(main())
67  \end{lstlisting}
68
69  \section{Main behavior}
70
71  \begin{lstlisting}[style=pythonstyle]
72
```

```
73 import asyncio
74
75 from spade.behaviour import CyclicBehaviour
76 import RPi.GPIO as GPIO
77 from spade.message import Message
78 import time
79 import signal
80
81 '''
82  - This is the core of the agent, here the agent will control de I/O of
    ↪  the RaspBerry.
83  - In this behaviour the agent will handle with the messages swapped,
    ↪  and decide what is the proper action.
84 '''
85
86 # Set-up I/O
87 INPUT_SENSOR = 24
88 OUTPUT_SENSOR = 25
89 MOTOR = 23
90
91 GPIO.setmode(GPIO.BCM)  # Set the way of how the pinOut will be
    ↪  represented
92 GPIO.setup(INPUT_SENSOR, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
93 GPIO.setup(OUTPUT_SENSOR, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
94 GPIO.setup(MOTOR, GPIO.OUT)
95
96
97 class ConveyorClicBehaviour(CyclicBehaviour):
98
```

```
99      def __init__(self, jid, agent):
100         super().__init__()
101         self.agent = agent
102         self.jid = str(jid)
103         self.my_token = -1
104         self.token_transition = 0
105         self.conveyed_last_piece = False  #
106         self.number_pieces = 0
107         self.fist_input = True  #
108         self.may_first_input = False
109         self.piece_transition = False
110         self.number_conveyors = 1
111         self.number_all_pieces_conveyed = 0
112         self.list_of_agents = []
113         self.client_mqtt = None
114
115         self.debounce_time = 0.2
116         self.time_out = 5  # change this parameter, it's the time to
            ↪ wait a message
117         self.time_out_first_conveyor = 5
118         self.time_of_sent = 0
119         self.time_conveyed_last_piece = None
120         self.waiting_message = False
121         self.swap_alert = False
122         self.swap_position = None
123
124         self.last_state_input = False
125         self.last_state_output = False
126
```

```
127     async def on_start(self) -> None:
128         print(' - The conveyor behaviour started - ')
129 
130         GPIO.output(MOTOR, GPIO.LOW)
131         # Inform other agents that it's alive
132         contact_list = self.presence.get_contacts()
133         msg = self.create_message("informAlive", content=self.jid)
134 
135         send_task = [self.send_message(contact=contact, message=msg) for
            ↪  contact in contact_list]
136         await asyncio.gather(*send_task)
137 
138         print(f'[{self.jid}] - sent - [informAlive] - to all contacts')
139         # Set-up callback to Ctrl+C
140         signal.signal(signal.SIGINT, self.handle_keyboard_interrupt)
141 
142     async def on_end(self) -> None:
143 
144         print(f'[{self.jid}] - keyBoardInterrupted - \n')
145         GPIO.output(MOTOR, GPIO.LOW)
146         GPIO.cleanup()
147         """"Send message to inform all contacts that this agents is
            ↪  dead"""
148         contact_list = self.presence.get_contacts()
149         body = self.jid + "*" + str(self.my_token)
150         msg = self.create_message("leftDF", body)
151 
152         send_task = [self.send_message(contact=contact, message=msg) for
            ↪  contact in contact_list]
```

```
153         await asyncio.gather(*send_task)
154         print(f'[{self.jid}] - sent - [leftDF] - to all contacts')
155         print(f'[{self.jid}] - behaviour ended - \n')
156         await self.agent.stop()
157 
158 
159     async def run(self) -> None:
160 
161 """                     Here is the main loop of the behaviour
162     Here the agent will handle the messages received and the
        ↪ input/output signals """
163 
164 """" 1 ------------------------ Handle the Input sensor
    ↪ ------------------------------------------------
165     When the piece first passes through the conveyor, is needed to learn
        ↪ the order and assign a token number
166 to each running agent. Therefore, the initial part of the code is
    ↪ responsible to lear the order of conveyors
167 and set a token number. After this, the agent sends a message to inform
    ↪ others of its token and that it received
168 the piece.
169     If It is not the first time that the piece has passed through the
        ↪ sensor, the agent will only send a message
170 to inform that the piece has entered in the input sensor so that the
    ↪ agent behind it can turn off its motor.
171 """
172 
173         if GPIO.input(INPUT_SENSOR) and not self.last_state_input:  #
            ↪ High State
```

```
174
175             GPIO.output(MOTOR, GPIO.HIGH)
176             self.last_state_input = True
177             self.number_pieces += 1
178             self.may_first_input = True
179             self.piece_transition = False
180             print(f'[{self.jid}] - Input sensor - number of pieces
                ↪  [{self.number_pieces}]')
181
182             # Learn the token position, this part of the code will run
                ↪  once in each agent
183             if self.my_token <= 0:
184                 if self.token_transition != 0:
185                     self.my_token = self.token_transition + 1
186                     print(f'[{self.jid}] - Token position -
                        ↪  [{self.my_token}]')
187                 else:
188                     self.token_transition = 1
189                     self.my_token = self.token_transition
190                     self.fist_input = True
191                     self.conveyed_last_piece = False
192                     print(f'[{self.jid}] - Token position -
                        ↪  [{self.my_token}]')
193
194             if self.swap_alert:
195                 self.swap_alert = False
196                 await self.inform_swap_found(self.my_token,
                    ↪  self.swap_position)
197
```

```
198             # Case the agent is the last token
199             if self.my_token == self.number_conveyors:
200                 """"Here the last agent will send a message to
                    ↪ inform the new last Agent that it needs to wait
201                       the number of pieces given before send a
                          ↪ conveyed all piece message
202                           ---------------- This part need to be
                              ↪ adapted to the new logic --------"""
203
204                 contact_list = self.presence.get_contacts()
205                 msg = self.create_message("PiecesToConveyor",
                    ↪ str(self.number_all_pieces_conveyed))
206                 for contact in contact_list:
207                     msg.to = str(contact)
208                     await self.send(msg)
209                 print(f'[{self.jid}] - sent - [PiecesToConveyor] - to
                    ↪ all contacts')
210                 send_task = [self.send_message(contact=contact,
                    ↪ message=msg) for contact in contact_list]
211                 await asyncio.gather(*send_task)
212
213             self.my_token = self.swap_position
214
215         if self.my_token == 1:
216             """"Here the conveyor will count the number os pieces
                ↪ that need to be convey
217                 and it will send a message to inform the last
                    ↪ conveyor the number of pieces"""
218             input_piece = 1
```

```
219                 if self.conveyed_last_piece:  # First input
220                     self.fist_input = True
221                     self.conveyed_last_piece = False
222 
223                     # Send message
224                     contact_list = self.presence.get_contacts()
225                     msg = self.create_message("PiecesToConveyor",
                        ↪ str(input_piece))
226                     send_task = [self.send_message(contact=contact,
                        ↪ message=msg) for contact in contact_list]
227                     await asyncio.gather(*send_task)
228 
229             # Send TokenTransitionInput message to all contacts
230             contact_list = self.presence.get_contacts()
231             msg = self.create_message("tokenTransitionInput",
                ↪ self.my_token)
232             send_task = [self.send_message(contact=contact, message=msg)
                ↪ for contact in contact_list]
233             await asyncio.gather(*send_task)
234 
235             print(f'[{self.jid}] - sent - [tokenTransitionInput] - to all
                ↪ contacts')
236             try:
237                 await asyncio.sleep(self.debounce_time)
238             except Exception as e:
239                 print(f'Error: {e}')
240 
241         elif not GPIO.input(INPUT_SENSOR) and self.last_state_input:  #
            ↪ Piece out of sensor, state low
```

```
242             self.last_state_input = False
243             try:
244                 await asyncio.sleep(self.debounce_time)
245             except Exception as e:
246                 print(f'Error: {e}')
247 
248         """ 2 ----------------------- Handle the Output sensor
        ↪  -----------------------------------------"""
249 
250         if GPIO.input(OUTPUT_SENSOR) and not self.last_state_output:  #
        ↪  High state
251 
252             contact_list = self.presence.get_contacts()
253             self.last_state_output = True
254             self.may_first_input = False
255             # If the Agent is the last conveyor
256             if self.my_token == self.number_conveyors:
257                 self.number_pieces -= 1
258                 self.number_all_pieces_conveyed -= 1
259                 if self.number_pieces <= 0 and not self.piece_transition:
260                     self.number_pieces = 0
261                     GPIO.output(MOTOR, GPIO.LOW)
262                     self.waiting_message = False
263 
264                     if self.number_all_pieces_conveyed <= 0:
265                     # Send a ConveyedAllPieces to all contacts
266                         self.number_all_pieces_conveyed = 0
267                         msg = self.create_message("conveyedAllPieces",
                         ↪  self.my_token)
```

```
268                         msg.set_metadata("protocol", "conveyedAllPieces")
269                         send_task = [self.send_message(contact=contact,
                            ↪  message=msg) for contact in contact_list]
270                         await asyncio.gather(*send_task)
271
272                         print(f'[{self.jid}] - sent - [conveyedAllPieces]
                            ↪  - to all contacts')
273
274             else:
275                 # Send a TokenTransitionOutput to all contacts
276                 msg = self.create_message("tokenTransitionOutput",
                    ↪  self.my_token)
277                 send_task = [self.send_message(contact=contact,
                    ↪  message=msg) for contact in contact_list]
278                 await asyncio.gather(*send_task)
279                 print(f'[{self.jid}] - sent - [tokenTransitionOutput] -
                    ↪  to all contacts')
280
281                 # Set waiting time and the time when the last message
                    ↪  was sent ######
282                 self.time_of_sent = time.time()
283                 self.waiting_message = True
284             try:
285                 await asyncio.sleep(self.debounce_time)
286             except Exception as e:
287                 print(f'Error: {e}')
288
289         elif not GPIO.input(OUTPUT_SENSOR) and self.last_state_output:  #
            ↪  Piece out of sensor, state low
```

```
290             self.last_state_output = False
291             try:
292                 await asyncio.sleep(self.debounce_time)
293             except Exception as e:
294                 print(f'Error: {e}')
295 
296         """" 3 ---------------------------------- Handle messages
            ↪  ------------------------------------------------"""
297 
298         message_received = await self.receive()
299         if message_received is None:
300             if (time.time() - self.time_of_sent > self.time_out) and
                ↪  self.waiting_message:
301                 self.waiting_message = False
302                 print("[timeOut reached] - checking if there is a
                    ↪  swap...")
303                 await self.conveyor_swapped()
304         else:
305             await self.handle_message(message_received)
306 
307 
308     """"" 4 ------------------------------- Functions used in the code
        ↪  ----------------------------------------"""
309 
310     async def handle_message(self, message_received):
311         if message_received.get_metadata("protocol") ==
            ↪  "tokenTransitionInput":
312             # First position change warning
313             if (self.my_token == 1 and not self.fist_input
```

```
314                     and (time.time() - self.time_conveyed_last_piece >
                        ↪  self.time_out_first_conveyor)):
315
316                 print(f'[{self.jid}] - changed its token - [old token:
                    ↪  {self.my_token}] - '
317                       f'[new token: {message_received.body}]')
318
319                 contact_list = self.presence.get_contacts()
320                 msg = self.create_message("firstTokenSwapped",
                    ↪  self.my_token)
321                 send_task = [self.send_message(contact=contact,
                    ↪  message=msg) for contact in contact_list]
322                 await asyncio.gather(*send_task)
323
324                 self.my_token = int(message_received.body)
325
326             elif self.my_token == 1 and not self.fist_input:
327                 self.fist_input = True
328
329                 contact_list = self.presence.get_contacts()
330                 msg = self.create_message("piecesToConveyor",
                    ↪  self.my_token)
331                 msg.set_metadata("protocol", "piecesToConveyor")
332
333                 send_task = [self.send_message(contact=contact,
                    ↪  message=msg) for contact in contact_list]
334                 await asyncio.gather(*send_task)
335
336             if self.my_token == (int(message_received.body) - 1):
```

```
337                 self.number_pieces -= 1
338                 if self.number_pieces <= 0:
339                     GPIO.output(MOTOR, GPIO.LOW)
340                     self.waiting_message = False
341                     self.number_pieces = 0
342                     print(f'[{self.jid}] - pieces conveyed, waiting a new
                        ↪  input...')
343
344             if message_received.get_metadata("protocol") ==
                ↪  "tokenTransitionOutput":
345                 if self.my_token == (int(message_received.body) + 1) or
                    ↪  (self.my_token <= 0):
346                     GPIO.output(MOTOR, GPIO.HIGH)
347                     self.token_transition = int(message_received.body)
348                     self.piece_transition = True
349                     print(f'[{self.jid}] - received - [tokenTransitionOutput]
                        ↪  - turn on the motor')
350
351             if message_received.get_metadata("protocol") ==
                ↪  "conveyedAllPieces":
352                 if self.my_token == 1 and self.number_pieces == 0:
353                     self.number_all_pieces_conveyed = 0
354                     self.conveyed_last_piece = True
355                     self.fist_input = False
356                     self.time_conveyed_last_piece = time.time()
357                     print(f'[{self.jid}] - All the pieces were conveyed,
                        ↪  waiting to a new piece...')
358
359             if message_received.get_metadata("protocol") == "thereIsSwap":
```

```
360             if self.my_token > int(message_received.body):
361                 self.swap_alert = True
362                 GPIO.output(MOTOR, GPIO.HIGH)
363                 self.swap_position = int(message_received.body) + 1
364                 print(f'[{message_received.metadata["protocol"]}] -
                    ↪  received from - [{message_received.sender}] - '
365                       f'position of swap - [{self.swap_position}]')
366
367         if message_received.get_metadata("protocol") == "swapTokenFound":
368             tokens = message_received.body.split('*')
369             self.swap_alert = False
370             print(f'[{message_received.metadata["protocol"]}] - received
                ↪  from - [{message_received.sender}] -'
371                   f'[its old position: {tokens[0]}] - [its new position:
                    ↪  {tokens[1]}]')
372             if self.my_token == int(tokens[1]):
373                 print(f'[{self.jid}] - old token [{self.my_token}] - new
                    ↪  token [{tokens[0]}]')
374                 self.my_token = int(tokens[0])
375
376         if message_received.get_metadata("protocol") ==
            ↪  "firstTokenSwapped" and self.may_first_input:
377             self.fist_input = True
378             print(f'[{self.jid}] - changed its token - [old token:
                ↪  {self.my_token}] - '
379                   f'[new token: {message_received.body}]')
380             self.my_token = int(message_received.body)
381
```

```
382             #Send a message to the last conveyor to inform the first
                ↪ piece input
383             contact_list = self.presence.get_contacts()
384             msg = self.create_message("piecesToConveyor", self.my_token)
385             msg.set_metadata("protocol", "piecesToConveyor")
386             send_task = [self.send_message(contact=contact, message=msg)
                ↪ for contact in contact_list]
387
388             await asyncio.gather(*send_task)
389             print(f'[{self.jid}] - sent - [PiecesToConveyor] - to the
                ↪ last conveyor')
390
391         if message_received.get_metadata("protocol") == "leftDF":
392             print(f'[{self.jid}] - the agent - [{message_received.body}]
                ↪ - is off.')
393             sender = str(message_received.sender).split("@")[0]
394             body = message_received.body.split("*")
395
396             if sender == "agent3":
397                 sender_mqtt = body[0]
398                 token_of_sender_jade = body[1]
399
400                 """Removing the agent from the contact list and
                    ↪ reducing the number of conveyors"""
401                 if self.my_token > int(token_of_sender_jade):
402                     self.my_token -= 1
403
404                 if sender_mqtt in self.list_of_agents:
405                     self.list_of_agents.remove(sender_mqtt)
```

```
406                 print(f'[{self.jid}] - new number of conveyors WITHOUT
                    ↪  JADE- [{len(self.list_of_agents) + 1}]')
407                 self.number_conveyors -= 1
408                 self.conveyed_last_piece = True
409
410             else:
411                 token_of_sender = body[1]
412                 if self.my_token > int(token_of_sender):
413                     self.my_token -= 1
414                 if sender in self.list_of_agents:
415                     self.list_of_agents.remove(sender)
416                 print(f'[{self.jid}] - new number of conveyors -
                    ↪  [{len(self.list_of_agents) + 1}]')
417                 self.number_conveyors -= 1
418             self.number_all_pieces_conveyed = 0
419             self.number_pieces = 0
420
421         if message_received.get_metadata("protocol") == "informAlive":
422             sender = str(message_received.sender).split("@")[0]
423             sender_mqtt = message_received.body
424             if self.my_token > 0:
425                 self.my_token = -1
426                 self.token_transition = 0
427                 self.fist_input = False
428                 print(f'[{self.jid}] - received - [informAlive] - from -
                    ↪  [{sender}] - learning new order...')
429
430             if sender == "agent3":
431                 self.list_of_agents.append(sender_mqtt)
```

```
432                 self.number_conveyors += 1
433                 print(f"[{self.jid}] - the conveyor - [{sender_mqtt}] -
                    ↪  was added in the contact list")
434
435             else:
436                 self.list_of_agents.append(sender)
437                 self.number_conveyors += 1
438             # Send a aliveToo to the agent that sent the message
439             contact_list = self.presence.get_contacts()
440             msg = self.create_message("aliveToo", self.jid)
441
442             send_task = [self.send_message(contact=contact, message=msg)
                ↪  for contact in contact_list]
443             await asyncio.gather(*send_task)
444
445             print(f'[{self.jid}] - number off conveyors live -
                ↪  [{self.number_conveyors}]')
446             self.number_all_pieces_conveyed = 0
447             self.number_pieces = 0
448
449         if message_received.get_metadata("protocol") == "aliveToo":
450             print(f'[{self.jid}] - received - [aliveToo]')
451             sender = str(message_received.sender)
452             sender_jid = sender.split("@")[0]
453             sender_mqtt = message_received.body
454             if str(sender_jid) == "agent3":
455                 await self.analise_inform(sender_mqtt)
456             else:
457                 await self.analise_inform(sender_jid)
```

```
458             print(f'[{self.jid}] - number off conveyors live -
                ↪ [{self.number_conveyors}]')
459
460         if message_received.get_metadata("protocol") ==
            ↪ "piecesToConveyor":
461             if self.my_token == self.number_conveyors or self.my_token <
                ↪ 0:
462                 self.number_all_pieces_conveyed +=
                    ↪ int(message_received.body)
463                 print(self.number_all_pieces_conveyed)
464             else:
465                 self.number_all_pieces_conveyed = 0
466
467     async def conveyor_swapped(self):
468
469         contact_list = self.presence.get_contacts()
470         msg = self.create_message("thereIsSwap", self.my_token)
471         msg.set_metadata("protocol", "thereIsSwap")
472
473         # Inform all agents that happened a change in the order
474         send_task = [self.send_message(contact=contact, message=msg) for
            ↪ contact in contact_list]
475         await asyncio.gather(*send_task)
476
477         print(f'[{self.jid}] - sent - [thereIsSwap] - to all contacts')
478
479     async def inform_swap_found(self, last_token, new_token):
480         body = str(last_token) + '*' + str(new_token)
481         msg = self.create_message("swapTokenFound", body)
```

```
482         contact_list = self.presence.get_contacts()
483         msg.set_metadata("protocol", "swapTokenFound")
484 
485         print(f'[{self.jid}] - found swap - [old token: {last_token}] -
            ↪  [new token: {new_token}]')
486         # Inform all agents that the swap was found
487 
488         send_task = [self.send_message(contact=contact, message=msg) for
            ↪  contact in contact_list]
489         await asyncio.gather(*send_task)
490         print(f'[{self.jid}] - sent - [swapTokenFound] - to all
            ↪  contacts')
491 
492     async def analise_inform(self, sender):
493 
494         if sender in self.list_of_agents:
495             print(f"[{self.jid}] - the conveyor - [{sender}] - is in the
                ↪  contact list")
496         else:
497             self.list_of_agents.append(sender)
498             self.number_conveyors += 1
499             print(f"[{self.jid}] - the conveyor - [{sender}] - was added
                ↪  in the contact list")
500 
501     def create_message(self, protocol, content):
502         msg_template = Message()
503         msg_template.sender = self.jid
504         msg_template.body = str(content)
505         msg_template.set_metadata("performative", "inform")
```

```
506         msg_template.set_metadata("protocol", protocol)
507         return msg_template
508 
509     async def send_message(self, contact, message):
510         msg = message
511         msg.to = str(contact)
512         await self.send(msg)
513 
514     def handle_keyboard_interrupt(self, sign, frame):
515         self.kill()
516 
517 
518 
519 
520 
521 
522 
523 
524 \end{lstlisting}
525 \section{Behavior for register and monitoring the contact list}
526 \begin{lstlisting}[style=pythonstyle]
527 
528 import asyncio
529 from spade.behaviour import CyclicBehaviour
530 
531 '''
532 ##################### This Class is responsible to inform and register to
    ↪  other agents  ######################
533 
```

```
534 '''
535
536
537 class ConveyorRegister(CyclicBehaviour):
538
539     def __init__(self, jid, agent):
540         super().__init__()
541         self.jid = jid
542         self._register_state = None  # Variable responsible to register
        ↪  once in contact list
543
544     def on_agent_subscribed(self, jid):
545         print(f'The agent {self.agent.name} accepted the subscription
        ↪  from {jid.split("@")[0]}')
546
547     def on_agent_subscribe(self, jid):
548         print(f'The agent {jid.split("@")[0]} wants to subscribe in the
        ↪  list of contacts')
549         self.presence.approve(jid)
550
551     async def on_start(self) -> None:
552         print(' - Register behaviour started - ')
553         self._register_state = True
554
555     async def on_end(self) -> None:
556         print(' - Register behaviour ended - ')
557
558     async def run(self) -> None:
559
```

```
560         self.presence.on_subscribe = self.on_agent_subscribe
561         self.presence.on_subscribed = self.on_agent_subscribed
562 
563         # Find all the jid's of agents and try to register in the
            ↪  contact list
564 
565         if self._register_state:
566             with open("Jid's.txt", "r") as file:
567                 lines = file.readlines()
568 
569             for line in lines:
570                 jid = line.strip()
571                 if str(self.jid) != jid:  # Compare the agent jid with
                    ↪  the jid from the list
572                     try:
573                         self.presence.subscribe(jid)
574                         print(f'Agent - [{self.agent.name}] - Sent a
                            ↪  message register to [{jid}] ')
575                     except Exception as e:
576                         print(f'Behaviour unexpected, cause {e}')
577 
578             self._register_state = False
579         print('register behaviour checked')
580 
581         await asyncio.sleep(20)
582 
```

Appendix C

JADE conveyor agent

C.1 Agent

```
1  package selfOrganization;
2
3  import jade.core.Agent;
4  import jade.core.behaviours.DataStore;
5  import jade.core.behaviours.SimpleBehaviour;
6  import jade.domain.DFService;
7  import jade.domain.FIPAAgentManagement.DFAgentDescription;
8  import jade.domain.FIPAAgentManagement.ServiceDescription;
9  import jade.lang.acl.ACLMessage;
10
11
12 /* This class is responsible to run the agent and inform other Agents
   ↪  that it is alive */
13
14 public class ConveyorAgent extends Agent{
15   private static final long serialVersionUID = 1L;
```

```
16
17    private DataStore myToken = new DataStore();
18
19    protected void setup() {
20      /*Here the agent will register itself into Df and It will send a
        ↪  message to inform Alive */
21
22      DFAgentDescription dfd = new DFAgentDescription();
23      ServiceDescription sd = new ServiceDescription();
24      sd.setName(getName());
25      sd.setType("resource");
26      dfd.setName(getAID());
27      dfd.addServices(sd);
28
29      // 1 - ADD the conveyor's skill
30      ServiceDescription sd1 = new ServiceDescription();
31      sd1.setName("skill");
32      sd1.setType("conveyor");
33      dfd.addServices(sd1);
34
35      // 2 - Register in DF
36      boolean registered = false;
37      try {
38        DFService.register(this, dfd);
39        registered = true;
40      } catch (Exception e) {
41        System.err.println(e);
42      }
43
```

```
44      if (!registered) {
45        try {
46          DFService.register(this, dfd);
47        } catch (Exception e) {
48          System.out.println(String.format("[%s] - Something went wrong
            ↪  with register", this.getLocalName()));
49          System.err.print(e);
50        }
51      }
52      System.out.println(String.format("The agent [%s] is alive",
        ↪  this.getLocalName()));
53
54      // 3 - Send InformAlive to all contacts
55      DFAgentDescription[] contactList = null;
56      DFAgentDescription df1 = new DFAgentDescription();
57      ServiceDescription sd2 = new ServiceDescription();
58
59      sd2.setName("skill");
60      sd2.setType("conveyor");
61      df1.addServices(sd2);
62
63      try {
64        contactList = DFService.search(this, df1);
65      } catch (Exception e) {
66        // TODO: handle exception
67      }
68
69      ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
70      msg.setSender(this.getAID());
```

```
71      msg.setProtocol("informAlive");
72      msg.setContent(this.getName());
73
74      for (int i=0; i< contactList.length;i++) {
75        DFAgentDescription contact = contactList[i];
76        if(!contact.getName().getName().equals(getName())) {
77          msg.addReceiver(contact.getName());
78        }
79      }
80      this.send(msg);
81      System.out.println(String.format("[%s] - sent - [informAlive] - to
        ↪  all contacts",this.getLocalName()));
82
83      SimpleBehaviour conveyorBehaviour = new ConveyorBehaviour(this);
84      conveyorBehaviour.setDataStore(myToken);
85      addBehaviour(conveyorBehaviour);
86
87    }
88
89    /* This method  is called when the Agent is killed, so it clean up
      ↪  what is needed*/
90    protected void takeDown() {
91
92      /*Here the agent will, deregister itself from DF and will send a
        ↪  message informing other
93       agents that it is leaving the platform */
94
95      DFAgentDescription dfd = new DFAgentDescription();
96      dfd.setName(getAID());
```

```
97      try {
98        DFService.deregister(this, dfd);
99      } catch (Exception e) {
100       System.out.println(String.format("[%s] - Something went wrong with
          ↪  deregister", this.getLocalName()));
101       System.err.print(e);
102     }
103
104     // Send a leftDF to all contacts
105     DFAgentDescription [] contactList = null;
106     DFAgentDescription dfd1 = new DFAgentDescription();
107     ServiceDescription sd = new ServiceDescription();
108
109     sd.setName("skill");
110     sd.setType("conveyor");
111     dfd1.addServices(sd);
112
113     try {
114       contactList = DFService.search(this, dfd1);
115     } catch (Exception e) {
116       System.out.println(String.format("[%s] - Something went wrong with
          ↪  the search in DF", this.getLocalName()));
117     }
118
119     ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
120     msg.setSender(getAID());
121     msg.setProtocol("leftDF");
122     msg.setContent(getName()+"*"+myToken.get("MyToken").toString());
123
```

```
124     for (int i = 0; i< contactList.length; i++) {
125       DFAgentDescription contact = contactList[i];
126       if(!contact.getName().getName().equals(this.getName())) {
127         msg.addReceiver(contact.getName());
128       }
129     }
130     send(msg);
131     System.out.println(String.format("[%s] - sent - [leftDF] - to all
        ↪  contacts", this.getLocalName()));
132     ConveyorBehaviour.MOTOR.low();
133     ConveyorBehaviour.outputSensor.removeAllListeners();
134     ConveyorBehaviour.inputSensor.removeAllListeners();
135   }
136
137 }
138
139 \end{lstlisting}
140
141 \section{Main behaviour}
142
143 \begin{lstlisting}[style=javastyle]
144 package selfOrganization;
145
146 import java.util.ArrayList;
147
148 import com.pi4j.io.gpio.GpioController;
149 import com.pi4j.io.gpio.GpioFactory;
150 import com.pi4j.io.gpio.GpioPinDigitalInput;
151 import com.pi4j.io.gpio.GpioPinDigitalOutput;
```

```
152 import com.pi4j.io.gpio.PinPullResistance;
153 import com.pi4j.io.gpio.PinState;
154 import com.pi4j.io.gpio.RaspiPin;
155 import com.pi4j.io.gpio.event.GpioPinDigitalStateChangeEvent;
156 import com.pi4j.io.gpio.event.GpioPinListenerDigital;
157
158
159 import jade.core.Agent;
160 import jade.core.behaviours.SimpleBehaviour;
161 import jade.domain.DFService;
162 import jade.domain.FIPAAgentManagement.DFAgentDescription;
163 import jade.domain.FIPAAgentManagement.ServiceDescription;
164 import jade.lang.acl.ACLMessage;
165 import jade.lang.acl.MessageTemplate;
166
167
168 public class ConveyorBehaviour extends SimpleBehaviour{
169   private static final long serialVersionUID = 1L;
170
171   final GpioController GPIO = GpioFactory.getInstance();
172   static GpioPinDigitalInput inputSensor;
173   static GpioPinDigitalInput outputSensor;
174   static GpioPinDigitalOutput MOTOR;
175   static PinState lastStateInput;
176   static PinState lastStateOutput;
177
178   MessageTemplate mt =
179       MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.INFORM),
180       MessageTemplate.or(MessageTemplate.MatchProtocol("informAlive"),
```

```
181         MessageTemplate.or(MessageTemplate.MatchProtocol("leftDF"),
182         MessageTemplate.or(MessageTemplate.MatchProtocol("tokenTransitionInput"),
183         MessageTemplate.or(MessageTemplate.MatchProtocol("tokenTransitionOutput"),
184         MessageTemplate.or(MessageTemplate.MatchProtocol("thereIsSwap"),
185         MessageTemplate.or(MessageTemplate.MatchProtocol("swapTokenFound"),
186         MessageTemplate.or(MessageTemplate.MatchProtocol("firstTokenSwapped"),
187         MessageTemplate.or(MessageTemplate.MatchProtocol("conveyedAllPieces"),
188         MessageTemplate.or(MessageTemplate.MatchProtocol("piecesToConveyor"),
189         MessageTemplate.MatchProtocol("aliveToo"))))))))));
190
191     ACLMessage messageReceived;
192     Integer myToken = -1;
193     Integer tokenTransition = 0;
194     Integer numberAllPiecesConveyed = 0;
195     boolean listenerStarted = false;
196     boolean conveyedLastPiece = false;
197     boolean firstInput = false;
198     boolean mayBFirstInput = false;
199     boolean waitingMessage = false;
200     boolean swapAlet = false;
201     boolean pieceTransition = false;
202     boolean debounce = false;
203     boolean debounce2 = false;
204
205     int numberPieces = 0;
206     int numberOfConveyors = 1;
207     int swaPosition;
208     long timeOut = 5000;
209     long timeOutFirstConveyor = 5000;
```

```
210   long timeOfSent;
211   long timeOfConveyedLastPiece;
212
213   ArrayList<String> listOfAgents = new ArrayList<>();;
214
215   public ConveyorBehaviour(Agent a) {
216     super(a);
217
218     MOTOR = GPIO.provisionDigitalOutputPin(RaspiPin.GPIO_04, "MOTOR",
        ↪  PinState.LOW);
219     inputSensor = GPIO.provisionDigitalInputPin(RaspiPin.GPIO_05,
        ↪  PinPullResistance.PULL_DOWN);
220     outputSensor = GPIO.provisionDigitalInputPin(RaspiPin.GPIO_06,
        ↪  PinPullResistance.PULL_DOWN);
221     System.out.println(String.format("[%s] - The conveyor Behaviour
        ↪  started", myAgent.getLocalName()));
222     lastStateInput = PinState.LOW;
223     lastStateOutput = PinState.LOW;
224   }
225
226
227   public void action() {
228     if(!listenerStarted) {
229
230       /* 1 ------------------------------- Handle Input sensor
          ↪  ----------------------------------------------*/
231       /*  - Here the agent will learn its own token and send a message
          ↪  to other agents*/
```

```
232         /*  - The agent send a message if occur one of the following
            ↪  cases:
233          *     1- Inform the last conveys that the piece arrived in the
             ↪  input sensor
234          *     2- That it found the swap of tokens
235          *     3- Inform, case it is the first token, the number of pieces
             ↪  that are entering in the input sensor
236          *     4- Case it is the last token, inform the number of pieces
             ↪  remaining to pass in the output sensor */
237
238         inputSensor.addListener(new GpioPinListenerDigital() {
239
240           public void
               ↪  handleGpioPinDigitalStateChangeEvent(GpioPinDigitalStateChangeEvent
               ↪  event) {
241             if(!debounce) {
242               if(event.getState().equals(PinState.HIGH) &&
                   ↪  lastStateInput.equals(PinState.LOW)) {
243                 debounce=true;
244                 MOTOR.high();
245                 lastStateInput = PinState.HIGH;
246                 numberPieces+= 1;
247                 mayBFirstInput = true;
248                 pieceTransition = false;
249                 System.out.println(numberPieces);
250                 // Learn the token position, this part of the code will
                    ↪  run once in each agent.
251                 if(myToken<=0) {
252                   if(tokenTransition != 0) {
```

```
253                 myToken =
                     ↪  Integer.valueOf(tokenTransition.intValue()+1);
254                 System.out.println(String.format("[%s] - Token
                     ↪  position - [%s]",myAgent.getLocalName(),myToken));
255                 getDataStore().put("MyToken", myToken);
256               }else {
257                 tokenTransition = Integer.valueOf(1);
258                 myToken = tokenTransition;
259                 getDataStore().put("MyToken", myToken);
260                 firstInput = true;
261                 conveyedLastPiece = false;
262               }
263             }else if(swapAlet) {
264               swapAlet = false;
265               informSwapFound(myToken.intValue(), swaPosition);
266
267               if(myToken.intValue() == numberOfConveyors){ // Case the
                   ↪  agent is the last token
268                 DFAgentDescription[] contactList = findContacts();
269                 ACLMessage piecesToConveyor =
                     ↪  createMessage("piecesToConveyor",
                     ↪  numberAllPiecesConveyed, contactList);
270                 myAgent.send(piecesToConveyor);
271               }
272               myToken = Integer.valueOf(swaPosition);
273               getDataStore().put("MyToken", myToken);
274
275             }
276
```

```
277                 if(myToken.intValue() == 1) { //Case the agent is the first
                    ↪  token
278                   Integer inputPiece = Integer.valueOf(1);
279                   if(conveyedLastPiece) {
280                     firstInput = true;
281                     conveyedLastPiece = false;
282                   }
283                   DFAgentDescription[] contactList =findContacts();
284                   ACLMessage numberOfpieces =
                      ↪  createMessage("piecesToConveyor", inputPiece,
                      ↪  contactList);
285                   myAgent.send(numberOfpieces);
286                 }
287
288                 // Send TokenTransitionInput message to all contacts
289                 DFAgentDescription[] contactList = findContacts();
290                 ACLMessage inputMessage =
                    ↪  createMessage("tokenTransitionInput", myToken,
                    ↪  contactList);
291                 myAgent.send(inputMessage);
292                 System.out.println(String.format("[%s] - send - [%s] - to
                    ↪  all contacts",myAgent.getLocalName(),
293                     inputMessage.getProtocol()));
294
295                 try {
296                   Thread.sleep(550);
297                   debounce=false;
298                   lastStateInput = PinState.LOW;
299                   System.out.println("Released the debouce");
```

```
300             } catch (Exception e) {
301               System.err.print(e);
302             }
303
304           }
305         }
306       }
307     });
308
309     /* 2 ------------------------------ Handle the Output sensor
        ↪  ---------------------------------*/
310     /*  - Here the agent will send a message to the agent behind it so
        ↪  the other agent can turn off its motor
311      *   - Moreover, if the agent is the last token, it will count the
         ↪  number the pieces that are remaining and,
312      * case the value is zero, the agent will turn off its motor and
         ↪  send a message to inform that the all pieces
313      * passed through it*/
314
315     outputSensor.addListener(new GpioPinListenerDigital() {
316       public void
          ↪  handleGpioPinDigitalStateChangeEvent(GpioPinDigitalStateChangeEvent
          ↪  event) {
317         if(!debounce) {
318           if(event.getState().equals(PinState.HIGH) &&
              ↪  lastStateOutput.equals(PinState.LOW)) {
319             debounce = true;
320             lastStateOutput = PinState.HIGH;
321             mayBFirstInput = false;
```

```
322
323             // if the agent is the last conveyor
324             if(numberOfConveyors == myToken.intValue()) {
325               numberPieces-=1;
326               numberAllPiecesConveyed-=1;
327               System.out.println(numberPieces);
328               if(numberPieces <= 0 && !pieceTransition) {
329                 MOTOR.low();
330                 waitingMessage = false;
331                 numberPieces = 0;
332
333                 //Send a ConveyedAllPieces message to all contacts
334                 if(numberAllPiecesConveyed.intValue() <= 0) {
335                   numberAllPiecesConveyed = 0;
336                   DFAgentDescription[] contactList = findContacts();
337                   ACLMessage msg = createMessage("conveyedAllPieces",
                      ↪ myToken, contactList);
338                   myAgent.send(msg);
339                   System.out.println(String.format("[%s] - send - [%s]
                      ↪ - to all contacts",myAgent.getLocalName(),
340                       msg.getProtocol()));
341                 }
342               }
343
344             } else {
345
346               // Send a TokenTransition message to all contacts
347               DFAgentDescription[] contactList = findContacts();
```

```
348                 ACLMessage outputMessage =
                    ↪  createMessage("tokenTransitionOutput", myToken,
                    ↪  contactList);
349                 System.out.println(String.format("[%s] - send - [%s] - to
                    ↪  all contacts",myAgent.getLocalName(),
350                     outputMessage.getProtocol()));
351                 timeOfSent = System.currentTimeMillis();
352                 waitingMessage = true;
353                 myAgent.send(outputMessage);
354               }
355               try {
356                 Thread.sleep(350);
357                 debounce = false;
358                 lastStateOutput = PinState.LOW;
359               } catch (Exception e) {
360                 System.err.print(e);
361               }
362             }
363
364           }
365         }
366       });
367       listenerStarted = true;
368     }
369
370     /* 3 ---------------------------------------- Handle message
        ↪  ----------------------------------------*/
371     /*  - This part of the code is responsible to handle with messages
        ↪  received
```

```
372         *   - Case the time out is reached, something went wrong and the
            ↪  agent inform that the tokens were swapped,
373         * otherwise it call a function to handle the message*/
374
375       messageReceived  = myAgent.receive(mt);
376
377       if(messageReceived == null) {
378         long realTime = System.currentTimeMillis();
379         if((realTime - timeOfSent > timeOut) && waitingMessage) {
380           waitingMessage = false;
381           conveyorSwapped();
382           System.out.println(String.format("[%s] - timeOut reached -
              ↪  checking if there is a swap...",myAgent.getLocalName()));
383         }
384       }else {
385         handleMessage(messageReceived);
386       }
387     }
388
389
390     public boolean done() {
391
392       return false;
393     }
394     public int onEnd() {
395       myAgent.doDelete();
396       return 1;
397     }
398
```

```
399   /* 4 -------------------- Function to handle with the message received
      ↪  ---------------------------------*/
400
401
402   private void handleMessage(ACLMessage message) {
403     if(message.getProtocol().equals("tokenTransitionInput")) {
404       // If the first token is swapped and the timeout is reached
405       long realTime = System.currentTimeMillis();
406       if((myToken.intValue() == 1) && !firstInput && (realTime -
          ↪  timeOfConveyedLastPiece > timeOutFirstConveyor)) {
407         ACLMessage firTokenSwapMessage = message.createReply();
408         firTokenSwapMessage.setPerformative(ACLMessage.INFORM);
409         firTokenSwapMessage.setProtocol("firstTokenSwapped");
410         firTokenSwapMessage.setContent(myToken.toString());
411         myAgent.send(firTokenSwapMessage);
412         System.out.println(String.format("[%s] - The first token was
            ↪  swapped, changing the token", myAgent.getLocalName()));
413         myToken = Integer.parseInt(message.getContent());
414         getDataStore().put("MyToken", myToken);
415
416       } else if((myToken.intValue() == 1) && !firstInput) {
417         firstInput = true;
418         // inform the last token to add a new piece
419         DFAgentDescription[] contactList = findContacts();
420         ACLMessage msg = createMessage("piecesToConveyor",1,
            ↪  contactList);
421         myAgent.send(msg);
422       }
```

```
423         if(myToken.intValue() == (Integer.parseInt(message.getContent()) -1
            ↪  )) {
424           numberPieces -= 1;
425           mayBFirstInput = false;
426           if(numberPieces <= 0) {
427             MOTOR.low();
428             numberPieces = 0;
429             waitingMessage = false;
430             System.out.println(String.format("[%s] - pieces conveyed,
                ↪  waiting a new input...",myAgent.getLocalName()));
431           }
432         }
433       }
434       if(message.getProtocol().equals("tokenTransitionOutput")) {
435         if((myToken.intValue() ==
            ↪  (Integer.parseInt(message.getContent())+1) ||
            ↪  myToken.intValue() <= 0)){
436           pieceTransition = true;
437           MOTOR.high();
438           tokenTransition = Integer.parseInt(message.getContent());
439           System.out.println(String.format("[%s] - received - [%s] - turn
              ↪  on the
              ↪  motor",myAgent.getLocalName(),message.getProtocol().toString()));
440         }
441       }
442       if(message.getProtocol().equals("conveyedAllPieces")){
443         if((myToken.intValue() == 1) && (numberPieces == 0)) {
444           firstInput = false;
445           conveyedLastPiece = true;
```

```
446           numberAllPiecesConveyed = 0;
447           System.out.println(String.format("[%s] - All the pieces was
               ↪  conveyed, waiting to a new
               ↪  piece...",myAgent.getLocalName()));
448           // Debounce time to change de firt piece
449           timeOfConveyedLastPiece = System.currentTimeMillis();
450         }
451       }
452       if(message.getProtocol().equals("thereIsSwap")) {
453         if(myToken.intValue() > Integer.valueOf(message.getContent())){
454           MOTOR.high();
455           swapAlet = true;
456           swaPosition = Integer.valueOf(message.getContent()) + 1;
457           Integer position = swaPosition;
458           System.out.println(String.format("[%s] - received from - [%s] -
               ↪  position of swap - [%s]",
459               message.getProtocol(),message.getSender().getName(),
                   ↪  position.toString()));
460         }
461       }
462       if(message.getProtocol().equals("swapTokenFound")) {
463         swapAlet = false;
464         String delims = "[*]";
465         String[] tokens = message.getContent().split(delims);
466         System.out.println(String.format("[%s] - the agent [%s] found the
               ↪  token",
               ↪  myAgent.getLocalName(),message.getSender().getLocalName()));
467         if(myToken.intValue() == Integer.valueOf(tokens[1])) {
468
```

```
469           myToken = Integer.parseInt(tokens[0]);
470           System.out.println(String.format("[%s] - new token -
              ↪  [%s]",myAgent.getLocalName(),tokens[0]));
471           getDataStore().put("MyToken", myToken);
472
473         }
474       }
475       if(message.getProtocol().equals("firstTokenSwapped") &&
          ↪  mayBFirstInput) {
476         firstInput = true;
477         myToken = Integer.parseInt(message.getContent());
478         getDataStore().put("MyToken", myToken);
479         System.out.println(String.format("[%s] - new token -
            ↪  [%s]",myAgent.getLocalName(),message.getContent()));
480
481         // Send a message to the last conveyor to inform the first piece.
482         DFAgentDescription[] contactList = findContacts();
483         ACLMessage msg = createMessage("piecesToConveyor", numberPieces,
            ↪  contactList);
484         myAgent.send(msg);
485         System.out.println(String.format("[%s] - sent - [PiecesToConveyor]
            ↪  - to the last conveyor",myAgent.getLocalName()));
486       }
487       if(message.getProtocol().equals("leftDF")) {
488         String delims = "[*]";
489         String[] content = message.getContent().split(delims);
490         String sender = content[0];
491         String token_of_sender = content[1];
492
```

```
493         if(myToken.intValue() >Integer.valueOf(token_of_sender)) {
494           int newToken = myToken.intValue() - 1;
495           myToken = Integer.valueOf(newToken);
496           getDataStore().put("MyToken", myToken);
497           System.out.println(String.format("[%s] - leftDF received - new
              ↪  position - [%s]",
              ↪  myAgent.getLocalName(),myToken.toString()));
498         }
499         for(int i=0; i<listOfAgents.size(); i++) {
500           if (listOfAgents.get(i).equals(sender)) {
501             listOfAgents.remove(i);
502             break;
503           }
504         }
505         numberOfConveyors--;
506         numberAllPiecesConveyed = 0;
507         numberPieces = 0;
508         conveyedLastPiece = true;
509       }
510       if(message.getProtocol().equals("informAlive")) {
511         String sender = message.getSender().getLocalName();
512         String senderMqtt = message.getContent();
513         if(myToken.intValue() > 0) {
514           myToken = Integer.valueOf(-1);
515           tokenTransition = 0;
516           firstInput = false;
517           System.out.println(String.format("[%s] - received - [%s] -
              ↪  learning new order...",
518               myAgent.getLocalName(),message.getProtocol().toString()));
```

```
519         }
520
521         if(message.getSender().getLocalName().equals("AgentMqtt")) {
522           listOfAgents.add(senderMqtt);
523           numberOfConveyors+=1;
524         }else {
525           listOfAgents.add(sender);
526           System.out.println(String.format("[%s] - the agent - [%s] - was
              ↪  added in the contact list", myAgent.getLocalName(), sender));
527           numberOfConveyors+=1;
528         }
529         DFAgentDescription[] contactList = findContacts();
530         ACLMessage reply = createMessage("aliveToo", myToken, contactList);
531         reply.setProtocol("aliveToo");
532         reply.setContent(myAgent.getName());
533         myAgent.send(reply);
534         System.out.println(String.format("Number of conveyors
           ↪  [%d]",numberOfConveyors));
535         numberAllPiecesConveyed = 0;
536         numberPieces = 0;
537       }
538
539       if(message.getProtocol().equals("aliveToo")) {
540         String sender = message.getSender().getLocalName();
541         String senderMqtt = message.getContent();
542         if(sender.equals("AgentMqtt")) {
543           analiseInfom(message,senderMqtt);
544         }else {
545           analiseInfom(message, sender);
```

```
546         }
547         System.out.println(String.format("Number of conveyors
            ↪  [%d]",numberOfConveyors));
548
549       }
550       if(message.getProtocol().equals("piecesToConveyor")) {
551         if(myToken.intValue() == numberOfConveyors || myToken.intValue()
            ↪  <0) {
552           numberAllPiecesConveyed += Integer.valueOf(message.getContent());
553           System.out.println(String.format("[%s] - number of pieces to
              ↪  convey -
              ↪  [%s]",myAgent.getLocalName(),numberAllPiecesConveyed));
554         }else {
555           numberAllPiecesConveyed = 0;
556         }
557       }
558     }
559
560     /* 5 ---------------------------- Others functions used in the code
        ↪  ------------------------------------*/
561
562     private void conveyorSwapped() {
563       DFAgentDescription [] contactList = findContacts();
564       ACLMessage swapMessage = createMessage("thereIsSwap", myToken,
          ↪  contactList);
565       myAgent.send(swapMessage);
566       System.out.println(String.format("[%s] - send - [thereIsSwap] - to
          ↪  all contacts",myAgent.getLocalName()));
567     }
```

```
568
569   private void informSwapFound(int lastToken, int newToken) {
570     DFAgentDescription[] contactList = findContacts();
571     ACLMessage swapFoundMessage = createMessage("swapTokenFound",
        ↪  myToken, contactList);
572     swapFoundMessage.setContent(String.valueOf(lastToken)+"*"+String.valueOf(newToken));
573     myAgent.send(swapFoundMessage);
574   }
575
576
577   private DFAgentDescription[] findContacts() {
578     DFAgentDescription[] contactList = null;
579     DFAgentDescription dfd = new DFAgentDescription();
580     ServiceDescription sd = new ServiceDescription();
581     sd.setName("skill");
582     sd.setType("conveyor");
583     dfd.addServices(sd);
584
585     try {
586       contactList = DFService.search(myAgent, dfd);
587     } catch (Exception e) {
588       System.out.println(String.format("[%s] - Something went wrong with
        ↪  the search", myAgent.getLocalName()));
589       System.err.print(e);
590     }
591     return contactList;
592   }
593   private ACLMessage createMessage(String protocol, Integer token,
      ↪  DFAgentDescription[] contactList) {
```

```
594     ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
595     msg.setSender(myAgent.getAID());
596     msg.setProtocol(protocol);
597     msg.setContent(token.toString());
598 
599     for(int i=0; i<contactList.length;i++) {
600       DFAgentDescription contact = contactList[i];
601       if(!contact.getName().getName().equals(myAgent.getName())) {
602         msg.addReceiver(contact.getName());
603       }
604     }
605     return msg;
606   }
607 
608   private void analiseInfom(ACLMessage message, String sender) {
609     boolean thereIsAgent = false;
610     for(int i=0; i<listOfAgents.size(); i++) {
611       if(listOfAgents.get(i).equals(sender)) {
612         thereIsAgent = true;
613         }
614       }
615     if(!thereIsAgent) {
616       listOfAgents.add(sender);
617       numberOfConveyors++;
618       System.out.println(String.format("[%s] - received - [aliveToo] ",
        ↪  myAgent.getLocalName()));
619       System.out.println(String.format("[%s] - the agent - [%s] - was
        ↪  added in the contact list", myAgent.getLocalName(), sender));
620     }
```

```
621     }
622
623 }
624
625
```

Appendix D

SPADE Iterface Agent

```
1  import asyncio
2
3  import spade
4  from spade.agent import Agent
5  from spade.behaviour import CyclicBehaviour
6  from spade.message import Message
7  from interface_register import InterfaceRegister
8  import paho.mqtt.client as mqtt
9  from configs.agentConfigs import *
10 import json
11 import signal
12
13 """"
14 ---------------------------------------- Interface Agent
   ↪  ------------------------------------------------------------
15 - SPADE 3.3.0
16
```

```
17 This agent is responsible for dealing with messages sent from your framework and topics to which it is subscribed.
18 It features two filter mechanisms:
19
20     1 - When a message arrives from his framework, he transforms it into a json following the FIPA-ACL standard and
21     publishes it in the topic related to the agent who sent it.
22
23     2 - When the message arrives in the topic to which it is subscribed, it reads the JSON containing
24     the FIPA-ACL parameters, create a native message and sends it to its contacts intra-framework.
25
26
27 """
28
29
30 class InterfaceAgent(Agent):
31     class InterfaceBehaviour(CyclicBehaviour):
32
33         def __init__(self, jid):
34             super().__init__()
35
36             self._host_mqtt = mqtt_broker_configs["HOST"]
37             self._port_mqtt = mqtt_broker_configs["PORT"]
38             self._ca_file = None if mqtt_broker_configs["CA_CERTS"] == "" else mqtt_broker_configs["CA_CERTS"]
39             self._cert_file = None if mqtt_broker_configs["CERT_FILE"] == "" else mqtt_broker_configs["CERT_FILE"]
```

```
40              self._key_file = None if mqtt_broker_configs["KEY_FILE"] ==
                ↪  "" else mqtt_broker_configs["KEY_FILE"]
41              self._user_name = None if mqtt_broker_configs["USERNAME"] ==
                ↪  "" else mqtt_broker_configs["USERNAME"]
42              self._password = None if mqtt_broker_configs["PASSWORD"] ==
                ↪  "" else mqtt_broker_configs["PASSWORD"]
43
44              self._jid = str(jid)
45              self._mqtt_client = None
46              self._acl_message = {}
47              self._message_to_send = []
48              self._new_message = False
49              self._mqtt_message_received = 0
50              self._frameworks = []
51
52          async def on_start(self) -> None:
53
54              print(f'[{self._jid}] - Behaviour MQTT started - ')
55              self.client_connection(self._host_mqtt, self._port_mqtt,
                ↪  self._jid, self._user_name,
56                                     self._password, self._ca_file,
                                       ↪  self._cert_file, self._key_file)
57              signal.signal(signal.SIGINT, self.handle_keyboard_interrupt)
58
59          async def on_end(self) -> None:
60              print("The agent ended")
61
62          async def run(self) -> None:
63
```

```
64              """This part the agent will send messages received from
                ↪  toSpics to its contacts"""
65              while self._message_to_send:
66                  contact_list = self.presence.get_contacts()
67                  if not contact_list:
68                      print("No contacts available to send messages.")
69                      break
70
71                  messages_to_send = self._message_to_send.pop(0)
72
73                  send_task = [self.send_message(contact=contact,
                    ↪  message_to_send=messages_to_send)
74                               for contact in contact_list]
75                  try:
76                      await asyncio.gather(*send_task)
77                  except Exception as e:
78                      print(f"Error to send the message {e}")
79
80              """" * ----------------------------- Handle message
                ↪  ------------------------------------------"""
81              message_received = await self.receive()
82              if message_received is not None:
83                  spade_message =
                    ↪  self.handle_msg_interface_ag(spade_message=message_received)
84                  topic = "SPADE/" + str(spade_message["sender"])
85                  message_json = json.dumps(spade_message)
86                  self._mqtt_client.publish(topic=topic,
                    ↪  payload=message_json, qos=0)
87
```

```
88          """" * ------------------------ Functions used in the code
            ↪  ------------------------------------"""
89
90          def client_connection(self, broker_ip: str, port: str,
            ↪  client_name: str, user_name=None, password=None,
91                                ca_file=None, certificate_file=None,
                                  ↪  key_file=None, keepalive=60) -> None:
92
93              self._mqtt_client = mqtt.Client(client_name)
94              if (user_name and password) is not None:
95                  self._mqtt_client.username_pw_set(user_name, password)
96              if(ca_file and certificate_file and key_file) is not None:
97                  self._mqtt_client.tls_set(self._ca_file, self._cert_file,
                    ↪  self._key_file)
98
99              # callback
100             self._mqtt_client.on_connect = self.on_connect
101             self._mqtt_client.on_message = self.on_message
102             self._mqtt_client.on_subscribe = self.on_subscribe
103
104             self._mqtt_client.connect(broker_ip, port, keepalive)
105             self._mqtt_client.loop_start()
106
107         def on_connect(self, client, userdata, flags, rc):
108             """"Read the config file to subscribe in topics of
                ↪  interest"""
109
110             print(f'[{self._jid}] - connected in the broker -')
111             for index, topic in enumerate(mqtt_broker_configs["TOPICS"]):
```

```
112                 client.subscribe(topic)
113                 framework = topic.split("/")
114                 if framework:
115                     self._frameworks.append(framework[0])
116 
117         def on_subscribe(self, client, userdata, mid, granted_qos):
118             print(f'[{self._jid}] - subscribed , Qos: [{granted_qos}]')
119 
120         def on_message(self, client, userdata, message):
121             message_received = None
122             mqtt_message = message.payload.decode()
123             topic = message.topic.split("/")
124             mqtt_json = json.loads(mqtt_message)
125 
126             """function to handle the incoming message from the
                ↪  topics"""
127             message_received =
                ↪  self.handle_msg_interface_ag(mqtt_message=mqtt_json,
                ↪  mqtt_topic=topic)
128 
129             if message_received is not None:
130                 self._message_to_send.append(message_received)
131 
132         def handle_msg_interface_ag(self, spade_message=None,
            ↪  mqtt_message=None, mqtt_topic=None):
133             """"This function is the heart of the agent. It will check
                ↪  which framework the message was sent
134              from and will take the appropriate action to forward it to
                ↪  the agents in its framework if it was sent by
```

```
135             another interface agent
136             """
137
138             acl_message = {
139                 "performative": "inform",
140                 "sender": None,
141                 "receiver": None,
142                 "reply-to": None,
143                 "content": None,
144                 "language": None,
145                 "encoding": None,
146                 "ontology": None,
147                 "protocol": None,
148                 "conversation-id": None,
149                 "reply-with": None,
150                 "in-reply-to": None,
151                 "reply-by": None
152             }
153
154             """Message sent from its framework. Prepare it to be in a
                ↪ JSON file following the FIPA-ACL standard"""
155             if spade_message is not None:
156                 if spade_message.get_metadata("performative") is not
                    ↪ None:
157                     acl_message["performative"] =
                        ↪ spade_message.get_metadata("performative")
158
159                 acl_message["sender"] = spade_message.sender
```

```
160                 acl_message["receiver"] =
                    ↪  spade_message.get_metadata("receiver")
161                 acl_message["reply-to"] =
                    ↪  spade_message.get_metadata("reply-to")
162                 acl_message["content"] = str(spade_message.body)
163                 acl_message["language"] =
                    ↪  spade_message.get_metadata("language")
164                 acl_message["encoding"] =
                    ↪  spade_message.get_metadata("encoding")
165                 acl_message["ontology"] =
                    ↪  spade_message.get_metadata("ontology")
166                 acl_message["protocol"] =
                    ↪  spade_message.get_metadata("protocol")
167                 acl_message["conversation-id"] =
                    ↪  spade_message.get_metadata("conversation-id")
168                 acl_message["reply-with"] =
                    ↪  spade_message.get_metadata("reply-with")
169                 acl_message["in-reply-to"] =
                    ↪  spade_message.get_metadata("in-reply-to")
170                 acl_message["reply-by"] =
                    ↪  spade_message.get_metadata("reply-by")
171
172             if mqtt_message is not None and ("JADE" in self._frameworks):
173                 if mqtt_topic[0] == "JADE":
174                     acl_message["performative"] =
                        ↪  mqtt_message.get("performative")
175                     acl_message["sender"] = mqtt_message.get("sender")
176                     acl_message["receiver"] =
                        ↪  mqtt_message.get("receiver")
```

```
177                     acl_message["reply-to"] =
                        ↪ mqtt_message.get("reply-to")
178                     acl_message["content"] = mqtt_message.get("content")
179                     acl_message["language"] =
                        ↪ mqtt_message.get("language")
180                     acl_message["encoding"] =
                        ↪ mqtt_message.get("encoding")
181                     acl_message["ontology"] =
                        ↪ mqtt_message.get("ontology")
182                     acl_message["protocol"] =
                        ↪ mqtt_message.get("protocol")
183                     acl_message["conversation-id"] =
                        ↪ mqtt_message.get("conversation-id")
184                     acl_message["reply-with"] =
                        ↪ mqtt_message.get("reply-with")
185                     acl_message["in-reply-to"] =
                        ↪ mqtt_message.get("in-reply-to")
186                     acl_message["reply-by"] =
                        ↪ mqtt_message.get("reply-by")
187
188             return acl_message
189
190         def create_template(self, message):
191             msg_template = Message()
192             msg_template.sender = self._jid  # message["sender"]
193             msg_template.set_metadata("performative",
                ↪ str(message.get("performative")))
194             msg_template.set_metadata("receiver",
                ↪ str(message.get("receiver")))
```

```
195             msg_template.set_metadata("reply-to",
                ↪  str(message.get("reply-to")))
196             msg_template.set_metadata("content",
                ↪  str(message.get("content")))
197             msg_template.set_metadata("language",
                ↪  str(message.get("language")))
198             msg_template.set_metadata("encoding",
                ↪  str(message.get("encoding")))
199             msg_template.set_metadata("ontology",
                ↪  str(message.get("ontology")))
200             msg_template.set_metadata("protocol",
                ↪  str(message.get("protocol")))
201             msg_template.set_metadata("conversation-id",
                ↪  str(message.get("conversation-id")))
202             msg_template.set_metadata("reply-with",
                ↪  str(message.get("reply-with")))
203             msg_template.set_metadata("in-reply-to",
                ↪  str(message.get("in-reply-to")))
204             msg_template.set_metadata("reply-by",
                ↪  str(message.get("reply-by")))
205             msg_template.body = str(message.get("content"))
206             return msg_template
207
208         async def send_message(self, contact, message_to_send):
209             for message in message_to_send:
210                 msg = self.create_template(message)
211                 msg.to = str(contact)
212                 await self.send(msg)
213
```

```
214         def handle_keyboard_interrupt(self, sign, frame):
215             self.kill()
216 
217     async def setup(self) -> None:
218         interface_behaviour = self.InterfaceBehaviour(self.jid)
219         behaviour_register = InterfaceRegister(self.jid, self)
220         self.add_behaviour(interface_behaviour)
221         self.add_behaviour(behaviour_register)
222 
223         print(f'[{self.jid}] - Agent It is alive - ')
224 
225 
226 async def main():
227     agent = InterfaceAgent(jid=xmpp_configs.get("JID"),
        ↪  password=xmpp_configs.get("PASSWORD"))
228     await agent.start(auto_register=True)
229     await spade.wait_until_finished(agent)
230 
231 
232 if __name__ == "__main__":
233     spade.run(main())
234 
235 
236 \end{lstlisting}
237 
238 \section{Behavior for register and monitoring the contact
239 list}
240 
241 \begin{lstlisting}[style=pythonstyle]
```

```
242 import asyncio
243 from spade.behaviour import CyclicBehaviour
244
245 '''
246 ##################### This Class is responsible to inform and register to
    ↪  other agents  ######################
247
248 '''
249
250
251 class InterfaceRegister(CyclicBehaviour):
252
253     def __init__(self, jid, agent):
254         super().__init__()
255         self.jid = jid
256         self._register_state = None  # Variable responsible to register
            ↪  once in contact list
257
258     def on_agent_subscribed(self, jid):
259         print(f'The agent {self.agent.name} accepted the subscription
            ↪  from {jid.split("@")[0]}')
260
261     def on_agent_subscribe(self, jid):
262         print(f'The agent {jid.split("@")[0]} wants to subscribe in the
            ↪  list of contacts')
263         self.presence.approve(jid)
264
265     async def on_start(self) -> None:
266         print(' - Register behaviour started - ')
```

```
267         self._register_state = True
268
269     async def on_end(self) -> None:
270         print(' - Register behaviour ended - ')
271
272     async def run(self) -> None:
273
274         self.presence.on_subscribe = self.on_agent_subscribe
275         self.presence.on_subscribed = self.on_agent_subscribed
276
277         # Find all the jid's of agents and try to register in the
            ↪  contact list
278
279         if self._register_state:
280             with open("Jid's.txt", "r") as file:
281                 lines = file.readlines()
282
283             for line in lines:
284                 jid = line.strip()
285                 if str(self.jid) != jid:  # Compare the agent jid with
                    ↪  the jid from the list
286                     try:
287                         self.presence.subscribe(jid)
288                         print(f'Agent - [{self.agent.name}] - Sent a
                            ↪  message register to [{jid}] ')
289                     except Exception as e:
290                         print(f'Behaviour unexpected, cause {e}')
291
292             self._register_state = False
```

```
293         print('register behaviour checked')
294
295         await asyncio.sleep(20)
```

Appendix E

JADE Interface Agent

E.1 Agent

```
1  package agent;
2
3  import jade.core.Agent;
4  import jade.core.behaviours.SimpleBehaviour;
5  import jade.domain.DFService;
6  import jade.domain.FIPAAgentManagement.DFAgentDescription;
7  import jade.domain.FIPAAgentManagement.ServiceDescription;
8
9
10
11 public class InterfaceAgent extends Agent{
12
13   private static final long serialVersionUID = 1L;
14
15   protected void setup() {
16
```

```
17      // Regirter in DF
18      Object[] args = getArguments();
19      String serviceName = (String) args[0];
20      String typeOfService = (String) args [1];
21
22      DFAgentDescription dfd = new DFAgentDescription();
23      ServiceDescription sd = new ServiceDescription();
24      dfd.setName(getAID());
25      sd.setName(serviceName);
26      sd.setType(typeOfService);
27      dfd.addServices(sd);
28
29      boolean registered = false;
30      try {
31        DFService.register(this, dfd);
32        registered = true;
33      } catch (Exception e) {
34        System.err.print(e);
35      }
36      if(!registered) {
37        try {
38          DFService.deregister(this);
39          DFService.register(this, dfd);
40        } catch (Exception e) {
41          System.err.print(e);
42        }
43      }
44      SimpleBehaviour mqttBehaviour = new InterfaceBehaviour(this);
45      this.addBehaviour(mqttBehaviour);
```

```
46      System.out.println("-ALIVE-");
47    }
48
49  protected void takeDown() {
50
51      /*Deregister from DF */
52
53      DFAgentDescription dfd = new DFAgentDescription();
54      dfd.setName(getAID());
55      try {
56        DFService.deregister(this, dfd);
57      } catch (Exception e) {
58        System.out.println(String.format("[%s] - Something went wrong with
          ↪  deregister", this.getLocalName()));
59        System.err.print(e);
60      }
61
62
63    }
64
65  }
66
67  \end{lstlisting}
68
69  \section{Behaviour}
70  \begin{lstlisting}[style=javastyle]
71  package agent;
72
73  import java.util.ArrayList;
```

```
74 import java.util.HashMap;
75 import java.util.Map;
76
77 import javax.net.ssl.SSLSocketFactory;
78
79 import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
80 import org.eclipse.paho.client.mqttv3.MqttCallback;
81 import org.eclipse.paho.client.mqttv3.MqttClient;
82 import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
83 import org.eclipse.paho.client.mqttv3.MqttException;
84 import org.eclipse.paho.client.mqttv3.MqttMessage;
85 import org.eclipse.paho.client.mqttv3.MqttPersistenceException;
86 import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;
87
88 import jade.core.AID;
89 import jade.core.Agent;
90 import jade.core.behaviours.SimpleBehaviour;
91 import jade.domain.DFService;
92 import jade.domain.FIPAAgentManagement.DFAgentDescription;
93 import jade.domain.FIPAAgentManagement.ServiceDescription;
94 import jade.lang.acl.ACLMessage;
95
96 import org.json.JSONArray;
97 import org.json.JSONObject;
98
99 import java.io.FileInputStream;
100 import java.util.Queue;
101 import java.util.LinkedList;
102 import java.util.List;
```

```
103 import java.util.Properties;
104
105
106 public class InterfaceBehaviour extends SimpleBehaviour{
107   private static final long serialVersionUID = 1L;
108
109   MqttClient client;
110   ACLMessage messageReceived;
111   String mqttMessage;
112   boolean connected = false;
113   final String CONF_FILE_PATH = "mqttConfig.properties";
114   Queue<ACLMessage> messageQueue = new LinkedList<ACLMessage>();
115   private List<String> frameworks = new ArrayList<>();
116
117
118   public InterfaceBehaviour(Agent a) {
119     super(a);
120   }
121
122   public void action() {
123     if(!connected) {
124       /*Read the configuration file to connect to the broker*/
125
126       Properties config = readConfigFile(CONF_FILE_PATH);
127       String serverUrl = config.getProperty("mqtt.host");
128       String caFilePath = config.getProperty("mqtt.ca.file");
129       String clientCrtFilePath = config.getProperty("mqtt.client.crt");
130       String clientKeyFilePath = config.getProperty("mqtt.client.key");
131       String mqttUserName = config.getProperty("mqtt.username");
```

```
132         String mqttPassword = config.getProperty("mqtt.password");
133         String topics = config.getProperty("mqtt.topics.to.subscribe");
134         String[] topicsToSub = topics.split(",");
135 
136         MemoryPersistence persistence = new MemoryPersistence();
137         try {
138           client = new MqttClient(serverUrl,
              ↪ myAgent.getLocalName(),persistence);
139         } catch (MqttException e) {
140 
141           System.out.println("Something went rong.");
142         }
143 
144         try {
145           // - Setting the broker parameters -
146           MqttConnectOptions options = new MqttConnectOptions();
147           options.setPassword(mqttPassword.toCharArray());
148           options.setUserName(mqttUserName);
149           options.setConnectionTimeout(60);
150           options.setKeepAliveInterval(60);
151           options.setMqttVersion(MqttConnectOptions.MQTT_VERSION_3_1);
152 
153           SSLSocketFactory socketFactory =
              ↪ SslUtil.getSocketFactory(caFilePath,
154               clientCrtFilePath, clientKeyFilePath, "");
155           options.setSocketFactory(socketFactory);
156 
157           client.setCallback(new MqttCallback() {
```

```
158         public void messageArrived(String topic, MqttMessage message)
            ↪  throws Exception {
159           String[] topics = topic.split("/") ;
160
161           try {
162             ACLMessage aclMessage;
163             mqttMessage = new
                ↪  String(message.getPayload());
164             JSONObject msg = new JSONObject(mqttMessage);
165
166             // handle the message received from the
                ↪  topics
167             Map <String, String> jsonMessage = createAclJson(msg);
168
169             // Translating the message to FIPA-ACL
170             aclMessage  = spadeToJade(jsonMessage, topics);
171
172             //Insert the message in the queue
173             if(aclMessage != null) {
174               messageQueue.add(aclMessage);
175             }
176
177           } catch (Exception e) {
178             System.err.println(e);;
179           }
180
181         }
182
183         public void deliveryComplete(IMqttDeliveryToken token) {
```

```
184             System.out.println(String.format("[%s] - the message was sent
                ↪  successfully",myAgent.getLocalName()));
185           }
186
187           public void connectionLost(Throwable cause) {
188             System.out.println(String.format("[%s] - connection lost ",
                ↪  myAgent.getLocalName()));
189             System.err.println(cause);
190           }
191
192         });
193         client.connect(options);
194         for(int i = 0; i<topicsToSub.length; i++) {
195           client.subscribe(topicsToSub[i],0);
196           String[] framework = topicsToSub[i].split("/");
197           frameworks.add(framework[0]);
198         }
199
200       } catch (MqttException e) {
201         e.printStackTrace();
202       } catch (Exception e) {
203         e.printStackTrace();
204       }
205       connected = true;
206     }
207     while (!messageQueue.isEmpty()) {
208
209       ACLMessage aclMessage = messageQueue.poll();
210       DFAgentDescription[] contactList = findContacts();
```

```
211       for(int i=0; i<contactList.length;i++) {
212
              ↪  if(!contactList[i].getName().getName().equals(myAgent.getName()))
              ↪  {
213             AID contact = contactList[i].getName();
214             aclMessage.addReceiver(contact);
215           }
216       }
217
218       myAgent.send(aclMessage);
219     }
220
221 // ----------------------------- Handle Message received from JADE
    ↪  ----------------------------------------------------------
222     messageReceived = myAgent.receive();
223
224     if(messageReceived != null) {
225       try {
226         handleJadeToSpade(messageReceived);
227       } catch (MqttException e) {
228         e.printStackTrace();
229       }
230     }
231   }
232 //
    ↪  ------------------------------------------------------------------------------------
233
234   public boolean done() {
235     return false;
```

```
236   }
237   public int onEnd() {
238     myAgent.doDelete();
239     return 1;
240   }
241
242   protected String findPerformative(int valuePerformative) {
243     String performative = null;
244     switch (valuePerformative) {
245     case 0:
246       performative = "accept-proposal";
247       break;
248     case 1:
249       performative = "agree";
250       break;
251     case 2:
252       performative = "cancel";
253       break;
254     case 3:
255       performative = "cfp";
256       break;
257     case 4:
258       performative = "confirm";
259       break;
260     case 5:
261       performative = "disconfirm";
262       break;
263     case 6:
264       performative = "failure";
```

```
265         break;
266       case 7:
267         performative = "inform";
268         break;
269       case 8:
270         performative = "inform-if";
271         break;
272       case 9:
273         performative = "inform-ref";
274         break;
275       case 10:
276         performative = "not-understod";
277         break;
278       case 11:
279         performative = "propose";
280         break;
281       case 12:
282         performative = "query-if";
283         break;
284       case 13:
285         performative = "query-ref";
286         break;
287       case 14:
288         performative = "refuse";
289         break;
290       }
291       return performative;
292     }
293
```

```
294   protected DFAgentDescription[] findContacts() {
295     DFAgentDescription[] listOfContacts = null;
296     DFAgentDescription dfd = new DFAgentDescription();
297     ServiceDescription sd = new ServiceDescription();
298     sd.setName("skill");
299     sd.setType("Conveyor");
300     dfd.addServices(sd);
301     try {
302       listOfContacts = DFService.search(myAgent, dfd);
303     } catch (Exception e) {
304       System.out.println(e);
305     }
306     return listOfContacts;
307   }
308
309   protected ACLMessage spadeToJade(Map<String, String> messageData,
      ↪  String [] topics) {
310
311     ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
312     String framework = topics[0];
313
314     if(framework.equals("SPADE")){
315       switch (messageData.get("performative")) {
316       case "Inform":
317         msg.setPerformative(ACLMessage.INFORM);
318         break;
319       case "agree":
320         msg.setPerformative(ACLMessage.AGREE);
321         break;
```

```
322         case "accept_proposal":
323           msg.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
324           break;
325         case "cpf":
326           msg.setPerformative(ACLMessage.CFP);
327           break;
328         case "confirm":
329           msg.setPerformative(ACLMessage.CONFIRM);
330           break;
331         case "cancel":
332           msg.setPerformative(ACLMessage.CANCEL);
333           break;
334         }
335         msg.setSender(myAgent.getAID());
336         msg.setProtocol(messageData.get("protocol"));
337         msg.setContent(messageData.get("content"));
338         msg.setLanguage(messageData.get("language"));
339         msg.setOntology(messageData.get("ontology"));
340         msg.setEncoding(messageData.get("encoding"));
341         msg.setInReplyTo(messageData.get("reply-to"));
342         msg.setConversationId(messageData.get("conversation-id"));
343         msg.setReplyWith(messageData.get("reply-with"));
344         msg.setInReplyTo(messageData.get("in-reply-to"));
345 
346       }
347       return msg;
348     }
349     protected void handleJadeToSpade (ACLMessage msgReceived) throws
      ↪ MqttException{
```

```
350     String performative =
        ↪  findPerformative(msgReceived.getPerformative());
351
352     if(frameworks.contains("SPADE")) {
353       Map<String, String> jsonMessage = new HashMap<>();
354       jsonMessage.put("performative", performative);
355       jsonMessage.put("sender", msgReceived.getSender().getName());
356       jsonMessage.put("reply-to", msgReceived.getInReplyTo());
357       jsonMessage.put("content", msgReceived.getContent());
358       jsonMessage.put("language", msgReceived.getLanguage());
359       jsonMessage.put("encoding", msgReceived.getEncoding());
360       jsonMessage.put("ontology", msgReceived.getOntology());
361       jsonMessage.put("protocol", msgReceived.getProtocol());
362       jsonMessage.put("conversation-id",
          ↪  msgReceived.getConversationId());
363       jsonMessage.put("reply-with", msgReceived.getReplyWith());
364       jsonMessage.put("in-reply-to", msgReceived.getInReplyTo());
365
366       String agentTopic = "JADE/" + jsonMessage.get("sender");
367       JSONObject messageToSend = new JSONObject(jsonMessage);
368       MqttMessage jadeMessage = new
          ↪  MqttMessage(messageToSend.toString().getBytes());
369       try {
370         client.publish(agentTopic, jadeMessage);
371       } catch (MqttPersistenceException e) {
372         e.printStackTrace();
373       }
374     }
375   }
```

```
376
377
378 // ---------------------------------- Convert message to JSON
    ↪  -------------------------------
379
380   protected Map<String,String> createAclJson (JSONObject msg){
381     Map <String, String> aclMessage = new HashMap<>();
382     for (String key: msg.keySet()) {
383       Object value = msg.get(key);
384       String valueStr = (value != null && value != JSONObject.NULL) ?
          ↪  value.toString() : "";
385
386       if(("sender".equals(key) || "receiver".equals(key)) && value
          ↪  instanceof JSONArray) {
387         JSONArray localName = (JSONArray) value;
388         valueStr = localName.getString(0);
389       }
390       aclMessage.put(key, valueStr);
391     }
392     return aclMessage;
393   }
394
395   private static Properties readConfigFile (String filePath) {
396     Properties config = new Properties();
397     try (FileInputStream fileInput = new FileInputStream(filePath)){
398       config.load(fileInput);
399
400     } catch (Exception e) {
401       e.printStackTrace();
```

```
402       }
403       return config;
404     }
405
406   }
407
408
```