Artificial Intelligence Review
https://doi.org/10.1007/s10462-023-10596-z

A middleware for providing communicability to Embedded MAS based on the lack of connectivity

Vinicius Souza de Jesus[1] · Nilson Mori Lazarin[2] · Carlos Eduardo Pantoja[2] · Fabian César Pereira Brandão Manoel[2] · Gleifer Vaz Alves[3] · José Viterbo[1]

© The Author(s), under exclusive licence to Springer Nature B.V. 2023

Abstract

An Embedded multi-agent system (Embedded MAS) is an embedded cognitive system based on agents cooperating to control hardware devices. These agents are autonomous and proactive entities capable of decision-making and can constantly acquire new knowledge via interaction with other agents and the environment. Since the interaction between agents is relevant for acquiring new knowledge, issues such as the communicability and mobility of agents from different Embedded MAS must be highlighted. The classification of a MAS as Open or Closed only considers the mobility of agents, but communicability also needs to be considered. For this, we extend the notion of openness in these systems to consider the existence of Totally Closed and Limited Open MAS, to consider agents from an Embedded MAS without the ability to move or communicate or when they lose the ability to communicate but still can move to other systems. In cooperative missions where several devices adopt Embedded MAS, they should not become totally closed since they lose the ability to cooperate and could put the mission at risk. Some existent works considering Embedded MAS relies upon IoT infrastructures to guarantee communicability and mobility. But, in cases where these infrastructures are temporarily or permanently unavailable, the system becomes totally closed. Even when alternatives exist, they do not use cryptography. Therefore, we present a middleware for supporting the development of Embedded MAS, considering radiofrequency ad-hoc communication to reduce the dependency on centralized infrastructures. An extended protocol supports message exchange between devices using cryptography. We also present a proof of concept application and a formalization of our model.

Keywords Embedded multi-agent systems · Embedded systems · Communication protocol

1 Introduction

A Multi-Agent System (MAS) is a group of agents pursuing common or conflicting goals and acting upon a sphere of influence projected on a specific area of a real or simulated environment (Wooldridge 2009). Since intelligent agents are autonomous and proactive entities built on software or hardware, they can be employed in embedded systems to

Extended author information available on the last page of the article

Published online: 22 September 2023

manage devices. Thus, an Embedded MAS is defined as a group of software agents organized in a MAS acting to manage a physical device where they are inserted, interfacing sensors and controlling actuators (Brandão et al. 2021).

Some agents can move from one MAS to another (Pham and Karmouch 1998). These mobile agents depend on network infrastructure, the MAS capacity to allow agents' entrance and exit, and permissions from the MAS to enter or not in the system at runtime. Then, a MAS—Embedded or not—can be Closed when its agents cannot move to other systems or platforms or Open when agents are allowed to leave and enter the system (Artikis and Pitt 2008). However, the social ability of agents to communicate with each other could affect the behavior of Closed and Open MAS. For example, Closed MAS can allow agent communication between different systems even if it is impossible for agents to move, as we can observe in many agent programming languages (Hindriks et al. 1999; Busetta et al. 1999; Bellifemine et al. 2007; Bordini et al. 2007; Dennis and Farwer 2008). But, if there is no native communication infrastructure available for that, or it is temporally unavailable, or the responsible technology is permanently damaged, the MAS becomes totally closed for both mobile agents and communication.

The same applies to Open MAS, which allows agent mobility, but communication could be somehow affected by external reasons or forbidden. For example, if an Open MAS has different infrastructures for agent mobility and communication with other systems, and, for any reason, the communication infrastructure fails, the system becomes a Limited Open MAS. Communication is intrinsic, in this case, since agents can move and communicate and then go back to their origin MAS. However, it is risky to move in domains where agents can die or be predated (Jesus et al. 2021). Our extended classification considering Limited Open and Totally Closed MAS is essential since it poses challenges and concerns when using Embedded MAS. The unpredictability of autonomous systems in real environments could lead to Embedded MAS becoming Limited Open, or even Totally Closed. It implies that the device is still functional but loses its social ability, which is crucial for collaboration in some scenarios such as rescue and military.

Several works explore agents and Embedded MAS on top of devices to act proactively at the edge of Internet of Things (IoT) systems (Savaglio et al. 2016; Hernández and Reiff-Marganiec 2016; de Castro et al. 2022). However, there is a dependency on communication network infrastructures (i.e., TCP/IP, IoT middleware, 3 G networks), which can compromise the system's autonomy when unavailable, turning the device into a Totally Closed system. Moreover, agent platforms (Lazarin and Pantoja 2015; Taboun and Brennan 2017; Zhang et al. 2019) and customized agent architectures (Jensen 2010; Pantoja et al. 2016a; Manoel et al. 2020) provide ad-hoc solutions to create Embedded MAS without social ability making the device rely on IoT Architectures (Pantoja et al. 2018; Hamdani et al. 2022; Palanca et al. 2022) to communicate with other devices. The main problem is that these works depend on IoT infrastructure and do not present alternatives in critical situations (network unavailability) or communication security in open channels. Besides, some of them consider ad-hoc solutions disregarding any type of communication (Jensen (2010); Lazarin and Pantoja (2015); Pantoja et al. (2016b); Manoel et al. (2020)).

One way to overcome this situation is to employ Radio Frequency (RF) gadgets on devices since they use the air to broadcast messages. RF has been widely used in robotics (Isma et al. 2022; Groshev et al. 2022) and telecommunications (Elgeziry et al. 2022; Kumawat 2022). Still, when it comes to Belief-Desire-Intention (BDI) agents embedded in devices, there is a lack of solutions and proposals that minimize infrastructure dependency and guarantee proper communication between Embedded MAS from different devices. The BDI is a cognitive model that organizes the agent reasoning considering beliefs, plans, and actions to achieve goals based on intention and desires (Bratman 1987). However, purely RF adoption raises some concerns. As it uses the air as a communication channel, anyone accessing the channel can capture the message since it travels without encryption to secure the message content during the communication process. Besides, in some situations, Embedded MAS needs to communicate in groups differently from direct communication with another device. Sometimes, it is interesting to adopt secure channels for specific groups.

For example, in a scenario where autonomous vehicles—controlled by embedded MAS—are on a faraway highway, and for any reason, the telecommunication infrastructure is not available. On this highway, vehicles traveling could be from two groups: civil vehicles from the general public, and public safety vehicles, from police, support, hospital, and ambulances. If an accident happens, these vehicles will not communicate unless equipped with RF to allow communication using data dissemination through the air. In this case, the damaged vehicle could send an unencrypted *broadcast* message to all nearby vehicles. If a public safety vehicle receives the message, it can send an encrypted *unicast* message to the damaged vehicle, informing that help is coming soon. Then, it could send an encrypted *multicast* message to the public safety vehicles group to coordinate how they will engage in helping the damaged vehicle. Notice that *unicast* and *multicast* also use RF and the air as a communication channel, but they use cryptography to obfuscate the message.

This paper presents a middleware that offers a novel alternative communication channel using RF to avoid Totally Closed and Limited Open MAS situations in Embedded MAS using BDI agents. The middleware works based on a new communication protocol that allows communication from controllers to controllers using RF messages and between controllers and high-level programming languages (the Embedded MAS). The protocol is not limited to RF messages. It runs along with other communication infrastructures (3G, IoT, etc.) or when there are no other methods available for communication, thus allowing a Totally Closed MAS to transmit and send information to other devices. This behavior is innovative in Embedded MAS since communication between devices without a central infrastructure has not been explored concerning groups and cryptography.

The BDI agents interpret serial messages from sensors as beliefs and RF messages as social messages with an illocutionary force that defines the message's purpose. Three types of messages are available, *broadcast* messages without encryption and *unicast* and *multicast* messages are encrypted using Advanced Encryption Standard (AES) 128 bits with Cipher Feedback (CFB) mode of operation.

With our middleware, it is possible to build Embedded MAS capable of keeping a public air communication channel active independent of network infrastructure. The device empowered by Embedded MAS can send messages in this public channel to reach nearby devices (broadcast). Furthermore, it can send secure messages (cryptographed) to single devices (unicast) or a pre-defined group of devices (multicast). All of this is possible using the new agent architecture and a diffuse action available for agents.

For this, we extend the Javino serial interface (Lazarin and Pantoja 2015) to allow end-to-end communication using Radio Frequency (RF) using the protocol for sending *broadcast*, *multicast*, and *unicast* messages. We also extend the JaCaMo framework (Boissier et al. 2013) by adding a customized agent architecture to create an Embedded MAS with hardware interfacing capability and, simultaneously, to send and receive RF messages. We assembled three vehicle prototypes considering the scenario described above as a proof of concept that the middleware properly works using RF channels. The main contributions and novelties of this paper are:

1. a novel middleware for supporting the development of Embedded MAS to prevent the lack of connectivity;
2. the protocol for message exchange;
3. the new BDI agent architecture able to understand protocols' messages as BDI constructions;
4. extended definitions for openness in MAS;
5. a complete proof of concept is presented, where prototypes are embedded with agents using encrypted messages for communication and endowed with a formalization model to guarantee the communication protocol properly works.

This work is organized as follows; in Sect. 2, some theoretical background is discussed and the new definitions of Open and Closed MAS are presented; in Sect. 3, we show some related works in the Embedded MAS domain considering the last five years; Sect. 4 shows details of the proposed protocol and its implementation; in Sect. 5 presents the proof of concept applied in a case study with vehicles prototypes. Finally, Sect. 6 concludes the paper by presenting the final considerations and directions for future works.

2 Theoretical background

The agent theory has been widely employed in many cyber-physical domains during the past years (Leitão et al. 2016). Autonomous agents have pro-activity, social abilities, autonomy, and also mobility (Wooldridge 2000). Mobile agents can move from one Open MAS to another if there is an external infrastructure available for agents to travel, and the Open MAS allows the entrance and exit of agents from it. Conversely, a Closed MAS is the common vision of MAS, where agents can only exist in their system without network infrastructure or permissions for agents to move from one system to another.

However, classifying a MAS as open or closed just by observing its ability to allow agents' mobility is limited when considering Embedded MAS. In these cases, a MAS is responsible for controlling a device guaranteeing autonomy and pro-activity by accessing actuators, sensors, and communication infrastructures. In another way, an Embedded MAS is an embedded system that depends on hardware, including communication infrastructure, to maintain the social ability and communicability of the device (Brandão et al. 2021). Therefore, agents from a Closed MAS can still communicate with agents from another MAS and be forbidden to leave the system. Similarly, an Open MAS could allow agent mobility, despite direct communication could be unavailable (considering that the MAS uses different infrastructures for communication and mobility). Direct communication happens in an Open MAS when two agents in different MAS exchange messages without leaving their systems. Then, we define direct communication as follows:

Definition 1 Direct Communication is an end-to-end message exchange between two agents hosted in different MAS using any communication infrastructure.

Afterward, we extend the traditional definitions of Open and Closed MAS and define two new classifications: Limited Open MAS and Totally Closed MAS, considering the social ability of communication.

Definition 2 An Open MAS is an agent-based system that allows mobile agents to enter and leave the system and direct communication between agents from different MAS.

In this case, if an Embedded MAS is an Open MAS, its agents could leave and enter the system anytime they want and perform direct communications with other Embedded MAS. For example, two independent, autonomous vehicles controlled by different Embedded MAS: one agent from one vehicle can communicate with another agent from the second vehicle performing direct communication. Still, some of their agents could move to the manufacturer's headquarter to improve their abilities.

Definition 3 A Closed MAS is an agent-based system that does not allow agents to enter and leave the system but allows direct communication between agents from different MAS.

This case is the most common appearance in MAS development since most agent-oriented programming languages and frameworks allow creating communication structures (e.g., artifacts) or using an existing one. Still, agents cannot move from one system to another. In Closed MAS, the behavior is quite similar to the Open MAS. However, the agents could not move because it was a design-time decision or the mobility ability was damaged or intentionally turned off.

Definition 4 A Limited Open MAS is an agent-based system that allows mobile agents to enter and leave the system but does not allow direct communication between agents from different MAS.

It is the opposite of the Closed MAS. Agents can move to a different MAS but cannot exchange messages by performing direct communication. However, agents can still communicate locally or move to another system and exchange messages into the addressed system. Real environments where Embedded MAS acts pose several challenges due to unpredictability and the amount of information and agents are exposed to many hazards when moving from one system to another. So, designing an Embedded MAS as a Limited Open MAS is not an immediate decision and sometimes depends on the communication infrastructure available. Regardless of the designer's decisions, any communication infrastructure can become unavailable for many reasons (e.g., physical attacks and malfunctioning). In these cases, Embedded MAS will become temporally or permanently a Limited Open MAS. For example, a vehicle could crash and damage the RF sensor responsible for direct communication with other vehicles, but they could still have available telecommunication infrastructure such as the Internet.

Definition 5 A Totally Closed MAS does not allow agents to enter and leave the system, nor does direct communication between agents from different MAS.

It is the more restrictive case possible that an Embedded MAS can assume. Despite the mobility and communicability restrictions, a Totally Closed MAS as an Embedded MAS is a typical configuration when considering ad-hoc solutions where there is no need for distributed computing among several devices. The Embedded MAS can sense the environment, act upon it, and be autonomous and proactive, but it does not communicate with any other entity. However, communication is essential when it is necessary for cooperation between several Embedded MAS, so Totally Closed MAS should be avoided. Moreover,

Table 1 MAS classification considering mobility and communicability

	Totally closed MAS	Limited open MAS	Closed MAS	Open MAS
Mobility	–	✓	–	✓
Communicability	–	–	✓	✓

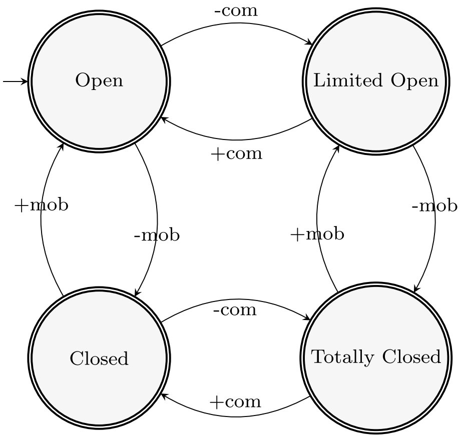

Fig. 1 FSM for the behavior of the Embedded MAS transitions

damages and unavailability can lead to a Totally Closed MAS situation. In Table 1, we present the new classification from a more restrictive to a less restrictive behavior of a MAS, comparing the existence of mobility and communicability.

Classifying MAS by considering both mobility and communicability when using Embedded MAS can help the device's designer understand which components to use and how to prepare backup functionalities based on what is necessary to attend to the application domain requirements. A Totally Closed MAS is an undesired configuration in distributed systems with collective goals because it can isolate a device from its team. Once missions and goals can change based on the unpredictability of dynamic environments, an incommunicable device cannot update its goals and missions, despite maintaining autonomy, pro-activity, and reasoning ability. It might even compromise the global mission without coordinating with its pairs. For example, all other MAS classifications have at least one way of communicating, which can be used for coordination.

Since an Embedded MAS manages hardware and uses external communication infrastructures, it is not guaranteed that they will be forever functional and available. Both hardware and communication infrastructure could experience malfunctioning, instabilities, and damage in an ongoing mission. When this happens, one Embedded MAS can move from one state to another, causing several concerns already discussed before. For this, we define a Finite-State Machine (FSM) to model the behavior of these transitions to guide the designer during the Embedded MAS and device creation. The FSM diagram can be seen in Fig. 1. The symbols of the alphabet used in the FSM are defined:

`-com` means lost of communicability.
`+com` means the communicability has been restored.
`-mob` means lost of mobility.

+mob means the mobility has been restored.

Therefore, it is important to provide mechanisms in practical applications of Embedded MAS to guarantee—or least avoid—that a MAS does not become a Totally Closed MAS. Although, it is tied and limited by the existing frameworks, middleware, programming languages, and platforms. They must provide constructions to allow agents to move, communicate, and connect to heterogeneous hardware devices. In this work, we present a middleware that provides support to prevent the Embedded MAS from becoming Totally Closed and Limited Open MAS.

3 Related work

The definition of Embedded MAS considers a whole MAS with several agents cooperatively controlling the hardware parts to provide autonomy to the device where the MAS is hosted or embedded. During the years, several works employed Embodied and Embedded MAS, where BDI agents are only capable of communicating to agents of their system (Castro et al. 2018; Heijmeijer and Alves 2018; Sakurada et al. 2019; Manoel et al. 2020). As Totally Closed MAS, these works are ad-hoc solutions built up for a dedicated purpose, and interactions with other embedded solutions are discarded. For these systems to become a Closed or Limited Open MAS, they need to add mechanisms at design or runtime to allow some communication or mobility; on the other hand, they would need both to become Open MAS. Castro et al. (2018) and Heijmeijer and Alves (2018) provided virtual solutions using JaCaMo that do not use hardware devices but could be extended to embedded solutions. Sakurada et al. (2019) and Manoel et al. (2020) use physical devices to implement embedded systems by adopting Java Agent DEvelopment Framework (JADE) (Bellifemine et al. 2007) and JaCaMo, respectively, on top of single-board computers. In these works, agents from one Embedded MAS accessing hardware resources can communicate with other agents from different Embedded MAS using RF if there are no available nearby networks. So, our solution avoids ad-hoc agents and MAS. Besides, all works above depend on the network structure and do not present alternatives for the MAS. By adopting our proposed middleware, any embedded system could adopt any communication infrastructure and use RF communication as an alternative infrastructure. In this way, the Embedded MAS will only become Totally Closed if it loses both communication infrastructures.

From this point on, works provided Closed MAS solutions dependable on external communication infrastructures. Pantoja et al. (2020) extended the notion of Things from IoT to Smart Things using a BDI agent and considered an architecture supported by an IoT middleware running at a centralized server. Moreover, Silva et al. (2020) and Michaloski et al. (2022) extended Jason and Gwendolen, respectively, to use Robot Operating System (ROS) as an interface to hardware components. Michaloski et al. (2022) also uses the Canonical Robot Command Language and Gazebo in robot planning. As these works employ ROS, they could be extended at design time to allow mobility and communication simultaneously since ROS is a robust middleware. Despite becoming Open, they will still be dependent on the network (local or internet). Our work allows us to employ an alternative channel to avoid this dependence on centralized architectures. In this channel, the Embedded MAS can cooperate, exchanging messages with nearby devices. We propose to avoid the Embedded MAS becoming Totally Close until it can find another available network or restore the original one.

The same occurs in the solutions proposed by Jesus et al. (2021), Brandão et al. (2021) and de Castro et al. (2022). These works allow the designer to create an Embedded MAS able to move to another Embedded MAS and to communicate using different mechanisms. But in the end, they rely on middleware and networks as well. Jesus et al. (2021) proposed bio-inspired protocols for transferring agents and their mental state to another hardware at runtime. Brandão et al. (2021) proposed edge-computing approaches considering BDI agents and artifacts running on top of devices. Both used JaCaMo and ContextNet (Endler et al. 2011).

Moreover, some works in the literature do not depend on network infrastructure, using a decentralized strategy, such as Wireless Sensor Networks (WSN). These works also explore energy efficiency to reduce the energy consumption of network nodes (each node operates only via in-built batteries) and routing network data traffic to find optimal routes (with the shortest path on the network). These characteristics are addressed in works that improve the resilience of a network. However, even though these works are fault-tolerant, they do not present agent mobility (characterizing them as Closed). Finally, they do not employ MAS, BDI agents, and Embedded MAS (Al-Otaibi et al. (2021); Mansour et al. (2022)).

Considering applications with agents and decentralized network strategies, the Ortiz et al. (2022) paper presents an architecture focused on IoT using different layers such as edge, fog, and cloud together to allow the management of resources. The authors provided a MAS application allowing agents to communicate via IoT using the JADE. However, as it does not allow the mobility of agents, the MAS is Limited Open. In the Machine Learning (ML) literature, several works focus on data traffic on the network, which allows the identification of fraudulent transactions et al. (2022) and tools to support the development of new solutions (Khalajzadeh et al. (2020)). However, these works do not have all mechanisms to allow the development of an Open Embedded MAS (with agents able to communicate and move to other MAS).

The proposed middleware fills a gap left by previous works in supporting the development of Embedded MAS considering the new classifications (Totally Closed, Closed, Open Limited, and Open). It allows the communication and mobility of agents between different MAS without the dependence on network infrastructure. Table 2 is shown the related works and their classification according to the communicability and openness of a MAS. Besides, it also highlights those dependent on any communication or mobility infrastructure and if it has an alternative mechanism to avoid Totally Closed MAS.

The recent solutions using Embedded MAS found in the literature are Closed or Open MAS. It is not trivial to develop Embedded MAS since various hardware, software, and middleware technologies need to be integrated to provide a functional platform for controlling devices autonomously. It gets more complicated when applying the BDI cognitive model once its reasoning cycle's processing is costly in several programming languages and frameworks (Stabile Jr. and Sichman 2015; Stabile Jr. et al. 2018). Then, it is straightforward to develop Totally Closed systems, and efforts have been made to overcome this issue. However, even systems there are Closed or Open (with some mobility or communication) can become Totally Closed at runtime. Our solution aims to overcome this issue by offering an alternative communication channel to the system's designer to create Embedded MAS.

Table 2 Comparison between the related work

Work	Classification	Dependency	Alternative
Castro et al. (2018)	Totally closed	–	–
Heijmeijer and Alves (2018)	Totally closed	–	–
Sakurada et al. (2019)	Totally closed	–	–
Manoel et al. (2020)	Totally closed	–	–
Pantoja et al. (2020)	Closed	✓	–
Silva et al. (2020)	Closed	✓	–
Khalajzadeh et al. (2020)	Closed	✓	✓
Jesus et al. (2021)	Open	✓	–
Brandão et al. (2021)	Open	✓	–
Al-Otaibi et al. (2021)	Closed	✓	✓
Mansour et al. (2022)	Closed	✓	✓
Ortiz et al. (2022)	Closed	✓	✓
Aziz et al. (2022)	Closed	✓	✓
de Castro et al. (2022)	Open	✓	–
Michaloski et al. (2022)	Closed	✓	–
Our approach	Open	✓	✓

4 The Javino middleware for Embedded MAS

Communication is essential in devices with embedded systems (Mundhenk et al. 2022), including MAS, as it allows data obtained from sensors' readings and internal conclusions to be forwarded to other devices. Otherwise, even if endowed with cognitive and autonomous capacity, any embedded system would depend only on its observations and conclusions for decision-making and learning. Besides, some communication channels depend on centralized infrastructure (Endler et al. 2011), which leads to a technological dependency. Totally Closed or Limited Open MAS can lead Embedded MAS to be disconnected from other sources of information and unable to share information with other devices. It could be solved by adopting dedicated hardware for communicating, such as radiofrequency devices, as an alternative communication channel when other channels are unavailable. It could also be a primary communication channel if all devices operate in a short range.

The middleware presented in this paper allows devices running Embedded MAS to communicate using RF as an alternative to avoid the embedded system from becoming a Totally Closed MAS. The middleware allows Embedded MAS to communicate with others Embedded MAS using a protocol to exchange messages between individuals or groups. As the air is the communication channel, information is still exchanged in clear messages. Then, the protocol provides encryption techniques to guarantee the privacy of individual and group messages by sending *unicast* and *multicast* messages. Thus, an Embedded MAS can send information individually to another Embedded MAS, to a group, or to everyone within the transmission range (*broadcast*), using one of the three sending modes:

- *Unicast*: Message sent from one device hosting an Embedded MAS to another specific device within the transmission range.
- *Multicast*: Message sent from one device hosting an Embedded MAS to an addressed group of devices within the transmission range.

- *Broadcast*: Message sent from one device hosting an Embedded MAS to all devices within the transmission range.

Therefore, an Embedded MAS is responsible for sensing an environment using a set of sensors, processing, and analyzing the captured data. After deliberation, it can act in the same environment or send messages to nearby devices. The Embedded MAS can send *unicast* messages directly to other devices or all nearby devices by broadcasting a message. Sometimes, devices could be organized in groups, and messages can be sent to a specific group instead of a device. In this case, all devices of the addressed group. For example, in the early scenario of autonomous vehicles running Embedded MAS, a damaged vehicle sends a *broadcast* message to all nearby autonomous vehicles. Any vehicle that receives the message could redirect it to a Support Point or a public safety vehicle. Once the accident is identified, the public safety vehicles can cooperate to help the damaged vehicle by exchanging *multicast* messages privately since they are organized into a group.

The middleware—named Javino—is a double-side library for exchanging serial messages between microcontrollers and high-level programming languages built over a secure end-to-end protocol, which addresses standalone and groups of devices. JaCaMo agents, based on a customized agent architecture to diffuse messages, access the Javino middleware to send and receive messages from another JaCaMo Embedded MAS. This *Diffuser* agent has a modified reasoning cycle to access the microcontroller to retrieve sensors' data as perceptions and messages and send actions to actuators and messages to other Embedded MAS. Every device with Embedded MAS has a unique identification stored in the microcontroller and *Diffuser* agent. This identification is part of the protocol and identifies the device and its group.

In this work, a physical architecture is adopted to allow the interconnection of hardware components so that it is possible to enable a flow of direct perceptions from sensors and messages from RF devices to agents in an Embedded MAS. In the opposite direction, the physical architecture allows agents to send actions performed by actuators and messages to other Embedded MAS. Physically, the architecture follows a traditional well-established architecture (Matarić 2007) using heterogeneous hardware devices and microcontrollers, serial communication, and a computer-based board capable of hosting an operational system and an Embedded MAS. This physical architecture is not an innovation and has been described and adopted in many works employing Embedded MAS (Barros et al. 2014; Pantoja et al. 2016b; Manoel et al. 2017; Pantoja et al. 2020; Jesus et al. 2021; de Castro et al. 2022).

Logically, agents from the Embedded MAS are in charge of all physical components of their device. The Embedded MAS is responsible for reasoning considering all gathered perceptions, messages exchanged, and internal conclusions. Agents can perform actions and send messages once a deliberation exists that should be forwarded to actuators or communicated. The decision to perform an action or send a message is part of an agent's reasoning, which forwards serial messages to the firmware using wired serial communication and the Javino. Since an Embedded MAS controls the local physical components, wired connections bring reliability and simplicity since there is no competition to access the hardware. Afterward, the microcontroller's firmware redirects the messages received to the RF responsible for broadcasting and all actions to be performed by actuators. The firmware is also responsible for the functions that read sensors' information, and RF messages received, mount them as perceptions, and send them to agents using serial communication. The architecture can be seen in Figure 2. It has four logical layers:

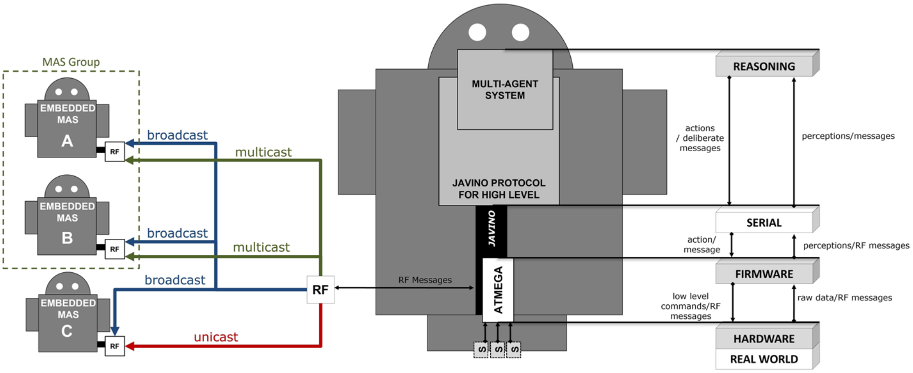

Fig. 2 The Embedded MAS architecture overview

1. *Reasoning*: It is the layer responsible for the device's cognition. An Embedded MAS interfaces the device's sensors and actuators using a serial interface, receiving perceptions or messages from other devices, deliberating and responding with actions to be performed on the actuators or messages to other devices. The *Diffuser* agent acts in this layer by accessing the high-level programming library of Javino middleware.
2. *Serial*: The layer interfaces the data traveling between the Embedded MAS and the adopted hardware. Serial communication complies with an information exchange protocol (Subsection 4.1) that guarantees messages exchange between individuals and groups and sensors' readings.
3. *Firmware*: The program is embedded in the microcontroller, where sensors and actuators' behaviors are programmed. Sensors' data become perceptions, and RF messages become agents' messages. They are sent to the Embedded MAS using the low-level library of Javino middleware when agents request. Moreover, agents can send actions to control actuators, and messages are sent to another device according to the agent's goals.
4. *Hardware*: These are all sensors and actuators available on the device and connected to the chosen microcontroller. These sensors and actuators produce the information that will be transformed into perceptions and receive messages sent to the Embedded MAS and other devices. This layer includes the RF sensor for communication with other devices.

The reasoning layer was extended by implementing *Diffuser* agents for managing messages via RF. Moreover, at the serial layer, the *Javino* middleware was extended to capture messages via RF with a new communication protocol that allows addressing messages to a specific (*unicast*), a group (*multicast*), or any receiver within the range of the RF emitter (*broadcast*).

The process for sending RF messages starts with the *Diffuser* agent capable of sending a text message to Javino (Serial layer). It encapsulates the text message by inserting fields to identify the sender, receiver, and message size (number of characters). After the message is encapsulated, the *Javino* sends them to the microcontroller (*Firmware* layers). By *Javino's* fields of the encapsulated message, the microcontroller identifies the type of message (*unicast*, *multicast*, or *broadcast*) and, based on that, encrypts or not the message. In case of the message is *unicast* or *multicast*, it is encrypted; otherwise, it is a *broadcast* type message, so it is not encrypted to travel in plain text for any RF receiver.

The receiver listens to the public channel using the RF receptor and, when capturing a message, forwards them to the microcontroller (*Firmware* layer). In the firmware layer, the message is decrypted, becoming an encapsulated message, and sent to *Javino*. In *Javino*, the encapsulated message fields are confirmed, and content integrity verification is performed based on the message size field. If all verifications are performed successfully, the text content of the encapsulated message is sent to the *Diffuser* Agent. Finally, the *Diffuser* agent transforms the text message into a belief, perception, or intention. In Figure 3, it is possible to see these procedures for sending and receiving messages via RF between two Embedded MAS.

We named the middleware Javino since its initial purpose was to exchange messages between the Java programming language and Arduino boards. Despite the name, it can be used with other microcontrollers (Guinelli and Pantoja 2016) and programming languages since it relies on a message-exchanging protocol. It showed reliability in supporting the development of Totally Closed and Limited Open MAS for Embedded MAS in previous versions (Lazarin and Pantoja 2015; Junger et al. 2016) and some agent architectures (Pantoja et al. 2016a, 2018; Manoel et al. 2020). But the Limited Open solutions depended on IoT infrastructures, and the Totally Closed solutions were too restrictive considering cooperative applications. It is important to remark that the Javino middleware works for any micro-controlled solution even though it does not use Embedded MAS.

4.1 End-to-end communication protocol

When one device sends a message to the other device, the *Diffuser* agent from the Embedded MAS accesses the serial interface using Javino, which redirects to the microcontroller the message with information from the sender, addressee, the illocutionary

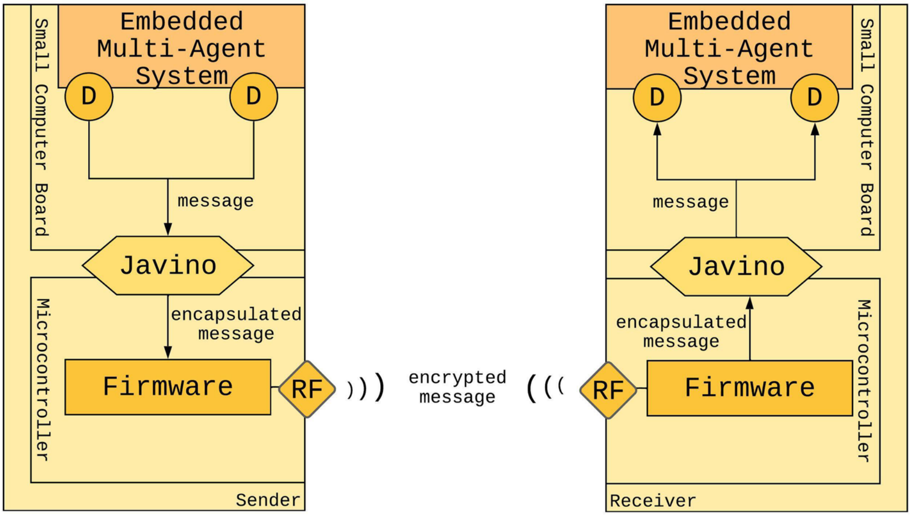

Fig. 3 Physical and logical architecture for an Embedded MAS considering the proposed protocol

force, and the message's content. At the microcontroller low-level, the Javino receives and diffuses the message through RF. Once the target device receives a message addressed to it, the message is handled by Javino and sent to a MAS agent using the serial port. The agent will be in charge of managing the message once it arrives at the Embedded MAS. Every RF message is executed in two steps: one serial communication from the Embedded MAS to the microcontroller (serial communication) and a diffusion message using the RF (air). The illocutionary forces are KQML performatives used by agents in communication. It represents the context and intention in a message exchange between agents.

The RF listens to the channel in the microcontroller until it receives a message and buffers it. Then, the Javino will evaluate the header of the message to verify if it is a valid message based on the expected format by the protocol. The message is evaluated by comparing the value in the size field, informed in the header and the received message size. Moreover, it identifies the message type, the destination address, and the addressed group. Messages of the type: *broadcast* are always accepted; the *multicast* messages are destined for the device's group, and; the *unicast* message is destined for the device's identification. If something goes wrong during the verification process, the message is discarded. Otherwise, the message is received and buffered until the *Diffuser* agent request it. The process can be seen in Fig. 4.

The protocol uses 24-bit addressing where the 12 most significant bits identify the group, and the 12 least significant bits represent the group member. The address uses the 64-Base Alphabet, so characters from A to Z—uppercase and lowercase—and + and / can be used. Up to 16,252,928 devices can be identified in 3968 groups of up to 4096 members each, starting in *AAAA* until *9///*. The special addresses *//AA* to *///+* are used for *multicast* messages, and a *multicast* address starts with *//* followed by two characters that represent the identification of the group. Besides, the special address *////* indicates *broadcast* communication, and the ++++ special address indicates serial communication from the microcontroller to high-level programming languages. The message's header has three fields: a 24-bit representing the device addressee, a 24-bit representing the device sender, and the message Size in bits using 12 bits. Figure 5 shows a

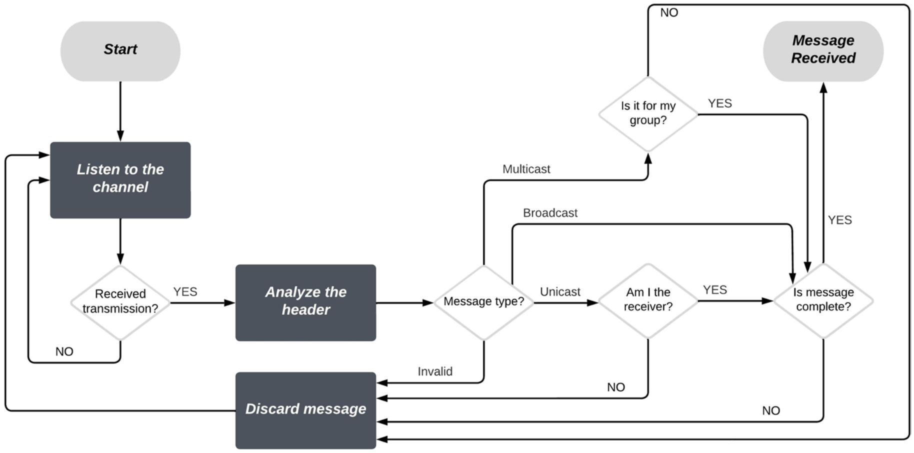

Fig. 4 Message reception on the broadcast medium

Table 3 The address range of the protocol

Type	From	To	Description
Addressable	A A A A	9 / / /	3968 groups [AA até 9/] 4096 member per group
Reserved	+ A A A	+ + + 9	For future use
Reserved	+ + + /	/ + / /	For future use
Multicast	/ / A A	/ / / +	Format //[group]
Special	+ + + +		Serial communication with hardware accessing
Special	/ / / /		*broadcast* communication

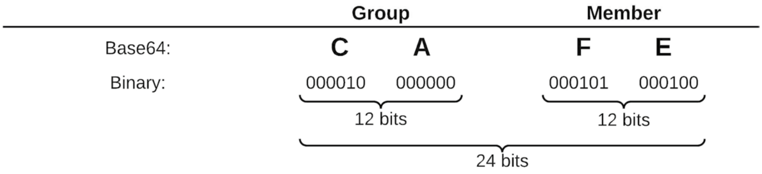

Fig. 5 The group and member addressing

Destination	Origin	Size	Message
////	CAFE	Aw	Hello!
24 bits	24 bits	12 bits	up to 4095 bits

Fig. 6 The addressing

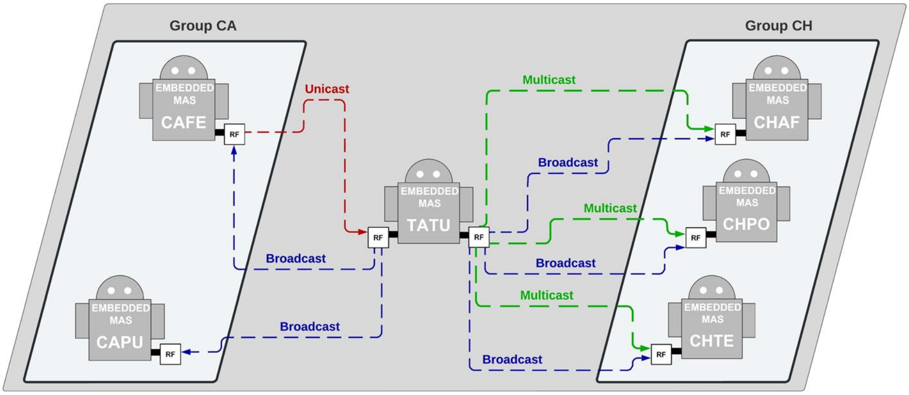

Fig. 7 General communication

broadcast message originated by the member *FE* of the *CA* group. The address format is shown in Fig. 6, and the addressing range is shown in Table 3.

Figure 7 shows five Embedded MAS that implements the Javino middleware with RF communication divided into three groups: CA group, with FE and PU members; CH group with AF, PO, and TE members; finally, the TA group, with only one member, the TU. In addition, CH members can be observed performing a *multicast* communication. The MAS CAFE sends a *unicast* message to the MAS TATU, and the MAS TATU sends a *broadcast* message to all MAS.

The special address for serial communication is important when considering Embedded MAS or Embedded Systems. First, serial communication is one way to transfer sensor data to a system and receive actions. Then, when using an Embedded MAS, the data can go direct to agents. In this paper, the serial communication is wired between the microcontrollers and the board that hosts the Embedded MAS. So, serial communication is dedicated to exchanging messages or data and actions transferring between hardware and embedded system. It avoids conflicts in accessing the serial ports and unauthorized access to the hardware.

The number of possible devices the protocol addresses is enough for small and medium applications in any domain. In domains that require a vast amount of intelligent devices in a wide-area cooperating, such as smart cities, 16 million could not be enough depending on the size of the city and devices, the number of groups necessary, and how many devices are in each group. However, we assert that, even in large applications, the protocol could assist in specific services. Smart cities are composed of heterogeneous solutions integrated to manage a city's resources. Then, Javino middleware and Embedded MAS could be used as one of these many possible solutions.

Using RF communication to exchange information between MAS presents some challenges; security is one of the most important. As the network is open, without infrastructure, and accessible to any member within the radio range, interference in this network can easily be carried out by malicious users. Therefore, making communication secure and protected by encryption is mandatory. To encrypt the information, we chose to use Advanced Encryption Standard (AES), as it presents suitable hardware and software performance on a wide variety of computing platforms and can be executed in environments with energy and processing restrictions, such as the Arduino platform (Guinelli et al. 2018). Encryption was implemented in the firmware layer of the embedded MAS, enabling secrecy in exchanging messages between MAS. For this, we used the library *Security-Vanets* (Freitas et al. 2021) as a base, a fork of the *Securino* library that implements the protocol presented in Lazarin et al. (2021), enabling *broadcast* communication via RF transmission for Arduino platforms. In this work, we carried out an expansion of the library that now allows the exchange of *broadcast* messages in clear, the exchange of *multicast* and *unicast* messages encrypted in Cipher Feedback Mode (CFB) with random initialization vector.

4.2 Diffuser agent architecture

The Javino middleware allows one Embedded MAS to exchange messages with another Embedded MAS using the RF addressing which device is the sender and the addressee. But, in practice, the Embedded MAS cannot send a message to another since it is a collective of agents managing physical resources. So, one agent from the Embedded MAS must become responsible for interfacing RF messages. This agent will store the identification of the protocol and every message diffused to the device the agent will handle. For this, this *Diffuser* agent must act and comply with the message format of the framework adopted, the JaCaMo. For an agent to send a message using JaCaMo, three mandatory fields must be informed: the receiver name, the illocutionary force, and the message itself. The *Diffuser* agent will adopt the same format and fields, but instead of the agent name, it will address the device identification provided by Javino's protocol.

Thus, we established that an Embedded MAS has only one *Diffuser* Agent responsible for handling messages coming from RF. This agent may have any name but holds the device identification. In this first moment, only the illocutionary forces *tell*, *untell*, *achieve*, and *unachieve* will be considered because they have a smaller message size than *tellHow* and *askHow*—used to send an entire plan with its actions—given the limit of the message field of the protocol. The *tell* and *untell* forces inform or ask to remove a belief from one Embedded MAS, and the *achieve* and *unachieve* inform an intention to be followed or unfollowed.

The *Diffuser* agent sends a message using the internal action named *Diffuser*, explicitly created to send messages via diffusion using the proposed protocol. For this, we present a customized agent architecture for JaCaMo where *Diffuser* agents can interface hardware resources by accessing the serial port to request perceptions and diffused messages using Javino middleware to send actuators' actions and diffuse messages. As BDI agents have a well-defined reasoning cycle (Bordini et al. 2007), it is not possible for the microcontroller to directly send or receive messages to the Embedded MAS without proper synchronization. The agent may have been deliberating an intention and lost the messages sent. Thus, we extended the *Diffuser* agent's reasoning cycle (Fig. 8), adding the Javino middleware before two steps at the beginning of the cycle: the *perceive*, responsible for gathering perceptions from an internal (endogenous) environment, and *checkMail*, which verifies if the agent received any messages from other agents of its MAS. The Javino replaces the endogenous with an exogenous environment by allowing perceptions to come from real sensors. Messages can now come from other agents of its Embedded MAS and other Embedded MAS by diffusion.

Moreover, we also added the Javino middleware at the end of the reasoning cycle after the *act* and *sendMsg* steps. The *act* is the execution of an action selected from a plan, and the *sendMsg* is a particular action that sends internal messages to other agents. Then, the Javino middleware allows actions to be forwarded to actuators and messages to be diffused to the RF. Adding Javino at both the beginning and end of the reasoning cycle synchronizes the data gathering avoiding the loss of messages during the serial communication. Besides, the microcontroller is set as slave hardware where the agent decides when to get data. For instance, agents get perceptions and messages once per cycle. But actions and messages are sent when they deliberate for so.

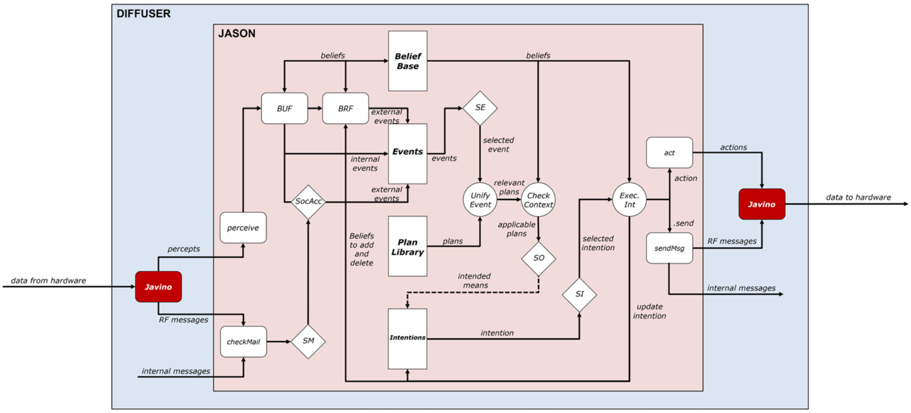

Fig. 8 The reasoning cycle of a *Diffuser* Agent

Every time an agent performs a perception or message request using Javino in the BDI cycle, it sends to the microcontroller a serial message using the address ++++, which answers back forwarding perceptions or messages available. Moreover, whenever an agent sends a message addressed to an Embedded MAS, it encodes the message content before forwarding it to the microcontroller. At the low level, the message is diffused using the receiver address identified by the sender agent. When the target device receives the message, it stores it in a temporary buffer until the *Diffuser* agent from its Embedded MAS requests the received messages.

To facilitate the programming of *Diffuser* agents using JaCaMO, we created the following internal actions:

– setMyRFId(CAFE) allows the agent to set their identification address for RF communication.
– getRFMessages(open|block) allows the agent open and block the RF communication channel.
– diffuse(recipient, force, message) allows the agent to diffuse messages with others Embedded MAS using RF communication.

For instance, the Embedded MAS must have one *Diffuser* agent so that the message addressing is centralized in only one entity. The existence of more than one *Diffuser* agent does not restrain the Embedded MAS operation. However, a competition for the acquisition of messages will be generated, obliging the Embedded MAS designer to reproduce the plans on all *Diffuser* agents or communicate if a message is not the responsibility of whoever captured it. Furthermore, it is recommended that each Embedded MAS adopt at least one dedicated microcontroller for RF communication to act separately as a support if the communication infrastructure or the main microcontroller fails. Again, using a *Diffuser* agent to manage other actuators and sensors in addition to RF does not interfere with capturing perceptions or messages, as these occur in different steps of the reasoning cycle.

5 Proof of concept

To demonstrate the middleware's functioning, we present a case study where a prototype embedded with a MAS does not become Totally Closed even if it does not have access to communications infrastructures. The case study considers the previous highway scenario where Embedded MAS controls civil and public safety vehicles without access to any communication infrastructure, except for the RF communication in each vehicle. In case of an accident, the car that suffers the accident sends a message (*broadcast*) to the other vehicles in proximity, asking for help. In addition, it can diffuse the message to public safety vehicles, which will be able to communicate with individuals from the same group (*unicast* and *multicast*) to provide the necessary assistance to the damaged vehicle. The scenario starts when a civilian vehicle (bus) suffers an accident and sends a message requesting help (*broadcast*). A police vehicle close to the scene receives the message and diffuses it to another member of the public safety group (*multicast*) named the support station. The support station diffuses the message to a hospital, and the hospital sends an ambulance to the place of the accident. Finally, the support station receives the message informing that the ambulance is on its way (*unicast*) and diffuses it to the police vehicle, which diffuses a message to the damaged vehicle.

5.1 Case study

As the Embedded MAS employs AgentSpeak (Rao 1996) interpreted by Jason framework to develop cognitive agents, each agent expected behavior must be codified in plans and messages that could activate these plans. Table 4 summarizes all expected behaviors from vehicles and their respective roles in the scenario. Each behavior can reflect a message sent or start with a triggering message.

For example, the damaged vehicle could be identified as *ROMA*, where *MA* is his identification in group *RO*. The public safety vehicles could be composed of the *Police*, *Support*, *Hospital*, and *Ambulance*. The *Police* vehicle could be identified as *SUPO*, the *Support* as *SUPP*, the *Hospital* as *SUHO*, and the *Ambulance* as *SUAM*. All vehicles are in the same group, the *SU*. When the damaged vehicle sends a broadcast message, it does not need to address a specific individual or group. It can send to the broadcast address (*////*). The public safety vehicles can send a multicast message to all group members by addressing just the

Table 4 The expected behaviors from the vehicles

Vehicles	Expected behavior	Triggering message
Bus	Requests help.	Send *broadcast*: +help
	Waits for Ambulance rescue.	Wait *unicast* from Police: +helpComing
Police	Receives the help request from the Bus and diffuses to Support the help request.	Wait *broadcast* from Bus: +help
	Receives the help request from the Bus and diffuses to Support the help request.	Send *multicast*: +someoneNeedHelp
	Receives that Ambulance is coming and diffuses to Bus that Ambulance is coming.	Wait *unicast* from Support: +helpComing
	Receives that Ambulance is coming and diffuses to Bus that Ambulance is coming.	Send *unicast* to Bus: +helpComing
Support	Receives the help request from Police and diffuses to Hospital the help request.	Wait *multicast* from Police: +someoneNeedHelp
	Receives the help request from Police and diffuses to Hospital the help request.	Send *unicast* to Hospital +someoneNeedHelp
	Receives that the Ambulance is going and diffuses to Police that Ambulance is going.	Wait *unicast* from Hospital: +helpComing
	Receives that the Ambulance is going and diffuses to Police that Ambulance is going.	Send *unicast* to Police: +helpComing
Hospital	Receives the help request from Support and diffuses to Ambulance the help request.	Wait *unicast* from Support: +someoneNeedHelp
	Receives the help request from Support and diffuses to Ambulance the help request.	Send *unicast* to Ambulance: +someoneNeedHelp
	Receives that Ambulance is going and diffuses to Support that Ambulance is going.	Wait *unicast* from Ambulance: +ambulanceIsGoing
	Receives that Ambulance is going and diffuses to Support that Ambulance is going.	Send *unicast* to Support: +helpComing
Ambulance	Receives help request from Hospital and diffuses to Hospital that Ambulance is going.	Wait *unicast* from Hospital: +someoneNeedHelp
	Receives help request from Hospital and diffuses to Hospital Ambulance is going.	Send *unicast* to Hospital: +ambulanceIsGoing

target group by canceling the first two characters of the identification. In this case, a message should address the identification *SU//*. The message should address all four characters for personal communication (unicast message). To send a message to the damaged vehicle, one should use *ROMA*. Figure 9 shows the complete scenario of our approach using Javino for the embedded MAS.

5.2 Formalization and formal verification

As previously mentioned, we need to use formal techniques to verify whether or not our model works as designed. In this section, we present the formalization model together with its corresponding simulation and formal verification. For that, we use the UPPAAL model checker Bengtsson et al. (1996). With this tool, we can use timed automata to model, represent and verify the expected behavior of all elements in our system. Mainly, we can formally verify the different communication methods between the senders and receivers, i.e. unicast, multicast or broadcast. Besides, we shall guarantee that our system respects the necessary constraints during the communication process. For example, a given multicast sender (when sending a multicast message) expects its message to be received only by a receiver that belongs to the same group as the (multicast) sender.

5.2.1 Building the model

Mainly, the construction of our model has three different senders and receivers, that is, a specific sender and receiver for each kind of communication: unicast, multicast and

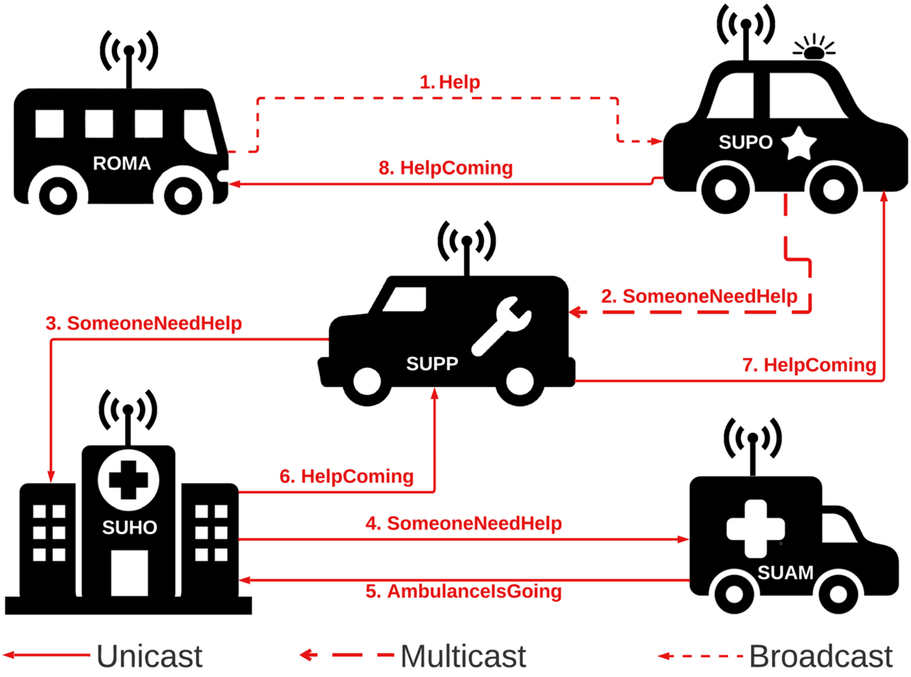

Fig. 9 The complete scenario of our case study

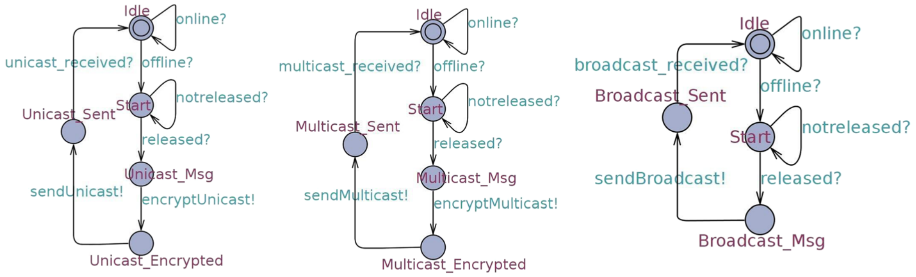

Fig. 10 Unicast, multicast and broadcast senders

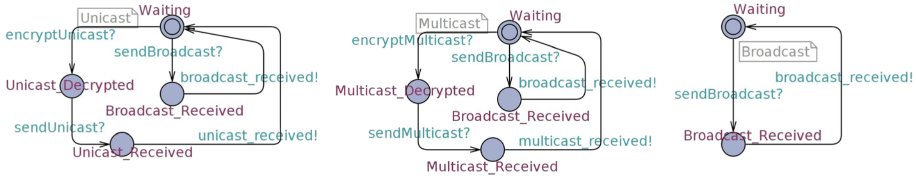

Fig. 11 Unicast, multicast and broadcast receivers

broadcast. Besides, we include into the network of automata, two simple models that are used to abstract the connection status, which can be *online* or *offline*; and the communication channel that can be *released* or *not released*. Notice that to start a communication process a given sender should check for two requirements: **i.** is the connection *offline*? and **ii.** is the communication channel *released*?

We use the UPPAAL model checker since we can easily define communication channels in our model using the so-called synchronization channels. For instance, a channel named `sendUnicast!` will synchronize with a channel named `sendUnicast?`, where the former is the sender while the latter is the receiver.

At the left of Fig. 10, we show the automaton for the `Sender-Unicast`. As soon as it is done the verification for connection status and communication channel, the sender encrypts the message (using the synchronization channel named `encryptUnicast!`) and sends the message (using the synchronization channel `sendUnicast!`). For verification purposes, we added an acknowledge message by using the channel `unicast_received?`, which is sent back to the sender by the corresponding receiver `Receiver-Unicast`.

Next, in the middle of Fig. 10 we present the automaton for the `Sender-Multicast`. The overall workflow is the same as the Sender-Unicast; the only difference is the use of a multicast type of message instead of a unicast one. So, the automaton uses the following three synchronization channels: `encryptMulticast!`, `sendMulticast!`, and `multicast_received?`.

At the right of Fig. 10 it is presented the automaton for the `Sender-Broadcast`; this model is quite simple because it does not require encrypting the message. Thus, the model presents only two synchronization channels: `sendBroadcast!` and `broadcast_received?`.

Now, we describe the corresponding receivers. As seen at the left of Fig. 11, the `Receiver-Unicast` can receive two kinds of messages: i. an encrypted unicast (using synchronized channel `encryptUnicast?`); or ii. a broadcast message (using synchronized channel `sendBroadcast?`). If the former message is used, it should be decrypted (using channel `encryptUnicast?`) and next it is properly received via channel `sendUnicast?` to conclude an acknowledge is sent to `Sender-Unicast` using channel `unicast_received!`. In case of a broadcast message, it is directly received using channel `sendBroadcast?` and an acknowledge is sent to the `Sender-Broadcast` using the channel `broadcast_received!`.

In the middle of Fig. 11 we illustrate the `Receiver-Multicast` which is quite similar to the `Receiver-Unicast`; the only difference is that it should receive a multicast message instead of a unicast message. Notice that `Receiver-Multicast` also can receive a broadcast message using the same channels previously described.

At the right of Fig. 11, we show the `Receiver-Broadcast` which is responsible to solely receive broadcast messages via channel `sendBroadcast?` and acknowledge the `Sender-Broadcast` by using the channel `broadcast_received!`.

5.2.2 Simulation scenario

Having built our model as a network of automata, now we explore the UPPAAL model checker to create a simulation scenario for a given case study.

For that, we define seven instances from the previously automaton (seen in 5.2.1) as follows:

1. `Police_Sender` as an instance from `Sender-Unicast`.
2. `Support_Service_Sender` as an instance from `Sender-Multicast`.
3. `Bus_Sender` as an instance from `Sender-Broadcast`.
4. `Support_Service_Receiver` as an instance from `Receiver-Unicast`.
5. `Ambulance_Receiver` as an instance from `Receiver-Multicast`.
6. `Hospital_Receiver` as an instance from `Receiver-Multicast`.
7. `Police_Receiver` as an instance from `Receiver-Broadcast`.

Figure 12 shows all the seven elements aforementioned plus two models which respectively represent the connection and communication status. The scenario simulation enables us to check how all different sorts of senders and receivers can properly work together and identify whether any change is necessary to be applied in our model.

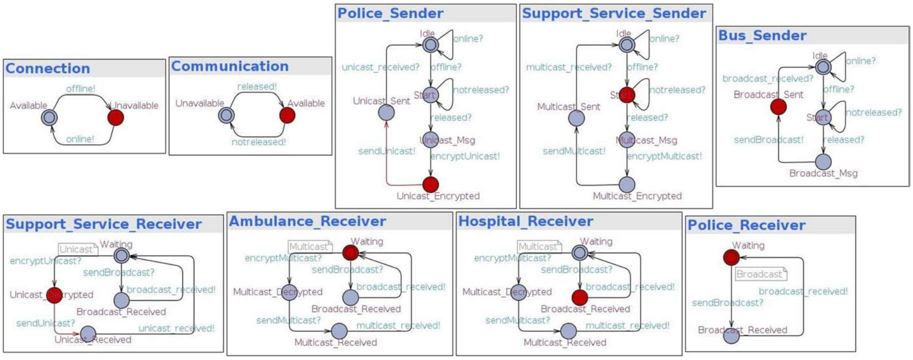

Fig. 12 Simulation Scenario

Notice that by using Model Checking, this simulation runs all possible instance for our model, enabling the verification of problems in the model. Besides, the diagram (see Fig. 13) illustrates that the exchange of messages occurs through the synchronization channels. Specifically, the image shows the exchange of messages between the Police Sender and the Support Service Receiver.

5.2.3 Verification of properties

Now, we specify a set of formal properties using temporal logic. With these properties, we can formally verify the behavior of our model and check whether or not the model respects the constraints related to the communication between senders and receivers. Before presenting the set of properties, we briefly introduce the temporal logic operators used in UPPAAL verifier.

5.2.4 Brief explanation of UPPAAL operators

Baier and Katoen (2008) mention that Timed Computation Tree Logic (TCTL) is a real-time variant of temporal logic used to express properties of timed automata. The UPPAAL Model Checker uses a simplified version of TCTL to specify verification properties. The syntax used in UPPAAL is provided in Table 5.

We have verified eight properties, and they were successfully verified. Nonetheless, the last two properties fail because we intend to show some of the restrictions that our model should respect.

1. `A[] not deadlock`
   description: this property verifies if the model has no deadlock.
2. `A[] (Bus_Sender.Broadcast_Sent) imply ((Police_Receiver.Broadcast_Received)`

|| (Police_Receiver.Waiting) || (Hospital_Receiver.Broadcast_Received)
|| (Hospita_Receiver.Waiting) || (Ambulance_Receiver.Broadcast_Received)

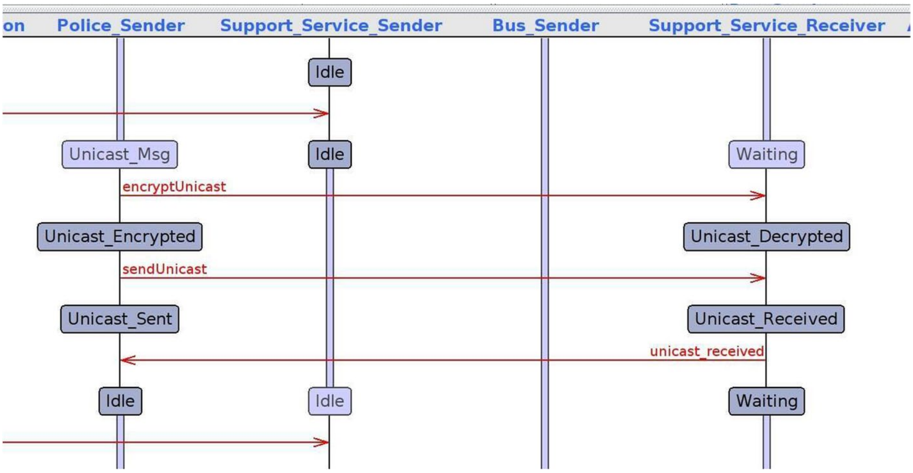

Fig. 13 Message exchange diagram

Table 5 UPPAAL syntax for TCTL

Operator	Meaning
&&	And
||	Or
==	Equivalence
imply	Conditional
not	Negation
A	Universal quantifier
E	Existential quantifier
[ ]	Always
<>	Eventually
-->	Leads to

|| (Ambulance_Receiver.Waiting) || (Support_Service_Receiver.Broadcast_Received)
|| (Support_Service_Receiver.Waiting))

description: it is always the case that when the Bus Sender sends a broadcast message, it will be received by any of the Receivers: Police, Hospital, Ambulance or Support Service.

3. A[] (Support_Service_Sender.Multicast_Sent) imply
((Hospital_Receiver.Multicast_Received) || (Hospital_Receiver.Waiting)
|| (Ambulance_Receiver.Multicast_Received) ||
(Ambulance_Receiver.Waiting))

description: it is always the case that when the Support Service sends a multicast message; it will only be received by the Hospital or Ambulance.

4. A[] (Support_Service_Sender.Multicast_Encrypted) imply
((Hospital_Receiver.Multicast_Decrypted) ||
(Ambulance_Receiver.Multicast_Decrypted))

description: it is always the case that when the Support Service sends an encrypted multicast message; only the Hospital or Ambulance will decrypt.

5. A[] (Police_Sender.Unicast_Sent) imply
((Support_Service_Receiver.Unicast_Received) ||
(Support_Service_Receiver.Waiting))

description: it is always the case that when the Police send a unicast message; it will only be received by the Support Service.

6. A[] (Police_Sender.Unicast_Encrypted) imply
(Support_Service_Receiver.Unicast_Decrypted)

description: it is always the case that when the Police send an encrypted unicast message; it will only be decrypted by the Support Service.

7. A[] (Police_Sender.Unicast_Sent) imply
((Hospital_Receiver.Multicast_Received) ||
(Ambulance_Receiver.Multicast_Received))

description: it is always the case that when the Police send an unicast message; it will only be received by the Hospital or Ambulance.

Remark: this verification fails because a multicast receiver (in this case, the Hospital or Ambulance) can not receive a message sent by a unicast.

8. A[] (Support_Service_Sender.Multicast_Sent) imply
((Police_Receiver.Broadcast_Received))

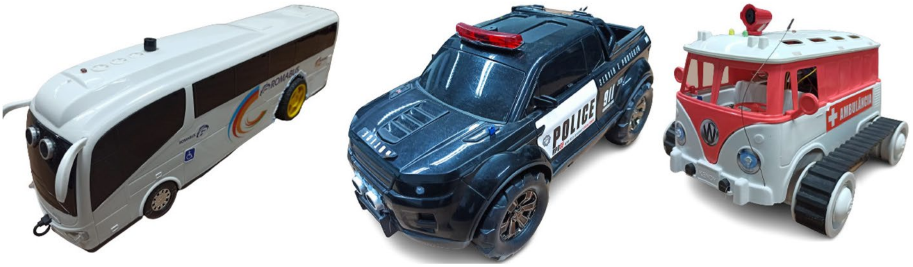

Fig. 14 The vehicles prototype

description: it is always the case that when the Support Service sends a multicast message; it will only be received by the Police.

Remark: this verification fails because a broadcast receiver (in this case, the Police) can not receive a message sent by a multicast.

5.3 The prototypes

The vehicles' prototypes are physically composed of a mini-computer (*Raspberry Pi zero*) capable of hosting an operating system and a microcontrolled platform (*Arduino Mega*) with a group of sensors and actuators connected to sending perceptions and receiving actions from the Embedded MAS. The prototypes employ Luminosity sensors (LDR) and ultrasonic HC-SR04 sensors. Furthermore, we employ the following actuators: a buzzer, two engines, and a group of LEDs to simulate the vehicles' headlights and sirens.

Then, five Embedded MAS were implemented: a bus, a police, an ambulance, the highway support station, and the hospital. The Embedded MAS from the hospital and the support station run each one in a different *notebook* with a microcontroller *Arduino Mega* attached to them since they just redirect messages and coordinates vehicles. In contrast, the bus, the police and the ambulance vehicles are prototypes using the previously explained sensors and actuators. Figure 14 shows the three built vehicles' prototypes.

5.4 Implementation

Each Embedded MAS has only one *Diffuser* agent interfacing the hardware with specific plans for deliberation depending on the embedded vehicle. This *Diffuser* agent has initial settings; it first informs the serial port and the microcontroller using the internal action *.port*. Then it is necessary to inform if the agent will have the ability to receive perceptions coming from sensors *openned* or *blocked* using the internal action *.percepts*. Considering the RF communication, it is necessary to set an identifier for the agent using the new internal action *setRFId*. Finally, the new internal action *.getRFMessages* is used to indicate whether the agent will be *open* or *block* for communication via RF. Plan 1 is the initial plan in AgentSpeak created to set all these agents' configurations. The new internal actions *setMyRFId* and *getRFMessages* allow agents to interface the RF messages and to identify themselves to other devices. They represent advances and contributions in how agents can deal with and interface messages from other devices.

Plan 1 The plan that configure the interaction with the *Diffuser* agent and the microcontroller

```
+!start : true <-
        .print("I am a Diffuser agent");
        .port(COM5);
        .percepts(open);
        .setMyRFId(ROMA);
        .getRFMessages(open).
```

In the case of the *Bus* agent, it uses two engines to move around while perceiving the environment (receiving perceptions from the sensors). Based on its perceptions, the *bus* agent can check the distance to any obstacle, and if this distance is less than 20 cms, it assumes that it was a collision. When a collision happens, the agent starts to believe that have suffered an accident and diffuses a *broadcast* message requesting help. Then, the *Bus* agent starts to wait for help to arrive. The Plan 2 shows the snippet of code of the *checkCollision* plan. The new internal action *.diffuse* allows agents to diffuse messages to other devices using RF using one of the three available operations (*unicast*, *multicast*, and *broadcast*) using illocutionary forces *tell* and *achieve*. So far, there is no approach allowing BDI agents to communicate directly with RF sensors. Thus, our approach allows it by guaranteeing security in *unicast* and *multicast* messages.

Plan 2 The object distance identification plan for collision detection

```
+!checkCollision : proximity(Status) AND Status < 20 <-
        .print("I have suffed an accident!!!");
        .act(stopBus);
        .act(accidentAlert);
        .print("Asking for help!!");
        .diffuse(broadcast, tell, help);
        +wait;
        !waitForHelp.
```

The *Police's* Embedded MAS has a *Diffuser* agent named *Police*. It implements a plan responsible for receiving help messages from RF. Once the agent receives any message from RF, it automatically decodes the message by extracting a belief or an intention depending on the message's illocutionary force. For example, a telling force indicates that the message carries a belief. Otherwise (achieve) carries an intention. So, the help message received was using the illocutionary force tell and automatically decoded as a belief that indicates to the agent that someone needs help. Consequently, Plan 3 is activated, which is responsible for diffusing an RF message to the other members of the public security group indicating that someone needs help.

Plan 3 The *Police* plan to intercept and forward the help request

```
+!askForHelp : help <-
    .print(”I got an accident message. Requesting support!”);
    .diffuse(”SU\\”, tell, someoneNeedHelp);
    !helping.
```

After that, the *Police* agent starts to wait for another message using *tell* force to indicate that help is coming. Since someone informs to *Police* agent that the ambulance is coming, the *Police* agent diffuses a *unicast* message using the *tell* force to the *Bus* informing them. Plan 4 shows the *helping* plan that waits for someone to send a *helpComing* belief to trigger the plan.

Plan 4 The *Police* agent sends to the *Bus* that the help is coming

```
+!helping : help AND helpComing <-
    .print(”The help is coming!”);
    .diffuse(”ROMA”, tell, helpComing).
```

The *Support* has a *Diffuser* agent named *Support* responsible for receiving help messages from the public safety group and forwarding them to the *Hospital* Embedded MAS (Plan 5). Another contribution of our approach is the possibility of adopting design time groups. *JaCaMo* framework allows the programming of the organizational dimension. Still, it is impossible to organize devices managed by Embedded MAS in groups mainly because the *Moise+* is built to organize agents from the same MAS in groups, with roles and missions, for example.

Plan 5 The Support plan is to intercept help messages in the public safety group and forward them to the Hospital

```
+!informHospital : someoneNeedHelp <-
    .print(”I got a help message in the public safety group. Forwarding to the Hospital!”);
    .diffuse(”SUHO”, tell, someoneNeedHelp);
    !helping.
```

Then, the *Support* agent starts to wait for the acknowledgment message from the *Hospital* announcing that the *Ambulance* is going to the place of the accident. It informs to the *Police* agent by sending an *unicast* message saying that the help is coming. Plan 6 shows the *informHelpComing* plan, which waits for *Hospital* agent to send the *ambulanceIsComing* belief to trigger the plan.

Plan 6 The *Support* agent plan sends to the *Police* agent that the help is coming

```
+!informHelpComing : ambulanceIsComing <-
    .print(”I understood! The Ambulance is coming!”);
    .diffuse(”POLI”, tell, helpComing).
```

The *Hospital* MAS has a *Diffuser* agent named *Hospital* agent. Like the other agents previously presented, the *Hospital* agent has a plan activated when receiving an RF message with the *tell* force indicating that someone needs help. Upon receiving the message, the plan is activated, and an RF message with the *tell* force is diffused to the available ambulance. In Plan 7, it is possible to see the source code of the sendAmbulance plan.

Plan 7 The *Hospital* agent plan that receives a help message from the *Support* agent and forward them to the *AmbulanceMAS*

```
+!sendAmbulance : someoneNeedHelp <-
    .print(”Asking for an ambulance!!!”);
    .diffuse(”SUAM”, tell, someoneNeedAmbulance);
    !checkAmbulanceIsComing.
```

Afterward, the *Hospital* agent waits for the confirmation message from the *Ambulance* agent informing that it is on its way and forwards it with a *unicast* message to the *Hospital* agent. Plan 8 shows the source code of the plan *checkAmbulanceIsComing* from the *Hospital* agent that sends the *ambulanceIsComing* belief to the *Support* agent.

Plan 8 The *Hospital* agent plan that sends to the *Support* agent that the Ambulance is coming

```
+!checkAmbulanceIsComing : ambulanceIsComing <-
    .print(”Sending an Ambulance!”);
    .diffuse(”SUPP”, tell, ambulanceIsComing);
    !wait.
```

Finaly, the *Ambulance* has a plan that receives messages from the *Hospital* agent informing them that someone needs help. Afterward, the *Ambulance* agent sends an acknowledgment message to the *Hospital* agent informing that it is moving to the place of the accident. The *!iAmGoing* plan can be seen in Plan 9.

Plan 9 The *Ambulance* agent plan that inform to the *Hospital* agent that it is going to the accident place

```
+!iAmGoing : someoneNeedAmbulance <-
    .print(”I am going!!!”);
    .diffuse(”SUHO”, tell, ambulanceIsComing);
    !wait.
```

In the proof of concept presented, Embedded MAS are applied in different prototypes—one per each—and they can act autonomously with individual objectives to be achieved. The road accident scenario was created to test the middleware and all its types of messages (*Unicast*, *Broadcast*, and *Multicast*). As expected, the proof of concept was successfully executed. The *Broadcast* message sent by the *Bus* was received by all the Embedded MAS within the range of the *Bus's* radiofrequency emitter. The *Multicast* message sent by the *Police* was received by members of the public safety group within range of the *Police*.

Ultimately, the several *Unicast* messages exchanged in the other interactions between the Embedded MAS also worked as expected.

6 Final considerations

This paper presented a middleware for supporting the development of hardware applications using Embedded MAS. It allows agents to automatically receive sensors' perceptions and transfer action commands to controllers. Moreover, it presents a protocol that allows different Embedded MAS to communicate even when no network infrastructure is available. The middleware provides a secure channel for *unicast* and *multicast* messages to guarantee the security and privacy of the exchanged messages. Our approach helps avoid Totally Closed MAS situations by providing ad-hoc mechanisms that remove the dependency on centralized architectures. Totally Closed MAS is inevitable in hardware applications since hardware can be damaged anytime. Therefore, it is essential to give options to the system designer, considering the agent domain where the approaches are scarce.

Although the architecture specifies a particular microcontroller in the firmware layer, by adopting Javino as the serial interface, it is possible to maintain the independence between the technologies used at high-level systems (MAS) and hardware layers. Moreover, the microcontroller can be changed without affecting the Embedded MAS programming since the protocol is based on serial message exchanging. For future works, it is necessary to implement the cryptography algorithms in the platform's operating system running as a service because there is a need to manage the cryptography keys—supported by a certification authority—outside the microcontroller once they have limited storage and memory processing.

Author contributions All authors contributed to the study's conception and design. VJ realized material preparation, and all authors contributed to the analysis and organization of the results. VJ wrote the first draft of the manuscript, and all authors commented and contributed to all the following versions. All authors read and approved the final manuscript.

Funding No funding was received for conducting this study.

Declarations

Conflict of interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial or non-financial interest in the subject matter or materials discussed in this manuscript.

Ethical approval All authors have read and abided by the statement of ethical standards for manuscripts.

Consent to participate All authors consented to participate in this manuscript.

Consent for publication All authors approved this manuscript for publication.

References

Al-Otaibi S, Al-Rasheed A, Mansour RF, Yang E, Joshi GP, Cho W (2021) Hybridization of metaheuristic algorithm for dynamic cluster-based routing protocol in wireless sensor networks. IEEE Access 9:83751–83761. https://doi.org/10.1109/ACCESS.2021.3087602

Artikis A, Pitt J (2008) Specifying open agent systems: a survey. In: International workshop on engineering societies in the agents world, pp 29–45

Aziz RM, Baluch MF, Patel S, Kumar P (2022) A machine learning based approach to detect the Ethereum fraud transactions with limited attributes. Karbala Int J Mod Sci 8:139–151

Baier C, Katoen JP (2008) Principles of model checking (representation and mind series). The MIT Press

Barros RS, Heringer VH, Lazarin NM, Pantoja CE, Moraes LM (2014) An agent-oriented ground vehicle's automation using Jason framework. In: 6th international conference on agents and artificial intelligence, pp 261–266

Bellifemine FL, Caire G, Greenwood D (2007) Developing multi-agent systems with JADE, vol 7. John Wiley & Sons

Bengtsson J, Larsen K, Larsson F, Pettersson P, Yi W (1996) UPPAAL - a tool suite for automatic verification of real-time systems. In: Alur R, Henzinger TA, Sontag ED (eds) Hybrid systems III, no. 1066 in lecture notes in computer science, Springer Berlin Heidelberg, pp 232–243, https://doi.org/10.1007/BFb0020949

Boissier O, Bordini RH, Hübner JF, Ricci A, Santi A (2013) Multi-agent oriented programming with JaCaMo. Sci Comput Program 78(6):747–761

Bordini RH, Hübner JF, Wooldridge M (2007) Programming multi-agent systems in agent speak using Jason. John Wiley & Sons Ltd

Brandão FC, Lima MAT, Pantoja CE, Zahn J, Viterbo J (2021) Engineering approaches for programming agent-based IoT objects using the resource management architecture. Sensors. https://doi.org/10.3390/s21238110

Bratman ME (1987) Intention, plans and practical reasoning. Cambridge Press

Busetta P, Rönnquist R, Hodgson A, Lucas A (1999) Jack intelligent agents-components for intelligent agents in java. Agent Link News Lett 2(1):2–5

Castro LFS, Borges AP, Alves GV (2018) Developing a smart parking solution based on a holonic multiagent system using JaCaMo framework. In: Anais do XII workshop-Escola de Sistemas de Agentes, seus Ambientes e apliCações - WESAAC 2018, Fortaleza, CE, vol XII, pp 226–231, http://uece.wesaac.com/

de Castro LFS, Manoel FCP, de Jesus VS, Pantoja CE, Borges AP, Alves GV (2022) Integrating embedded multiagent systems with urban simulation tools and IoT applications. Revista de Informatica Teorica e Aplicada 29(1):81–90. https://doi.org/10.22456/2175-2745.110837

Dennis LA, Farwer B (2008) Gwendolen: A BDI language for verifiable agents. In: Proceedings of the AISB 2008 symposium on logic and the simulation of interaction and reasoning, society for the study of artificial intelligence and simulation of behaviour, pp 16–23

Elgeziry M, Costa F, Genovesi S (2022) Radio-frequency guidance system for path-following industrial autonomous guided vehicles. In: 2022 16th European conference on antennas and propagation (EuCAP), pp 1–5

Endler M, Baptista G, Silva LD, Vasconcelos R, Malcher M, Pantoja V, Pinheiro V, Viterbo J (2011) ContextNet: context reasoning and sharing middleware for large-scale pervasive collaboration and social networking. In: Proceedings of the workshop on posters and demos track, p 2

Freitas J, Souza L, Sardou P, Lazarin N (2021) Comunicação segura em VANET. Anais da XIX Escola Regional de Redes de Computadores. SBC, Porto Alegre, RS, Brasil, pp 109–114. https://doi.org/10.5753/errc.2021.18551

Groshev M, Baldoni G, Cominardi L, de la Oliva A, Gazda R (2022) Edge robotics: are we ready? Digital communications and networks, an experimental evaluation of current vision and future directions https://doi.org/10.1016/j.dcan.2022.04.032, https://www.sciencedirect.com/science/article/pii/S2352864822000888

Guinelli JV, Pantoja CE (2016) A middleware for using pic microcontrollers and jason framework for programming multi-agent systems. In: I Workshop de Pesquisa em Computação dos Campos Gerais (WPCCG)

Guinelli JV, Aguiar OV, Lazarin NM (2018) Análise e comparação de algoritmos criptográficos simétricos embarcados na plataforma arduino. In: Anais Estendidos do XVIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais, SBC, Porto Alegre, RS, Brasil, pp 167–176, https://sol.sbc.org.br/index.php/sbseg_estendido/article/view/4153

Hamdani M, Sahli N, Jabeur N, Khezami N (2022) Agent-based approach for connected vehicles and smart road signs collaboration. Comput Inf 41(1):376–396

Heijmeijer AvH, Alves GVAZ (2018) Development of a middleware between SUMO simulation tool and JaCaMo framework. Adv Distrib Comput Artif Intell J 7(2):5–15. https://doi.org/10.14201/ADCAIJ201872515

Hernández MEP, Reiff-Marganiec S (2016) Towards a software framework for the autonomous internet of things. In: Future Internet of Things and Cloud (FiCloud), 2016 IEEE 4th international conference on, pp 220–227

Hindriks KV, De Boer FS, der Hoek WV, Meyer JJC (1999) Agent programming in 3APL. Auton Agent Multi-Agent Syst 2(4):357–401. https://doi.org/10.1023/A:1010084620690

Isma A, Kamel A, Abderrahmane S (2022) Hardware in the loop simulation for robot navigation with RFID. In: 2022 7th international conference on image and signal processing and their applications (ISPA), pp 1–6, https://doi.org/10.1109/ISPA54004.2022.9786321

Jensen AS (2010) Implementing Lego agents using Jason. CoRR abs/1010.0150, http://arxiv.org/abs/1010.0150

Jesus VS, Pantoja CE, Manoel F, Alves GV, Viterbo J, Bezerra E (2021) Bio-inspired protocols for embodied multi-agent systems. In: ICAART (1), pp 312–320

Junger D, Guinelli JV, Pantoja CE (2016) An analysis of Javino middleware for robotic platforms using Jason and JADE frameworks. In: 10th software agents, environments and applications school

Khalajzadeh H, Simmons AJ, Abdelrazek M, Grundy J, Hosking J, He Q (2020) An end-to-end model-based approach to support big data analytics development. J Comput Lang 58:100964. https://doi.org/10.1016/j.cola.2020.100964

Kumawat P (2022) Radio frequency identification technology used to monitor the use of water point for grazing cattle. In: Integrated emerging methods of artificial intelligence & cloud computing, pp 270–276

Lazarin NM, Pantoja CE (2015) A robotic-agent platform for embedding software agents using raspberry PI and Arduino boards. In: 9th software agents, environments and applications school

Lazarin NM, Pantoja C, de Jesus V (2021) Um Protocolo para Comunicação entre Sistemas multi-agentes embarcados. In 15th workshop-school on agents, environments, and applications (WESAAC)

Leitão P, Karnouskos S, Ribeiro L, Lee J, Strasser T, Colombo AW (2016) Smart agents in industrial cyber-physical systems. Proc IEEE 104(5):1086–1101. https://doi.org/10.1109/JPROC.2016.2521931

Manoel F, Pantoja CE, Samyn L, de Jesus VS (2020) Physical artifacts for agents in a cyber-physical system: a case study in oil & gas scenario (EEAS). In: SEKE, pp 55–60

Manoel FCPB, Nunes PdSM, de Jesus VS, Pantoja CE, Viterbo J (2017) Applying multi-agent systems in prototyping: programming agents for controlling a smart bathroom model with hardware limitations. Revista Jr de Iniciação Científica em Ciências Exatas e Engenharia (ICCEEg)

Mansour RF, Alsuhibany SA, Abdel-Khalek S, Alharbi R, Vaiyapuri T, Obaid AJ, Gupta D (2022) Energy aware fault tolerant clustering with routing protocol for improved survivability in wireless sensor networks. Comput Netw 212:109049. https://doi.org/10.1016/j.comnet.2022.109049

Matarić MJ (2007) The robotics primer. MIT Press

Michaloski J, Schlenoff C, Cardoso R, Fisher M, others (2022) Agile robotic planning with Gwendolen. Technical Note (NIST TN), National Institute of Standards and Technology, Gaithersburg, MD https://doi.org/10.6028/NIST.TN.2222

Mundhenk P, Hamann A, Heyl A, Ziegenbein D (2022) Reliable distributed systems. In: 2022 design, automation & test in Europe conference & exhibition (DATE), pp 287–291. https://doi.org/10.23919/DATE54114.2022.9774734

Ortiz G, Zouai M, Kazar O, de Prado AG, Boubeta-Puig J (2022) Atmosphere: context and situational-aware collaborative IoT architecture for edge-fog-cloud computing. Comput Stand Interfaces 79:103550. https://doi.org/10.1016/j.csi.2021.103550

Palanca J, Rincon J, Julian V, Carrascosa C, Terrasa A (2022) Developing IoT artifacts in a MAS platform. Electronics 11(4):655

Pantoja CE, Stabile MF, Lazarin NM, Sichman JS (2016) ARGO: an extended Jason architecture that facilitates embedded robotic agents programming. In: Matteo B, Müller JP, Ingrid N, Rym ZW (eds) Engineering multi-agent systems: 4th international workshop, EMAS 2016. Springer, pp 136–155

Pantoja CE, Stabile Jr MF, Lazarin NM, Sichman JS (2016b) Argo: a customized Jason architecture for programming embedded robotic agents. Fourth international workshop on engineering multi-agent systems (EMAS 2016)

Pantoja CE, Soares HD, Viterbo J, Seghrouchni AEF (2018) An architecture for the development of ambient intelligence systems managed by embedded agents. In: The 30th international conference on software engineering & knowledge engineering, San Franscisco, pp 214–215

Pantoja CE, Viterbo J, Seghrouchni AEF (2020) From thing to smart thing: towards an architecture for agent-based AMI systems. In: Gordan J, Chen-Burger YHJ, Mario K, Šperka Roman, J HR, C JL (eds) Agents and multi-agent systems: technologies and applications 2019, Springer, Singapore, pp 57–67

Pham VA, Karmouch A (1998) Mobile software agents: an overview. IEEE Commun Mag 36(7):26–37

Rao AS (1996) AgentSpeak(L): BDI agents speak out in a logical computable language. In: Carbonell JG, Siekmann J, Goos G, Hartmanis J, van Leeuwen J, Van de Velde W, Perram JW (eds) Agents breaking away, vol 1038, Springer, Berlin, Heidelberg, pp 42–55, https://doi.org/10.1007/BFb0031845, series Title: Lecture Notes in Computer Science

Sakurada L, Barbosa J, Leitão P, Alves G, Borges AP, Botelho P (2019) Development of agent-based CPS for smart parking systems. In: IECON 2019—45th annual conference of the IEEE industrial electronics society, vol 1, pp 2964–2969, https://doi.org/10.1109/IECON.2019.8926653

Savaglio C, Fortino G, Zhou M (2016) Towards interoperable, cognitive and autonomic IoT systems: an agent-based approach. In: Internet of Things (WF-IoT), 2016 IEEE 3rd world forum on, pp 58–63

Silva GR, Becker LB, Hübner JF (2020) Embedded architecture composed of cognitive agents and ROS for programming intelligent robots. IFAC-Papers OnLine 53(2):10000–10005. https://doi.org/10.1016/j.ifacol.2020.12.2718

Stabile Jr MF, Sichman JS (2015) Evaluating perception filters in BDI Jason agents. In: 4^{th} Brazilian conference on intelligent systems (BRACIS)

Stabile MF Jr, Pantoja CE, Sichman JS (2018) Experimental analysis of the effect of filtering perceptions in BDI agents. Int J Agent-Orient Softw Eng 6(3–4):329–368

Taboun MS, Brennan RW (2017) An embedded multi-agent systems based industrial wireless sensor network. Sensors 17(9):2112

Wooldridge M (2009) An introduction to multi-agent systems. Wiley

Wooldridge MJ (2000) Reasoning about rational agents. MIT Press, NY

Zhang X, Tang S, Liu X, Malekian R, Li Z (2019) A novel multi-agent-based collaborative virtual manufacturing environment integrated with edge computing technique. Energies 12(14):2815

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Vinicius Souza de Jesus[1] · Nilson Mori Lazarin[2] · Carlos Eduardo Pantoja[2] · Fabian César Pereira Brandão Manoel[2] · Gleifer Vaz Alves[3] · José Viterbo[1]

✉ Vinicius Souza de Jesus
vsjesus@id.uff.br

Nilson Mori Lazarin
nilson.lazarin@cefet-rj.br

Carlos Eduardo Pantoja
pantoja@cefet-rj.br

Fabian César Pereira Brandão Manoel
fabiancpbm@gmail.com

Gleifer Vaz Alves
gleifer@utfpr.edu.br

José Viterbo
viterbo@ic.uff.br

[1] Fluminense Federal University, Niterói, Brazil

[2] Federal Center for Technological Education Celso Suckow da Fonseca, Rio de Janeiro, Brazil

[3] Federal University of Technology–Paraná, Ponta Grossa, Brazil