Article

Multi-Modal Decentralized Interaction in Multi-Entity Systems

Andrei Olaru * and Monica Pricope

Department of Computer Science and Engineering, University Politehnica of Bucharest, 313 Splaiul Independentei, 060042 Bucharest, Romania
* Correspondence: andrei.olaru@upb.ro

Abstract: Current multi-agent frameworks usually use centralized, fixed communication infrastructures for the entities that are deployed using them. This decreases the robustness of the system but is less challenging when having to deal with mobile agents that can migrate between nodes. We introduce, in the context of the FLASH-MAS (Fast and Lightweight Agent Shell) multi-entity deployment framework, methods to build decentralized interaction infrastructures which support migrating entities. We discuss the WS-Regions (WebSocket Regions) communication protocol, a proposal for interaction in deployments using multiple communication methods, and a mechanism to facilitate using arbitrary names for entities. The WS-Regions Protocol is compared against Jade (the Java Agent Development Framework), the most popular agent deployment framework, with a favorable trade-off between decentralization and performance.

Keywords: multi-agent systems; decentralized interaction

Citation: Olaru, A.; Pricope, M. Multi-Modal Decentralized Interaction in Multi-Entity Systems. *Sensors* **2023**, *23*, 3139. https://doi.org/10.3390/s23063139

Academic Editor: Roberto Teti

Received: 19 December 2022
Revised: 7 March 2023
Accepted: 10 March 2023
Published: 15 March 2023

1. Introduction

Software agents are programs with the ability to run autonomously to fulfil their goals. An agent-oriented approach is especially appropriate in the context of heterogeneous, open systems in which agents appear and disappear regularly, and in which the network topology changes often.

Mobile agents have the additional ability to migrate between different nodes in the network. Using mobile agents or not is a choice of paradigm. Mobility is an orthogonal property of agents, independent of other properties. While any distributed application can be implemented by only using static agents, development and deployment can be improved by using agents that are mobile, with the effect of transferring not only data, but also behaviors, through the network, while keeping both the behavior and the data encapsulated in an autonomous entity [1].

There are many uses for mobile agents, among the most recent being the utilization in the Internet of Things (IoT) domain, where mobile agents can move between devices to perform computation locally, avoiding the need to consume bandwidth [2–5]; can be used for security purposes, detecting attacks or ensuring security [6,7]; or for the management of highly adaptable energy micro-grids [8–10].

Let us take a running example to use throughout the paper. A smart building contains several thousands of devices—desktop computers, laptops, smartphones, sensors, actuators, networking devices, printers, and projectors—all connected to the network. In order to be able to perform maintenance tasks on those machines that are managed by the organization, two types of mobile agents can be used. Some agents move through the system on various machines to evaluate the state of the machine, maintain the installed software, and review performance information. When a problem is detected, a specialized agent can be created, containing tools dedicated to that specific problem, and is sent to the machine in question. The advantage of mobile agents here is that they can transport both data and code from one machine to another, and they can use autonomous behaviors to evaluate and decide locally on the actions to take.

Multi-agent system frameworks assist in the development and deployment of multi-agent systems (MAS) by allowing the developer to focus on the design and implementation of the behavior of agents and offers functionality for the discovery of other agents and for delivering messages to other agents [11,12]. Frameworks supporting mobility allow agents to interrupt their execution and migrate to other nodes, where their execution is resumed.

The most popular current agent frameworks (Jade, JIAC, JaCaMo, SPADE [13–16]) use centralized communication—all messages pass through a central node, which routes messages to their destinations. This method is simple, and it is effective, especially in the context of mobile agents, as all the migrating agents also pass through the same central node en route to their destinations, and the central node is aware of the location of each agent. However, in larger scenarios, the central node becomes a bottleneck and a single point of failure. For communication, Jade supports splitting a large system into several *platforms*. Agents can communicate between platforms; however, they must use their complete names (including the IP address of the platform) and cannot migrate between platforms.

While frameworks generally use agents as the only abstraction available to the developer (for instance, in Jade there are only agents and containers that the developer can create and manage), other abstractions may exist. Among these, are artifacts, as proposed by the Agents and Artifacts (A&A) model [17], agent groups and organizations [18], but also the communication infrastructure itself [19]. Instead of limiting our discussion to agents, let us generalize to *entities*—any persistent unit that can be abstracted in a framework.

Research in decentralized agent systems includes dealing with delivering messages to mobile agents and also with coordination and consensus between agents [20–22]. However, they do not deal with creating a framework for *general use*, nor do they deal with practical issues that arise in deployments in real-life networks, such as the lack of routability for nodes in the system.

The main challenges in creating a distributed, decentralized mechanism for interactions between entities deal with entity identifiers and with the current locations of entities—how can an entity be addressed by the same identifier while it is migrating between nodes (machines) and how to localize the destination for a message between entities.

This paper introduces mechanisms for the communication between and migration of entities in a multi-entity system, which is decentralized in the sense that the system can be partitioned into any number of regions which do not depend on one another. The mechanism is interoperable with other communication methods, and it is layered in terms of capabilities to be appropriate for a system with a heterogeneous deployment. We call this the WS-Regions Protocol (*WS* stands *WebSocket*—https://en.wikipedia.org/wiki/WebSocket, accessed on 7 March 2023).

We address applications which need to be decentralized, which use mobile agents, and in which mobile agents need to both send and receive messages from any other entity in the system, regardless of their current location.

We have implemented and validated this mechanism using FLASH-MAS, a multi-entity deployment framework developed at our university [23]. FLASH-MAS originally stood for a Flexible and Lightweight Agent Shell and the motto of the project is *Easy for beginners, powerful to experts*. FLASH-MAS is open source, and the code is hosted at https://github.com/andreiolaru-ro/FLASH-MAS, accessed on 7 March 2023. The main goal of the framework is to support maximum flexibility in implementation and interaction, while remaining easy to deploy and efficient in execution.

More concretely, in this paper we present the following:

- A region-based support infrastructure for decentralized interaction between entities, using modern communication paradigms and supporting frequently migrating mobile entities;
- A solution for integrating multiple support infrastructures in the same multi-entity system, resulting in a multi-modal framework;
- A means for entities to be able to use arbitrary names to address each other in the context of a decentralized multi-modal system.

Following the general philosophy of FLASH-MAS, we have developed our solution in a modular manner organized in independent modules and layers. The decentralized, region-based support infrastructure is implemented on top of any server–client communication infrastructure. The multi-modal communication infrastructure can work in conjunction with any support infrastructure, and the arbitrary entity naming solution can be used with any decentralized support infrastructure to translate short names into complete names and work across multiple support infrastructures. An architectural perspective of these components is presented in Figure 1.

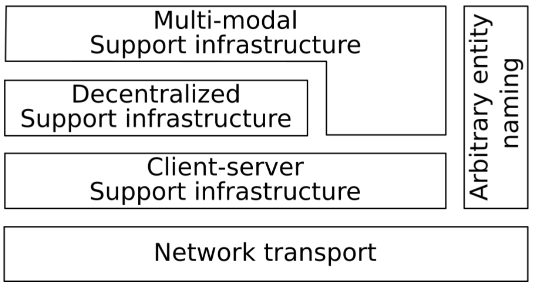


Figure 1. An architectural view of the components presented in this paper.

To evaluate the quality of the solution we propose for decentralized interaction, we perform a performance comparison, both qualitatively and quantitatively, between FLASH-MAS and Jade, the current most popular MAS framework. The results of the comparison were satisfactory and proved that our solution compares well to Jade in the various scenarios, while having the additional advantage of robustness in the case of failure.

After inspecting related work in the next section, we present the main architecture and features of FLASH-MAS in Section 3. Our main contributions are introduced in Section 4, and the validation and analysis of results follow in Section 5. Section 6 draws the conclusions.

2. Related Work

Mobile agents are currently used in several application areas, but most of them have characteristics in common, such as being heterogeneous, having a frequently changing topology (some of the devices/hosts are mobile), and suffering from low bandwidth network link between at least some of the devices. The most relevant fields are IoT, vehicular networks, wireless sensor networks, and energy micro-grids.

In wireless sensor networks, mobile agents are used to gather information from WSN nodes. Some works only use mobility along a pre-computed itinerary, with no communication [24]. In the cases where communication is needed, centralized communication methods are used [25,26], many times using Jade—the Java Agent Development Framework [27]. This works for small setups or for when the number of messages is low but is not adequate for city-scale WSNs. Research also exists in the field of distributed computing regarding decentralized messaging [28], however these are made to support only fixed message receivers, which are not able to migrate through the network. This makes the problem that we address specific to the field of mobile multi-agent systems.

In energy micro-grids, agents are used thanks to their autonomous capacity to make decisions in the face of arising issues or system faults [10]. Applications in this domain that use a framework use Jade, however, which is not in itself a robust communication infrastructure.

In the field of IoT, Salah et al. use a microservice-based architecture in which each agent exposes a microservice which responds to HTTP requests [2]. Besides having each agent (or at least each node) deal with the issues of exposing a web server to the Internet, routability means that IPV6 addresses need to be used to identify agents, but these addresses do not remain the same when mobile agents migrate to other hosts, hence being unable to keep an invariable identifier. Indeed, other authors building mobile agent systems rely directly on TCP/IP for communication [3–5], making it possible only for mobile agents to contact fixed nodes, but impossible for mobile agents to be contacted from the exterior.

Bosse proposes a framework for the deployment of agents written in JavaScript [29,30], allowing for mobility and interactions with IoT, the cloud, and machine learning services. However, it suffers from limitations in terms of addressing and the limited parallelism of JS, being more adequate for simulations than real-life deployments.

Existing models for message delivery in mobile agent systems have been surveyed and compared previously. Deugo [31] compares several classic delivery models from a theoretical point of view, without actual experiments. Hidayat [32] compares some models from a qualitative point of view of the features offered and challenges solved and proposes a new model. Similarly, Virmani [33] and Rawat et al [20] make qualitative comparisons of the state-of-the-art models at the time.

In the qualitative analyses, the features that are evaluated are generally a subset of the following: solution to the tracking problem (when an agent moves after the message is sent, but before the message reaches it), guaranteed message delivery, support for asynchronous communication, delivery in reasonable time, and transparency of location.

In previous work [34], we have implemented several decentralized message delivery models from the literature and performed a detailed, quantitative comparison of their performance in terms of delivery rate, delivery time, and network load. This comparison has been conducted in a simulated environment with different settings for network and node performance.

Message delivery models for communication in mobile agent systems are generally built around several well-known schemata:

- The *centralized* solution, in which one server is tracking the whereabouts of all agents and forwards their messages accordingly; this is improved by the *home server scheme* where different servers are assigned to different partitions of the agent set, offering a more balanced solution than centralization [35]; hierarchical solutions, based on *domains* or *regions* also exist [36];
- *Blackboard* solutions in which agents need to visit or contact the blackboard explicitly to receive their messages [37,38];
- *Forwarding proxy* solutions in which each host remembers the next location to which an agent migrated, and messages will be forwarded along the path of the agent [39]; the *Shadow Protocol* combines the proxy model with the home server model by using proxies, but agents regularly send updates of their location to their home server [40]; *search-by-path-chase* also adds regions for improved location management [41]; a combination of forwarding proxies and location servers is used by MEFS [42,43].

Distributed decision and control are important aspects in MAS-based control of physical robots [44–46]. However, this differs from our work in that, while in robotic systems agents remain static to their machines, in our work, agents move between machines. In the former case, distributed communication does not bring issues in terms of addressing agents because agents usually keep the same identifier (e.g., their IP address).

Regarding robustness and resiliency, there is a significant body of research in multi-agent consensus and protection of multi-agent systems against cyberattacks [47–49]. While our goal is to build mechanisms for robustness in MAS, we currently only analyze the possibility of creating tools for a general-purpose distributed MAS framework with no single point of failure. At this point, we do not go into the subject of attacks on such a framework, focusing on the analysis of the performance of the framework during normal operation. An element of future work is to verify how the proposed methods function in the face of attacks.

3. FLASH-MAS Architecture

FLASH-MAS is a flexible and lightweight agent deployment framework [23]. It is currently implemented in Java, the same as Jade and many other MAS frameworks. FLASH-MAS models a system as a collection of *entities*—persistent, potentially autonomous processes which offer services, make decisions, perform actions, or help in the interaction with the framework or with the environment. FLASH-MAS defines a basic set of *default* entities: nodes, pylons, agents, artifacts, and shards, the implementation of all of which extends a single Java class—the `Entity`. However, any instance of any other class-extending `Entity` can be added to the system at runtime and can interact with any other entities without the need of changing the code of the framework. Some instances of entities inspired by our running example are shown in Figure 2.

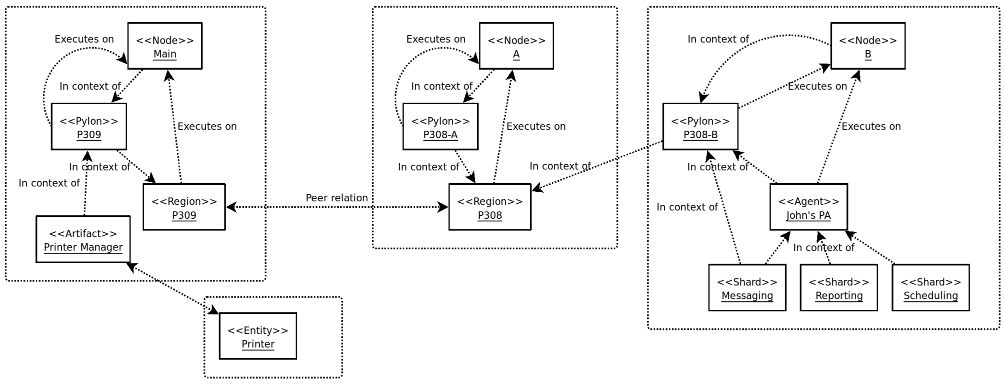


Figure 2. Examples of entities and their interconnections in FLASH-MAS (the Fast and Lightweight Agent Shell). Some relations between entities have been omitted.

Entities in FLASH-MAS exist in the context of one another. There are several standard entities:

- *Nodes* represent the machines that FLASH-MAS executes on, and any entity runs in the context of the local node;
- *Support infrastructures* are virtual entities spanning multiple nodes, offering services (such as interaction, discovery, etc.) to other entities;
- *Pylons* are the materialization of support infrastructures. On every node which exists in the context of a support infrastructure, the infrastructure is represented by a pylon. For entities to use the services offered by the support infrastructure, they must exist in the context of their local pylon;
- *Agents* are autonomous entities that perceive the environment, make decisions, and perform actions;
- *Artifacts* are entities that facilitate the interaction between agents and the environment, offer reactive services, or offer support for agent groups, organizations, or other interactions between entities, according to the A&A model [17];
- *Shards* are sub-agent entities which implement specific functionality or behaviors, existing in the context of agents, nodes, or other entities. Shards implement, for example, functionality for messaging, monitoring, remote control, user interaction, etc.

All entities in FLASH-MAS interact via *waves*. A *wave* has a source, a destination, and any number of other $\langle name, value \rangle$ arguments. Messages between entities can be transferred as waves, but other types of interaction may exist in the system (in our context, for instance, entities also migrate via waves). Currently, FLASH-MAS offers several types of communication mechanisms (including WebSocket, web services-based, ROS-based, MPI), monitoring and control facilities (including remote monitoring and control via a central web-based interface), and automatic GUI (Graphical User Interface) generation (including the ability to display a graphical interface for an entity on a different node than where the entity currently executes). Thanks to its flexibility, the actual method of communication between entities can be decided at runtime, at the code of entities stays the same regardless of how they communicate.

To ensure flexibility in the interaction method, a support infrastructure is implemented via specific pylons and shards. Shards are driven by events in the entity that contains the shard and can submit events to that entity. While usually shards are part of agents—a specific agent implementation which relies only on shards is also defined and is called a *composite agent*—any entity can use a shard to access its functionality.

Most support infrastructures in FLASH-MAS are implemented in two parts: a node-specific part, materialized as a pylon on each node that is part of the support infrastructure; and an entity-specific part, materialized as a shard that must be added to every entity that exists in the context of the support infrastructure and which needs to access its services. While indeed the shard added to the entity must sometimes be specific to the support infrastructure, FLASH-MAS contains a mechanism which can dynamically load the *appropriate* shard for the support infrastructure without needing to modify the code of the entity. Namely, if an entity needs to use a `messaging` shard for communication, then when it is added in the context of the pylon offering communication services, it will ask the framework to provide it with the appropriate shard implementation. The interaction between entity and shard is standardized as event-driven, while the interaction between the shard and the pylon can be specific to that support infrastructure.

3.1. Challenges and Requirements

Previously, FLASH-MAS supported a variety of communication methods based on client–server or otherwise centralized architectures (WebSocket, RESTful web services, the ROS robot operating system).

The objective that has led to this research was to have a means of interaction which is robust, reliable, and decentralized, such that no specific node is a bottleneck or a single point of failure for the system, all while continuing to support entity mobility.

Of course, centralized interaction brings many advantages that are lost with decentralization. The most important one is the ability of the central node to know on which node an entity is located. In the decentralized case, several challenges arise:

[C1] *The infrastructure must know where (to which node) to deliver a message for any entity in the system.*
When mobile entities exist in the system, their location can only be determined at runtime; in the decentralized case, there cannot exist a single entity which tracks all mobile entities and can know their location at any time.

[C2] *The infrastructure must allow an entity to move to any node in the system.*
A large multi-entity system may be formed of several regions employing different communication mechanisms, or using separate servers, central to each region. Migration across regions is in this case more difficult to implement. In Jade, for instance, although messages may be exchanged between different Jade *platforms*, migration across platforms is not possible without the use of third-party plugins.

[C3] *The infrastructure must be able to deliver all messages meant for an entity, even if that entity is sometimes in the process of migrating.*
Migration is a process that does not happen instantaneously, and in that time, messages may be sent to the migrating entity; these messages must not be discarded, even if the entity is not currently able to receive and process them.

[C4] *The infrastructure must mitigate damage when a node is lost and should be able to repair connections affected by losing that node*
If any node in the system is lost, the system must be able to recover partially or entirely. In the case of centralized systems, loss of the central node is not recoverable. In the case of region-based systems, loss of the server central to a region usually means losing all the information on that region and orphaning mobile entities originating from that region and which are currently located in other regions.

[C5] *(Optional) The entities can be addressed (as message destinations) by an arbitrary name, which is not required to contain routing data for any node (e.g., the IP address of a particular node).*
Addressing entities by a simple, arbitrary name is easy for developers in frameworks such as Jade. In decentralized systems (including the case of using multiple, interacting Jade platforms), entities must usually be referred to by an identifier which contains the identifier of the region of that entity.

The Jade framework, currently the most popular MAS deployment framework, as well as other frameworks which it inspired, covers the first three challenges for a single-platform deployment, but our experiments have shown that C3 is not fulfilled for cases when agents migrate very quickly (see Section 5.2). In a multi-platform deployment, it only covers challenges C1 and C3 and needs a plugin provided by a third party to cover challenge C2.

4. Multi-Modal Decentralized Interaction between Entities

This section introduces a series of proposals for the improvement of the interaction between entities in a decentralized, heterogeneous system.

- The WS-Regions Protocol can be used for routing messages in a decentralized system containing mobile entities;
- Solutions for heterogeneous, multi-modal systems, in which multiple communication mechanisms are used in different areas of the same multi-entity system;
- The Arbitrary Entity Naming mechanism allows entities to be addressed using arbitrary identifiers, which do not need to contain the identifier of their home region;
- Finally, we propose a layer of increased efficiency for message routing, which trades space for increased speed in communication.

4.1. The WS-Regions Protocol

We have developed the WS-Regions Protocol, taking inspiration from protocols, such as Home Server, Shadow, and RAMD [35,38,40]. Our initial intention was to use the Shadow Protocol as a decentralized interaction mechanism for FLASH-MAS; however, there have been several issues with that approach.

Like other mechanisms for decentralized interaction between mobile agents, Shadow assumes uniform lower-level communication between nodes in that any node can communicate directly with any other. As such, it makes sense for the agent to leave proxies to its previous locations. However, in the modern world, relying on this approach works only for PCs in a local network and, in general, for machines with routable IP addresses but fails when using mobile devices (e.g., smartphones), non-TCP/IP communication (e.g., for Bluetooth devices), or machines behind Network Address Translation gateways. Therefore, in FLASH-MAS we have relied heavily on the WebSocket Protocol [50], which allows full duplex communication between any client and a routable server. However, since communication passes through a server, the method proposed by the Shadow Protocol becomes inefficient—a proxy for an entity does not actually forward messages directly to that entity; instead they go through the region's WebSocket server anyway. In fact, using WebSocket for communication is more like how region-based protocols work, each region having a server which keeps track of agents created within that region.

As such, we have developed the WS-Regions Protocol, in which the multi-entity system is partitioned into several *regions*, each region containing several *nodes* and a single *WebSocket server*.

4.1.1. Entities

There are several types of entities with specific roles in the WS-Regions Protocol:

- *Regions* are partitions of the multi-entity system. Each region spans one or multiple physical machines (hence, it spans multiple nodes) and contains a server on one of its nodes—the *region server*; each region has a *region identifier* which is a URI (Uniform Resource Identifier), routable from the region server of any other region;
- Each *node* in the system contains a *pylon* which belongs to one region;
- The *WS-Regions pylon* contains a client which connects to the region server at startup;
- Any *entity* which is *created* within a region has a *name* which is unique to that region and encapsulates a *WS-Regions shard* which interfaces with the local WS-Regions pylon. The region where the entity was created is its *home region*.

Leveraging the fact that the FLASH-MAS architecture is modular and flexible, the WS-Regions Protocol can be deployed on top of any client–server communication protocol, be it WebSocket or something different. It only needs to offer a server entity and a client entity that can be used by the region servers and the WS-Regions pylons, respectively.

Formally, a MAS using the WS-Regions Protocol contains regions, nodes, and entities:

$$MAS = \langle \mathcal{R}, \mathcal{N}, \mathcal{E} \rangle, \text{with } \mathcal{R} \text{ a partition of } \mathcal{N} \text{ and } \mathcal{N} \text{ a partition of } \mathcal{E}:$$

$$\forall N \in \mathcal{N}.\exists R \in \mathcal{R}.N \in R \land \nexists R'.(R' \neq R \land N \in R')$$

$$\forall E \in \mathcal{E}.\exists N \in \mathcal{N}.E \in N \land \nexists N'.(N' \neq N \land E \in N')$$

That is, any node belongs to one and only one region, and any entity executes on one and only one nodes.

For this formalization sketch, we will use the notation $MAS^t = \langle \mathcal{R}^t, \mathcal{N}^t, \mathcal{E}^t \rangle$ for the state of the system at time t and the relation $\in_t$ for a membership relation which is true at time t. We will also use the membership relation for the relation between entities and regions, i.e., for $E \in \mathcal{E}, N \in \mathcal{N}, R \in \mathcal{R}$, we will have $E \in R \iff E \in N \land N \in R$.

For each physical node there is a `Node` entity that embodies it and is fixed to that physical node (i.e. cannot migrate):

$$\forall N \in \mathcal{N}.\exists E_N \in E, embodiment(N) = E_N, \text{and } \forall t.E_N \in_t N$$

Each entity has a home region, where the entity was created:

$$\forall E \in \mathcal{E}.\exists R_E \in R, home(E) = R_E, \text{with } E \in_0 R_E$$

Each region is identified by a URI. This makes it easy for regions to address each other. Each entity (including entities that embody nodes) is also identified by a URI. The URI of the entity must contain, as a prefix, the URI of its *home region*, where the entity has been created initially. For instance, an entity `Printer308` which has been created in region `build.ing/P308` will be identified by the URI `build.ing/P308/Printer308`. This way, it is easy to route messages to entities based on their identifiers. In the paper, we use simplified names for the sake of readability. A discussion on using simpler identifiers is presented in Section 4.3.

4.1.2. Protocol Sequences

We have illustrated in Table 1 the path taken by a wave in every case of home region/current location for the source and destination entities.

Table 1. The path taken by a wave from entity S to entity D, considering the same or different home regions, and various situations for their location. In every case, we call the actual nodes where the entities are located $Node_S$ and $Node_D$, without meaning that those nodes are the same across different cases.

S Home	D	S Status	D Status	Wave Path
R_{SD}		HOME	HOME	$Node_S \rightarrow R_{SD} \rightarrow Node_D$
R_{SD}		HOME	REMOTE/$R2$	$Node_S \rightarrow R_{SD} \rightarrow R2 \rightarrow Node_D$
R_{SD}		REMOTE/$R1$	REMOTE/$R1$	$Node_S \rightarrow R1 \rightarrow Node_D$
R_{SD}		REMOTE/$R1$	REMOTE/$R2$	$Node_S \rightarrow R1 \rightarrow R_{SD} \rightarrow R2 \rightarrow Node_D$
R_S	R_D	HOME	HOME	$Node_S \rightarrow R_S \rightarrow R_D \rightarrow Node_D$
R_S	R_D	HOME	REMOTE/R_S	$Node_S \rightarrow R_S \rightarrow Node_D$
R_S	R_D	REMOTE/$R1$	REMOTE/$R2$	$Node_S \rightarrow R1 \rightarrow R_S \rightarrow R_D \rightarrow R2 \rightarrow Node_D$

However, a decentralized communication protocol is especially challenging when the entities are migrating. The specific challenge here is how to ensure that messages are not lost. For instance, in the last case in Table 1, suppose that S sends a message to D, and while the message is in transit, D migrates to a region $R3$, different from $R1$, R_S, R_D, or $R2$. Migration also takes some time, so entity D must suspend processing, package itself, get the package sent to the other node, unpack itself, and resume processing of events. Each of these phases takes some time in which messages may try to reach it. The protocol must ensure the message is buffered and is able to reach region $R3$ and, eventually, D. More precisely:

1. An entity's home server must be aware of the change in region *before* sending messages to a region where the entity is not ready to receive them anymore;
2. An entity must suspend processing *after* receiving confirmation that no other messages will arrive in its current region.

Principle 2 above could be avoided or relaxed if messages arriving in the former region of the entity could be sent by that region to the future region of the entity. However, this means that messages coming from the former region and from the home server would have to be synchronized to arrive to the entity after migration in the correct order (supposing, for instance, that multiple messages were sent by the same entity). Keeping principle 2 above also means that message-chasing problems, which occur when an entity migrates extremely frequently [51], can be solved at the level of the entity's home server rather than other regions sending messages on the path taken by the entity.

For their home server, mobile entities can have one of three states:

- HOME—the entity is located within its home region and has a reference to a WebSocket client connected to the region server;
- REMOTE—the entity is located within another region than its home region;
- IN-TRANSIT—the entity is currently in the process of migration between nodes.

For other region servers, mobile entities can be one of the following:

- GUEST—when the entity is in this region, which is not its home region; or
- IN-TRANSIT—when the entity is preparing to move away from this region.

To apply the two principles expressed previously, we have implemented the following protocol. Table 2 lists the messages in the protocol; Table 3 presents the pseudocode of the processes that happen in a migrating entity and in the relevant regions; and Figure 3 gives a visual perspective on the migration of an agent. The protocol functions are as follows:

- An entity E, with the home region R_E, currently on $Node_1$ in region R_1, which intends to migrate to $Node_2$ in region R_2, starts by suspending operations; any interactions beyond this point (both incoming and outgoing) will be buffered and processed after migration is completed;
- E sends to its host server R_1 a REQ_LEAVE message announcing the migration; any messages for E that do not come from its home server will be sent to the home server, as if E had already left;
- The message is relayed to E's home server R_E as a REQ_BUFFER message, which starts buffering any messages for E; the entity is from this point considered IN-TRANSIT; then, R_E sends a REQ_ACCEPT message to E confirming the buffering of messages;
- Upon receiving the confirmation from R_E, E completely halts, packages its data, and asks $Node_1$ to deliver the data to $Node_2$;
- The package follows the direct path $Node_1 \rightarrow R_1 \rightarrow R2 \rightarrow Node_2$, as a wave of type AGENT_CONTENT between node entities;
- Upon arriving on $Node_2$, the package is unpacked by the node and the entity is started;
- E sends a CONNECT message to R_2, which registers it as a GUEST entity and sends an AGENT_UPDATE to R_E; E then starts resuming operations and processes any interactions buffered when migration was starting;
- R_E registers E's new location, changes its state to HOME or REMOTE depending on the current region, and sends all the buffered messages to E.

Table 2. Types of *waves* sent between entities in the system in the context of the migration of an entity between two host regions, different from its home region.

Message Type	Direction	Semantics
REGISTER	entity → home server	A new entity has been created in the region
CONNECT	entity → host server	The entity has arrived in the region after migration
REQ_LEAVE	entity → host server	The entity wants to move to another region
REQ_BUFFER	host server → home server	An entity in the source region wants to move to another region
REQ_ACCEPT	home server → entity	Acknowledgment of buffer request
AGENT_UPDATE	host server → home server	An entity has arrived in the region
AGENT_CONTENT	node → node	A wave that contains a packaged entity
CONTENT	entity → entity	A wave that is a normal message between entities

The process is, of course, somewhat simplified when the region migrates from, to, or within its home region.

Table 3. Pseudocode for communication and migration-related processes in the WS-Regions Protocol. The various cases for messages received by region servers are organized in the figure for increased readability of the entire process. The cases in the middle column apply to the case when the current region of an entity is not its home region.

Entity E	Current Region of E, If Different from R_E	Home Region R_E

$\langle normal\ operation \rangle$	receive normal message m for E,
	    with $home(E) \neq R_1$
	  if $E \in R_1$
	    then send m to $node(E)$
	    else send m to $home(E)$
		receive normal message m for E,
		    with $home(E) = R_E$
		  case $state(E)$ of:
		    HOME: send m to $node(E)$
		    REMOTE on R_1: send m to R_1
		    IN-TRANSIT: add m to buf[E]

$\langle E \in N_1 \in R_1 \rangle$
 	
$\langle E$ *needs to migrate to* $N_2 \in R_2 \rangle$
start buffering in / out operations
send REQ_LEAVE to R_1
$\langle wait\ for$ REQ_ACCEPT$\rangle$
suspend all sub-entities
package into p
request N_1 to send p to N_2
	$\langle R_1 \neq R_E \rangle$
	
	receive REQ_LEAVE from E
	    with $home(E) = R_E \neq R_1$
	  $state(E) :=$ IN-TRANSIT
	  send REQ_BUFFER to R_E
	
	receive REQ_ACCEPT to E from R_E
	  send REQ_ACCEPT to N_1
	  remove E from R_1
		receive REQ_BUFFER from $E \in R_1$
		    with $home(E) = R_E$
		  create buf[E]
		  $state(E) :=$ IN-TRANSIT
		  send REQ_ACCEPT to R_1
		
		receive REQ_LEAVE from E
		    with $home(E) = R_E$
		  create buf[E]
		  $state(E) :=$ IN-TRANSIT
		  send REQ_ACCEPT to E

$\langle E \in N_2 \in R_2 \rangle$
 	
$\langle E$ *arrives on* $N_2 \in R_2 \rangle$
unpack all sub-entities
send CONNECT to R_2
run buffered in/out operations
resume normal operation
	$\langle R_2 \neq R_E \rangle$
	
	receive CONNECT from E
	    with $home(E) = R_E \neq R_2$
	  register that $E \in R_2$
	  $state(E) :=$ GUEST
	  send AGENT_UPDATE to R_E
		receive AG_UPDATE from $E \in R_2$
		    with $home(E) = R_E$
		  $state(E) :=$ REMOTE
		  for each m in buf[E]
		    send m to E
		    delete buf[E]
		
		receive CONNECT from E
		    with $home(E) = R_E$
		  $state(E) :=$ HOME
		  for each m in buf[E]
		    send m to E
		    delete buf[E]

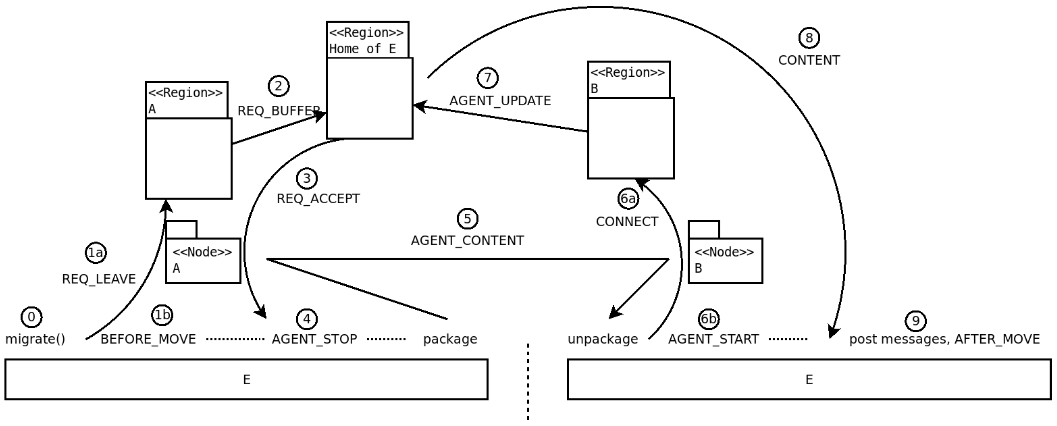


Figure 3. A representation of the process of migration of an entity between two regions of which none is its home region. The phases shown are presented in Sections 4.1.2 and 4.1.3.

4.1.3. Entity Mobility

A mobile entity which must be able to migrate between two nodes brings some implementation challenges also from the point of view of internal functionality. Note that entities in FLASH-MAS, unlike agents in Jade or in JIAC, have no *pre-established* internal structure. Some standard structures exist, but they are not mandatory.

One of these standard structures most used in FLASH-MAS applications, is the *Composite Agent*, an agent which is constructed exclusively as a set of shards, brought together by a queue of events; shards are able to *post* events in a queue, and an event-processing thread takes each event from the queue and disseminates it to each of the shards in the agent before moving on to the next event (see Figure 4).

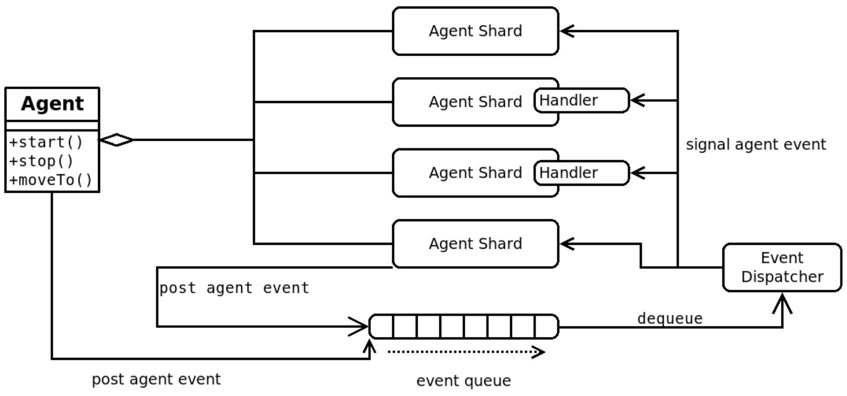


Figure 4. A perspective on how events are processed in a Composite Agent instance.

A similar structure can be assumed for other entities containing shards or other entities containing sub-entities in that a shard can post events to its parent entity, and the shard can receive events which occur in the parent entity. The structure of the entity, however, is arbitrary, and any entity (including shards) can launch several threads, timers, and so on. This brings challenges when having to suspend the execution of an entity so that it can be resumed later.

FLASH-MAS already offers an implementation for the migration process, which is used primarily by the *Mobile Composite* agent implementation, and that serves as a guideline for any entity that wishes to implement mobility. In the rest of this section, we will refer to the implementation of the Mobile Composite Agent, but the same principles apply to other implementations.

The main challenge when supporting mobility of an entity with an arbitrary implementation is suspending its activity and its interactions without losing events or messages during the suspension period. Considering entities as event-based, we implement a solution to this challenge via four *events*:

- When the process of migration is triggered by some part of the entity, a BEFORE_MOVE event is disseminated to all the shards (or other components) of the entity. This is a signal that a suspension of activity follows, and they should stop or buffer any further interaction. The messaging shard sends the REQ_LEAVE to the region server. Any messages which the entity still wishes to send or which are received by the messaging shard are now buffered according to the principle of suspending interactions.
- When the confirmation for migration is received by the messaging shard, it closes all means of communication, and it posts an AGENT_STOP event to the entity, signaling that all activity should stop. When this event is processed by all components of the entity, it is ready to migrate.
- The entity packages itself and passes the package to its current node, which sends the package to the destination node, which unpacks the entity and starts it.
- Upon startup, the entity posts an AGENT_START event to all the shards. This means that they should initialize, but it may be that not all the other shards have been started, and interactions are not yet resumed. The messaging shard posts all buffered received messages and sends all buffered outgoing messages. These events will be processed by the entity after the processing of the AGENT_START event is completed. It also posts an AFTER_MOVE event, which at the time when it will be disseminated to other shards will mean that all the shards and components have started, that pending interaction has been processed, and that the activity of the entity is fully resumed.

This process ensures that all sub-agent entities are aware of the current state of the entity and, if they abide by the semantics of the events, no interaction is lost. For instance, a shard will know that after the BEFORE_MOVE event it can still send messages, but these will not be processed until after the migration, and it will know that any interaction it initiates when processing the AGENT_STOP event may be lost because other shards have already ceased activity.

4.2. Multi-Modal Interaction

In a complex, decentralized, multi-entity deployment, it may be that various parts of the deployed system use different communication mechanisms. For instance, in a smart building, there may be local area networks (LANs), Bluetooth-connected sensors and devices, wireless sensor networks using proprietary protocols, and so on. Our goal is to be able to model in FLASH-MAS all the entities in such a deployment, including the support infrastructures themselves, and for entities to be able to interact with each other, via the FLASH-MAS framework, regardless of where in the system the entities are located. The main challenges in this case are the following:

- Being able to address entities in the entire system via unique identifiers;
- Being able to deliver messages between entities in different areas of the system;
- Being able to migrate entities between nodes in different areas of the system.

To ensure that any entity can be addressed via a unique identifier, we adopt the solution of prefixing all identifiers with a URI. This URI may be the address of the central server for a centralized support infrastructure, or it may be the address of a region server in a decentralized support infrastructure such as WS-Regions.

Everything would be quite simple if all entities were identified, in general, by a URI and a name, as in the WS-Regions Protocol; however, this is not always the case. Some support infrastructures may support addressing by simple names. Let us take the following example situation in the context of our running example: in a research laboratory located in room *P308* in the smart building, machines are integrated within FLASH-MAS via a support infrastructure using WebSocket communication; each entity in the laboratory has a unique name, as given by the researchers there. For an experiment, a researcher wishes entity *Sun-seer* to access the value read by a sensor *T* in a micro-meteorological station situated on the roof of the building so that it can predict the weather; there, devices communicate via Bluetooth. Communication between regions in the building is carried out using RESTful web services. Of course, *Sun-seer* may use a specific protocol to read the value from *T*, but integration via FLASH-MAS would mean that it would be sufficient for *Sun-seer* to send a message to *T* as with any other entity, regardless of where *T* is located and how communication is actually performed. Here, there are multiple situations:

- *T* could be a unique name throughout the building, or it may be addressable as entity *T* within the larger context *MSM* (the micro meteorological station);
- *Sun-seer* could be a unique name throughout the building, or it may be addressable as entity *Sun-seer* within the larger context of room *P308*.

We will suppose the second case for each of the two entities. Moreover, in case arbitrary names are not available (see Section 4.3), *MSM* and *P308* are identified within the building by their URIs `build.ing/msm` and `build.ing/p308`. However, entity *Sun-seer* is ignorant of the identifier of the room, and within the room's support infrastructure, its identifier remains just *Sun-seer*.

To support an addressing system and communication in a multi-modal environment, we introduce *Bridge* entities, relying on previous work that we have conducted in the field [52]. They exist in the context of two or more support infrastructures and are able to route messages among these. They also translate the names of sender or receiver entities, making them usable within the target support infrastructure.

Let us use the same example to illustrate how routing using Bridge entities works (also illustrated in Figure 5):

- *Sun-seer* sends a message $\langle \textit{Sun-seer}, \texttt{build.ing/MSM/}T, \textit{getT} \rangle$, where the components of a message are the source, the destination, and the content. It is necessary for the entire identifier of *T* to be known if a directory is not used. The message reaches the WebSocket server in room *P308*;
- The WebSocket server tries to find the name of the destination entity by iterating through progressively shorter prefixes of the destination name. It has no registrations for `build.ing/MSM/T` or `build.ing/MSM`, but an entity (call it *Bridge 1*) has registered to *P308* with the identifier `build.ing`, as an interface to the rest of the building. In the web services infrastructure of the building, *Bridge 1* registered as `build.ing/P308/`;
- *Bridge 1* receives the message and uses the web services support infrastructure to route the message to the meteorological station, but not before prefixing the source of the message with the identifier of *Bridge 1* as a web services end point, transforming the message into $\langle \texttt{build.ing/P308/}\textit{Sun-seer}, \texttt{build.ing/MSM/}T, \textit{getT} \rangle$;
- The message is routed to another bridge entity, which had registered as a web services endpoint with the identifier `build.ing/MSM`;
- *Bridge 2* relays the message to the other support infrastructure it is registered in (*MSM*), but it first removes the prefix with which it is registered as a webservice end point, so the message is now $\langle \texttt{build.ing/P308/}\textit{Sun-seer}, T, \textit{getT} \rangle$;
- *MSM* also serves as a gateway for the Bluetooth devices in the meteorological station and knows the identifier *T*, so it sends the message to *T*, which can reply;
- The reply will follow the same path backwards.

Regarding mobility, again leveraging the principles in FLASH-MAS, mobility is implemented regardless of the actual communication protocol, provided that the support infrastructure can buffer messages for migrating entities. As such, as long as the entity itself supports migration and the source and destination node support migration (so as to send and receive/unpack the entity), the transfer between the two nodes is performed as a *wave* between the nodes and is routed in the same manner. Moreover, since the entity can *ask* the local pylon which messaging shard it should load, there is no problem in changing support infrastructures after migration.

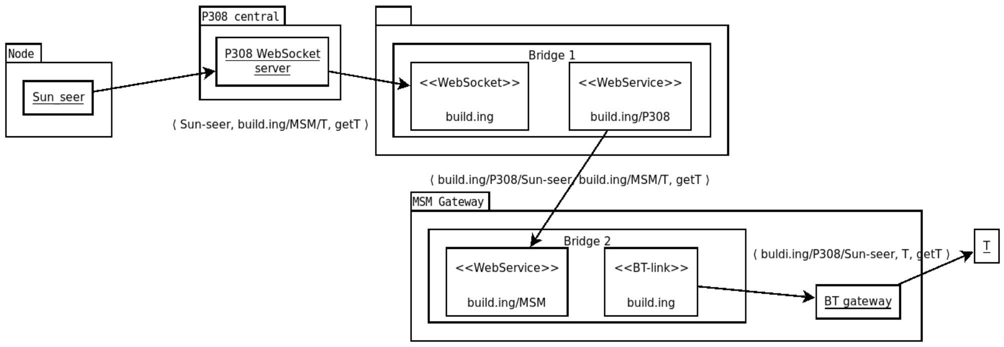


Figure 5. An illustration of the path taken by a message while crossing multiple communication modalities.

4.3. Arbitrary Entity Naming

In a centralized multi-entity system, using arbitrary names is easy. By arbitrary names, we mean that there is no requirement for a relation between the name of an entity and its origin, location, etc. In a centralized multi-entity system in which migration is supported, there can exist a centralized directory which holds a correspondence between the names and, for instance, the nodes where the entities are located.

In a decentralized system, a directory would be a single point of failure, and if it were to fail, any further communication would be hindered. However, having multiple directories is difficult in a system with migrating entities because latency in synchronization would mean that immediately after an entity migrates, messages may reach its previous host, and if the entity migrated more frequently, messages may even reach hosts several migrations ago. A solution is to use a home server solution, such as the one in the WS-Regions Protocol. Having a home server for each entity means that entities still need to know which is the home server for an arbitrary named entity, but home servers do not change when an entity migrates. It is then the home server which serves as a directory for all entities originating from that region.

We use the following algorithm for arbitrary entity naming: all entities have a *short name*, which is an arbitrary sequence of characters, and a *home server*, which is identified by a routable URI. We can construct an entity's *long name* by concatenating the identifier of the home server, a slash, and the short name. The short name must be unique to the home server. For example, entity `Printer` using home server `equipment.build.ing/floor3/P309` , would have the long name `equipment.build.ing/floor3/P309/Printer`. If the short name of the entity contains slashes, then a double slash should be used when assembling it to the URI of the home server, so that the short name can be correctly derived from the long name by other entities.

Different entities situated in different areas of the system may have the same short name, e.g., there may be a `Printer` on the third floor, but also another `Printer` entity on the seventh floor. Directories where both entities wish to publish their name will have to accept only one of the registrations.

There are several *directory servers*. Each directory server is itself an entity and exists in the context of at least one support infrastructure. It holds a lookup table with the correspondence between short names and long names. There are several methods in which the lookup table can be populated in the following ways:

- By synchronizing with other directory servers, whose URIs are given in the configuration.
- By direct request from an entity that wishes that its name is stored by the directory server. The entity may specify that its name:
  - Should be published only in that specific directory server; or
  - Should be published in that server or in any directory servers it synchronizes with at a given maximum number of hops; or

    - Should be published in that server or in any other directory servers synchronized with each other in a given domain.
- Manually by configuring the directory server with a file containing a table.

When an entity sends a message, if the destination is not routable by the support infrastructure and if the support infrastructure is connected to a directory server (or more), the name of the destination will be looked up in the respective servers in order to find its home server.

Optionally, a *local directory* entity may be added to a pylon or to a centralized support infrastructure, which splits the long names that are the sources of incoming messages into short names and home servers and stores these correspondences. The same may be done for the destination of outgoing messages. In the case of incoming messages, a short name will only be saved if it does not clash with an existing identical short name for a different entity.

This method does not necessarily need communication to be conducted via the WS-Regions Protocol. It may be applied regardless of the actual communication method to know what location to send a message to, depending on the arbitrary-named destination for that message. Of course, it is more appropriate for decentralized support infrastructures (such as WS-Regions or web services-based methods) or multi-modal communication.

4.4. Increased Communication Efficiency

The routing algorithm discussed in Section 4.1 brings a significant penalty in message delivery time in the case in which messages are sent between entities originating in different regions, and the destination entity is migrating. This is because all messages for an entity must pass through the entity's home region server. This can be mitigated using the *Short Path* Protocol.

The protocol is active on a per-entity basis. For an entity E, the protocol is activated when it sends a request to its home server R_E using a SHORT-PATH_ACTIVATE message (and later canceled by means of a SHORT-PATH_DEACTIVATE message). Say entity E is currently located in a different region R_1, and entity A located in region R_A wishes to send a message to E. There are two paths which could be taken by the message: the "long path" $A \to R_A \to R_E \to R_1 \to E$ and the "short path" $A \to R_A \to R_1 \to E$. However, using the short path would encounter issues when entity E would leave region R_1. The initiation of the *Short Path* Protocol works through the following steps:

- When A sends a message to E, R_E sends an AGENT_LOCATION message to R_A containing the URI of R_1. R_A stores the information that it can use the short path for entity E (regardless of what entities in region R_A will send messages to E from now on).
- R_A sends a LONG-PATH-STOP message to R_1 via the long path (via R_E) and a SHORT-PATH-BEGIN message directly to R_1, both containing the name E.
- After R_1 receives the SHORT-PATH-BEGIN, it will buffer any messages received via the short path for E until the LONG-PATH-STOP message is received to be sure that messages received via the short path do not get delivered to E before messages sent via the long path before the Short Path Protocol is initiated.
- R_1 will store the information that region R_A is sending messages to entity E via the short path.

When entity E announces to R_1 that it is leaving for another region, the protocol must be ended:

- R_1 sends a SHORT-PATH-STOP message to all regions that were using the Short Path Protocol with entity E.
- R_A cancels the Short Path Protocol for entity E and responds with a SHORT-PATH-END message meaning that this will be the last message sent via the short path. It also sends a LONG-PATH-BEGIN to R_E.
- R_1 relays all incoming messages for E, including all SHORT-PATH-END messages, to R_E.

- When entity E arrives in a new region, R_E will send, for each region, first the saved messages received before the SHORT-PATH-END from that region and then any messages received from that region after the LONG-PATH-BEGIN message. This ensures that messages sent to E from a region arrive in the same order in which they were sent.

5. Discussion and Experimental Results

We have analyzed the trade-offs brought by the solutions presented in Section 4, especially in the WS-Regions Protocol. The goals for this analysis were twofold: first, to verify the validity of the protocol with respect to the challenges in Section 3.1; second, to evaluate how decentralization affects the performance of the system, especially in terms of message delivery times. For the latter of the two goals, we made an in-depth comparison with Jade, which is currently the most popular MAS framework and uses centralized communication. The results were very satisfactory.

5.1. Experimental Setup

To perform the tests, we developed two testing scenarios, called "intensive" and "mobility". In both scenarios, there are four nodes, each located on a different physical machine. There are two regions, each containing two of the four nodes. In each scenario, a number of messages is exchanged very quickly to test how responsive the communication protocol is. Moreover, in the second scenario, an agent moves very quickly between nodes to test how well the protocol handles the message-changing problem.

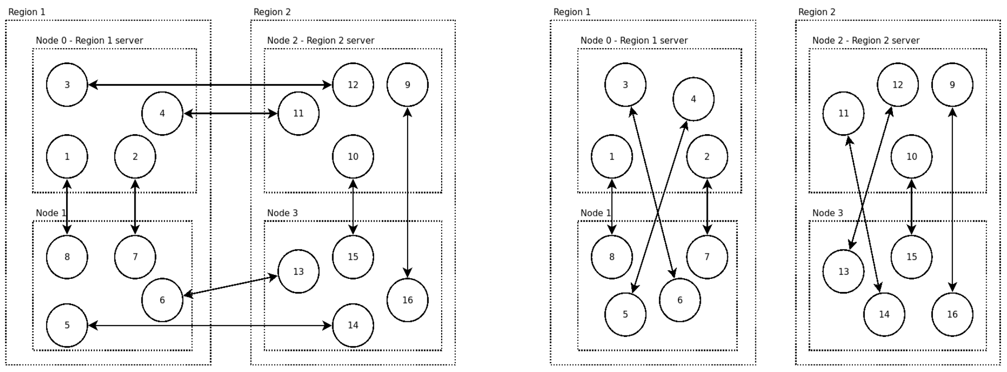
Scenario "intensive": There are 16 agents, with 4 agents located on each of the 4 nodes (see Figure 6): A total of 8 pairs of agents exchange 200 messages each. Exchanges happen very quickly: whenever an agent receives a message, it immediately replies. The scenario has two variants—when half of the messages are exchanged across the two regions and when all the messages are exchanged within the same region (see Figure 6a,b, respectively).

Scenario "mobility": There are four agents—A, B, C, D—with each agent located on one of the four nodes (see Figure 7a). Agent A migrates between the nodes very frequently, waiting almost no time at all between two subsequent migrations. Agents B, C, and D each exchange messages with A, in the same manner as in the "intensive" scenario.

For each scenario, each agent receives a script. In Jade, the scripts were implemented by hand. In FLASH-MAS, each agent is provided with a shard that reads the script from a `.yaml` file containing actions triggered by agent events or delays (see Figure 8). The exact same scenarios were used both with the WS-Regions Protocol and with Jade. For Jade, we used version 4.5.0, the most recent version at the time of writing.

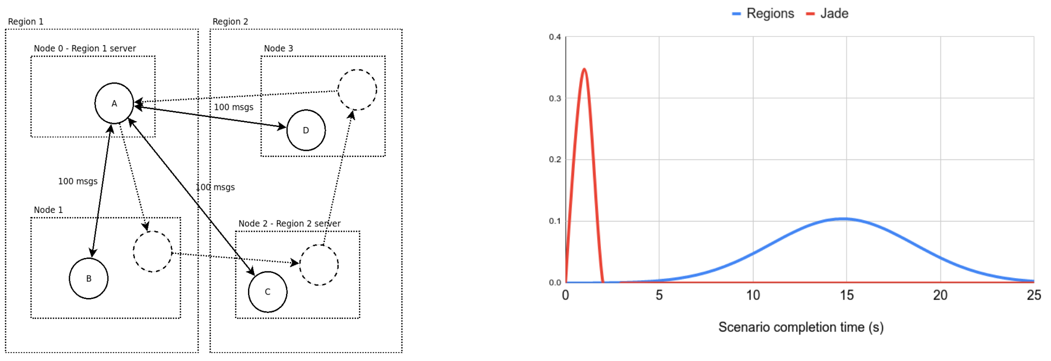
We tested the scenarios using four physical machines connected via a local area network. The machines have varied capabilities—two machine learning servers, one desktop PC, and a low-end laptop. The exact capabilities are not important, as the goals are to check that messages are delivered and to compare the times with the Jade framework, using the exact same setup.

To have a correct test methodology and to have all communication performed simultaneously, the script for each agent was started, with good approximation, at the same time by having an agent send a synchronization message that triggers the beginning of the script after some delay. A local entity is used to *mark* the times when the script starts and when the script is completed. In the analysis, we considered how long it takes for each agent to complete its scenario.



(**a**) (**b**)

Figure 6. Communication patterns in the "intensive communication" scenario for the normal version (**a**) and the "isolated" version (**b**). Each pair of agents exchanges 200 messages. Note that although the arrows are direct between agents, messages pass through region servers.



(**a**) (**b**)

Figure 7. (**a**) The "mobility" scenario. Agent A passes 4 times by each of the 4 nodes. Agents B, C, D each exchange 100 messages with A; (**b**) the distribution of scenario completion times (for agents B, C, D) when using the WS-Regions Protocol and Jade, respectively. However, there are some caveats.

```
script:
 one-R1:
  actions:
  - { action: "MARK", trigger: "BOOT", arg: 15000, arguments: {with: "BOOTING one-R1"} }
  - { action: "SEND_MESSAGE", trigger: "NEXT", arguments: {to: "one-R2", with: "start"} }
  - { action: "SEND_MESSAGE", trigger: "NEXT", arguments: {to: "three-R1", with: "start"} }
  - { action: "SEND_MESSAGE", trigger: "NEXT", arguments: {to: "five-R1", with: "start"} }
  - { action: "SEND_MESSAGE", trigger: "NEXT", arguments: {to: "seven-R1", with: "start"} }

  - { action: "MARK", trigger: "DELAY", arg: 5220, arguments: {with: "one-R1"} }
  - { action: "SEND_MESSAGE", trigger: "NEXT", arguments: {to: "eight-R1", with: "check"} }
  - { times: 100, action: "SEND_MESSAGE", trigger: "EVENT", arg: "AGENT_WAVE", arguments: {to: "eight-R1", with: "check #times"} }
  - { action: "MARK", trigger: "EVENT", arg: "AGENT_WAVE" }
```

Figure 8. An example `yaml` script for an agent. Simpler names are used instead of URIs.

5.2. *Analysis of Results*

We analyzed the results obtained in terms of the distribution of scenario completion times. The completion times for each agent vary greatly, mainly because the machines used in the experiments are quite different. That is why we performed the experiments several times, and we analyzed the distribution and the quartiles for the scenario completion times.

For the "intensive" scenario, the results are shown in Figure 9. We compared the "cross-region" and the "isolated" variant of the scenario running on the WS-Regions Protocol, with the same scenario running on the existing FLASH-MAS centralized WebSocket-based infrastructure and with the same scenario implemented using Jade agents. We observe the following:

- The times for the cross-region variant of the scenario when using WS-Regions are by far the longest, longer by a factor of about three compared to using Jade. This is because cross-region communication for some agent pairs can take three hops (node—region server—regions server—node).

- However, in the isolated variant, when only communicating within the same region, the times are significantly shorter when using WS-Regions rather than Jade, by a factor of three on average. This is because in Jade, all messages must pass through the main container, whereas in WS-Regions the regions share the effort of routing messages.
- The centralized WebSocket infrastructure performs better than Jade, but with comparable results. Just as in the case with Jade, WS-Regions performs significantly worse in the cross-region variant and better in the isolated variant.

For the "mobility" scenario, the results are shown in Figure 7b. Because the number of experiments is large, the figure shows a normal distribution based on the completion times using the two infrastructures. We can see that Jade had significantly better performance than WS-Regions. However, there is an important caveat: when inspecting the console output, we could see that, except for the first message, *all of the message exchanges* happened *after* agent A was finished with all migration. In WS-Regions, this does not happen: logs show that almost all the messages are delivered as intended as soon as an agent reaches a node, messages are delivered in the correct order, and in the end all messages reach their destination.

We further evaluated how Jade handles the message-changing problem. We changed the behavior of agent A to keep migrating for 20 cycles. What happens is that message delivery fails completely because Jade cannot find the agent and does not buffer the messages for A. Hence, very few or no messages were delivered. This happened even if we allowed a delay of up to 50 ms in which A would have had time to receive its messages and reply to the other agents. Figure 10 shows the console output for Node 0. In this experiment, only four messages were delivered. In contrast, the WS-Regions Protocol was particularly concerned with the message-changing problem, and all messages were buffered until they were delivered to the destination.

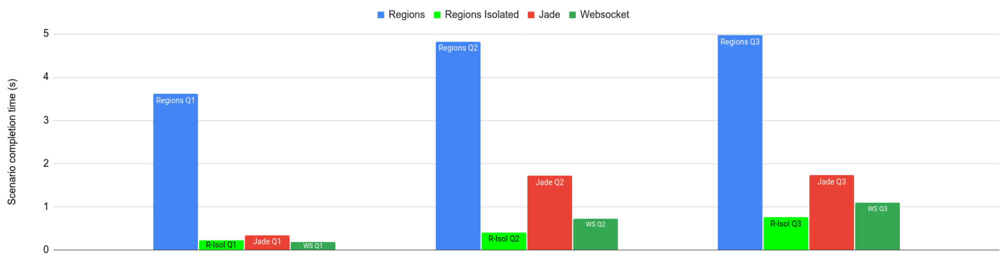


Figure 9. Distribution of scenario completion times (first, second, and third quartiles) for agents in the "intensive communication" scenarios. *Regions*—agents communicate mostly across regions, using the *WS-Regions* Protocol; *Regions Isolated* (*R-Isol*)—agents only communicate within the same region, using the *WS-Regions* Protocol; *Jade*—agents communicate using Jade; *WS*—agents communicate using a centralized WebSocket-based infrastructure.

We can see from these results that the FLASH-MAS with the WS-Regions Protocol and the Jade framework are comparable. There is indeed a significant penalty in delivery times when messages move across regions, but these are offset by delivery times for messages exchanged within the same region. Moreover, WS-Regions solves the problem of message chasing much more elegantly (and effectively) than Jade. Of course, decentralization also brings the advantage of robustness in the face of node failure.

We can also discuss here what would be an appropriate size for regions. Region size is a trade-off between efficiency and robustness. When there are many regions, and one region server fails, the other regions are unaffected, save for any agents that were guests on the failed node. However, having too many regions means more communication overhead. The correct balance is, of course, up to the developer of the multi-entity system.

```
moving to 1
<...> jade.imtp.leap.NodeSkel executeCommand
WARNING: Error serving H-Command jade.core.messaging.Messaging/1:
 jade.core.NotFoundException: Messaging service slice failed
 to find ( agent-identifier :name A@<...>/JADE )
B Received: [B 00 reply] from ( agent-identifier :name A@<...>/JADE  :addresses (sequence <...> ))
moving to 1
<...4 more migrations...>
moving to 1
A Received: C 00 from ( agent-identifier :name C@<...>/JADE  :addresses (sequence <...> ))
moving to 1
<...4 more migrations...>
moving to 1
<...> jade.imtp.leap.NodeSkel executeCommand
WARNING: Error serving H-Command jade.core.messaging.Messaging/1:
jade.core.NotFoundException: Messaging service slice failed
to find ( agent-identifier :name A@<...>/JADE  :addresses (sequence http://<...>/acc ))
moving to 1
<...7 more migrations...>
```

Figure 10. A snippet of the output on Node 0 in the "mobility" scenario, where Jade fails to deliver messages to a fast-migrating agent. Every time the agent is migrating, it is moving to the next node—Node 1.

5.3. Discussion on Robustness

The goal of having created the WS-Regions Protocol is to ensure that there is no single point of failure in a deployed MAS. We have made some observations on the issues that arise when a node fails, both when using WS-Regions and when using Jade, in Table 4.

Table 4. A qualitative analysis of the robustness of the WS-Regions Protocol compared to Jade for cases when a node fails completely.

Infrastructure	**Case**	**Entities Lost**	**Interactions Lost**
WS-Regions	node N which is not region server fails	· entities on node N are lost, but their names are known.	· messages or migrating entities currently on node N.
WS-Regions	node N_R containing the server of region R fails	· entities on node N_R are lost; · guest entities on N_R are lost, but their name is known; · all entities $E_i \in R, E_i \notin N_R$ have invalid names until R is recreated.	· messages and migrating entities in transit in N_R are lost; · entities in R on other nodes than N_R cannot interact; · entities $E_i \in R$ which are remote cannot receive messages and cannot migrate.
Jade	node without main container fails	· agents currently on the node are lost.	· messages or migrating agents currently on node N are lost.
Jade	node with main container fails	· all agents are lost.	· all interactions are lost.

The WS-Regions Protocol has obvious advantages with respect to Jade. When a node in a deployment using WS-Regions fails, it never brings down the entire system. Of course, this does not happen in a centralized communication infrastructure, where the central node stores all information about the deployment.

In the case in which the failed node is not a region server, the failure is detected immediately by the region server because the WebSocket connection with the server is broken. The region server contains all the names of the entities which were in execution on the node, so these could be re-created if needed. Messages arriving for the lost entities are buffered, or for guest entities, sent back to their home servers.

In the case in which the failed node is a region server, the failure is detected immediately by other region servers because region servers form a mesh of WebSocket connections. Entities in that region will not be able to interact until the region server is recreated. Messages for entities in the region will be buffered by other regions. The same happens to messages for entities in the region that are currently remote. If the region server can be re-created, the system can continue to function normally. If not, nodes in that region will remain orphans, with entities functioning but unable to interact; if support for changing the names of entities exists, remote entities from the lost region could change their names and be adopted by their current regions.

6. Conclusions and Future Work

Our general research goal is to make FLASH-MAS a flexible, modular, easy-to-use framework for multi-agent and multi-entity systems in which the model of the system contains abstractions for all the elements in the deployment, including but not limited to support infrastructures, sub-agent entities, deployment partitioning, and so on.

In this context, we developed WS-Regions as a robust, decentralized communication infrastructure for FLASH-MAS entities, which couples principles from older protocols with the use of modern communication methods. This infrastructure is designed so that it is reliable in terms of message delivery, regardless of how frequently the destination entity migrates.

We compared the performance of the WS-Regions Protocol to the Jade framework, as well as to a centralized WebSocket-based infrastructure, on identical scenarios and observed that: (1) the performance for cross-region communication is of the same order of magnitude with that of Jade, albeit worse; (2) the performance for in-region communication is better than that of Jade; and (3) the reliability with which messages reach a frequently migrating destination is much better than in Jade. Hence, we see the trade-off in performance as worth the gain in robustness and reliability.

From our results and the observations that we made, we can conclude that the WS-Regions Protocol has the advantage that it is more robust, and in some cases, it has better performance than a centralized communication infrastructure. The drawbacks of the protocol are increased complexity and the need for message buffers to keep messages while destination entities are migrating.

Our research continues with improvement of the performance in the WS-Regions Protocol and with the design of large-scale scenarios to test the other proposed methods—an enabler for arbitrary entity names and a method for interoperation between multiple communication infrastructures which is transparent to both the source and the destination of the communication. We will further study the robustness of the WS-Regions Protocol, the capacity to recover entities and messages after a failure, and the capacity to react to partial node failures.

Author Contributions: Conceptualization, A.O. and M.P.; methodology, A.O.; software, A.O. and M.P.; validation, A.O. and M.P.; formal analysis, A.O.; writing—original draft preparation, review and editing, A.O.; supervision, A.O.; funding acquisition, A.O. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by a grant of the Ministry of Research, Innovation and Digitization, CNCS - UEFISCDI, project number PN-III-P1-1.1-TE-2021-1422, within PNCDI III, and this work has been partially funded by UEFISCDI project Cornet (1/2018, PN-III-P3-3.6-H2020-2016-0120).

Acknowledgments: The authors would also like to thank Maria Buiac, who contributed to the implementation of entity mobility, and Cosmin Mihai and Florina Năstăsoiu who contributed to the development of WebSocket communication for FLASH-MAS and in the architecture of the framework in general.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Lange, D.B.; Oshima, M. Seven good reasons for mobile agents. *Commun. ACM* **1999**, *42*, 88–89. [CrossRef]
2. Salah, T.; Zemerly, M.J.; Yeun, C.Y.; Al-Qutayri, M.; Al-Hammadi, Y. IoT applications: From mobile agents to microservices architecture. In Proceedings of the 2018 International Conference on Innovations in Information Technology (IIT), Al Ain, United Arab Emirates, 18–19 November 2018; pp. 117–122.
3. Yousefi, S.; Derakhshan, F.; Karimipour, H.; Aghdasi, H.S. An efficient route planning model for mobile agents on the internet of things using Markov decision process. *Ad. Hoc. Netw.* **2020**, *98*, 102053. [CrossRef]
4. Alsboui, T.; Qin, Y.; Hill, R.; Al-Aqrabi, H. Enabling distributed intelligence for the Internet of Things with IOTA and mobile agents. *Computing* **2020**, *102*, 1345–1363. [CrossRef]
5. Ismail, L.; Materwala, H. IoT-edge-cloud computing framework for QoS-aware computation offloading in autonomous mobile agents: Modeling and simulation. In *Proceedings of the International Conference on Mobile, Secure, and Programmable Networking*; Springer: Berlin/Heidelberg, Germany, 2021; pp. 161–176.
6. Boopathi, M.; Seetha, R. Accurate Detection of Multi-layer Packet Dropping Attacks Using Distributed Mobile Agents in MANET. In *Proceedings of the Journal of Physics: Conference Series*; IOP Publishing: Bristol, UK, 2021; Volume 1979; p. 012040. [CrossRef]
7. Uddin, M.; Memon, J.; Alsaqour, R.; Shah, A.; Rozan, M.Z.A. Mobile agent based multi-layer security framework for cloud data centers. *Indian J. Sci. Technol.* **2015**, *8*, 1. [CrossRef]
8. Colson, C.M.; Nehrir, M.H. A review of challenges to real-time power management of microgrids. In Proceedings of the 2009 IEEE Power & Energy Society General Meeting, Calgary, AB, Canada, 26–30 July 2009; pp. 1–8.
9. Brearley, B.J.; Prabu, R.R. A review on issues and approaches for microgrid protection. *Renew. Sustain. Energy Rev.* **2017**, *67*, 988–997. [CrossRef]
10. Coelho, V.N.; Cohen, M.W.; Coelho, I.M.; Liu, N.; Guimarães, F.G. Multi-agent systems applied for energy systems integration: State-of-the-art applications and trends in microgrids. *Appl. Energy* **2017**, *187*, 820–832. [CrossRef]
11. Pal, C.V.; Leon, F.; Paprzycki, M.; Ganzha, M. A Review of Platforms for the Development of Agent Systems. *arXiv* **2020**, arXiv:2007.08961.
12. Savaglio, C.; Ganzha, M.; Paprzycki, M.; Bădică, C.; Ivanović, M.; Fortino, G. Agent-based Internet of Things: State-of-the-art and research challenges. *Future Gener. Comput. Syst.* **2020**, *102*, 1038–1053. [CrossRef]
13. Bellifemine, F.; Poggi, A.; Rimassa, G. JADE—A FIPA-compliant agent framework. In Proceedings of the Fifth International Conference on Autonomous Agents, AGENTS 2001, Montreal, Canada, 28 May–1 June 2001; pp. 216–217.
14. Lützenberger, M.; Küster, T.; Konnerth, T.; Thiele, A.; Masuch, N.; Heßler, A.; Keiser, J.; Burkhardt, M.; Kaiser, S.; Albayrak, S. JIAC V: A MAS framework for industrial applications. In Proceedings of the 2013 International Conference on Autonomous Agents and Multi-Agent Systems. International Foundation for Autonomous Agents and Multiagent Systems, Saint Paul, MN, USA, 6–10 May 2013 ; pp. 1189–1190.
15. Boissier, O.; Bordini, R.H.; Hübner, J.F.; Ricci, A.; Santi, A. Multi-agent oriented programming with JaCaMo. *Sci. Comput. Program.* **2013**, *78*, 747–761. [CrossRef]
16. Palanca, J.; Terrasa, A.; Julian, V.; Carrascosa, C. Spade 3: Supporting the new generation of multi-agent systems. *IEEE Access* **2020**, *8*, 182537–182549. [CrossRef]
17. Ricci, A.; Viroli, M.; Omicini, A. Give agents their artifacts: The A&A approach for engineering working environments in MAS. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*; ACM: Honolulu, HI, USA, 2007; p. 150.
18. Hannoun, M.; Boissier, O.; Sichman, J.S.; Sayettat, C. MOISE: An organizational model for multi-agent systems. In *Advances in Artificial Intelligence*; Springer: Berlin/Heidelberg, Germany, 2000; pp. 156–165.
19. Olaru, A. tATAmI-2—A Flexible Framework For Modular Agents. In *Proceedings of the AgTAmI 2015, the International Workshop on Agent Technology for Ambient Intelligence, the 20th International Conference on Control Systems and Computer Science, Bucharest, Romania, 27–29 May 2015*; Dumitrache, I., Florea, A.M., Pop, F., Dumitrascu, A., Eds.; IEEE Computer Society: Washington, DC, USA; 2015, Volume 2, pp. 703–710. . [CrossRef]
20. Rawat, A.; Sushil, R.; Sharm, L. Mobile agent communication protocols: A comparative study. In *Computational Intelligence in Data Mining*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 131–141.
21. Kia, S.S.; Rounds, S.; Martinez, S. Cooperative localization for mobile agents: A recursive decentralized algorithm based on Kalman-filter decoupling. *IEEE Control Syst. Mag.* **2016**, *36*, 86–101.
22. De Gennaro, M.C.; Jadbabaie, A. Decentralized control of connectivity for multi-agent systems. In Proceedings of the 45th IEEE Conference on Decision and Control, San Diego, CA, USA, 13–15 December 2006; pp. 3628–3633.
23. Olaru, A.; Sorici, A.; Florea, A.M. A Flexible and Lightweight Agent Deployment Architecture. In Proceedings of the 22nd International Conference on Control Systems and Computer Science, Bucharest, Romania, 28–30 May 2019; pp. 251–258.
24. Qadori, H.Q.; Zulkarnain, Z.A.; Hanapi, Z.M.; Subramaniam, S. Multi-mobile agent itinerary planning algorithms for data gathering in wireless sensor networks: A review paper. *Intl. J. Distrib. Sens. Netw.* **2017**, *13*, 1550147716684841. [CrossRef]
25. Derakhshan, F.; Yousefi, S. A review on the applications of multiagent systems in wireless sensor networks. *Int. J. Distrib. Sens. Netw.* **2019**, *15*, 1550147719850767. [CrossRef]
26. Outtagarts, A. Mobile agent-based applications: A survey. *Int. J. Comput. Sci. Netw. Secur.* **2009**, *9*, 331–339.

27. Bellifemine, F.; Poggi, A.; Rimassa, G. Developing multi-agent systems with JADE. In *Intelligent Agents VII Agent Theories Architectures and Languages*; John Wiley & Sons: Hoboken, NJ, USA, 2001; pp. 42–47.
28. John, V.; Liu, X. A Survey of Distributed Message Broker Queues. *arXiv* **2017**, arXiv:1704.00411.
29. Bosse, S. Mobile multi-agent systems for the internet-of-things and clouds using the javascript agent machine platform and machine learning as a service. In Proceedings of the 2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud), Vienna, Austria, 22–24 August 2016; pp. 244–253.
30. Bosse, S. Self-organising Urban Traffic control on micro-level using Reinforcement Learning and Agent-based Modelling. In *Proceedings of the SAI Intelligent Systems Conference*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 745–764.
31. Deugo, D. Mobile Agent Messaging Models. In Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems, ISADS 2001, Dallas, TX, USA, 26–28 March 2001; IEEE Computer Society: Washington, DC, USA; 2001; pp. 278–286.
32. Hidayat, A. A review on the communication mechanism of mobile agent. *Int. J. Video Image Process. Netw. Secur* **2011**, *11*, 6–9.
33. Virmani, C. A comparison of communication protocols for mobile agents. *Int. J. Adv. Technol* **2012**, *3*, 114–122.
34. Olaru, A.; Petrescu, D.; Florea, A.M. Comparing the Performance of Message Delivery Methods for Mobile Agents. In *Proceedings of the International Conference on Practical Applications of Agents and Multi-Agent Systems*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 188–199. [CrossRef]
35. Wojciechowski, P.T. Algorithms for location-independent communication between mobile agents. In Proceedings of the AISB'01 Symposium on Software Mobility and Adaptive Behaviour, AISB'01 Convention, York, UK, 21–24 March 2001 .
36. Yousuf, A.Y.; Hammo, A. Developing a new mechanism for locating and managing mobile agents. *J. Eng. Sci. Technol.* **2012**, *7*, 614–622.
37. Cabri, G.; Leonardi, L.; Zambonelli, F. Mobile-agent coordination models for internet applications. *Computer* **2000**, *33*, 82–89. [CrossRef]
38. Choi, S.; Kim, H.; Byun, E.; Hwang, C.; Baik, M. Reliable asynchronous message delivery for mobile agents. *IEEE Internet Comput.* **2006**, *10*, 16–25. [CrossRef]
39. Desbiens, J.; Lavoie, M.; Renaud, F. Communication and tracking infrastructure of a mobile agent system. In Proceedings of the Thirty-First Hawaii International Conference on System Sciences, Kohala Coast, HI, USA, 6–9 January 1998; Volume 7, pp. 54–63.
40. Baumann, J.; Rothermel, K. The Shadow Approach: An Orphan Detection Protocol for Mobile Agents. *Pers. Ubiquitous Comput.* **1998**, *2*, 100–108.
41. Di Stefano, A.; Santoro, C. Locating mobile agents in a wide distributed environment. *IEEE Trans. Parallel Distrib. Syst.* **2002**, *13*, 844–864. [CrossRef]
42. Jingyang, Z.; Zhiyong, J.; Daoxu, C. Designing reliable communication protocols for mobile agents. In Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops, Providence, RI, USA, 19–22 May 2003; pp. 484–487.
43. Cao, J.; Xu, W.; Chan, A.T.; Li, J. A reliable multicast protocol for mailbox-based mobile agent communications. In Proceedings of the 2005 International Symposium on Autonomous Decentralized Systems, ISADS 2005, Chengdu, China, 4–8 April 2005; pp. 74–81.
44. Ge, X.; Han, Q.L. Distributed formation control of networked multi-agent systems using a dynamic event-triggered communication mechanism. *IEEE Trans. Ind. Electron.* **2017**, *64*, 8118–8127. [CrossRef]
45. Ismail, Z.H.; Sariff, N.; Hurtado, E. A survey and analysis of cooperative multi-agent robot systems: Challenges and directions. In *Applications of Mobile Robots*; IntechOpen: London, UK, 2018 ; pp. 8–14.
46. Jiménez, A.C.; García-Díaz, V.; Bolaños, S. A decentralized framework for multi-agent robotic systems. *Sensors* **2018**, *18*, 417. [CrossRef] [PubMed]
47. Qin, J.; Ma, Q.; Shi, Y.; Wang, L. Recent advances in consensus of multi-agent systems: A brief survey. *IEEE Trans. Ind. Electron.* **2016**, *64*, 4972–4983. [CrossRef]
48. Zhang, D.; Feng, G.; Shi, Y.; Srinivasan, D. Physical safety and cyber security analysis of multi-agent systems: A survey of recent advances. *IEEE/CAA J. Autom. Sin.* **2021**, *8*, 319–333. [CrossRef]
49. Murugesan, S.; Liu, Y.C. Resilient finite-time distributed event-triggered consensus of multi-agent systems with multiple cyber-attacks. *Commun. Nonlinear Sci. Numer. Simul.* **2023**, *116*, 106876. [CrossRef]
50. Fette, I.; Melnikov, A. The WebSocket Protocol. Technical Report rfc6455, Internet Engineering Task Force (IETF), 2011. Available online: https://www.rfc-editor.org/rfc/rfc6455 (accessed on 7 March 2023).
51. Cao, J.; Feng, X.; Lu, J.; Chan, H.C.; Das, S.K. Reliable message delivery for mobile agents: Push or pull? *IEEE Trans. Syst. Man Cybern.-Part A Syst. Humans* **2004**, *34*, 577–587. [CrossRef]
52. Olaru, A.; Florea, A.M. A Framework for Integrating Heterogeneous Agent Communication Platforms. In Proceedings of the ACSys 2015, the 12th Workshop on Agents for Complex Systems, in Conjunction with SYNASC 2015, the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, 21–24 September 2015; pp. 399–406. [CrossRef]