Middleware for the communication between Distributed Agents in JaKtA

Project for the course *Intelligent Systems Engineering*

Angela Cortecchia
angela.cortecchia@studio.unibo.it

Leonardo Micelli
leonardo.micelli@studio.unibo.it

Filippo Vissani
filippo.vissani@studio.unibo.it

Academic year 2022/2023

Contents

1 Introduction

1.1 Agents

1.1.1 Definition

An agent is an autonomous computational entity endowed with decision-making and independent execution capabilities. Agents are designed to operate in a specific environment, assuming defined responsibilities and tasks. The key characteristic of an agent is its autonomy, which means it can make decisions independently, without requiring external intervention for every action undertaken. This computational autonomy implies intrinsic proactivity, wherein agents don't passively await events but actively act to change the state of the Multi-Agent System (MAS).

A MAS, or Multi-Agent System, is a computational system composed of multiple autonomous agents that interact with each other to achieve common or individual goals. In a MAS, each agent is an autonomous entity with perception, reasoning, decision-making, and action capabilities. Agents in a MAS can operate independently and, simultaneously, collaborate to address complex problems or achieve goals that may be difficult or impossible for a single agent to accomplish.

A crucial aspect is the situational nature of agents, meaning their actions depend on the context in which they find themselves. Agents, by interacting with each other and the surrounding environment, become inherently social entities. This sociality emerges because autonomy makes sense only when an agent is immersed in a society of agents, highlighting the absence of truly autonomous agents in isolation. Interaction occurs through the exchange of knowledge and information, while the proactivity of agents drives them to change the state of the MAS, influencing other agents or the environment. In this context, agents become autonomous, interactive, and social components that, through their self-regulation capabilities, are essential for managing complex systems.

1.1.2 The Framework Belief-Desire-Intention

The Belief-Desire-Intention (BDI) framework is widely accepted as one of the most popular and successful frameworks in the field of agent technology. Defined by Rao and Georgeff, the framework is based on three key concepts: belief, desire, and intention. Agents that adhere to this framework are commonly referred to as BDI agents. The following are the characteristics of the framework:

- **Belief**: To make decisions, agents must have a representation of the world in which they exist. Beliefs are the knowledge about the environment that agents gather and can store as part of their belief base.
- **Desire**: Desires represent the main goals of the system. Agents have a set of plans at their disposal that they can use to achieve their goals, and the choice of the most suitable plan depends on the current situation of the agent and the environment.
- **Intention**: Intentions capture the deliberative component of the system. They represent the actions to which the agent has committed to executing to achieve its goals. Intentions keep track of the progress of actions and changes in the environment during execution.

1.2 JaKtA

JaKtA[5] is a library that provides the capability to develop agents that perceive and act within a shared environment and communicate with each other through a message-based communication system.

The library offers a framework for building multi-agent systems (MAS) composed of agents conforming to the Belief-Desire-Intention (BDI) model and inspired by the implementation of Jason[6]. A Multi-Agent System consists of two fundamental entities:

- **Agents**: Agents are the main entities of the library.
- **Environment**: The environment in which agents live.

During their execution, agents in a MAS autonomously observe the environment's state and choose to act based on perceived information. Their actions can influence the state of the environment, and this change will be perceived by all other agents operating on it. Agents are equipped with goals that they are committed to achieving, contributing to the overall goal of the Multi-Agent System. As for the environment, a user can:

- Create an environment.
- Add functionalities to the environment, such as:
  - Define the actions that agents within the environment can use.
  - Define the state of the environment, including the set of information that agents can perceive and potentially modify.

- Add and remove agents from the environment.

Moreover, agents located in the same environment can communicate with each other through messages. Messages allow agents to share information about their state or share the completion of a goal, making the Multi-Agent System a true society of agents. Regarding agents, a user can:

- Create and customize agents' behavior, following the BDI model.
- Add actions related to an agent.
- Customize the behavior of agents during their execution.

The library enables users to define how their BDI agents will be executed. This means that the same agents can be executed either in a single thread or among multiple threads, in a nearly transparent manner. To enable the MAS to do this, users must define its execution model, i.e., how its agents are executed.

1.3 Agent Life Cycle in JaKtA

The life cycle of an agent essentially serves as the mechanism responsible for managing the execution of the entity. Indeed, at each iteration of the latter, a series of operations must be performed to guarantee the functioning of its BDI model. The first operation that the agent must perform, at each iteration of its life cycle, is to observe whether any changes have occurred within the environment. Indeed, to make the agent as responsive as possible to these changes, before it decides which operations to perform, it must observe the current state of the environment. Once the agent knows the new state of the environment, it must update its knowledge base, i.e. the belief base, with the new information retrieved. To react to changes in the environment's state, and therefore to be a reactive entity, the agent must generate events. These events are determined by evaluating differences between its knowledge base and the new perception of the state around itself. Events can be of two types: addition or deletion of a belief. Another event that the agent must respond to is the receipt of a new message. Messages are the components that agents can use to communicate with each other within the environment. This means that, at each iteration of the life cycle, the Agent has to check if there are any new messages that it needs to handle. Messages can trigger two different scenarios related to their type, indeed, there could be *achieve messages* that trigger a new *achievement goal invocation event* or also *tell messages* that are handled exactly like adding a new belief, the only difference with a generic

one of them is given by the annotation, because it refers to the sender of that message. As with the perception phase, the latter type of message also generates a *belief addition event*. After checking all the event sources that have occurred, the agent selects one event to handle. If there is an event to handle, then it is removed from the agent's event set, and a suitable plan is searched to fulfill it. The plan selection phase follows three steps.

- First, the plans that have a triggering event that unifies with the selected event are filtered among plans available to the agent, these are the relevant plans, and then the applicable ones are selected; this is done by verifying that the plan's context is a logical consequence of the agent's belief base.
- Of the remaining plans, the one applied is chosen. If it is possible to outline a plan that satisfies all these three steps, then this is assigned to an intention. If the event that triggered the plan application was external, then a new intention, with the partially instantiated goals of that plan's body, is added to the intention set. Otherwise, if the event was an internal one, the life cycle retrieves the related intention and adds another activation record, with those goals inside of it, on the top of its stack.
- Lastly, the agent executes one intention goal. The intention to execute is selected and if there's at least one of them to execute then it's scheduled for execution. The execution phase is where the agent could potentially perform changes over its state or on the environment's one.

However, agent life cycle does not operate directly over the environment state, as for action's side effect, the whole reasoning procedure described above returns a list of environment changes, which application will be managed by the Multi-Agent System execution strategy.

1.4 Environment in JaKtA

The *environment* is the entity in which agents live, its state is changed by agents to achieve their goals. Although its behavior can be manipulated by users, this entity has only the goal of agents' synchronization, this means that the update of its state must be managed appropriately by users in multi-threaded systems, although immutable modeling helps to avoid critical runs. The environment encapsulates four important pieces of information: a set of agents living within the environment, the agent's mailboxes on which the

others can send messages, the set of external actions visible at the environment level, and, finally, the component that returns the current state of the environment itself, in terms of belief.

1.5 Middleware for Communication among Distributed Agents

The current implementation of JaKtA focuses on managing multi-agent systems locally, limiting consideration to agents distributed over a network. To address this limitation and enable effective communication among distributed agents, the introduction of a dedicated middleware is proposed. A middleware for communication among distributed agents would act as an intermediate layer between the agents themselves and the network, providing an abstraction that makes the complexities of distributed communication transparent to the programmer. The middleware aims to handle the sending and receiving of messages between agents located on different machines, ensuring information consistency. The middleware would be responsible for establishing and maintaining connections between distributed agents. This includes managing communication protocols, handling lost connections, and addressing any delays or transmission errors. Implementing middleware for communication among distributed agents would enhance JaKtA's ability to support scenarios where agents operate on heterogeneous or geographically distant networks. This would allow the creation of multi-agent systems that can collaborate and coordinate their actions on a larger scale, opening up new application possibilities for the JaKtA library.

2 Requirements Analysis

2.1 Goals

The main goal of the project is to extend the JaKtA library to allow the construction of distributed multi-agent systems over a network while avoiding the library user from having to deal with the complexities of network communications.

2.2 Requirements

Based on the main objective of the project, the following functional, non-functional and implementation requirements were identified.

2.2.1 Functional Requirements

Functional requirements outline the specific functions of the system, that is, what the system is to do:

1 Managing the distribution of agents:

- 1.1 The *middleware* must support the distribution of agents across heterogeneous machines, enabling their coordinated execution on different geographical locations.

2 Implementation of a mechanism to *dispatch* messages between agents:

- 2.1 The *middleware* must exploit a *broker* for network communication:
  - 2.1.1 The *broker* must allow the addition of *publisher* associated with a given *topic*.
  - 2.1.2 The *broker* must allow the removal of *publisher* associated with a specific *topic*.
  - 2.1.3 The *broker* must support the addition of *subscriber* associated with a specific *topic*.
  - 2.1.4 The *broker* must allow the removal of *subscriber* associated with a specific *topic*.
  - 2.1.5 The broker must provide a list of all available topics.
- 2.2 The *middleware* must incorporate a *client* that interfaces with the *broker*:
  - 2.2.1 The *client* must provide the functionality of posting a message on a given *topic*.

2.2.2 The client must provide the ability to subscribe to a specific *topic*.

2.2.3 The system must allow the *client* to perform a broadcast *broadcast*, sending a message to all other connected *clients*.

2.2.4 The *client* must be able to establish and manage *WebSocket* connections to a *broker* specified by *IP* address and port.

3 Distributed Communication and Coordination:

3.1 The *middleware* must handle distributed communication between agents, enabling message exchange and coordination of activities.

3.2 It must support synchronization of distributed agents to ensure cohesive and coordinated interaction.

3.3 It needs to manage disconnections and reconnections of agents, ensuring that the system remains consistent.

3.4 It needs to manage the disconnection of the broker, ensuring that the system remains consistent.

2.3 Non-Functional Requirements

Nonfunctional requirements define quality attributes, constraints, and general properties of the system:

1 The *middleware* must be implemented in separate modules so that it can be easily maintained and extended.

2 The *middleware* must be designed to scale horizontally, easily supporting the addition of new nodes without degrading performance.

2.4 Implementation Requirements

Implementation requirements concern the technological and methodological aspects of system implementation:

1 The project must be implemented using the Kotlin programming language to ensure consistency and compatibility with the JaKtA library.

2 Use the Ktor *framework* to implement the *middleware* for interaction between distributed multi-agent systems.

3 Design

In the following section, the design of the project is presented, starting from the architecture-wise decisions and proceeding with the detailed design.

3.1 Architectural Design

As we have seen in the previous chapters, the project aims to develop a mechanism for the creation of distributed multi-agent systems using *JaKtA*. The team has decided to achieve this goal by providing an extension of the framework and its *Domain Specific Language* that is as minimal as possible. From an architectural point of view, the project is a standalone module, as shown in figure 1:

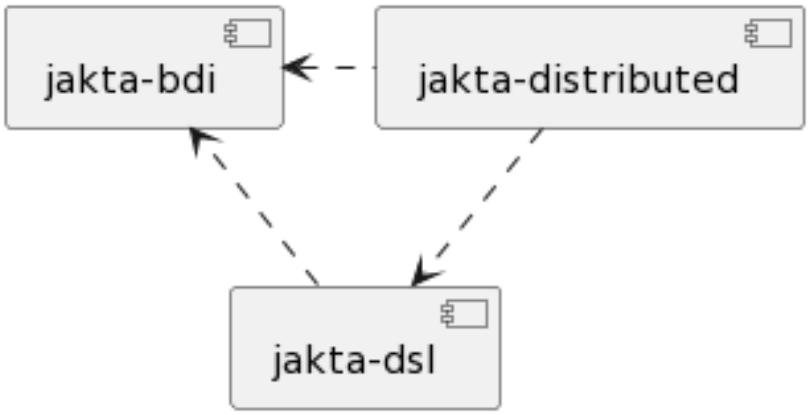

Figure 1: Architecture of the project

In particular, the developed module is composed of three main components, as shown in figure 2:

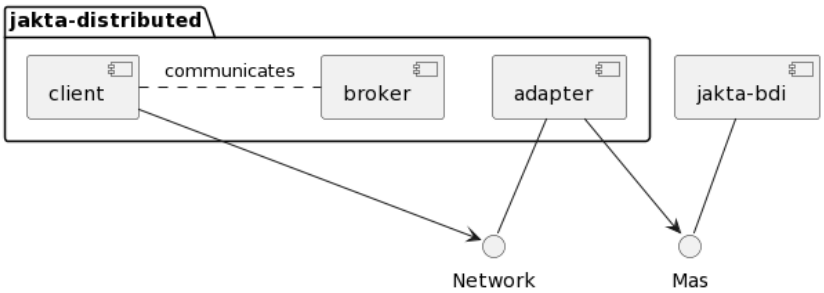

Figure 2: Detailed architecture of the project

Of the modules represented in figure 2, the *Adapter* module is the one that defines and contains all the concepts to be implemented to achieve the project goal, while the *Client* and *Broker* modules provide a first implementation of the communication logic between distributed multi-agent systems.

As previously mentioned, the project also presents an extension of the JaKtA *Domain Specific Language*, whose structure is represented in figure 3.

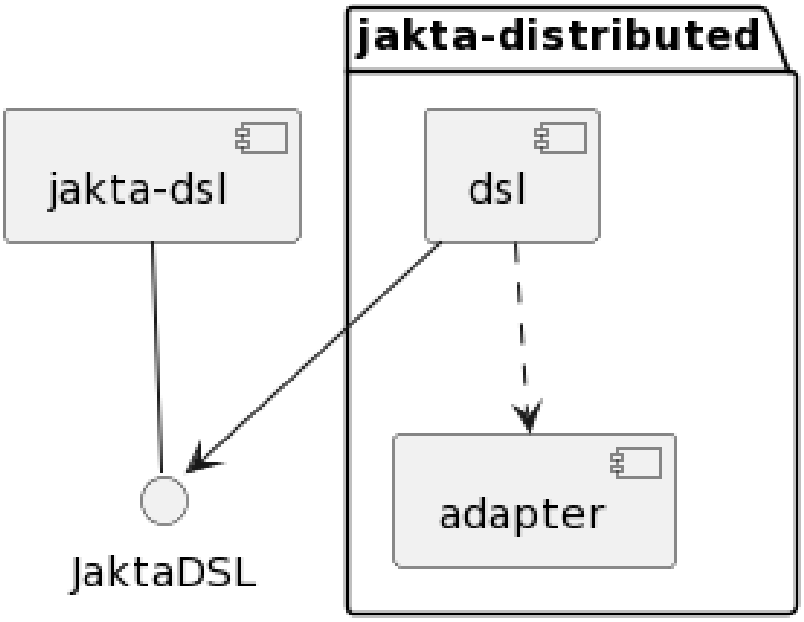

Figure 3: Domain Specific Language architecture

3.1.1 Network Architecture

The component diagram shown in Figure 4 provides an overview of the network architecture.

The *Broker* exposes four interfaces: the */subscribe/topic* and */publish/-topic* interfaces allow respectively the subscription and publication on specific topics, while the topics interface allows to retrieve the list of available topics. Furthermore, the */subscribe-all/*{*except...*} interface offers the possibility to subscribe to all topics except those specified. The *MAS*, on the other hand, connects to the *Broker* through these interfaces, using them to subscribe, publish and obtain information from the other *MAS*. This setup allows the management of distributed communications, with the *Broker* acting as a cen-

tral hub to facilitate the subscription, publication and management of topics within the system.

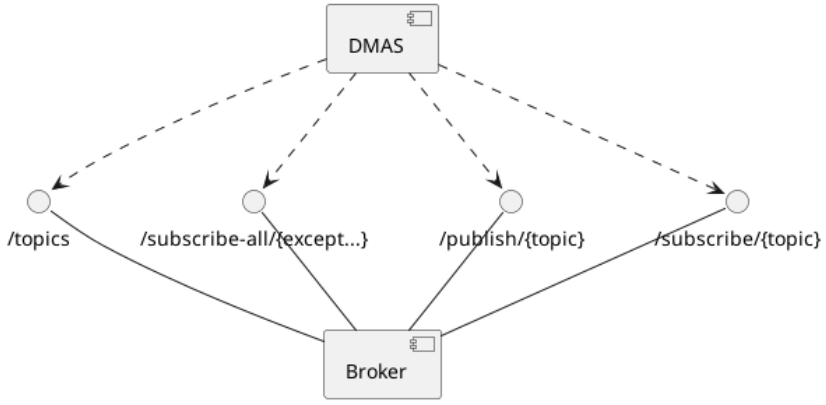

Figure 4: Network architecture

3.2 Detailed Design

When designing an extension of the framework that allows communication between distributed multi-agent systems, it is important to establish what is shared between these systems. To this end, there are several possibilities to consider:

Sharing the Environment : in this case, the agents of each multi-agent system operate in the "same" *Environment*. This means that any modification that may happen on the environment is known by each distributed Mas, for example:

- agent creation;
- agent deletion;
- actions;
- messages exchanged between agents;
- any data added or removed from the environment.

This solution allows for easy programming of the distributed system, since the environment and the agents it contains are the same for each distributed Mas, making it possible for the programmer to write the program as if it were a single Mas. However, this solution presents some technical problems:

1 The environment is a complex data structure, and its sharing between multiple systems requires very robust management of data consistency between the various systems;

2 If the environment is shared, this means that any sort of modification is propagated to the network, to keep the environments in each distributed Mas consistent. This can lead to a lot of network traffic, putting pressure on the broker services;

3 This solution does not allow for the creation of multiple environments, which could be useful in some cases.

Sharing messages : in this case, different multi-agent systems have different environments and agents, but the agents of a system can send messages to the agents of another system, both in a one-to-one fashion and in a one-to-many fashion. This solution embodies the idea of "do not communicate by sharing memory; instead, share memory by communicating" and is a technically easier solution to implement than the previous one, since it does not require a robust management of data consistency between the various systems. Also, it could be more flexible regarding programmer needs since it allows for the creation of multiple environments, and the programmer can decide which messages to share between the various systems.

However, this solution is somewhat limited in some respects:

- The only way to share data between the various systems is through communication, so the programmer needs to instruct the agents to send messages to agents in the other systems;
- Since the environments are not shared, the programmer needs to take care of the consistency of the data between the various systems;
- Again, not sharing the environments means that the programmer needs to design multiple distributed Mas', as opposed to the previous solution.

In summary, this solution removes some complexity from the distributed infrastructure design but puts some on the programmer's shoulders. Regardless, after some discussion, taking into account the project's scope and technical difficulties, the team decided to implement the second solution, valuing its simplicity and flexibility.

So the proposed solution consists of the implementation of an alternative version of the `Mas` interface, called `Dmas`, to be used in distributed contexts. This extension should behave like a `Mas`, but in addition, should be able

to communicate with other `Dmas` through the network. The communication model chosen is the *publish-subscribe* model, which allows for a simple and effective communication between the various `Dmas`. In this context, every agent is associated with a unique *topic*, so to enable an effective communication, it is necessary for two agents to know each other's *topic* in advance. This association enables targeted and organized communication, aiming for agents to focus solely on information relevant to their scope, thereby facilitating the coordination of activities in the whole system.

3.2.1 Adapter

This module defines the concepts of:

- **Dmas**: represents a distributed multi-agent system, i.e. a system that can communicate with other `Dmas` through the network;
- **Network**: encapsulates the logic for the communication between `Dmas`;
- **RemoteService**: represents a remote agent, i.e. an agent that belongs to a `Dmas` different from the current one, but reachable through the network.

The relationships between these concepts are exemplified by the class diagram in figure 5.

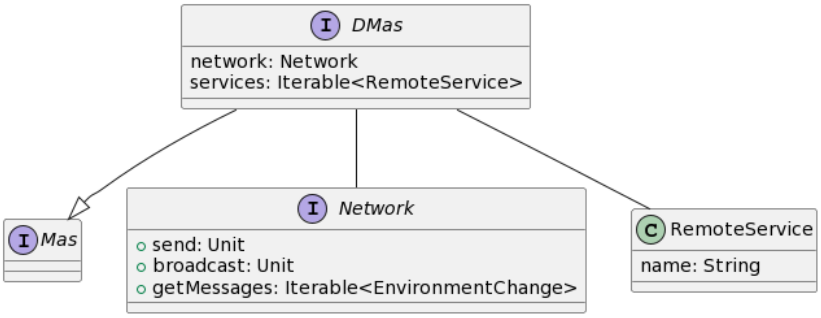

Figure 5: Adapter class diagram

3.2.2 Broker

In the design and implementation of our system, a dedicated broker has been introduced to cater to specific requirements and unique behavioral patterns.

This decision was driven by the need for nuanced control over message publication and subscriptions, differentiating our broker from conventional MQTT or AMQT brokers. Several motivations underpin this choice:

Our broker deviates from the conventional MQTT or AMQT model by enforcing a distinct behavior: for a given topic, only a single client can publish messages at any given time. This ensures a controlled and orderly flow of information, avoiding conflicts that may arise when multiple clients attempt simultaneous publications on the same channel.

While adhering to the exclusive publishing behavior for most topics, exceptions are made for specific topics, such as the "broadcast" topic. Topics falling under this category do not follow the stringent single-client publishing rule, allowing for broader dissemination of information.

A pivotal aspect of our dedicated broker lies in the introduction of specialized error-handling mechanisms. Notably, in the event of a publisher disconnecting, the broker takes proactive measures to notify the disconnection to the subscribers. This tailored approach to error management ensures that clients are promptly informed of disruptions, facilitating responsive and effective handling of disconnection scenarios.

The *broker*'s class diagram (Figure 6) presents a modular and organized structure. Inside the `model` package, two key interfaces are defined: `Cache<T>` for registering, releasing and reading data associated with a *topic*, and `SubscriptionManager<T>` for managing the subscription, addition and removal of *publishers* and *subscribers*, as well as obtaining the available *topics* and the *subscribers* associated with a *topic*. The `plugins` package includes the `Routing` and `Websockets` classes, which are used to define the *Web APIs*.

3.3 Behavior

The behavior of the various components of the system can be described through the following steps.

3.3.1 System Initialization

1 *Network* initialization;

2 connection between Network and *Broker*;

3 definition of the *RemoteServices* of interest;

4 *Dmas* creation;

5 for each RemoteService, the Dmas subscribes to the according *topic*;

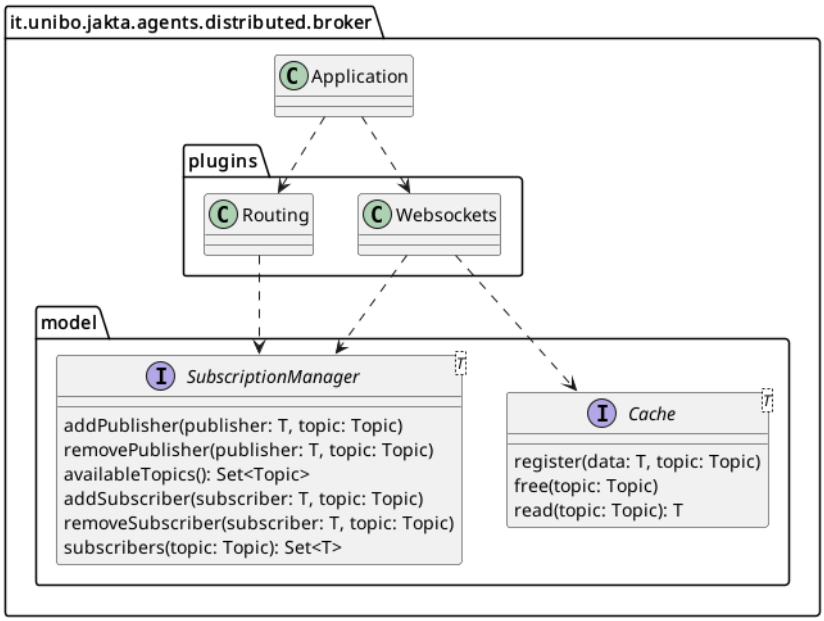

Figure 6: Broker class diagram

6 *ExecutionStrategy* dispatching;

These activities are showcased in the figure 7.

3.3.2 Dmas execution cycle

1 the `Dmas` collects all the external events received from the `Network`, i.e. the messages published by the `RemoteServices` to which it has subscribed and the eventual disconnection notifications of a `RemoteService`, in the form of `EnvironmentChange`;

2 external events are concatenated to the queue of internal EnvironmentChanges of the Dmas;

3 one by one, the `EnvironmentChanges` are processed by the `Dmas`. In case the event consists of sending a message to a `RemoteService` or a *broadcast*, the message is sent to the `Network`, which will forward it to the *broker*;

These activities are showcased in the figure 8.

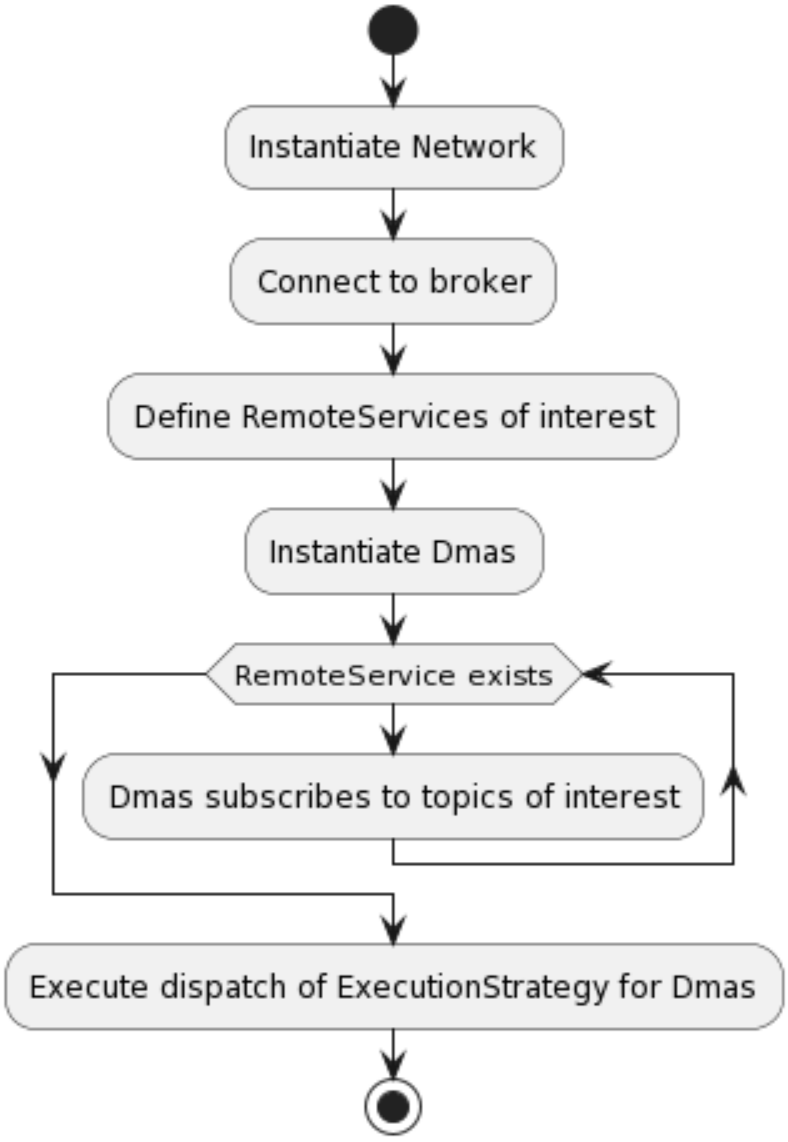

Figure 7: System initialization

3.4 Interaction

The interaction model between the various `Dmas` connected to the network is of the *publish-subscribe* type: each `Dmas`, which assumes the role of *client*, can subscribe to one or more *topics*, and can publish messages on one or more *topics*. Published messages are received by all *clients* that have subscribed to the *topic* to which the message was published. It is relevant to note that

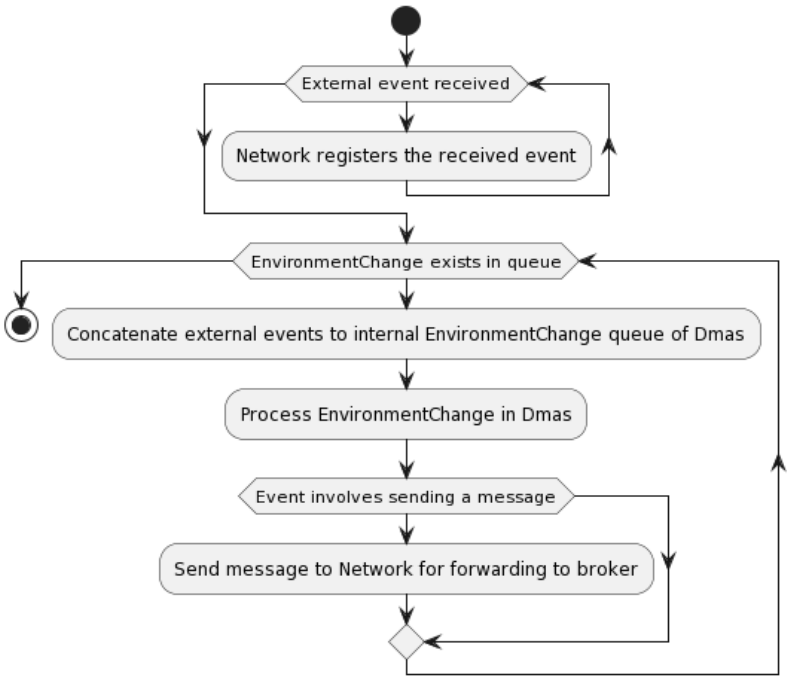

Figure 8: Dmas execution cycle

agents know each other's *topic*.

The *broker*, which assumes the role of server, is responsible for managing the communication between the *clients*, in particular, it is responsible for:

- receiving the messages published by the *clients*;
- sending the messages to the *clients* subscribed to the corresponding *topics*;
- managing the connections and disconnections of the *clients*.
- managing the subscription of the *clients* to the various *topics*.

The behavior of *client* and *broker* in situations of *broadcast* and sending of messages with a single recipient is illustrated in figures 9 and 10.

3.4.1 Publishers with the same name

From the point of view of the Dmas, the interaction in the case of a *sendmessage* is comprised of the following steps:

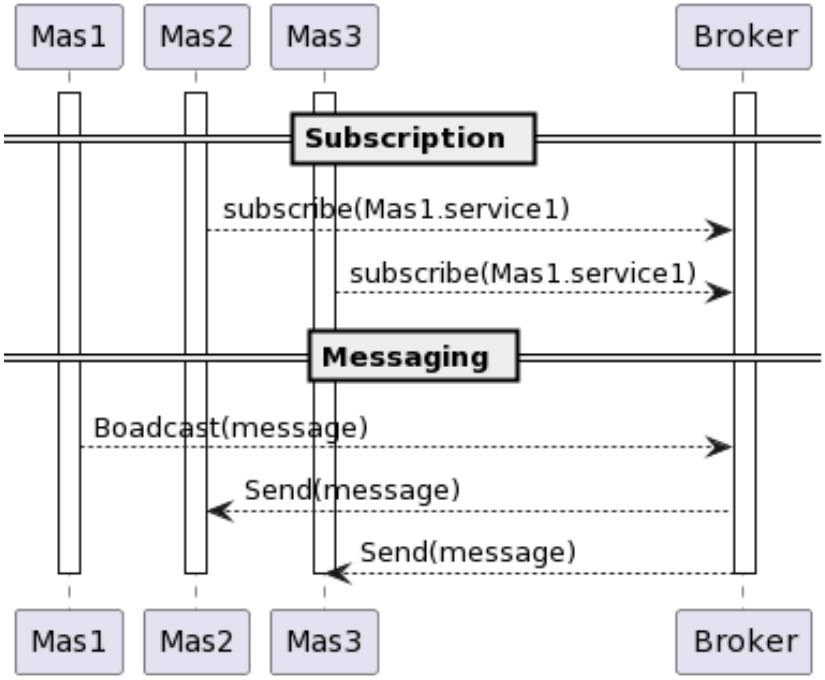

Figure 9: Interaction between DMas, *client* and *broker* in case of *broadcast*

- when an agent in the Dmas sends a message to an agent outside of it, the Dmas sends a message to the *broker* via his *Client* containing the message to be sent and the *topic* of the recipient;
- the broker checks whether or not the publisher's name is valid;
- if the check ended with a success, the message is forwarded, otherwise the broker closes the connection with the client;

this interaction is illustrated in figure 11.

3.4.2 Disconnection management

Since the system is distributed, it is important to assume that disconnections of any participant in the system may occur, be it a `Dmas` or the *broker*. The general strategy of the system is to create, in the event of a disconnection, an `EnvironmentChange` of type `SendMessage` for each agent of the `Dmas` subscribed to the disconnected participant, so that they can be informed of the disconnection and act accordingly. It is important to note that this strategy leaves the task of managing failures to the programmer of the agents of the `Dmas`.

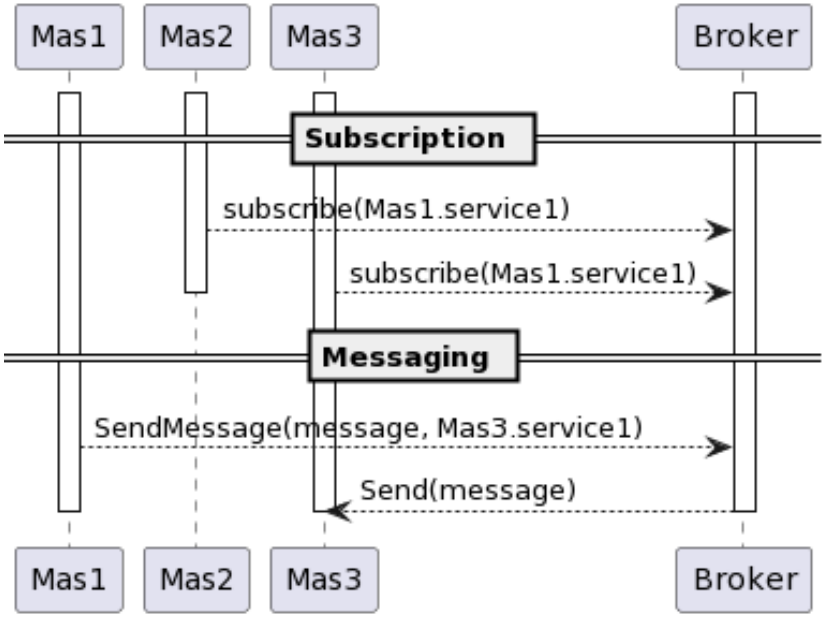

Figure 10: Interaction between DMas, *client* and *broker* in case of sending a message to a single recipient

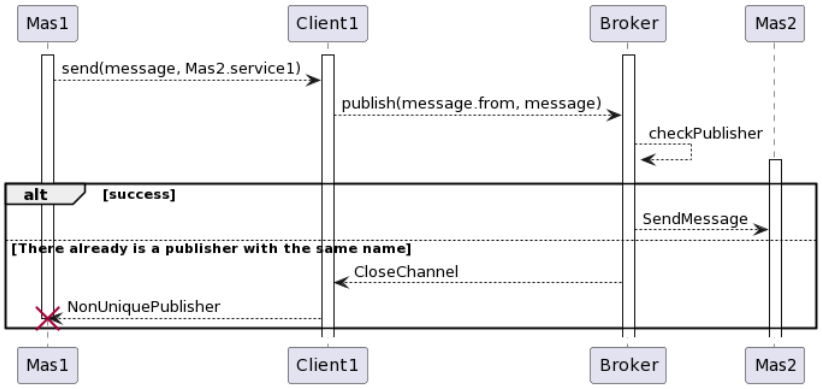

Figure 11: Sequence diagram of the interaction between `Dmas`, his *client* and *broker* in case of sending a message to a single recipient

Figures 12 and 13 show how the system manages the disconnections from `Dmas` and *broker*.

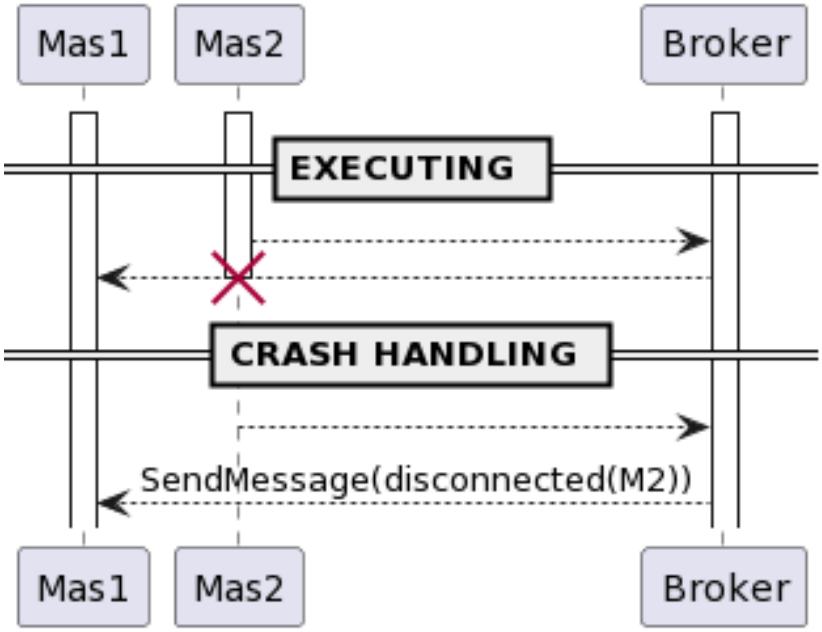

Figure 12: Interaction between `Dmas`, *client* and *broker* in case of *client* disconnection

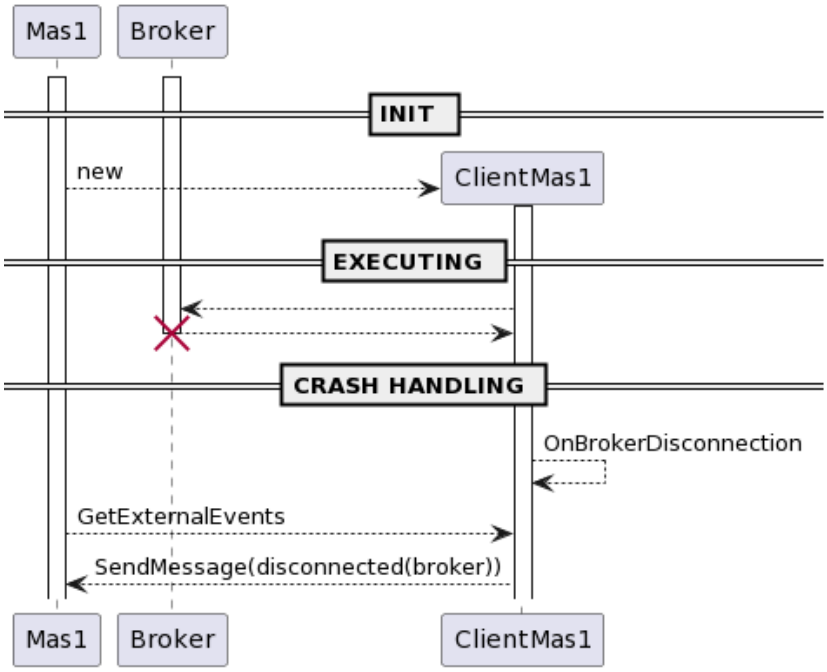

Figure 13: Interaction between `Dmas`, *client* and *broker* in case of *broker* disconnection

4 Implementation Details

4.1 Technologies

4.1.1 Kotlin

Kotlin[4] was chosen as the programming language for the development of the project, mainly because the codebase in JaKtA is also written in Kotlin.

Kotlin is a modern and highly expressive programming language designed to work seamlessly with Java[3]. With clear and concise syntax, advanced nullability handling, function extensions, and native support for object-oriented programming, Kotlin provides a robust Kotlin provides a robust and flexible development environment.

4.1.2 Gradle

The *build system* chosen was the one already present in JaKtA, namely Gradle[1].

Gradle is a build automation and dependency management system. Using a declarative configuration model, Gradle simplifies the process of creating and managing projects by providing a clear and powerful syntax based on Groovy[2] or Kotlin.

4.1.3 Ktor

For the management of network communications between *client* and *broker*, the Ktorcitektor library was chosen.

Ktor is a modern, lightweight Web development *framework* written in Kotlin, designed to simplify the creation of robust, high-performance Web applications. It offers a declarative and concise approach to handling *HTTP* requests and building web services. The *framework* allows asynchronous and non-blocking programming, plus it has the ability to integrate effortlessly with other Kotlin technologies, such as *coroutines*

4.1.4 JUnit

The *framework* JUnitcitejunit was used to test the operation of the developed modules.

Created to simplify and improve the process of *testing*, JUnit provides a set of *annotation* and *assert* that allow developers to easily define and execute *unit tests*.

4.2 Broker

The *broker* implemented in Kotlin uses the Ktor *framework* to handle asynchronous communication through *WebSocket* and *HTTP* requests. This distributed *broker* is based on two main components: *Cache* and *SubscriptionManager*.

Interface `Cache<T>`:

- It is responsible for the temporary storage of data associated with specific *topic*.
- Provides operations such as logging data, releasing resources, and reading data associated with a specific *topic*.

`SubscriptionManager<T>`:

- Provides operations to add and remove *publisher* and *subscriber* associated with specific *topic*.
- Provides information about available *topic* and sections associated with a specific *topic*.

Endpoint and Routing:

- The module `configureWebSockets` defines the *WebSocket endpoints* for publishing and subscribing
- to data related to specific *topic*.
- The *configureRouting* module configures the *route HTTP* to obtain information about the available *topic*.

4.3 Adapter

To realize a multi-agent system version that can communicate with the outside world, the `Mas` interface was extended with the `Dmas` interface:

```
interface Dmas : Mas {
    val network: Network
    val remoteServices: List<RemoteService>
}
```

An implementation of the `Dmas` can thus be used in place of the `Mas` by the main application logic of the *framework*, but in addition it can make use of the `Network` and `RemoteService` to implement the external communication logic.

As for the implementation of the `Network` interface, which encapsulates the communication logic between `Dmas` across the network, a solution based on the use of of a *client WebSocket*, which will be discussed in detail in the following section.

4.4 Client

The *client* application is designed to handle communication through *WebSocket*, making use of the *framework* Ktor in Kotlin. To address the distributed nature of the project, key entities were defined, including `Client`, `WebSocketSession`, and data models `SerializableSendMessage` and `SerializableBroadcastMessage`.

The `Client` interface plays the role of the main access point, encapsulating the core functionality for communication. Its key implementation is represented by the `WebSocketsClient` class.

The `WebSocketsClient` class is a central component for managing *WebSocket* connections. It is responsible for creating and managing *WebSocket* sessions for publishing, subscription and transmission of messages. It also keeps track of active connections, disconnections, and incoming messages.

The models `SerializableSendMessage` and `SerializableBroadcastMessage` are essential to represent the messages exchanged between the *client* and the *server*. Their structure is designed to ensure proper serialization and deserialization of data.

Three main features are implemented:

- `publish`: deals with sending a message to a given *topic* through a *WebSocket* connection;
- `broadcast`: similar to the above function but is intended to send a message to all connected *clients*;
- `subscribe`: handles the subscription of the *client* to a particular *topic* through a *WebSocket* connection.

Regarding the handling of incoming messages, there is the `subscribe` function. This function listens for messages sent to the specified *topic*, deserializes the received messages, and stores them for later use. This mechanism is critical to ensure effective two-way communication between the *client* and the *server*.

4.5 Implementation Notes

This section seeks to shed light on some implementation choices that may be considered suboptimal, which were made for reasons of time and/or complexity.

4.5.1 RemoteService

The `RemoteService` serves as a representation of a remote agent, i.e., an agent that belongs to a different `Dmas` than the current one, but can be reached through the network. It is identified by its name and is specified when creating the `Dmas`. It is important to note that, at present, no mechanism has been implemented to make unique `RemoteService` names for all `Dmas`, so it is possible for two different `Dmas` to have a `RemoteService` with the same name.

The option of prefixing the name of the `RemoteService` with a unique code was evaluated, however, since in the current design the `RemoteService` is specified at the time of creation of the `Dmas`, it is not possible to know a priori the unique code to be used. There is also a risk that two different `Dmas`, being in separate processes will generate the same code prefix.

So, as a temporary solution, a control mechanism has been implemented at the broker level that prevents two different `Dmas` from publishing messages on a topic (corresponding to the remote service) with the same name. Design notes related to this mechanism can be found in the section: 3.4.1.

4.5.2 Reserved names

. There are two `RemoteService` names that are to be considered reserved: `broker` and `broadcast`. The former is the name of the `RemoteService` of the Broker, while the latter is used to perform broadcasting of messages.

Currently, there is no mechanism in place to prevent a programmer from calling an agent of their `Dmas` with the above names, however, doing so could cause behaviors unexpected system behavior.

5 System Validation

The JUnit testing framework in combination with the Ktor test engine was used to test the developed software modules.

Ktor provides a special *test engine* that does not create a *web server*, does not connect to the *sockets*, and does not make any *HTTP* requests. Instead, it connects directly to the internal mechanisms and directly processes an application call. This results in faster test execution than starting a full *webserver*.

The Listing 1 presents the test class `BrokerBehaviorSpec`, which tests the behavior of the *broker*.

The test creates two simulated *client*, *fooMas* and *barMas*, which are configured to support *JSON* serialization and *WebSockets*. Next, *topic* (`fooTopic` and `barTopic`) and a test message (`message`) are established. Sessions are then created *WebSocket* for subscription and publishing operations between the two *clients*.

The test verifies the following behavior:

- Obtains the list of available *topic* through a *HTTP GET* request.
- Verifies that the obtained *topic* matches the previously established `fooTopic` and `barTopic`).
- Send a message from `fooMas` to `barTopic` through a subscription session, checking the error `BAD_REQUEST` in response.
- Send a message from `fooMas` to `fooTopic` through a publish session and verify that `barSubscribeSession` receives the message.
- Send a message from `barMas` to `barTopic` through a publishing session and verify that `fooSubscribeSession` receives the message.

```
1    testApplication {
2        val fooMas = createClient {
3            install(ContentNegotiation) {
4                json()
5            }
6            install(WebSockets)
7        }
8        val barMas = createClient {
9            install(ContentNegotiation) {
10               json()
11           }
12           install(WebSockets)
```

```
13          }
14          val fooTopic = "fooTopic"
15          val barTopic = "barTopic"
16          val message = "testMessage"
17          val fooSubscribeSession = fooMas.webSocketSession("/
    subscribe/$barTopic")
18          val barSubscribeSession = barMas.webSocketSession("/
    subscribe/$fooTopic")
19          val fooPublishSession = fooMas.webSocketSession("/
    publish/$fooTopic")
20          val barPublishSession = barMas.webSocketSession("/
    publish/$barTopic")
21          val availableTopics: Set<Topic> = fooMas.get("/topics
    ").body()
22
23          assertEquals(setOf(fooTopic, barTopic),
    availableTopics)
24
25          fooSubscribeSession.send(Frame.Text(message))
26          val response = fooSubscribeSession.incoming.receive()
27          assertEquals(Error.BAD_REQUEST, Json.decodeFromString
    ((response as Frame.Text).readText()))
28
29          fooPublishSession.send(Frame.Text(message))
30          val response2 = barSubscribeSession.incoming.receive
    ()
31          assertEquals(message, (response2 as Frame.Text).
    readText())
32
33          barPublishSession.send(Frame.Text(message))
34          val response3 = fooSubscribeSession.incoming.receive
    ()
35          assertEquals(message, (response3 as Frame.Text).
    readText())
36      }
```

Listing 1: Test class `BrokerBehaviorSpec`.

6 Usage Examples

To showcase the actual usefulness of the introduced middleware, some examples have been developed; among these, one of the most significant is *Telephone Game*2. The example implements the following behavior:

- there are N agents, distributed among N *MAS*;
- each agent is assigned an ID, ranging from 0 to N - 1;
- each agent waits for a message from the agent with id $ID + N - 1$;
- when a message is received, the agent sends it to the agent with id $(ID + 1)\%N$;
- the program ends when there has been a fixed amount of messages sent.

```
1   fun main(args: Array<String>) {
2       val host: String = args[0]
3       val port: Int = args[1].toInt()
4       val id: Int = args[2].toInt()
5       val nAgents: Int = args[3].toInt()
6       val nMessages = nAgents * 2
7       dmas {
8           environment {
9               actions {
10                  action(send, 2) {
11                      val receiver: Atom = argument(0)
12                      val message: Struct = argument(1)
13                      this.sendMessage(receiver.value,
    Message(this.sender, Tell, message))
14                  }
15              }
16          }
17
18          agent("ag$id") {
19              beliefs {
20                  fact { receiver("ag${(id + 1) % nAgents}"
    ) }
21              }
22              if (id == 0) goals { achieve(start) }
23              plans {
24                  +achieve(start) onlyIf { receiver(source(
    self), R) } then {
25                      execute(print("Starting..."))
26                      execute(send(R, count(1)))
27                  }
```

```
28                  +count(source(`_`), X) onlyIf {
29                      (X lowerThan nMessages) and (M `is` (
    X + 1)) and receiver(source(self), R)
30                  } then {
31                      execute(print("Sending ", M))
32                      execute(send(R, count(M)))
33                  }
34                  +count(source(`_`), X) onlyIf { X
    greaterThanOrEqualsTo nMessages } then {
35                      execute(print("Done!"))
36                      execute("stop")
37                  }
38              }
39          }
40
41          service {
42              name("ag${(id + nAgents - 1) % nAgents}")
43          }
44
45          network {
46              websocketNetwork(host, port)
47          }
48      }.start()
49  }
```

Listing 2: Usage example: `TelephoneGame`.

6.1 Broadcast

The next example showcases a different situation:

- there is a Broadcaster agent, executing in his own *MAS*;
- there are 2 Receiver agents, both executing in the same *MAS* which is different from the Broadcaster's;
- the Broadcaster broadcasts a message to the system;
- the receivers receive the message and print it.

```
1   fun main() {
2       val broadcastAction = object : AbstractExternalAction
    ("broadcast", 2) {
3           override fun action(request: ExternalRequest) {
4               val type = request.arguments[0].castToAtom()
5               val message = request.arguments[1].
    castToStruct()
6               when (type.value) {
```

```
                    "tell" -> broadcastMessage(Message(request.sender, Tell, message))
                    "achieve" -> broadcastMessage(
                        Message(request.sender, Achieve, message),
                    )
                }
            }
        }

        val env = Environment.of(
            externalActions = mapOf(
                broadcastAction.signature.name to broadcastAction,
            ),
        )

        val broadcaster = Agent.of(
            name = "broadcaster",
            events = listOf(
                Event.ofAchievementGoalInvocation(
                    it.unibo.jakta.agents.bdi.goals.Achieve.of(
                        Jakta.parseStruct("broadcast"),
                    ),
                ),
            ),
            planLibrary = PlanLibrary.of(
                Plan.ofAchievementGoalInvocation(
                    value = Jakta.parseStruct("broadcast"),
                    goals = listOf(
                        ActInternally.of(Jakta.parseStruct("print(\"Broadcast message\")")),
                        Act.of(Jakta.parseStruct("broadcast(tell, greetings)")),
                    ),
                ),
            ),
        )

        DMas.fromWebSocketNetwork(
            ExecutionStrategy.oneThreadPerAgent(),
            env,
            listOf(broadcaster),
            emptyList(),
            "localhost",
            8080,
        ).start()
```

```
}
```

Listing 3: Usage example: `Broadcast`: Broadcaster's code.

```
fun main() {
    val env = Environment.of()

    val alice = Agent.of(
        name = "alice",
        planLibrary = PlanLibrary.of(
            Plan.ofBeliefBaseAddition(
                belief = Belief.from(Jakta.parseStruct("greetings(source(Sender))")),
                goals = listOf(
                    ActInternally.of(Jakta.parseStruct("print(\"Received message from: \", Sender)")),
                ),
            ),
        ),
    )

    val bob = Agent.of(
        name = "bob",
        planLibrary = PlanLibrary.of(
            Plan.ofBeliefBaseAddition(
                belief = Belief.from(Jakta.parseStruct("greetings(source(Sender))")),
                goals = listOf(
                    ActInternally.of(Jakta.parseStruct("print(\"Received message from: \", Sender)")),
                ),
            ),
        ),
    )

    DMas.withEmbeddedBroker(ExecutionStrategy.oneThreadPerMas(), env, listOf(alice, bob), emptyList()).start()
}
```

Listing 4: Usage example: `Broadcast`: Receiver's code.

7 Conclusions

The goal of this project was to give users of the *framework* JaKtA the possibility to create distributed multi-agent systems easily and quickly, with the least possible *overhead*.

Therefore, an extension of the *framework* was developed, providing a new implementation of the multi-agent system concept capable of communicating with other systems using an asynchronous *publish-subscribe* communication model.

This communication model was implemented through the *framework* Ktor and the *WebSocket* protocol.

The team believes that the goal has been achieved; however, they are aware of some possible improvements to the proposed solution, which will be discussed in the next section.

In conclusion, the project has provided a good opportunity to deepen the knowledge acquired in the field of multi-agent systems, agent-oriented programming, and distributed systems.

7.1 Future Works

The team believes that the project has achieved its set objectives; however, it is aware that the proposed solution can be improved in various aspects, some of which are listed below:

- **Support for Multiple Communication Protocols**: Currently, the *framework* supports only one communication protocol, namely *WebSocket*. However, it would be interesting to provide users with the option to choose from different protocols, such as *MQTT*. The project's architecture would easily allow adding *MQTT* support through the implementation of a new `Network` and potentially a new *broker*.

- **Handling Uniqueness of Remote Service Names**: As mentioned in the "Implementation Details" section, the *framework* does not provide any mechanism to ensure the uniqueness of remote service names. One possible solution could be to add a coordination step during the connection to the *broker*, allowing `Dmas` to register their services and check that the chosen name has not been used already.

References

[1] Gradle build tool. `https://gradle.org/`.

[2] Groovy language. https://groovy-lang.org/.

[3] Java language. https://www.java.com/en/.

[4] Kotlin language. https://kotlinlang.org/.

[5] Martina Baiardi, Samuele Burattini, Giovanni Ciatto, and Danilo Pianini. Jakta: Bdi agent-oriented programming in pure kotlin. In Vadim Malvone and Aniello Murano, editors, *Multi-Agent Systems*, pages 49–65, Cham, 2023. Springer Nature Switzerland.

[6] Rafael H. Bordini, Jomi F. Hübner, and Renata Vieira. *Jason and the Golden Fleece of Agent-Oriented Programming*, pages 3–37. Springer US, Boston, MA, 2005.