Delft University of Technology
Master's Thesis in Embedded Systems

# MAES: A Multi-Agent Systems Framework for Embedded Systems

**Carmen Chan-Zheng**

embedded software

TU Delft Delft University of Technology

# MAES: A Multi-Agent Systems Framework for Embedded Systems

Master's Thesis in Embedded Systems

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Carmen Chan-Zheng
c.chanzheng@student.tudelft.nl

September, 2017

**Author**
Carmen Chan-Zheng (c.chanzheng@student.tudelft.nl)
**Title**
MAES: A Multi-Agent Systems Framework for Embedded Systems
**MSc presentation**
September 29th, 2017

**Graduation Committee**

| | |
|---|---|
| Prof. dr. Koen Langendoen | Delft University of Technology |
| Dr. Matthijs Spaan | Delft University of Technology |
| Dr. Alessandra Menicucci | Delft University of Technology |
| M.Eng. Johan Carvajal-Godnez | Delft University of Technology |

**Abstract**

Miniaturization and cost reduction of hardware components have created a trend in the space industry where the traditional centralized computer is being replaced by distributed computer architecture. However, this trend comes with a cost: the on-board software complexity of the space missions has increased. The complexity has origins in the requirements of the missions where in general, these are coordination and control-related processes. As the coordination and the control of the satellite's activities are not trivial tasks, the Multi-Agent Systems(MAS)-approach has been proposed as a new architectural style due to its distributed nature. There are several existing frameworks for implementing MAS-based applications, however, most of them are neither designed to satisfy real-time requirements nor designed to be implemented in highly-constrained embedded systems. Therefore, the purpose of this thesis is to develop a new tool for MAS-based applications: A Multi-Agent Framework for Embedded Systems (MAES).

The framework was implemented on top of a Real-Time Operating System: TI-RTOS, therefore, applications implemented with MAES have real-time characteristics. Experiments have shown that the execution time of an Attitude Determination algorithm is consistent on each call with a variance value of the order of $10^{-5}$ $[s^2]$, demonstrating the predictability of the framework. Furthermore, the user coding effort is reduced as several routines are standardized and encapsulated into MAES' API. However, the predictability and ease-of-use come with a slight cost: experiments have shown that MAES-based applications lead to an increase of 6.7 KB in average in Flash memory and 4.5 KB in average in SRAM memory with respect to its non-agent implementation. Also, the CPU utilization increases as inter-agent communication requires additional processing time, also increasing the power consumption. However, the increase is low as the results have shown that is less than 1% in average.

# Preface

My passion for the Embedded Systems is driven by the eagerness to understand the interaction between the software and the hardware. Moreover, I find exciting the multidisciplinary nature of this field as I am a learning-lover and a curious person. Driven by the curiosity, I felt attracted to this thesis topic as this not only involves the field that I love but it also involves the Aerospace field. Little did I know that the magnitude of the knowledge acquired during this thesis would be that large. Not only I have improved my technical skills but also I learned to manage a project from scratch and how to communicate effectively with my supervisors. However, I believe that the most valuable lesson I learned is to be critical with every aspect of the thesis.

The journey in this master would not have been possible without all the support from my family and friends. Beside them, there are few people that I would like to acknowledge specially for the success of this Master degree. First, I'd like to thank the University of Costa Rica staff who trusted me blindly with all the academic choices I've made during this master thesis. Secondly, I would like to thank the committee and the members of MICITT (Ministerio de Ciencia, Tecnologia y Telecomunicaciones) for their decision to trust and to grant me the scholarship. Then, I would like to thank Professor Koen Langendoen for his valuable advice during our bi-weekly meetings. Also, I want to express my sincere gratitude to Johan Carvajal-Godinez for his patience and kindness with guiding me through this journey. I thank both of them for teaching me this invaluable lesson: Always keep questioning and never take anything for granted at first. Last but not least, I thank to God, where my faith has kept me strong when my friends and family have been far away. I want to dedicate this to my sister, mom and dad who have been the keystone of all of this success.

# Contents

# Chapter 1

# Introduction

This chapter provides the motivation to create a Multi-Agent Systems-based framework for satellite's on-board software application development. Then, the subsequent sections introduce important concepts of Multi-Agent Systems and provide a brief description of current application implementations. Next, the specific research problem and the characteristics of the framework are discussed. Afterwards, the research strategy and the thesis contribution are further explained. Finally, the outline of this research is presented.

## 1.1 Motivation

Miniaturization and cost reduction of hardware components have created a trend in the space industry where the traditional centralized computer architecture is shifting towards a distributed architecture. As the space environment is dynamic and unpredictable, there could be scenarios where the operation of a centralized satellite system might be disrupted if the control satellite loses its functionality and the rest of the nodes are disconnected. Therefore it is necessary a distributed architecture where each node should be able to perform intelligent improvements based on the circumstances [1].

A new generation of picosatellites called CubeSat is exploited by the industry and the academy for its cost effective platform in which is possible to test advance mission concepts using constellations, swarms disaggregated systems [2]. The picosatellites weight between 1-2kgs [3], and it has a highly-constrained embedded processor such MSP430 [4].

The decentralization of the architecture offers several advantages: lower cost in hardware, improvement in fault tolerant capabilities by simplifying resource sharing among subsystems, reconfigurability and upgradability [1][5]. However, the decentralization increased the on-board software complexity. The main cause of the growth in the on-board software complexity comes from the requirements of the missions where, in general, these requirements are fundamentally about coordination and control [6]. While

the coordination is done among the many components of the spacecraft, the control of its activities is done by a deterministic operating system (Real-Time Operating System). The on-board software is an embedded real-time software, i.e. is characterized by being autonomous, which can make decisions without human intervention. In particular, the real-time characteristic is important for fault protection, identification and recovery tasks as this ensures the safety of the spacecraft by responding to failures in a timely manner [7][8].

As coordinating and controlling the activities among all the satellites are not trivial, a Multi-Agent System(MAS) approach is proposed as a new architectural style as this enables artificial-intelligence capabilities [9]. The key feature of these systems lies in their capacity to address computing problems by distributing them into different components named agents [10]. The agents are autonomous computational entities, which are assigned to a specific "role" within the system, communicate with other agents and perceive their environment. These features allow them to work "proactively" and "reactively" to their environment and therefore, to achieve a wide range of goals [11].

Currently, most of the applications do not use any dedicated platform or tool to develop Multi-Agent Systems applications, but instead, these are developed from scratch [12]. However, the advantage of using an existing platform or framework is the reduced time of development since the MAS's services and communication mechanism are already implemented, therefore, the developer needs only to focus on each agent's implementation.

For this reason, the goal of this thesis is to develop a software framework capable to implement MAS-based applications for highly-constrained systems.

## 1.2 State-of-the-Art: Multi-Agent Concepts and current frameworks

The term *agent* has been debated among the members of the scientific community, hence, there is no universally-accepted definition. However, common concepts are found from within all these definitions: *autonomy* and *environment*. The authors of [13] compiled several definitions among the community and presented their own generalized definition:

*"An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."*

Moreover, an *agent* is characterized by the following properties [14]:

- autonomy: an agent operates without direct human intervention, and it has control over its actions and internal state.

- social: an agent cooperates with humans or with other agents to achieve a goal.
- reactivity: an agent perceives its environment and responds in a timely manner to changes that occur in the environment.
- pro-activity: an agent is able to exhibit goal-directed behavior by taking initiative.

Each agent is a micro-system living in an environment and two or more agents build up a macro-system named *Multi-Agent System* (MAS) [15]. According to [16], the characteristics of these systems are: each agent is subjective of its environment, thus, each agent has incomplete information of the complete system. Also, the computation performed by an agent is asynchronous. Lastly, there is no global control and the data is decentralized, therefore, each agent possesses its own internal state not accessible by other agents.

A wide range of applications has been developed with this technology due to the distributing nature of the MAS-based architecture. Some applications target condition monitoring and fault record on power system grids [17], economics [18], e-commerce [19], automation of SCADA [20], robotics [21], traffic and transportation [22], among any others.

In the aerospace field, there have been notable implementations from NASA such as the Livingstone Project, Orbital Communication Adaptor and The Lights out Ground Operation Systems [23]. Other implementations are found in the literature such as: fault detection and recovery in gyroscope's drift in small satellites [24], flying formation [25] and a teleoperated robot [26].

Although the large majority of applications do not use any dedicated platform or tool [12], there are several available frameworks and platforms used for MAS-based applications development. Some existing platform/framework designs are based on the specifications established by The Foundation for Intelligent Physical Agents (FIPA), which is an IEEE Computer Society standards organization, that creates specifications for generic agent technologies. For example, SPADE [27], JADE [28][29] and Mobile-C [30] are FIPA-compliant platforms. On the other hand, it also reports in literature successful non-FIPA compliant MAS framework designs such BESA-ME [31], EmSBot [32] and ObjectAgent [5].

## 1.3 Research Problem

Albeit there are several existing frameworks on the market for implementing MAS-based applications, most of them are not designed to satisfy real-time requirements. Furthermore, these are not designed for embedded implementation. Therefore, the aim of this thesis is to enable a software framework for

the implementation of MAS-based applications with real-time requirements compatible with highly resource-constrained embedded systems. After the framework is created, an attitude determination reference application is implemented as a demonstration of the framework's operation.

The framework shall satisfy requirements proposed by the stakeholder of the project. The following list encompasses requirements for the software framework named MAES (Multi-Agent Framework for Embedded Systems):

1. **FIPA - based components**: MAES shall contain the minimum mandatory technology specified by the FIPA 23 specification for agent lifecycle management.
2. **Lightweight Real-Time implementation**: MAES shall enable the implementation on a Class-2 IoT devices and on Real-time operating systems.
3. **Single embedded platform execution**: MAES shall deploy a **single** Agent Platform in a local embedded-hardware device. As only a single Agent Platform is deployed, the communication among the agents is intra-platform only.
4. **Scalability**: The number of agents living in the Agent Platform is not determined by the framework but by the user's application needs.
5. **Portability**: MAES shall provide C++ agent-based API for interfacing with application developers.

For this research, a Multi-Agent System application is provided by the stakeholder; therefore, the number of agents of the system is assumed to be known a-priori at compile time.

Once that the motivation, requirements and the goal of the project are defined, the main challenge of the framework development is identified as the mapping process of the concepts/components from the Multi-Agent System context into the Embedded Real-Time Operating System context. Therefore, the research questions that this thesis addresses are:

1. **How to map the MAS-based software architecture components onto an Embedded Real-Time Operating System?**
2. **How does the mapping strategy affect the CPU's memory, load and power consumption?**

## 1.4 Research methodology

In order to answer the research questions discussed in Section 1.3 a methodology comprising four phases is proposed.

The first phase is the preparation where a literature review is conducted in order to determine the minimum required components to build a Multi-Agent Systems, and also to determine the current mapping strategies implemented in state-of-the-art platforms and frameworks. Additionally, testing applications are implemented in JADE. JADE is chosen as a MAS framework baseline and used in this research in order to understand the process of building a MAS application from an agent developer's perspective. Also, JADE is selected as the benchmark framework to be compared against. The second phase consists a theoretical work of the MAS components mapping where the mapping criteria are discussed. Third, the framework is developed using the C++ language according to the mapping discussed in the second phase. Lastly, framework testing and benchmarking are conducted to verify the framework's functionality and to identify required future capabilities.

## 1.5 Thesis Contribution

- Develop MAS-based framework that satisfies the requirements propsed in Section 1.3.
- Report how the mapping methodology might affect the CPU's load, power consumption and memory.

## 1.6 Thesis Structure

The thesis is organized as follows: Chapter 2 presents the literature review conducted for this research. It discusses the main architectural components of Multi-Agent Systems, the programming languages used to implement them and a survey of current platforms/frameworks.

Then, Chapter 3 is divided into two main sections: Mapping Strategy and Framework Implementation. In the first section, the mapping strategy for each MAS architectural component is discussed and selected based on a trade-off analysis. Then, the framework implementation on top of a Real-Time Operating System is described in detail.

Chapter 4 presents and analyzes the framework verification results. Next, Chapter 5 presents the benchmark results (Memory footprint, CPU Utilization and Power consumption) of an application implemented with the framework against the same application implemented in a non-agents environment. Lastly, the conclusions, recommendations and future work are discussed in Chapter 6.

# Chapter 2

# Multi-Agent Systems: Literature Review

In this chapter the architectural components are introduced. Then, the MAS programming languages are described accordingly. Lastly, a survey of current MAS platform/framework implementations is provided.

## 2.1 Architecture components

Regardless the application domain where a MAS is implemented, these systems have several common architectural characteristics. Shehory et al. [33] comprise the key properties of a MAS architecture:

- **MAS Organization**: The agents are organized in one of the following ways: hierarchical, flat, subsumption or modular organization.
- **Communication**: MAS platforms implement proprietary and standard communication languages or protocols for its agents interaction. Standard communication facilitates inter-platform agent conversations, but it is less efficient since the message overhead is larger than propietary communication.
- **System openness**: The ability to introduce new agents to a platform and the capability of agents to leave.
- **Infrastructure services**: MAS might provide optional services such as agent naming, agent location, mobility (across different MAS platforms), security, privacy and trust.
- **System Robustness**: The failure of a single agent does not necessarily imply the failure of the whole system as other agents can take actions for the fallen agent.

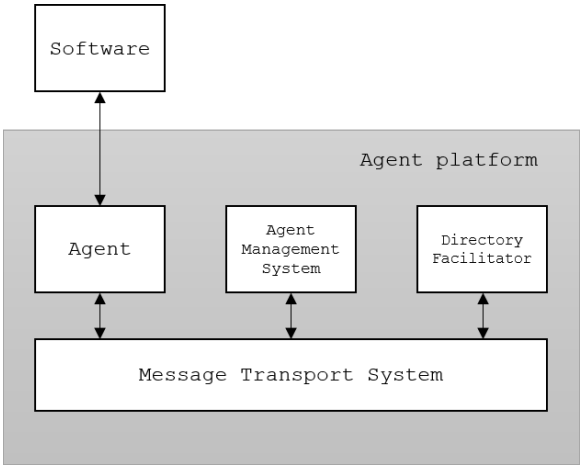


Figure 2.1: The FIPA reference model for an Agent Platform [35].

Argente et al. [34] propose that each agent has a affiliation and a role within an organization. An "affiliation" defines a long-term relationship with the organization: owner (creator of the organization with total control over it), admin (controls access of agents to the organization) and member (agents inside of the organization). On the other hand, a "role" defines the role of the agent within the organization: moderator (controls the agents communication), participant(agents allowed to talk), visitor (agents not allowed to communicate). Also, according to [34], a Hierarchy topology is formed by a supervisor and several subordinates. The supervisor coordinates all the subordinate agents tasks by capturing all the message exchanges. While the subordinates carry out the tasks and only can communicate with the supervisor. On the other hand, the Team is a topology in which all members collaborate to reach a common goal. The communication in a Team organization is limited to the members of that team.

Although all existing MAS implementations share the same set of key properties, there are several different approaches about how the MAS concept should be implemented. Therefore, FIPA established a set of specifications to standardize agent-based technology. The specifications are categorized in: agent communication, agent transport, agent management, abstract architecture and applications. These specifications are the minimum amount of technology necessary for the agents management.

The FIPA23 specification [35] provides the normative framework in which agents exist and operate. This model is used as reference for the creation, registration, location, communication, migration and retirement of agents. The components of the reference model are shown in Figure 2.1.

The *Agent Platform* (AP) is the technological architecture that provides the physical infrastructure where the agents are deployed. The AP consists

of the physical device(s) where the system is deployed, its operating system, agent support software and management components.

The *agent* is the fundamental actor of the AP. Each agent has a unique *Agent Identifier* (AID) so it can be distinguished among all the agents within the platform. Moreover, the FIPA97 specification [36] establishes that an overall *agent's behaviour* is defined by a set of one or more *tasks* that contain the computation code to generate the desired agent's action. Also, these are internal components of an agent that can not be directly accessed by other agents unless the agent offers them as services.

The agents communication is done through by exchanging messages which service is provided by the *Message Transport System* (MTS). On any Agent Platform, the MTS is provided by the *Agent Communication Channel* (ACC). According to the FIPA67 specification [37], there is no standard for the internal transportation of messages, however, since the agents might run on different APs and different technologies (CAN, HTTP, IIOP, etc), FIPA established that the messages transported between platforms should be encoded in a textual form, moreover, it specifies a standard language for messaging Agent Communication Language (FIPA ACL).

The *Agent Management System* (AMS) is a mandatory component of the AP. Only one AMS exists per AP. This component provides agent life-cycle management services and supervisory control on the AP [38]. The AMS is seen as a white pages service provider since it maintains a directory of AIDs registered within the AP, therefore, each agent must register and de-register with the AMS before entering and leaving the platform, respectively. The agent operations supported by the AMS are: register, de-register, modify, search, get description and agent's life-cycle related operations: create, invoke, destroy, suspend, resume, wake up and execute. The operations that can be initiated by the agent itself are: quit, suspend, wait, move. The agents life-cycle possible states are shown in Figure 2.2.

In *active* state, the MTS delivers messages to the agent normally. While in *Initiated/Waiting/Suspended/Transit* state, the MTS buffers messages until the agent returns to the *active* state or forwards the messages to a new location. The *transit* state is only available for mobile agents. Finally, the *Directory Facilitator* (DF) is an optional component of the AP. This component provides yellow page services to other agents. Agents may register in the DF their offered services or query from the DF the list of services provided by other agents. The functions supported by the DF are: register, de-register, modify and search.

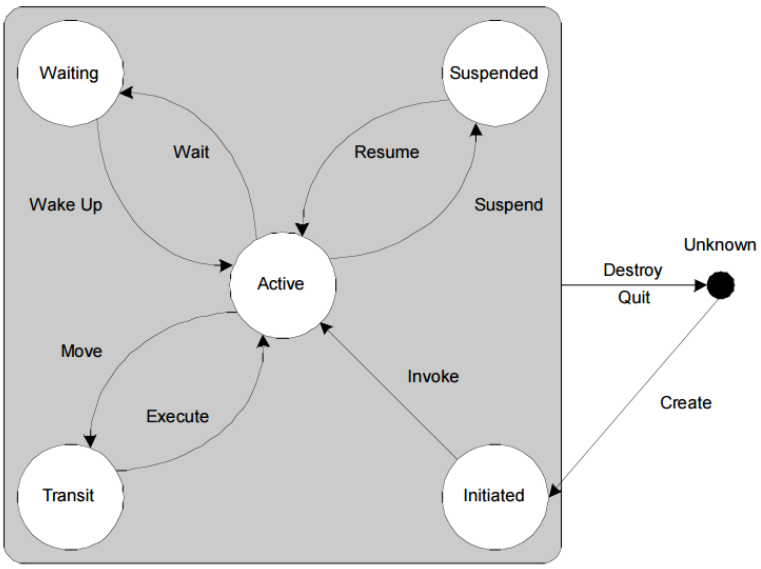


Figure 2.2: The agent Life-cycle [35].

## 2.2 Programming Languages

Most of the MAS applications are implemented by using Object-Oriented Languages (OOLs). Although they can be realized by any programming language, OOLs are the most used because the concept of an *agent* shares similarities with the concept of an *object*. *Objects* and *agents* are similar in the sense that they are computational entities encapsulating a state and perform methods on this state [39]. However, they also differ on important aspects. Woodldridge [40] identifies three main distinctions: first, the agents incorporate a stronger notion of autonomy than objects. When a method or service is made available to the public, an object can not control the access from other objects to its method whereas an agent can decide whether or not to perform the action on request from another agent. Second, agents have flexible, reactive, pro-active and social behavior. Lastly, MAS inherently is multi-threaded where each agent is assumed to have at least one thread of control. However, despite that the autonomy or the behaviours are not inherently features for a basic standard object, it can be built or be programmed.

The surveys discussed [12] and [41] show that most of the available applications, frameworks and platforms are created with Java while C++ is the second most employed. Rousset et al. [42] and Sturm et al. [43] provide surveys and evaluations of the current platforms and frameworks for MAS applications development such as JADE (Java) [28], Cougaar (Java) [44], Pandora (C++) [45]. The criterion for language selection depends of the

user needs, for example, Poggi [46] presents a FIPA-compliant agent library written in both C++ and Java for developing MAS-based applications. The Java version facilitates application development such as: voice on demand service or system prototyping. On the other hand, the C++ version is used when it is required to integrate different software modules performing sensory data interpretation, planning and fault diagnosis. For embedded control applications, C/C++ is preferred since it allows low-level hardware communication and in general, the hardware's drivers are already providen/written in that language.

## 2.3 A Survey of mapping strategies

This section discusses some technical details about the implementation of current MAS platforms and frameworks.

### 2.3.1 SPADE: Smart Python Multi-Agent Development Environment

SPADE [27][47] is a FIPA-compliant multi-agent platform written in the Python programming language. The platform offers a library that contains tools and classes to manage and to create new agents. The main communication protocol used by an agent to communicate inside and outside of the AP is the Jabber technology [48].

Each thread is the base for the creation of an *abstract agent* that is later used to create regular agents or management agents (AMS,DF). Each agent is composed of: a connection mechanism to the platform, a message dispatcher and a set of tasks [49]. The connection mechanism is the agent's communication interface with other agents. The message dispatcher redirects the messages to the correct destination, for example, when a message is received, it is redirected to the correct task's message queue. On the other hand, if the agent needs to send a message, the message dispatcher redirects it to the correct receiver agent. Finally, the agent's behaviour is implemented as a set of tasks that can be run simultaneously.

### 2.3.2 JADE: Java Agent Development Framework

JADE is a Java-written and FIPA-compliant multi-agent platform and provides a set of APIs and a graphical user interface to develop MAS-based applications [28][29]. A JADE Agent Platform is built up of one or more containers that can be distributed over a network (i.e. the Agent Platform can be hosted by several devices). Each agent lives in a container, which is a Java process that provides the services required for an agent's life cycle. Each JADE platform should have a mandatory *main container*, which is the first container to be launched when the application is executed. Other agents

can live in the *main container*, but the AMS, DF and ACC components shall run exclusively in this container. The intra-platform communication is managed by each container's message dispatcher, whereas the inter-platform communication is managed by the ACC by using the IIOP protocol.

Each JADE agent is mapped into a single thread and it can execute several behaviours. Each behaviour is implemented as a task and it is scheduled according to a cooperative non-preemptive scheduling policy. JADE uses the concurrency model *agent-per-thread* instead of *behaviour-per-thread* in order to reduce the number of threads running in the agent platform. This model becomes important in particular for resources-constrained environments. Furthermore, the *agent-per-thread* model enables improved performance since a behaviour switching is faster than Java thread switching. Also, it eliminates synchronization issues such behaviours accessing the same resource. However, when a behaviour is blocked, the rest of the agent's behaviours are also blocked. If this is not a desired feature in the application, JADE provides a solution to implement *behaviour-per-thread*,i.e., behaviours that are executed in their own thread, therefore, if a behaviour is blocked, the rest of the agent's behaviours will not be blocked.

### 2.3.3 Mobile-C: Multi-Agent Platform for Mobile C/C++ Agents

Mobile-C [30] is a MAS FIPA-compliant platform written in C/C++. This platform is specifically developed to support mobile agents that are software components capable to move between different execution environments. The implementation of this platform and its functionality are explained by its developers in [50] and [51]. The library is supported by Windows, Linux, Solaris, Mac OS X, QNX, HP-UX and Linux OS running on general-purpose or tiny and single-board computers.

The major building block of a Mobile-C system is an *agency* whose core is the agent platform that contains the minimal software components to support the execution of a stationary or a mobile agent. Additionally to the FIPA-specified components, Mobile-C implements two extra components: the Agent Security Manager (ASM) that supervises security policies for the platform and the Agent Execution Engine (AEE) that serves as the interface with the mobile agents.

The AP is a multi-threaded program that resides in a host computer running any of the abovementioned operating systems. Each component resides in its own thread and each agent (stationary or mobile) also has its own thread. An AEE thread is started per each mobile agent.

### 2.3.4 BESA-ME Framework

Flores et al. [31] present a software middleware in order to facilitate the construction of robotic control systems based on MAS techniques. The framework named BESA-ME (Behavior-Oriented, Event-Driven And Social-Based Agent Micro Edition) supports the development of MAS-oriented applications for microcontrollers. Functional tests were run successfully in PIC18F8720 and PIC18F8620 running FreeRTOS.

A BESA-ME agent is modeled by at least three components: the channel, one or more behaviours and the agent state. The Channel and the Behaviour components are implemented as RTOS tasks. The Channel component manages a message queue that blocks the task until an event is received. Once an event is received, the task assigns it to a respective treatment function and redirects the message to the corresponding behaviour. The behaviour is unblocked by the message and performs the action defined by the user. Also, the behavior is interconnected with the agent's state where it stores information about the agent, its environment and the global system. Given that two BESA-ME components are modeled as tasks, each agent can have two or more tasks, therefore, the agent is mapped to at least two threads.

### 2.3.5 EmSBoT

EmSBot (Embedded modular Software framework for networked robotic systems) [32][52] is a lightweight embedded C-written framework that provides an API to develop networked robotic systems. An EmsBoT system is composed by one or more independent *nodes* across the network. Each *node* contains processing power, memory, at least one communication channel and it may also contain sensors and actuators.

The EmsBoT core layer provides the API for agents creating and message routing. The agents only interact between them by using messages. The message structure is fixed since it is beneficial for the determinism of the system by preventing dynamic memory allocation. Furthermore, an EmsBoT agent may have one or more threads: one mandatory thread and optional worker threads. An implementation is available for the ARM Cortex-A8 and the Cortex-M4 processors running $\mu$C/OS-III. The memory usage of the firmware is 13 KB of flash memory and 5KB of data memory when there are two agents. One extra agent will cost 1KB.

### 2.3.6 ObjectAgent

The ObjectAgent (OA) project is an agent-based software architecture for autonomous distributed systems that was developed and implemented in C++ by Princeton Satellite Systems [5]. A demonstration scenario was implemented by using a PowerPC 750 processor and Enea's OSE real-time

operating system. The platform provides classes for agent creation, skills (behaviour) manipulation and agent communication.

The communication and the synchronization among the agents is done by message passing. Each agent has one or more skills that are basic functions that trigger one or more actions that produce a message to be sent across the platform. Several skills can be grouped into a activities where each of them always runs in a separate thread. An agent can carry one or more activities, therefore, an agent is mapped into one or more threads.

### 2.3.7 Discussion

Despite that JADE is the most used MAS-based application development platform [12] and its agent platform runtime's memory footprint is around 100KB (making it suitable for embedded devices) [53], it does not fullfil the real-time requirements. Similarly, SPADE does not offer real-time support either. Moreover, the Mobile-C framework runs on OSs that are not suitable for highly-constrained embedded platforms. On the other hand, MAS-based platforms on real-time embedded system approaches have been implemented successfully in the robotic field such as BESA-ME and EmSBot, whereas ObjectAgent has been successfully implemented in space applications. However, they do not discuss the implementation of the minimum technology required to build a MAS-based platform specified by FIPA. Therefore, in this thesis, a framework is developed in order to fulfill the above-mentioned gaps found during the literature study. In the following chapters, the MAS components mapping into a RTOS context and its implementation are discussed in detail.

# Chapter 3

# Multi-Agent System Framework for Embedded System Design

This chapter presents the component mapping process from the Multi-Agent System (MAS) context into the Real-Time Embedded Systems context, which addresses the first research question of this thesis: "How to map the MAS-based software architecture components onto an Embedded Real-Time Operating System?". First, the mapping strategy is discussed and selected based on a set of criteria. Then, the implementation of the framework on top of a chosen Real-Time Operating System (RTOS) is discussed.

## 3.1 Mapping strategy

The minimum components required for an Agent Platform to operate correctly are specified by the FIPA23 specification as discussed in Chapter 2. For the scope of this research, the components to be implemented are: Agent, Agent Management System and Message Transport System. On the other hand, the Directory Facilitator is useful when an application is running in a multi-platform context as this facilitates an agent's services visibility to other agents on other platforms. However, since the Directory Facilitator's implementation is optional and the MAES framework is a single-platform implementation, this component is not implemented.

Then, a trade-off analysis is conducted in order to select the RTOS onto which the MAS components are mapped. The available options are: TI-RTOS [54] and FreeRTOS [55]. Although there exists several other open source RTOSs, these options are the only RTOSs that support the microcontroller (TI MSP432). The RTOS is chosen based on three criteria: Memory footprint, Tool Support and Usability.

Table 3.1: RTOS trade-off analysis.

| Criterion (Weight) | TI-RTOS | FreeRTOS |
|---|---|---|
| Memory footprint(40) | 0 | + |
| Support(40) | + | 0 |
| Usability (20) | + | - |
| Total (100) | 60 | 20 |

The memory footprint is an important criterion as memory is a scarce resource in embedded systems. Therefore it is given a weight factor of 40.

Then, the tool support is also considered as one of the most relevant since it accelerates framework development. This criterion is given a weight factor of 40.

Lastly, the usability of the tool is considered during this analysis since an easy-to-use tool costs less time to learn and it is less error-prone. Thus, this criterion is assigned to a weight factor of 20. Each criterion can be scored by three possible scores: + (equals to 1), 0 and - (equals to -1). Table 3.1 shows the trade-off analysis result.

First, according to the FreeRTOSs webpage [56], FreeRTOS has a minimal memory footprint as a binary image will be in the region of 6KB to 12KB. On the other hand, the memory footprint for TI-RTOS is at least 15KB (obtained from a TI-RTOSs template project containing the minimum necessary files to use the kernel). For this criterion, FreeRTOS is scored positively as its memory footprint is lower than that of TI-RTOS.

For the second criterion, the TI-RTOS performs better as it comes with its own Integrated Development Environment (IDE). Also, TI-RTOS already includes tested device drivers (UART, SPI, I2C, etc.) for its MSP432 Family and additional middleware components such as the TI-RTOS Instrumentation. The TI-RTOS Instrumentation is a key feature for the development of the framework as this includes the Real-Time Analyzer, which visualizes the execution of the running tasks. Furthermore, the analyzer also includes a power performance monitoring tool, which is used during benchmarking. On the contrary, FreeRTOS is a kernel-only solution. Therefore, additional effort would be required to implement drivers.

Lastly, the third criterion is the usability in which TI-RTOS also performs better. Since TI-RTOS is already included in its own IDE (Code Composer Studio), the IDE provides a graphic user interface for the operating system configuration. For example, only the used API modules can be selected from the user interface. Furthermore, the threads can also be configured using the interface. FreeRTOS being a kernel-only solution, does not provide any user interface to facilitate the user development. Based on the trade-off analysis, the TI-RTOS (v2.20.00.06) was selected for this research.

TI-RTOS is developed by Texas Instruments and its core is the real-time multitasking kernel SYS/BIOS. The kernel is a fixed-priority preemptive scheduler. The RTOS' APIs are modularized so only those APIs that are implemented by the user are bounded into the executable program. Thus, the memory footprint is minimized. This operating system is a *soft* real-time system that ensures a process will meet a deadline most of the time and the variability is due to jitter.

The applications written in SYS/BIOS are structured as a collection of threads. Each thread carries a modularized function. The term *thread* is defined within the SYS/BIOS environment as any independent stream of instructions executed by the processor.

TI-RTOS supports four different thread-types (from the highest to the lowest priority): Hardware Interrupts (Hwi), Software Interrupts(Swi), Tasks and Background thread(idle). Each of them has different execution and preemption characteristics. For example, the Hwi thread preempts Swi, tasks and the idle thread [54].

When a thread is created, a corresponding handle to the thread is also created (for example: for Swi threads, a `Swi_Handle` is returned when the thread is created). This handle points to the thread and it has a unique address within the SYS/BIOS environment. The handle can be used for operations such deleting the thread or getting internal state of the thread (for example: get a thread's priority).

Once the kernel starts to execute, the created threads are scheduled and run based on their priority. On a single-processor environment, the SYS/-BIOS scheduler allows the preemption of higher-priority threads on lower-priority threads. However, two same-priority threads cannot preempt each other unless one explicitly yields the control of the processor.

The data sharing methods on Hwi and Swi are implemented through simple mechanisms such as: global variables and lists. However, for inter-task communication, TI-RTOS provides more complex solutions such as: semaphores, events, gates, mailboxes and queues.

In the next section, each component is mapped into TI-RTOS' context.

### 3.1.1 Agent mapping

The mapping strategies survey discussed in Section 2.3 showed that an agent can be mapped into a single thread or into multiple threads. Three criteria are selected in order to choose the agent mapping strategy where each criterion was given a weight factor. The set of criteria (with the corresponding weight factor) are: scalability (50), concurrency (30) and performance (20). Each criterion can be scored by three possible scores: + (equals to 1), 0 and - (equals to -1). The most important criterion with the largest weight factor is the scalability since it is featured in the list of requirements described in Section 1.2. Albeit the performance and the concurrency criteria

Table 3.2: Agent mapping strategy trade-off.

| Criterion (Weight) | Single-Thread | Multiple-Thread |
|---|---|---|
| Scalability (50) | + | 0 |
| Concurrency (30) | - | + |
| Performance (20) | + | - |
| Total (100) | 40 | 10 |

are not featured in the list of requirements, these are considered during the mapping selection process since these criteria affect the overall functionality of the framework. The second most important criterion is the concurrency, which is a desired feature in the MAES framework as this exploits the distributed nature of a Multi-Agent System. Lastly, the performance criterion directly affects the determinism of the Real-Time Operating System. The performance is directly affected by the frequency of context switches in the system. In general, the context switches are computationally intensive as they require CPU time to switch from one thread to another. The results for scoring each strategy are presented in Table 3.2.

In general, on the **multiple-threads strategy**, each agent's behaviour is mapped to its own thread. The advantage of using this strategy is the improvement of the concurrency. When a behaviour performs a blocking operation, the agent itself is not blocked and the other agent's behaviours might run. However, the rate of context switching is higher as an agent has many behaviours. In consequence, the processor spends more time switching between threads rather than executing them. Thus, the system performance is decreased. Besides, additional issues (i.e. race conditions) might arise when two or more behaviours are accessing their agent's resources. Also, the scalability is reduced as fewer agents can be added to the system.

On the other hand, with the **single-thread strategy**, the concurrency level is reduced as one blocked behaviour causes the blocking of the rest of the agent's behaviours. But, the performance is improved as this strategy reduces the number of context switches as the behaviours are running in the same thread. Lastly, the number of threads running in the Agent Platform is reduced in a single thread strategy. Therefore, more agents can be added leading to a more scalable system.

Based on the discussion above and the results of Table 3.2, the **single-thread strategy** is chosen as the mapping strategy.

After choosing the mapping strategy, the mapping process of the Agent component into the Real-time context is described accordingly. Both *agent* and TI-RTOS *thread* share similar characteristics: a unique identification, an internal state, a computation execution code and a method to communicate with other agents.

For this research, only the *task* module is enabled to be used in the framework. Even though tasks have lower priority and are less memory efficient (tasks require their own stack) than Hwi and Swi (they use the system's stack), these are the only threads that can wait (or block) until an event occurs or a resource becomes available, while the Hwi and Swi threads run until completion. The blocking behaviour is a key feature for any agent since an agent cannot act independently, but operates according to its environment and its internal state. Therefore, an agent requires a waiting mechanism until a certain event occurs: such as coordination and negotiation with other agents.

Furthermore, as a consequence of the $agent-to-task$ mapping, the agent obtains a new characteristic from the module: the priority. Since the SYS/-BIOS kernel schedules the operation of each agent according to its priority, the developer is required to take additional consideration on the agent priority-assignment during the design stages of the application to avoid any incorrect operation of the system. The agent's priority can be assigned up to 33 levels: with 0 being the lowest and 31 being the highest priority, while priority -1 corresponds to an inactive task.

A `Task_Handle` object is returned when a task instance is created. Since each task's handle is unique within the TI-RTOS, the handle is used as the agent's identification(AID) within the Agent Platform. The AID/handle can be used to retrieve information of an agent/task such as the priority and its execution mode.

### Behaviour mapping

During task creation, a pointer to a function is passed as a parameter. This function encapsulates the computation code that is executed by the task when it takes control of the processor. Also, this function can execute more than one subroutine defined in its body. Since an agent might have one or more behaviours, each behaviour is mapped to a subroutine. The execution order of the subroutines/behaviours is defined by the developer by implementing design patterns such as a Finite State Machine.

### Agent's state mapping

An agent's life cycle is characterized by the set of internal states described in Section 2 (see Figure 2.2). These states are mapped directly into the *task execution states*. The task can be either in one of the following states:

- Running: the current running thread on the processor.
- Ready: the thread is scheduled for execution. The scheduler always executes the highest priority task in the queue.

Table 3.3: Agent's state mapped into Task's execution state.

| Agent State | | Task Execution state |
|---|---|---|
| Active | → | Ready |
| Waiting | → | Blocked |
| Suspended/Initiated | → | Inactive |
| Terminated | → | Terminated |

- Blocked: the task cannot execute further until a particular event occurs (such as an arriving message in the mailbox).
- Inactive: the task has priority equal to -1. Thus, it is not scheduled for execution.
- Terminated: the task is not executed anymore.

The agent's states (Figure 2.2) mapping into the task's execution states is shown in Table 3.3.

It is noteworthy that none of the agent's state is mapped into the `Running` execution state. This design choice is because the chosen hardware platform is single-processor, which means only one agent is running at the time (the highest priority-ready agent). For example, if the running agent requests the state of another agent, the latter can be in any of the execution states of Table 3.3, but never in the `Running` state since the running agent has control over the processor. Also, the transit state is not mapped since the framework will not implement *mobile agents*.

### 3.1.2 Message Transport System Mapping

The TI-RTOS offers several modules for inter-task communication including: semaphores, events, mailboxes and queues. The semaphore and event modules are only signal-based modules, i.e. they cannot exchange data. On the other hand, the queue module passes data, but it does not have any signalling mechanism for new incoming data. The mailbox module combines the features of the queue module plus the semaphore module; a signal is posted when a new message is set so the subscribers to that mailbox are aware of a new incoming message.

Since a Multi-Agent System requires signaling and message exchange, the mailbox module is used as the Message Transportation Service of the framework.

A mailbox instance can be accessed by several readers and several writers. For the framework design, the number of readers of a mailbox instance is fixed to one, but the instance might have several writers. By using this design choice, a mailbox instance is assigned uniquely to an agent. Thus, each agent has its own mailbox address. When an agent is waiting for

a message, it is blocked during a user-defined time or until a message is received from any other agent.

Similar to the task creation, when a mailbox is created, the kernel returns a `Mailbox_Handle`. This handle serves as the mailbox address that other agents use in order to set the message's destination.

### 3.1.3 Agent Management System mapping

The last component to be mapped into the TI-RTOS system is the Agent Management System (AMS). The AMS is a mandatory component of any FIPA-based Agent Platform and only one exists per platform. The AMS is responsible for any agent's life cycle management. Each agent shall be registered through the AMS into the platform in order to execute its behaviour(s). Additionally, the AMS maintains an index that contains all the agents' AIDs residing in the platform.

The Agent Management System is mapped into two elements of the TI-RTOS: the kernel scheduler and special task. The scheduler specifies the execution order of the agents registered in the platform, while the task implements management operations.

The AMS task is considered as a special agent with the priority set to the highest possible as the AMS represents the management authority within the platform. Since the AMS agent has the highest priority, it preempts any other running agent when it is invoked. Moreover, this agent also uses the mailbox module as a communication method to interact with other agents. The AMS agent/task is implemented as a state machine that is in the `waiting` state for any request from other agents. If a request arrives, the AMS can perform any of the following actions:

- Register/De-register: Agent registration/deregistration into/from the Agent Platform.
- Suspend/Resume: Set the agent to suspended/active state.
- Kill: Terminate the agent's execution. The agent cannot be invoked again.
- Restart: Restart the agent's execution.

Additionally, the index that contains all the agents' AIDs living in the platform is mapped as an array of task handles/Agent's AIDs.

Table 3.4 summarizes the results of the mapping process.

Table 3.4: Multi-Agent System component mapping into TI-RTOS.

| **Multi-Agent System Component** | | **TI-RTOS component** |
|---|---|---|
| Agent | → | Task |
| Agent AID | → | Task handle |
| Agent state | → | Task execution mode |
| Behaviour | → | Subroutine in a function wrapper |
| Message Transport System | → | Mailbox Module |
| Agent Management System | → | 1. SYS/BIOS scheduler<br>2 .Task |

## 3.2 Framework Implementation

The integrated development environment (IDE) used for the framework implementation is the Code Composer Studio (Version: 7.1.0.00016) with the Texas Instruments Compiler (Version: TI v16.9.1 LTS) and XDCtools (Version: 3.32.0.06_core). The latter software component provides the technology for the configuration of SYS/BIOS modules or generation/compilation of source code files. Also, the IDE integrates the System Analyzer tool suite that provides visibility of the real-time execution and performance of the threads, which is useful for debugging and benchmarking purposes. The tool also allows the developer to analyze the load, execution sequence and timing of the application by providing visualization capabilities and advanced analysis features.

The provided hardware platform for this research is the SimpleLink™ MSP432P401R LaunchPad™ Development Kit. The microcontroller used by the development board is the MSP432P401R microcontroller that features an ARM Cortex-M4 processor running at 48MHz. The consumed power during active operation is 80 $\mu$A/MhZ and during standby operation is 660nA. Also, the microcontroller features a Floating Point Unit(FPU), a 64KB RAM memory, a 256KB Flash memory, four I2C modules, eight SPI modules and four UART modules. The microcontroller choice criteria are based mainly on the processor power consumption due to the power constraints on satellites. Also, the FPU makes the processor suitable for highly computational demanding applications such applications with a Kalman Filter.

As discussed in Section 2.2, most of the Multi-Agent Systems are implemented through Object-Oriented Programming Languages due to similarities in their concepts. Since any TI-RTOS applications can be written either in C or C++, the latter is used as this is an object-oriented programming language.

The framework provides an API to implement the Multi-Agent System components explained in the last section. The API is built up by four

main classes: the Agent class, the Agent Message class, the Agent Platform class and the Behaviour-related classes. Two additional classes are added to implement additional features: the User condition class and the Agent Organization class.

The User condition class is created for developers to implement their customized conditions to determine whether or not an AMS action shall be executed. By default, when the AMS agent is invoked by the requests of other agents, it always performs the action requested (for example: suspending a specific agent). However, there could be the case that the AMS operation can only be performed under certain conditions. For example: an agent can be de-registered only if it is in blocked state or an agent can only be killed if the the requester agent has a priority larger than a threshold value. As the conditions vary per application, the framework provides the User condition class that is a set of methods that the user can override in order to set their own conditions for the AMS to perform a certain operation. If these are not overriden, the methods always return `TRUE`. In consequence, the action is always performed. The AMS agent always look up to the return values of the methods of this class in order to determine whether or not the AMS action shall be executed.

On the other hand, the Agent Organization class groups the agents into specific ways such as modular (team) or hierarchical. Therefore, the communication among agents is constrained within the organization.

The following sections further explain the implementation of each class.

### 3.2.1 Agent Class

An instance of the Agent class contains variables associated with the agent information and variables associated with the agent's stack. These variables are declared as **private** members of the class in order to restrict access and to prevent unauthorized modifications. However, some classes such the Agent Platform, Agent Message and Agent Organization need access to these variables, hence, these are declared as friend classes of the Agent class. A class in C++ can access private and protected member of another class in which the first is declared as a `friend` class.

The agent's information is stored in a `typedef struct` as shown in Listing 1.

The `aid` variable corresponds to the `Task_Handle` or the Agent's AID. The `Task_Handle` is renamed into `Agent_AID` for consistency. Then, the `mailbox_handle` variable corresponds the mailbox assigned to the agent. The `AP` variable denotes the AMS agent's AID of the platform where the agent belongs to. The variables `org`, `affiliation` and `role` are Agent Organization-related variables.

It is mentioned in Section 3.1.1 that each task requires its own runtime stack. This stack is used for storing local variables and for storing the

```
typedef struct Agent_info{
    Agent_AID aid;
    Mailbox_Handle mailbox_handle;
    String agent_name;
    int priority;
    Agent_AID AP;
    org_info *org;
    int affiliation;
    int role;
}Agent_info;
```

Listing 1: Agent's information `typedef struct`.

context in case that the task is preempted. The TI-RTOS allows the user to create a stack by using dynamic memory allocation or by using static memory allocation. When a stack is created dynamically, the heap memory is used. If a task is destroyed, the user is required to free manually the task's stack, otherwise, a memory leak and memory fragmented might occur. Also, dynamic memory allocation is slower than static memory allocation. Thus, in order to prevent memory leaks and fragmentation, and to improve the system performance, the agent implementation is done by static memory allocation. Due to this approach, the designer is required to define the task stack and its size during the design phase of the application. These values are stored in a `typedef struct` as shown in Listing 2.

```
typedef struct{
    Agent_Stack *stack;
    int stackSize;
}Agent_resources;
```

Listing 2: Agent's information `typedef struct`.

In TI-RTOS, the stack is declared as an array of char. For consistency, the `char` type is redefined as `Agent_Stack` in the framework.

When the constructor of this class is called, the user passes as parameters: the agent's name, the agent's priority, the agent's stack pointer and the agent's stack size. The agent's stack size parameter is required as the size of the stack cannot be determined directly from the agent's stack pointer.

An agent object must be declared outside of the `main()` function since this must be in a persistent location. The agent's scope needs to be global since its information needs to be accessed from other classes for operations such as management or message exchange. As a consequence of the agent's

global scope, the stack also has to be defined globally. In order to prevent a memory stack corruption, the developer is responsible to allocate a stack per agent.

Although the agent is already constructed, the Agent's AID and the Mailbox Handle are still `NULL`. The task creation and mailbox creation processes are managed by the Agent Platform class.

### 3.2.2 Agent Platform Class

An instance of the Agent Platform class contains private member variables and methods to initialize the platform and to perform services.

The private members provide the information of the instance. The members are shown in Listing 3.

```
class Agent_Platform{
private:
    /*Class variables*/
    Agent agentAMS;
    Agent_Stack task_stack[4096];
    Agent_AID Agent_Handle[AGENT_LIST_SIZE];
    int subscribers;
    USER_DEF_COND cond;
    USER_DEF_COND *ptr_cond;

public:
...
};
```

Listing 3: The private members of an Agent Platform class object.

The `agentAMS` and `task_stack` variables are the AMS' Agent object and its stack, respectively. Then, the `Agent_Handle` is an index containing all the members' AIDs. The number of agents living in the Agent Platform is constrained by the definition of AGENT_LIST_SIZE macro. This value is chosen to be specified at compile-time in order to prevent the developer to overuse the memory resources available in the hardware platform. The `subscribers` variable denotes the number of active agents living in the platform.

Before the Agent Platform's execution is started, the developer must construct an instance of the Agent Platform class. Similar as an Agent object, an Agent Platform instance needs to be declared in a global location. There are two versions of the constructor: one **without** the user conditions and the other **with** the user conditions. If the user does not specify its own conditions, the `User Condition` pointer variable points by default to the

instance defined in the class. Otherwise, if the user constructs and overrides the methods of a `User Defined Condition` instance and sets the instance as a parameter in the Agent Platform Constructor, the pointer variable points to that object instead.

After the Agent Platform object construction, the developer must initialize each of the created agents using the Agent initialization method. This method creates the mailbox instance and the task instance associated with each agent. Therefore, a Mailbox_handle and an Agent_AID are also created during this method. TI-RTOS provides an additional environmental variable that can be used by the developer during the task creation to store a pointer to any `typedef struct`. Later, the value of the environmental variable can be retrieved from the Agent's AID/Task handle. For the framework implementation, a pointer to the Agent object is stored in the environmental variable. Therefore, the Agent_AID and the Agent instance are coupled internally since the Agent object has the AID information and from the Agent_AID, the pointer to the associated Agent instance can be retrieved. Initially, an agent is created with priority -1. This prevents the agent to execute before the Agent Platform is booted. The assigned priority is set until the agent is registered into an Agent Platform.

Once that all the agents are initialized, the developer must boot the Agent Platform using the Agent Platform boot method. This method creates and initializes the AMS agent. Also, the method registers all the previously initialized agents. The AMS agent is created with a stack of 4096 bytes and the priority is set to 31.

Unlike all the methods described so far, the AMS agent's wrapper function is hidden to the developer. The scope of this function is limited to the use-only within the Agent Platform class. This restriction is created in order to prevent any user's change on the AMS agent's services. With this restriction, the consistency with the services described in FIPA23 specification is ensured.

The flow diagram of the AMS agent's behaviour is shown in Figure 3.1.

The AMS agent's behaviour performs the actions described in Section 3.1.3. Also, this behaviour is designed so the agent can be either in `running` or `inactive` state. In the `running` state, the AMS agent is serving a request from an agent, while in the `inactive` state, the AMS agent is waiting for any incoming request.

The platform services are available once that the platform is initialized. The services are methods available to be used by the agents living in the platform, but, some of them are restricted for the AMS agent use-only. In order to distinguish the services, these are classified into two categories: public services and private services. The private services can only be performed by the AMS agent, while the public services can be accessed by any caller agent. Further information of the Agent Platform class' methods is found in Appendix B.

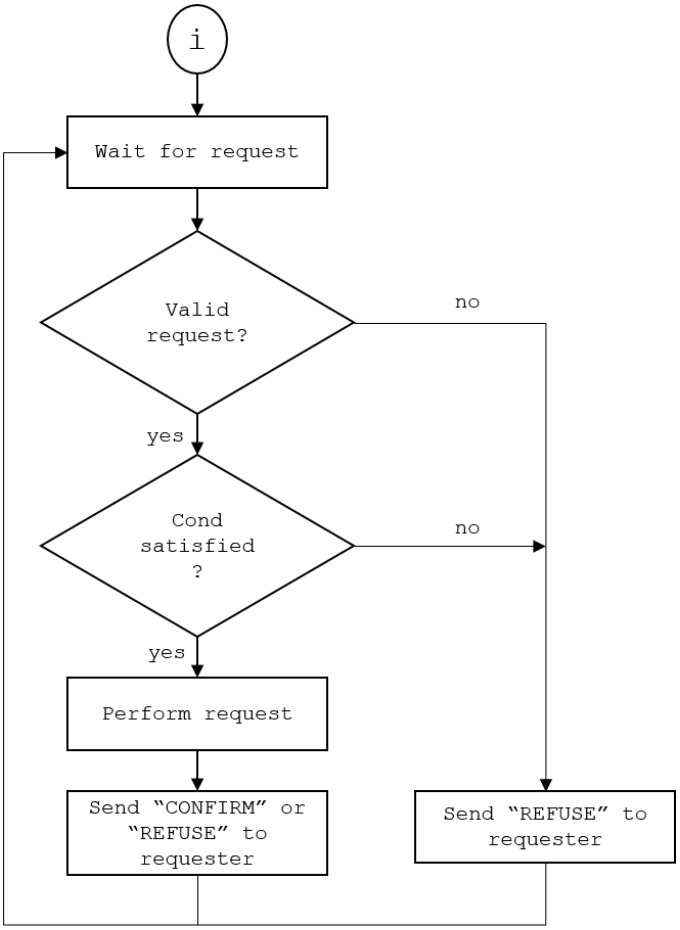


Figure 3.1: AMS agent behaviour.

### 3.2.3 Agent Organization Class

The Agent Organization class allows the developer to create an organization that groups the agents into one of the following topologies: Hierarchy and Team (Modular). Without an Agent Organization, a Flat structure is implemented by default in which there is no restriction on agents communication. In a Hierarchy organization, there are several subordinates and one supervisor (also acting as the moderator and the administrator of the organization). The subordinates can only communicate with the supervisor and the communication outside of the organization (such as with other agents or with the AMS agent) can only be performed by the supervisor. In a Team organization, there is no restriction in the communication between the agents members (unless the moderator explicitly restricts the conversation of a specific member). However, only the administrator or the organization creator can communicate with the agents outside of the organization.

Beside grouping the agent into a specific topology, the class also has permissions to modify the variables `affiliation`, `role` and `org` of an Agent instance (See Listing 1). An instance of the Agent Platform class has no visibility over the Agent Organization instances. Therefore, an Agent Organization instance is decoupled from the Agent Platform. This approach minimizes the communication overhead that would result if the AMS agent decides whether or not to create the organization.

The Agent Organization is described by the `typedef struct org_info`. This variable is defined in Listing 4.

```
typedef struct org_info{
    enum org_type;
    int members_num;
    int banned_num;
    Agent_AID members[AGENT_LIST_SIZE];
    Agent_AID banned[AGENT_LIST_SIZE];
    Agent_AID owner;
    Agent_AID admin;
    Agent_AID moderator;
}org_info;
```

Listing 4: Message format `typedef struct`.

The variable `org_type` denotes the organization type of the instance. The possible organization types are: `HIERARCHY` or `TEAM`. The `members_num` variable describes the number of members in the organization and the `banned_num` variable describes the number of banned members. There are also lists containing all the members' AID and all the banned agent's AID. Finally, there are three `Agent_AID` variables that correspond to the owner, admin and moderator of the organization.

Although this class defines the relationship of the agent with an organization, the communication exchange is managed by the Agent Message class. According to the organization characteristic of an agent, the Agent Message class determines whether or not a message can be exchanged.

Further information of the Agent Organization class' methods is found in Appendix B.

### 3.2.4 Agent Message Class

The communication method of each agent is set up by the Agent initialization method from the Agent Platform class. During the initialization, the mailbox is created and assigned to the associated Agent object. However, the message object to be passed during an exchange is not constructed

yet. Although the message object can be constructed during the Agent initialization or Agent construction, a separate class was created in order to encapsulate the methods related to an agent's communication exchange. Therefore, the framework provides methods for the message object creation and management through the Agent Message class.

An instance of the Agent Message class contains the message object to be exchanged between two agents and the methods to manipulate the object. The message object is defined by the `typedef struct` shown in Listing 5.

```
typedef struct MsgObj{
      Agent_AID sender_agent;
      Agent_AID target_agent;
      MSG_TYPE type;
      String content;
}MsgObj;
```

Listing 5: Message format `typedef struct`.

The message format is composed by the sender's AID, the receiver's AID and the message type. Also, the struct features a `content` variable that allows the user to set the payload of the message. The message type is set to any value of the `enum` type `MSG_TYPE`: `INFORM, CONFIRM, REQUEST`, etc. These values are based on the message type established by the FIPA69 specification [57]. The `MsgObj` object is declared as a private member in the Agent Message class as this prevents unauthorized manipulation. Additionally, the instance also maintains three private variables: the agent's AID who created the object, the number of receivers and an index of the receivers' AID of the message.

Also, the Agent Message class determines whether or not a message can be exchanged according to each agent's organization characteristics. Further information of the Agent Message class' methods is found in Appendix B.

### 3.2.5 Behaviour-related classes

As mentioned in Section 3.1.1, an agent's behaviour is encapsulated as a subroutine into a function wrapper. Several behaviours can be implemented as separate subroutines in the wrapper function and its execution order is determined by the developer by using patterns such as Finite State Machine. The framework facilitates classes that can be used by the developer to implement customized behaviours.

To implement generic-type behaviours, the developer can create an instance of the Generic Behaviour class. The class provides methods that can be overwritten by the user: `setup()`, `action()` and `done()`. Optionally,

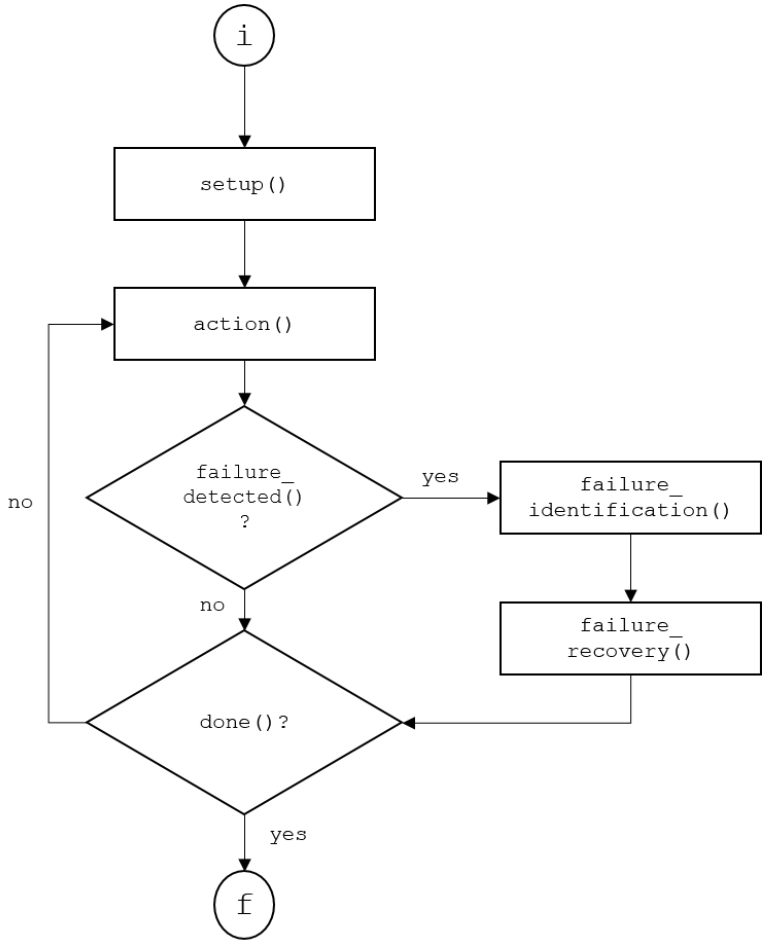


Figure 3.2: Generic behaviour execution flow diagram.

three methods for Failure Detection, Identification and Recovery (FDIR) are included for user overriding. The FDIR methods' goal is to detect faults and identify the origin of the fault in the shortest time possible. Therefore, reducing the diagnostic time and increasing the system availability [58]. With the FDIR methods, the autonomy is increased for all the agents by allowing each of them to detect, identify and recover from its own failure instead of having a centralized managing authority (such the AMS agent) to perform those actions.

Finally, the class provides the method `execute()` that executes the above-mentioned methods according to the diagram flow of Figure 3.2.

In the `setup()` method, the user may include any initialization-related code. The method `action()` comprises the main behaviour's computation code. Then, the method `failure_detected()` checks if there is a failure occurred. If there is a failure, the methods `failure_identification()`

and `failure_recovery()` are executed. Lastly, the `done()` method returns `TRUE` if any condition set by the developer is met. When condition is met, the behaviour's execution is finalized.

When an object of the Generic Behaviour class is created, an Agent Message object instance is also created. Thus, the message object is already built-in into an instance of this class.

Additionally, the framework provides two customized behaviour classes derived from the Generic Behaviour class: `One-Shot Behaviour` class and `Cyclic Behaviour` class. For `One-Shot Behaviour`, the method `done()` is designed to return `TRUE` always, so the `action()` method executes only once. As for the `Cyclic Behaviour`, the `action()` always returns `FALSE`, so the `action()` method executes repetitively.

To use the behaviour instance, the developer only needs to call the method `execute()` from the wrapper function. The wrapper function is required to encapsulate the class method as it is not possible to pass a class member as a parameter to TI-RTOS's function Task_create().

# Chapter 4

# MAES Framework verification

One of the requirements described in Section 1.3 specifies that the MAES framework should be FIPA-based, i.e., the framework should contain the minimum mandatory components specified by the FIPA-23 [35] specification (see Figure 2.1). Table 4.1 demonstrates that each of the mandatory components is implemented accordingly through different classes provided in MAES.

The *Agent* component is implemented by using three classes: the Agent class, the Generic Behaviour class and the Agent Organization class. The Agent class contains the agent description: AID, name and organization relationship (role and affiliation). The Generic Behaviour class encapsulates the computation code of an agent's behaviour, and the Agent Organization class groups the agent into a pre-determined communication topology (Hierarchy or Team).

The *Agent Platform* component is implemented through the Agent Platform class. An instance of this class provides services to the agents living in the platform such as search agent and get agent's description.

The *Agent Management System* is created when an instance of the Agent

Table 4.1: Class implementation for each component.

| **FIPA component** | **Class** |
|---|---|
| Agent | - Agent Class<br>- Generic Behaviour Class<br>- Agent Organization Class |
| Agent Platform | - Agent Platform Class |
| Agent Management Service | - Agent Platform Class |
| Message Transport Services | - Agent Platform Class<br>- Agent Message Class |

Platform class is booted. The AMS oversees the lifecycle of each agent living in the platform. Therefore, the AMS is the only agent that can perform lifecycle related operations (kill, register, modify, suspend, resume).

Lastly, the *Message Transport Service* is implemented through the Agent Platform class and the Agent Message class. The Agent Platform class creates the mailbox associated with each agent, while the Agent Message class creates the object to be exchanged among the agents.

Following the components verification, a static library `MAES.lib` was created and used in several test applications described in the current and the next chapter. Section 4.1 and Section 4.2 describe two applications that verify the basic functionalities of the framework: communication, agent organization and agent's failure detection, identification and recovery features. For this, the RTOS Analyzer's Execution Analysis tool from Code Composer Studio is used in order to visualize the agents/threads execution and interactions in TI-RTOS. Then, in Section 4.3 the framework functionality is compared with a known framework: JADE. The purpose of this test is to demonstrate that an algorithm implemented in MAES behaves similarly as the algorithm implemented in JADE. Also, a performance analysis is conducted for both frameworks.

## 4.1 Blink LED Application

The main purpose of the application is to verify the communication among the agents on two different scenarios. For each scenario, an execution trace from the Execution Analysis tool is obtained and analyzed. The proposed scenario descriptions and their outcome are listed below:

1. Simple two-agents communication: This scenario consists of a writer agent and a LED agent. The writer agent is assigned priority 1 and the LED agent is assigned priority 2. The writer agent is preempted when a message is sent to the LED agent, as the latter has a higher priority than the writer agent. The execution of this scenario is shown in Figure 4.1.

   On mark 1 of Figure 4.1, the writer is woken up and it preempts the idle task. Then, the writer agent posts a message and it is preempted by the LED agent (mark 2). The LED agent switches the LED and suspends itself until a new message arrives. On mark 3, the scheduler takes control of the processor. The reason of why the scheduler is seen in the execution trace is because there are still threads (Writer Agent and Idle thread) in which execution have been preempted by a higher priority thread and the scheduler is determining which preempted thread shall be executed next.

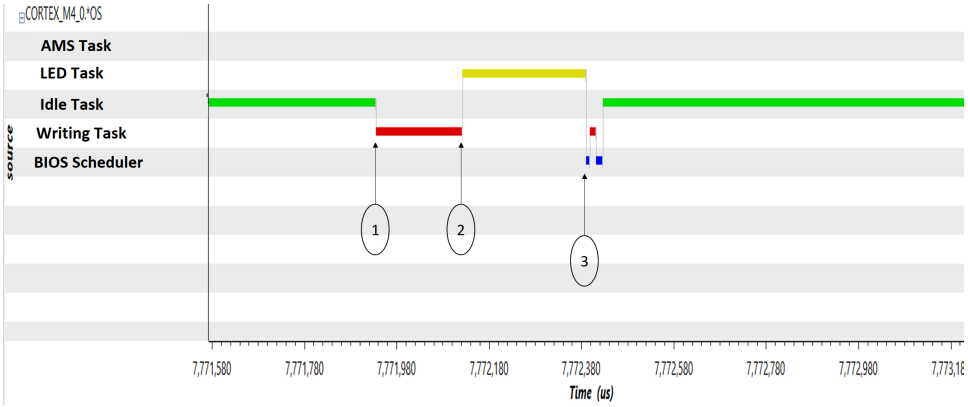


Figure 4.1: The execution graph of two-agents communication.

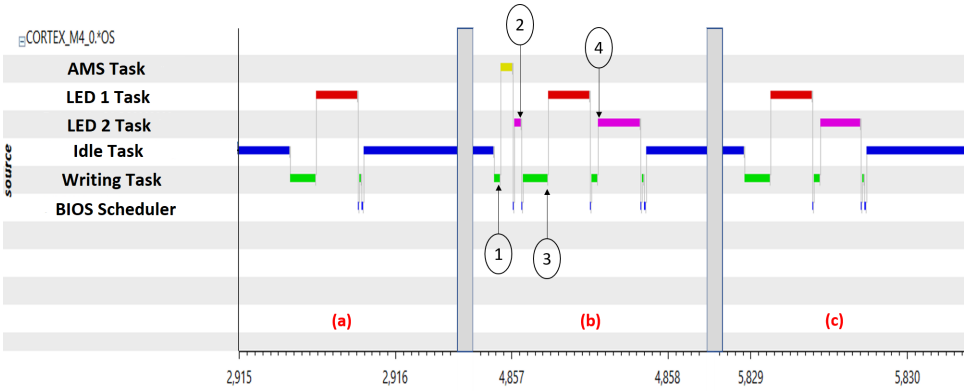


Figure 4.2: The execution graph of Agent Multicasting/AMS.

2. Agent multicasting/AMS: This scenario demonstrates the communication between an agent with the AMS agent and also demonstrates an agent multicasting to several agents. The scenario consists of a writer agent (priority 1) and two LED agents (priority 2). First, during the first 5 seconds, the writer agent posts a message to a LED agent. When 5 seconds have elapsed, the writer agent sends a request to the AMS to register a second LED agent. After the second LED agent registration, the writer agent posts to both LED agents. The execution is shown in Figure 4.2.

   The part (a) of Figure 4.2 shows the same behaviour as the "Simple two-agents communication" scenario as there is one active LED agent. In part (b), approximately about 5 seconds after the application has started, the writer agent sends a message request to the AMS agent to register a

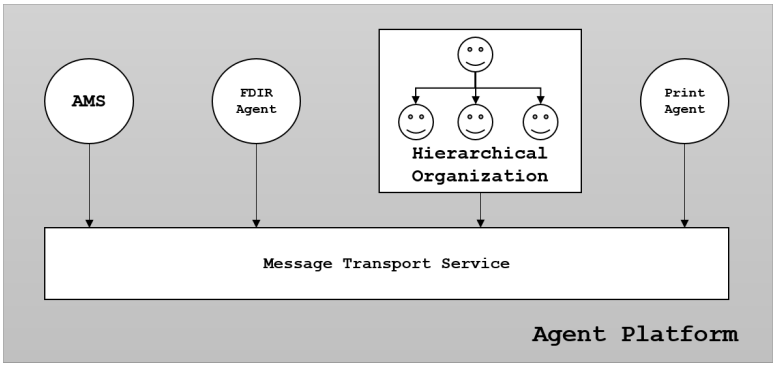


Figure 4.3: The architecture of the demonstration application.

second LED as shown on mark 1. The AMS agent preempts any threads since it has the largest priority of the system. Then, after the AMS agent execution (mark 2), the second LED agent runs as it has a priority higher than the writer agent. The second LED agent executes the initialization steps and suspends itself waiting for an incoming message. On mark 3, the writer agent takes control again and posts a message to the first LED agent. Lastly, on mark 4, after the first LED agent has toggled the LED, the writer agent takes back control and posts a second message to the second LED agent. The third part (c) shows a normal agent multicasting behaviour. As mentioned before, the scheduler is executed as it determines which of the preempted threads is executed next.

The communication in both scenarios worked as expected since the LED agents and the AMS agent received the messages and executed their functions correctly. If the communication is not executed properly, the `send()` method from the Agent Message class returns an error code.

## 4.2 FDIR/Agent Organization application

The purpose of this application is to demonstrate the capabilities of the framework for Fault Detection, Identification and Recovery and for Agents Organization. The application consists of six agents. The architecture is shown in Figure 4.3.

The FDIR Agent is linked to a button connected to the GPIO port of the development board. When pressed, a failure of the agent is simulated. Then, the agent increments an internal counter and proceeds to execute the FDIR methods described in Section 3.2.5. When five failures have occurred, the agent sends a restart request to the AMS in order to restart its execution. The FDIR agent sleeps for 500 milliseconds if no failure is detected.

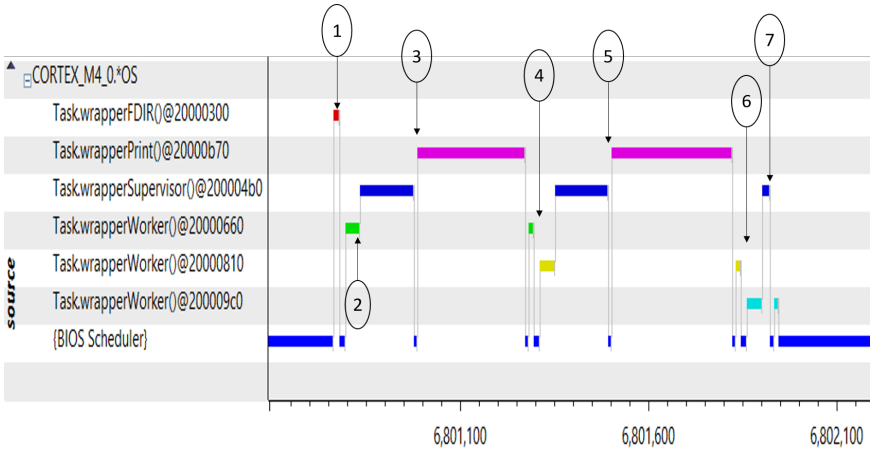


Figure 4.4: The execution graph of the demonstration application(1).

Then, four agents are grouped into a hierarchical organization. One agent is designated as the Supervisor (this agent is also the creator/administrator/-moderator) of the group and three "Worker" agents. Each of the "Worker" agent simulates a thermocouple and sends the simulated readings to the Supervisor at a specific rate. If the Supervisor detects a temperature more than 70 degrees Celsius, it sends a message to the Print Agent (free-agent) that prints the warning to the user. The Supervisor agent and Print agent are assigned priority 3, whereas the Worker agents are assigned priority 2.

Similar to Section 4.1, the RTOS Analyzer's Execution Analysis tool is used in order to visualize the agents/threads execution in TI-RTOS. The Figure 4.4 shows an extract of the execution graph obtained from the application.

The following list explains the execution trace:

- Mark 1: The FDIR agent executes and does not detect any fault. Therefore, it sleeps for 500 millisecond.
- Mark 2: The first Worker executes and sends the thermocouple value to the Supervisor agent. The first Worker is preempted by the Supervisor agent.
- Mark 3: The Supervisor agent posts a message to the Print agent as the first worker's value is above 70 degrees Celsius.
- Mark 4: The first Worker resumes and finishes its execution. Then, the second Worker starts its execution and sends the value to the Supervisor agent.
- Mark 5: The Supervisor agent posts a message to the Print agent as the second Worker's value is above 70 degrees Celsius.

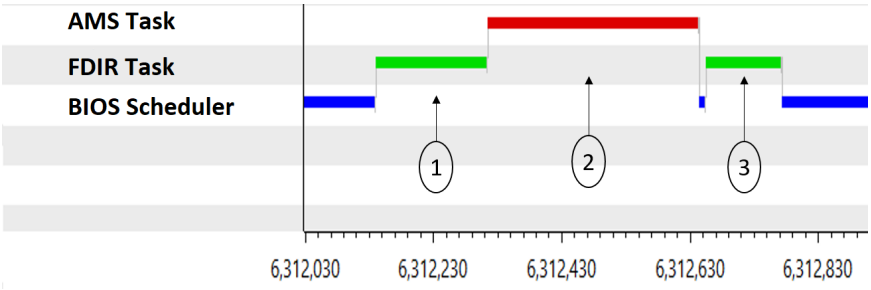


Figure 4.5: The execution graph of demonstration application(2).

- Mark 6: The second Worker resumes and finishes its execution. Then, the third Worker starts its execution and sends the value to the Supervisor agent.

- Mark 7: The Supervisor agent does not post a message as the third Worker's value is below 70 degrees Celsius.

As mentioned above, if five failures have been detected, the FDIR agent sends an AMS request to restart its execution. The execution trace of this exchange is seen in Figure 4.5.

On mark 1, the FDIR agent executes and sends a restart request to the AMS agent as five failures have occured. On mark 2, the AMS agent is woken up and executes the request. Lastly, on mark 3, the FDIR agent resumes its execution.

The communication between agents of the same organization and with the AMS agent are tested successfully as the AMS agent, the Supervisor agent and the Print agent execute correctly their functions when these are invoked by other agents as shown in Figure 4.4 and 4.5. As explained in Section 4.1, if there is a communication irregularity, an error would return from the method `send()` of the Agent Message class.

## 4.3 Attitude Determination application

In order to verify the functionality of the MAES framework against a known-Multi-Agent framework (JADE), the quaternion-based Extended-Kalman filter (EKF) algorithm described by A.M Sabatini [59] is proposed as a reference application. This algorithm was selected since it is a fundamental/common tool implemented in aerospace applications [60].

The goal of the EKF algorithm is to determine the orientation of a rigid body by using a gyroscope. The algorithm corrects the measurement value by using the values of two aiding sensors: an accelerometer and a magnetometer. The gyroscope measurements are used in the EKF's prediction, while the aiding sensors measurements are used in the EKF's estimation.

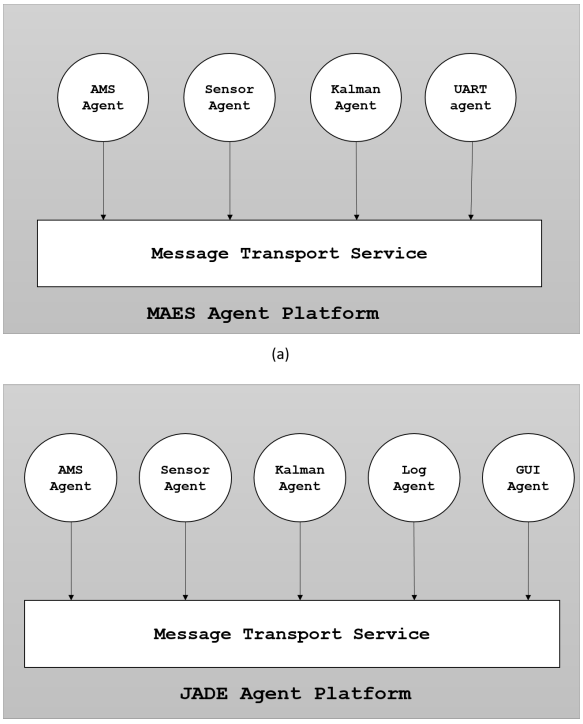


Figure 4.6: The implementation of the attitude determination application: (a) MAES implementation (b) JADE implementation.

The algorithm's output is a quaternion that consists of a 4-tuple that describes the orientation of a rigid body. The algorithm's expressions and initial conditions are further described in Appendix A.

In order to implement the algorithm into the LaunchPad, an additional plug-in module is connected: the Sensors BoosterPack Plug-In Module. This module contains a light sensor, an infrared sensor, an Inertial Measurement Unit (IMU) (featuring an accelerometer and gyroscope), a magnetometer and an environmental sensor [61]. For the purpose of this research, only the IMU and the magnetometer are used. The architectures for both implementations are shown in Figure 4.6.

On both applications, the sensor agent acquires the measurement data and posts a message to the Kalman agent. Then, the Kalman agent proceeds to execute the filter code. When the quaternion is available, the Kalman agent posts a message to the UART Agent (for the MAES implementation) or to the Log and GUI agents (for the JADE implementation).

Once that the application is defined, two experiments are conducted in order to compare both framework's functionality for the same reference application. The first experiment consists of comparing the quaternions obtained

from both frameworks. On the second experiment, the execution time for the EKF algorithm is obtained from both implementations. Further details of each experiment will be given in the following sections.

### 4.3.1 Algorithm's output comparison experiment

The first experiment intends to demonstrate that the algorithm implemented in the MAES framework outputs the same value in the JADE framework. In order to compare the outputs, a dataset is obtained from each framework. Both frameworks are fed by the same sensors's measurement (obtained at a 10Hz sampling rate) and the obtained datasets are composed of the quaternions. Five pairs of datasets are acquired for each of the scenarios explained below:

- Neutral position: 600 samples are acquired in a static position.
- Moving around the yaw axis: acquired 600 samples.

The MAES dataset is compared against the JADE dataset for each scenario by using the "Two-Sample t-test" method [62]. The result from this test accepts or rejects a null hypothesis. The null hyphothesis states that there is no significant difference between the datasets and any observed difference are due to experimental error. The "Two-Sample t-test" results are obtained via `MATLAB` by using the `ttest2` command by using both datasets as parameters. The command returns as results: the result of the null hypothesis (returns 0 if the null hypothesis is accepted, otherwise it returns 1), the *p-value* and the confidence interval. The *p-value* is the probability under a specified statistical model that the mean would be equal to or more extreme than its observed value [63] (a low *p-value* provides enough evidence to reject the null hypothesis). On the other hand, the confidence interval (ci) provides the range of values that it is likely to contain the difference between MAES value and JADE value.

The results for each scenario is shown in the next page in Table 4.2 and Table 4.3 [1].

As shown in these tables, the confidence interval is at least of the order of $1x10^{-2}$. This indicates that the difference between both dataset is small. Since the command returns h=0, it can be concluded that both datasets are statistically equivalent and the functionality of the algorithms is the same.

### 4.3.2 Algorithm's execution time comparison experiment

The second experiment consists of comparing the execution time of the EKF algorithm in both frameworks. The execution time is obtained from the difference between two variables placed specifically in different parts of the

[1] For all the tests, the command returns h=0. Hence, this parameter is not shown in these tables

Table 4.2: The `ttest2` outputs for Neutral Position scenario.

| | **Test 1** | **Test 2** | **Test 3** | **Test 5** | **Test 5** |
|---|---|---|---|---|---|
| **q1** | p=0.9459 ci= [-0.1884e-4, 0.2020e-4] | p=0.9512 ci= [-0.2174e-4, 0.2313e-4] | p=0.9987 ci= [-0.2792e-4, 0.2788e-4] | p=0.8584 ci= [-0.3718e-4, 0.3098e-4] | p=0.7647 ci= [-0.2079e-4, 0.2827e-4] |
| **q2** | p=0.8869 ci= [-0.4470e-4, 0.3865e-4] | p=0.9116 ci= [-0.5192e-4, 0.4635e-4] | p=0.8741 ci= [-0.5747e-4, 0.4888e-4] | p=0.9006 ci= [-0.5662e-4, 0.6432e-4] | p=0.9646 ci= [-0.4753e-4, 0.4543e-4] |
| **q3** | p=0.9861 ci= [-0.1776e-3, 0.1745e-3] | p=0.9920 ci= [-0.4149e-3, 0.4107e-3] | p=0.9702 ci= [-0.1475e-3, 0.1532e-3] | p=0.6192 ci= [-0.1534e-3, 0.0914e-3] | p=0.7826 ci= [-0.1366e-3, 0.1029e-3] |
| **q4** | p=0.9614 ci= [-0.5875e-6, 0.5592e-6] | p=0.9767 ci= [-0.3060e-5, 0.2970e-5] | p=0.8571 ci= [-0.3819e-6, 0.3177e-6] | p=0.7181 ci= [-0.2104e-6, 0.3054e-6] | p=0.9331 ci= [-0.2211e-6, 0.2029e-6] |

Table 4.3: The `ttest2` outputs for Yaw scenario.

| | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** |
|---|---|---|---|---|---|
| **q1** | p=0.6693 ci= [-0.0010, 0.0007] | p=0.9456 ci= [-0.0012 0.0011] | p=0.9142 ci= [-0.0012, 0.0011] | p=0.6770 ci= [-0.4479e-3, 0.2910e-3] | p=0.8724 ci= [-0.2375e-3, 0.2015e-3] |
| **q2** | p=0.7156 ci= [-0.8668e-3, 0.5952e-3] | p=0.7559 ci= [-0.2795e-3 0.3848e-3] | p=0.9287 ci= [-0.8666e-3, 0.7910e-3] | p=0.8113 ci= [-0.2139e-3, 0.1675e-3] | p=0.8701 ci= [-0.1745e-3, 0.2062e-3] |
| **q3** | p=0.9520 ci=[-0.0307, 0.0289] | p=0.9591 ci= [-0.0336, 0.0319] | p=0.9049 ci= [-0.0279, 0.0315] | p=0.9680 ci= [-0.0283, 0.0272] | p=0.9795 ci=[-0.0295, 0.0287] |
| **q4** | p=0.9873 ci=[-0.0046, 0.0047] | p=0.9830 ci= [-0.0056, 0.0055] | p=0.8348 ci= [-0.0041, 0.0051] | p=0.9582 ci= [-0.0034, 0.0032] | p=0.9789 ci= [-0.0042, 0.0041] |

code; the initial time value is set when new measurement enters the filter, while the final time value is set after the quaternion is output. The execution time of the algorithm is measured each time that a new data enters. The data is logged during 5 minutes at 10Hz sampling rate (3,000 samples

Table 4.4: The mean and variance of the execution time from both frameworks.

| | **MAES Framework** | **JADE Framework** |
|---|---|---|
| **Min (ms)** | 2.564792 | 0.054312 |
| **Max(ms)** | 2.593562 | 64.982879 |
| **Mean(ms)** | 2.574432 | 0.182392 |
| **Variance($[ms]^2$)** | 0.000023 | 2.328159 |

are recorded). The mean and variance of both frameworks' execution time dataset are shown in Table 4.4.

As seen in Table 4.4, in average the JADE Framework executes the Kalman Filter algorithm approximately 14 times faster than the MAES Framework. This is mainly due to the processor difference on both frameworks: JADE runs on a 64-bit Intel i7-5500U CPU at 2.40GHz, while MAES runs on a 32-bit ARM Cortex M4F at 48MHz. The factor between the frameworks' best times (minimum values) is approximately 47 times (Intel i7 runs 50 times faster than ARM).

On the other hand, the variance is approximately 100,000 times lower in the MAES Framework. The difference in the variance is because JADE runs on top of a Java Virtual Machine (JVM) in Windows 10 (a General Purpose Operating System), whereas MAES runs on top of TI-RTOS. JADE's execution time varies on each call as the processor might be busy with other processes, while MAES' execution time is consistent due to the determinism of the TI-RTOS. Figure 4.7 shows that MAES' execution time duration is almost constant over 300 seconds, while in JADE there are several peaks. Furthermore, Figure 4.8 shows that the algorithm's execution time is widely spread in JADE, while it is concentrated in MAES. Even though that the algorithm executes faster in JADE on average, JADE cannot guarantee that on each call the algorithm execution time will be consistent. On the other hand, the execution time in MAES is consistent as it lies on top of a RTOS that ensures predictable execution pattern or deterministic behaviour.

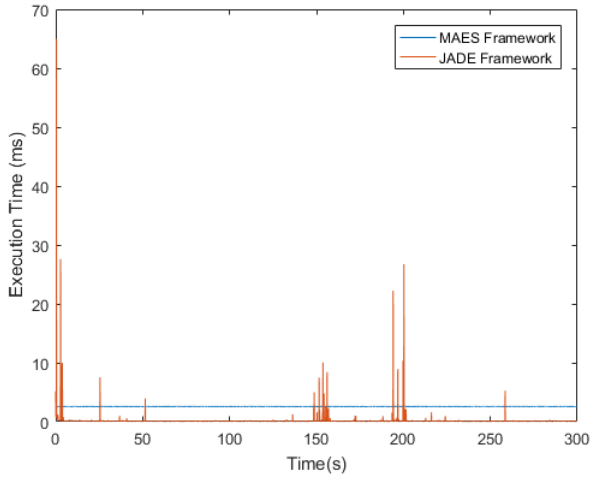


Figure 4.7: The execution times of the Kalman filter during 300 seconds.

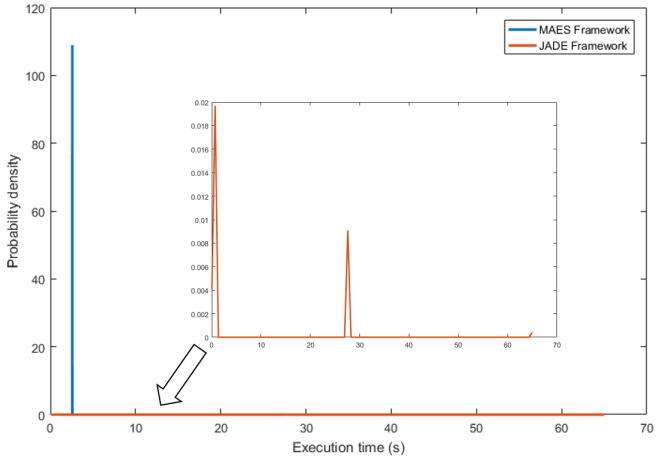


Figure 4.8: The probability density function for both frameworks[2].

[2]A probability density function can take values larger than 1. However, the result of the integral of the function over the time should be 1.

# Chapter 5

# MAES Framework benchmark

One of the research questions defined in Chapter 1 states: *"How does the mapping strategy affect the CPU's memory, load and power consumption?"*. In this chapter, several benchmark experiments are conducted in order to answer that research question.

For this, four applications are used in order to show the memory performance, CPU utilization and the power consumption of the MAES Framework. Additionally, these applications are also implemented without the MAES framework. Then, a comparison of the results between the implementation with MAES against its without-MAES counterpart on the same execution platform is described. The following list describes the used applications:

- The Blink LED Application: This application is described in Section 4.1 for Agent multicasting/AMS scenario. For its non-agent implementation, the signaling is implemented using semaphores since the communication does not require message exchange.
- Attitude Determination Application: This application is described in Section 4.3 and its non-agent counterpart's signaling is implemented by using semaphores.
- Telemetry Logger Application: This application simulates a telemetry logger of a satellite. The application has four functions: a current logger, a voltage logger, a temperature logger and a measurement function. Each of the loggers is woken up at a specific rate and posts a message to the measurement function to request a measurement. The measurement function sends the data back to the logger that outputs the value to the UART interface.
- Command and Data Handling System Application: This application simulates a command and data handling system (CDHS) of a satellite. The

Table 5.1: The code size for each class.

| Class | Size (bytes) |
|---|---|
| Agent Platform | 2,364 |
| Agent Msg | 1,280 |
| Agent Organization | 1,738 |
| Agent | 236 |
| Behaviour | 136 |
| User-predefined-condition | 72 |
| **Total** | **5,826** |

application has the following functions: a mission function, two LED functions (LED toggling action) and a sensor function (retrieves gyro/accelerometer/magnetometer measurements). The mission function receives commands from the user through the UART port. According to the command, the mission function signals either to the sensor function or to any of the LED functions. The commands `comm_led0` and `comm_led1` signal the LED0 function and the LED1 function, respectively. On the other hand, the commands `comm_gyro`, `comm_accel`, `comm_magn` signal the sensor function. For its non-agent implementation, the signaling is implemented using semaphores. Lastly, in order to benchmark this application, a Python script is developed to automate the command sending.

The benchmark tests are explained in further detail in the following sections. In order to achieve consistency with the results, all the experiments are run on the same LaunchPad board and were implemented in the same environment with the same compiling tools.

## 5.1 Memory performance

During the compilation and linking process, a memory map file is generated that establishes how the memory is allocated. The LaunchPad includes 256KB Flash memory and 64KB SRAM memory and its memory allocation and usage can be visualized by the Memory Allocation tool from Code Composer Studio.

The code segment size for each class is determined from the Flash Memory view tool; the results are shown in Table 5.1.

As observed, when the MAES library is used almost 6KB are additionally allocated in the Flash memory. Table 5.2 shows the `main()` function size per application.

As seen in Table 5.2, the `main()` function in the non-agent implementations requires more memory than the MAES implementation. This increment is because the non-agent implementations require additional coding to create the tasks and the signaling mechanisms (semaphore or mailbox),

Table 5.2: The `main()` function code size per application.

| Application | Without MAES (bytes) | With MAES (bytes) | % Decrease |
|---|---|---|---|
| **Blink LED** | 252 | 120 | 52.39 |
| **Attitude Determination** | 232 | 152 | 34.48 |
| **Telemetry Logger** | 444 | 236 | 46.85 |
| **CDHS** | 344 | 156 | 54.65 |

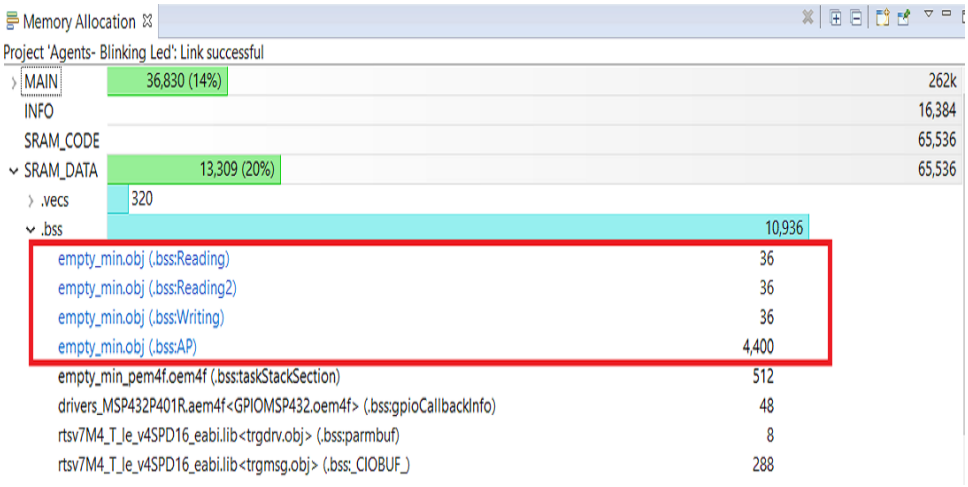


Figure 5.1: The memory allocation view extract for Blink LED application.

while in the MAES implementation, these routines are already encapsulated in the method `agent_init()` of the Agent Platform class (see Section 3.2.2). Therefore, a MAES implementation requires less user coding, but the memory usage is incremented as a result of the usage of the framework's classes.

Moreover, the SRAM memory contains all the statically allocated data such the task/agent stack. When the MAES framework is used, additional memory is also allocated. It is seen from the applications that an Agent Platform object requires **4400 bytes** and an extra agent object costs **36 bytes**. Figure 5.1 shows an example of the additional SRAM memory space required for an agent object and for an Agent Platform object in the Blink LED application.

Next, Table 5.3 shows the memory usage for the application on both implementations. The static memory allocation from the applications implemented with the MAES framework is larger than its counterpart due to

Table 5.3: The total memory allocation per application.

| Application | | Without MAES (bytes) (% memory used) | With MAES (bytes) (% memory used) |
|---|---|---|---|
| **Blink LED** | **Flash** | 30,984(11%) | 36,886(14%) |
| | **SRAM** | 8,809(13%) | 13,309(20%) |
| **Attitude** | **Flash** | 67,842(25%) | 75,236(28%) |
| **Determination** | **SRAM** | 21,034(32%) | 25,546(38%) |
| **Telemetry** | **Flash** | 43,726(16%) | 50,022(19%) |
| **Logger** | **SRAM** | 16,501(25% ) | 21,005(32%) |
| **CDHS** | **Flash** | 67,360(25% ) | 75,186(28%) |
| | **SRAM** | 19,743(30%) | 24,275(37%) |

the additional MAES code and objects (see Table 5.1 and Figure 5.1). The increment in the Flash memory is 6.7 KB in average and in the SRAM memory is 4.5 KB in average when the MAES Framework is used.

## 5.2 CPU Utilization

In order to measure the CPU utilization for each application, the RTOS Analyzer's Load Analysis tool from Code Composer Studio is used to capture the average CPU utilization for each task/agent. The tool loads the data by default at 2Hz. For better resolution, the sample rate was set to 10Hz and the data was acquired during 5 minutes. The average CPU utilization of the applications on MAES implementation and its non-agent counterpart are shown in tables 5.4, 5.5, 5.6 and 5.7.

Table 5.4: The CPU Utilization per function for the Blink LED application.

| Source | Without MAES | With MAES |
|---|---|---|
| **Reading function** | 0.01% | 0.02% |
| **Reading2 function** | 0.01% | 0.02% |
| **Writing function** | 0.01% | 0.02% |
| **Idle function** | 99.98% | 99.94% |

Table 5.5: The CPU Utilization per function for the Attitude Determination Application.

| Source | Without MAES | With MAES |
|---|---|---|
| **Kalman function** | 2.43% | 2.44% |
| **Sensor function** | 0.26% | 0.28% |
| **UART function** | 1.60% | 2.16% |
| **Idle function** | 95.71% | 95.11% |

Table 5.6: The CPU Utilization per function for the Telemetry Logger Application[2]

| Source | Without MAES | With MAES |
|---|---|---|
| **Measurement_gen function** | 0.07% | 0.07% |
| **Logger_amp function** | 0.05% | 0.06% |
| **Logger_volt function** | 0.02% | 0.03% |
| **Logger_temp function** | 0.01% | 0.02% |
| **Idle function** | 99.84% | 99.83% |

Table 5.7: The CPU Utilization per function for the CDHS Application[3]

| Source | Without MAES | With MAES |
|---|---|---|
| **Mission function** | 0.02% | 0.05% |
| **LED0 function** | 0.00% | 0.00% |
| **LED1 function** | 0.00% | 0.00% |
| **Sensor function** | 0.02% | 0.05% |
| **Idle function** | 99.95% | 99.89% |

As seen in these tables, there is an increment of the CPU load in the functions of the MAES implementation. The additional load is due to the communication method implemented in MAES since it not only implements

[2]The current logger is set to 500ms rate, the voltage logger is set to 5 seconds and the temperature logger is set to 10 seconds.

[3]Due to the sampling rate of the Load Analysis tool (100ms), the LED0 and LED1 functions' utilization could not be captured.

Table 5.8: The average duration for different communication methods.

| | **Average duration ($\mu$s)** | **Average duration (cycles)** |
|---|---|---|
| **Semaphore Post/Pend** | 8.188 | 393 |
| **Mailbox Post/Pend** | 17.412 | 836 |
| **Agent_msg Send/Receive** | 27.375 | 1,314 |

the mailbox module, but also, additional instructions (to check the recipient validity) are enclosed in the method `send()` from the Agent Message class (see Section 3.2.4). Table 5.8 shows the average duration for each communication method.

As shown in Table 5.8, the semaphore post/pend pair is the most efficient since it takes fewer cycles to execute. Then, an increase of approximately 112% is obtained by using a mailbox since this module not only has a signalling mechanism, but also, transfers data. Lastly, although the `Agent_msg` Send/Receive methods implement the Mailbox module, this methods have additional instructions to verify the recipient validity. Therefore, this increases the average duration in cycles for this communication method. As a result, the CPU load is also increased. From this benchmark, it can be seen that MAES framework represents an additional CPU utilization as a result of the communication method implemented.

## 5.3 Power Consumption

In order to verify the framework's impact on the power consumption, the power profile from the application implemented with MAES is compared against its non-agent implementation. The power profile is obtained by using the EnergyTrace™Technology analysis tool from Code Composer Studio. This technology measures the current used by the microcontroller. The power is obtained by assuming an ideal power supply of 3.3V, i.e., the tool does not take into account the temperature, aging or other factors that could negatively affect the power supply.

Each of the applications (both MAES implementation and non-agent implementation) was run five times with each run lasting 5 minutes. After running the application, all the data from each run was parsed into a single file. Next, the MAES implementation datafile is compared against its counterpart by using the "Two-Sample-t-test" method (see Section 4.3.1). The result of this comparison is obtained by using `Matlab` and these are shown in Table 5.9.

Table 5.9 shows that the null hypothesis is rejected as h=1. Thus, the data obtained from the MAES implementation and the data obtained from the non-Agents implementation are **not** statistically equivalent. Next, the

Table 5.9: The `ttest2` result per application.

| Application | `ttest2` result |
|---|---|
| Blink LED | h =1<br>p =3.7e-98 |
| Attitude Determination | h =1<br>p =0 |
| Telemetry Logger | h =1<br>p =1.2e-18 |
| CDHS | h =1<br>p =0 |

Table 5.10: The mean power consumption for each application.

| Application | Without MAES (mW) | With MAES (mW) | Difference |
|---|---|---|---|
| **Blink Led** | 152.60 | 152.85 | 0.160% |
| **Attitude Determination** | 149.21 | 149.57 | 0.245% |
| **Telemetry Logger**[4] | 134.17 | 134.18 | 0.004% |
| **CDHS** | 158.95 | 160.20 | 0.785% |

mean values are obtained in order to analyze the differences between both implementations. The difference is calculated as

$$d = 1 - \frac{NM}{M} \tag{5.1}$$

where NM corresponds to non-agent value and M corresponds to the MAES value. These results are shown in Table 5.10.

In general, there is an increase in the power consumption in the MAES implementation since the calculated difference is positive. The additional power consumption in the MAES framework is the consequence of the extra CPU utilization required for the MAES's communication method as reported in Section 5.2. However, the impact of the MAES framework on the power performance is low as the mean difference for all the applications is lower than 1%.

[4]The current logger is set to 500ms rate, the voltage logger is set to 1 second and the temperature logger is set to 2 seconds

Table 5.11: The FDIR's benchmark time per application.

| | **Reference Application** | **Blink LED Application** |
|---|---|---|
| **Detection Mean (ms)** | 55.19 | 49.97 |
| **Detection Variance** ($ms^2$) | 8470.13 | 811.94 |
| **I/R Mean(ms)** | 0.74 | 0.40 |
| **I/R Variance**[5]($ms^2$) | 0.0000 | 0.0000 |

## 5.4 Additional Benchmark: Failure Detection, Identification and Recovery time

One of the features added in the MAES framework is the agent's self-ability to recover from a detected and identified failure. In this section, a failure is introduced by using a button. In the Attitude Determination application, the failure is introduced in the Kalman agent, while in the Blink LED application, the failure is introduced in the Writing Agent. The detection time is measured from the time the button is pressed to the time that the failure is identified. The identification and recovery time are measured from the time that the agent starts the identification method (see Figure 3.2) to the time that the agent is redeployed by the AMS agent. 100 samples are acquired for each of the measured times. The results are shown in Table 5.11.

As seen in Table 5.11, the detection time within the same application varies greatly on each iteration. The large variance value is because when the failure has occurred, the processor might be executing other processes and the failure can be detected only until the `failure_detected()` method (see Figure 3.2) is called. On the other hand, the identification and recovery time is consistent as long as there are no other agents preempting the running agent.

[5]The value is less than 1x10$^{-5}$.

# Chapter 6

# Conclusions and Future Work

## 6.1 Conclusions

This research proposes the Multi-Agent Framework for Embedded Systems (MAES) as a new tool to implement MAS-based architecture applications for embedded real-time applications. During the early stages of this research, a literature study was conducted in order to determine the state-of-art of Multi-Agent System frameworks/platforms. Albeit there are several frameworks, most of them are designed neither to satisfy real-time requirements nor for embedded systems.

The development of the framework was divided into two phases: the mapping and the implementation. During the mapping phase, the *single thread* strategy was shown to improve performance and scalability. Based on that strategy, the rest of the FIPA components were mapped accordingly and showed in Table 3.4.

During the implementation phase, in order to comply with the real-time requirements, the framework was implemented on top of a Real-Time Operating System(TI-RTOS). This operating system was chosen as it provides tested drivers (UART, SPI, I2C) and instrumentation for debugging purposes. Since the TI-RTOS's scheduler is fixed-priority-based, the applications developed with this framework contain *soft* real-time characteristics, i.e., a process will meet a deadline most of the time. The framework's real-time characteristic was demonstrated in the benchmark analysis where an Attitude Determination application was implemented by using the MAES framework. Then, the results were compared against JADE, a well-known Multi-Agent Platform. The experiments demonstrated that algorithm execution time in MAES is consistent with a variance of the order $10^{-5}$ $[s^2]$, while the variance is approximately 100,000 larger in JADE. Based on that experiment, MAES ensures predictability or deterministic behaviour in its

execution. Furthermore, the user coding effort is reduced as the tasks and communication routines are standardized and encapsulated into MAES' class methods. However, the results have shown that MAES-based applications lead to an increase of 6.7 KB in average in Flash memory and 4.5 KB in average in SRAM memory with respect to its non-agent implementation. Also, the framework requires additional CPU utilization as MAES' communication methods implement extra routines to check the recipient validity. As a consequence of the increased CPU load, the power consumption is thereby increased. Nonetheless, MAES' impact on the power consumption is low as the results show that the increase is less than 1% in average. Although there is an increase in memory allocation, it is demonstrated that the framework is lightweight as this only requires additionally 5,826 bytes in the Flash memory. Moreover, an Agent Platform object requires 4,400 bytes and an extra Agent object requires 36 bytes in the SRAM memory. Thus, making the framework scalable.

Additionally, the time for detection and for identification/recovery methods were measured. Experiments showed that both times vary per application. Furthermore, the detection time varies within the same application as the fault might occur while the processor is executing other processes. On the other hand, it is seen that the identification and recovery time is consistent as MAES ensures predictability in these processes.

In conclusion, MAES is a real-time, lightweight and scalable framework compatible with highly resource-constrained embedded systems. Therefore, making it suitable for small satellites mission application development.

## 6.2 Future Work

Although the MAES framework components are based on FIPA specifications, the framework is not-fully FIPA-compliant. Specifically, the MAES framework messages are not compliant with the FIPA Agent Communication Language (ACL). Thus, it presents an implementation opportunity to further MAES' functionality. The framework can be expanded to perform agents' inter-platform communication when FIPA ACL is integrated.

Another implementation opportunity is to expand the Agent Management System functionality to handle errors and to log agent's failure information. As several methods of the API already return error codes used for debug purposes, the AMS can overview these errors and perform actions accordingly.

Lastly, as the priority is not a native agent's characteristic, the agent-developer is required to take additional consideration on assigning priority to the agent during the design stages of the application. Therefore, it presents an opportunity to create a methodology for agent-priority assignment.

# Bibliography


[1] R. Radhakrishnan, W. W. Edmonson, F. Afghah, R. M. Rodriguez-Osorio, F. Pinto, and S. C. Burleigh. Survey of Inter-Satellite Communication for Small Satellite Systems: Physical Layer to Network Layer View. *IEEE Communications Surveys Tutorials*, 18(4):2442–2473, 2016.

[2] Elizabeth Mabrouk. What are SmallSats and CubeSats?, March 2015.

[3] Hank Heidt, Jordi Puig-Suari, Augustus Moore, Shinichi Nakasuka, and Robert Twiggs. CubeSat: A New Generation of Picosatellite for Education and Industry Low-Cost Space Experimentation. *AIAA/USU Conference on Small Satellites*, August 2000.

[4] T. Vladimirova, X. Wu, and C. P. Bridges. Development of a Satellite Sensor Network for Future Space Missions. In *2008 IEEE Aerospace Conference*, pages 1–10, March 2008.

[5] D. M. Surka, M. C. Brito, and C. G. Harvey. The real-time ObjectAgent software architecture for distributed satellite systems. In *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, volume 6, pages 2731–2741 vol.6, 2001.

[6] Daniel Dvorak. NASA Study on Flight Software Complexity. In *AIAA Infotech@Aerospace Conference*. American Institute of Aeronautics and Astronautics, 2009.

[7] Christopher Krupiarz, Annette Mirantes, Doug Reid, Adrian Hill, and Roger Ward. *Flight Software*, pages 471–491. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[8] K Schilling. Perspectives for miniaturized, distributed, networked cooperating systems for space exploration. *Robotics and Autonomous Systems*, 2016.

[9] C. P. Bridges and T. Vladimirova. Agent computing applications in distributed satellite systems. In *2009 International Symposium on Autonomous Decentralized Systems*, pages 1–8, March 2009.

[10] Philippe Lalanda, Julie A. McCann, and Ada Diaconescu. *Sources of Inspiration for Autonomic Computing*, pages 57–94. Springer London, London, 2013.

[11] Philippe Lalanda, Julie A. McCann, and Ada Diaconescu. *Autonomic Computing Architectures*, pages 95–128. Springer London, London, 2013.

[12] Jörg P. Müller and Klaus Fischer. *Application Impact of Multi-agent Systems and Technologies: A Survey*, pages 27–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[13] Stan Franklin and Art Graesser. Is It an agent, or just a program?: A taxonomy for autonomous agents. In *Intelligent Agents III Agent Theories, Architectures, and Languages*, pages 21–35. Springer, Berlin, Heidelberg, August 1996.

[14] Michael Wooldridge and Nicholas R. Jennings. Agent theories, architectures, and languages: A survey. In *Intelligent Agents*, pages 1–39. Springer, Berlin, Heidelberg, August 1994.

[15] Michael Schumacher. *Multi-Agent Systems*, pages 9–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[16] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, March 1998.

[17] S. D. J. McArthur, E. M. Davidson, V. M. Catterson, A. L. Dimeas, N. D. Hatziargyriou, F. Ponci, and T. Funabashi. Multi-Agent Systems for Power Engineering Applications #x2014;Part I: Concepts, Approaches, and Technical Challenges. *IEEE Transactions on Power Systems*, 22(4):1743–1752, November 2007.

[18] D. Koesrindartoto, Junjie Sun, and L. Tesfatsion. An agent-based computational laboratory for testing the economic reliability of wholesale power market designs. In *IEEE Power Engineering Society General Meeting, 2005*, pages 2818–2823 Vol. 3, June 2005.

[19] Jarok Koo. Intelligent multiagent systems in E-commerce. In *Proceedings 6th Russian-Korean International Symposium on Science and Technology. KORUS-2002 (Cat. No.02EX565)*, pages 134–136, 2002.

[20] E. M. Davidson, S. D. J. McArthur, J. R. McDonald, T. Cumming, and I. Watt. Applying multi-agent system technology in practice: automated management and analysis of SCADA and digital fault recorder data. *IEEE Transactions on Power Systems*, 21(2):559–567, May 2006.

[21] J. H. Kim, H. S. Shim, H. S. Kim, M. J. Jung, I. H. Choi, and J. O. Kim. A cooperative multi-agent system and its real time application to robot soccer. In *Proceedings of International Conference on Robotics and Automation*, volume 1, pages 638–643 vol.1, April 1997.

[22] B. Burmeister, A. Haddadi, and G. Matylis. Application of multi-agent systems in traffic and transportation. *IEE Proceedings - Software Engineering*, 144(1):51–60, February 1997.

[23] Emil Vassev and Mike Hinchey. *Software Engineering for Aerospace: State of the Art*, pages 1–45. Springer International Publishing, Cham, 2014.

[24] Johan Carvajal-Godinez, Jian Guo, and Eberhard Gill. Agent-based algorithm for fault detection and recovery of gyroscope's drift in small satellite missions. *Acta Astronautica*, 139:181 – 188, 2017.

[25] S. Mandutianu, F. Hadaegh, and P. Elliot. Multi-agent system for formation flying missions. In *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, volume 6, pages 2793–2802 vol.6, 2001.

[26] Kejun Ning and Ruqing Yang. MAS based embedded control system design method and a robot development paradigm. *Mechatronics*, 16(6):309–321, July 2006.

[27] Gustavo Aranda and Javier Palanca. SPADE User's Manual.

[28] Telecom Italia Lab. Jade Site | Java Agent DEvelopment Framework.

[29] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. JADE: A White Paper. *EXP in search of innovation*, 3(3):6–19, 2003.

[30] Integration Engineering Laboratory. Mobile-C.

[31] David M. Flrez, Guillermo A. Rodrguez, Juan M. Ortiz, and Enrique Gonzlez. Besa-me: Framework for robotic multiagent system design. In *Proceedings of the 3rd International Workshop on Multi-Agent Robotic Systems - Volume 1: MARS, (ICINCO 2007)*, pages 64–73. INSTICC, ScitePress, 2007.

[32] L. Peng, F. Guan, L. Perneel, H. Fayyad-Kazan, and M. Timmerman. EmSBoT: A lightweight modular software framework for networked robotic systems. In *2016 3rd International Conference on Advances in Computational Tools for Engineering Applications (ACTEA)*, pages 216–221, July 2016.

[33] Onn Shehory and Arnon Sturm. *Multi-agent Systems: A Software Architecture Viewpoint*, pages 57–78. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[34] Estefania Argente, Javier Palanca, Gustavo Aranda, Vicente Julian, Vicente Botti, Ana Garcia-Fornes, and Agustin Espinosa. Supporting Agent Organizations. In *Multi-Agent Systems and Applications V*, Lecture Notes in Computer Science, pages 236–245. Springer, Berlin, Heidelberg, September 2007.

[35] FIPA Architecture Board. FIPA Agent Management Specification.

[36] FIPA Architecture Board. FIPA Design Process Documentation Template.

[37] FIPA Architecture Board. FIPA Agent Message Transport Service Specification.

[38] Stefan Poslad and Patricia Charlton. *Standardizing Agent Interoperability: The FIPA Approach*, pages 98–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[39] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[40] Michael Wooldridge. Intelligent agents. In Gerhard Weiss, editor, *Multiagent Systems:A Modern Approach to Distributed Modern Approach to Artificial Intelligence*, pages 27–77. MIT Press, Cambridge, MA, USA, 1999.

[41] Kalliopi Kravari and Nick Bassiliades. A Survey of Agent Platforms. *Journal of Artificial Societies and Social Simulation*, 18(1):11, 2015.

[42] Alban Rousset, Bndicte Herrmann, Christophe Lang, and Laurent Philippe. A survey on parallel and distributed multi-agent systems for high performance computing simulations. *Computer Science Review*, 22:27 – 46, 2016.

[43] Arnon Sturm and Onn Shehory. *The Evolution of MAS Tools*, pages 275–288. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[44] Todd M. Carrico. Cognitive Agent Architecture (Cougaar).

[45] Xavier Rubio-Campillo. Pandora: An HPC Agent-Based Modelling framework | BSC-CNS.

[46] Agostino Poggi. Developing Real Applications With Agent Technologies. *Journal of Systems Integration*, 9(4):311–328, December 1999.

[47] Miguel Escriva Javier Palanca, Gustavo Aranda. SPADE API Documentation.

[48] Matthew Wild Jerry Pasker Jonathan Siegle Edwin Mons Peter Saint-Andre, Kevin Smith and Jeremie Miller. jabber.org - the original XMPP instant messaging service.

[49] Miguel Escrivá Gregori, Javier Palanca Cámara, and Gustavo Aranda Bada. A jabber-based multi-agent system platform. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '06, pages 1282–1284, New York, NY, USA, 2006. ACM.

[50] Yu-Cheng Chou, David Ko, and Harry H. Cheng. An embeddable mobile agent platform supporting runtime code mobility, interaction and coordination of mobile agents and host systems. *Information and Software Technology*, 52(2):185–196, February 2010.

[51] Bo Chen, Harry H. Cheng, and Joe Palen. Mobile-C: a mobile agent platform for mobile C/C++ agents. *Software: Practice and Experience*, 36(15):1711–1733, December 2006.

[52] Long Peng, Fei Guan, Luc Perneel, and Martin Timmerman. EmSBot: A modular framework supporting the development of swarm robotics applications. *International Journal of Advanced Robotic Systems*, 13(6):1729881416663662, 2016.

[53] Agostino Poggi and Michele Tomaiuolo. *Integrating Peer-to-Peer and Multi-agent Technologies for the Realization of Content Sharing Applications*, pages 93–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[54] Texas Instruments. SYS/BIOS (TI-RTOS Kernel) v6.46. User's Guide, June 2016.

[55] Richard Barry. *Mastering the FreeRTOS Real Time Kernel. A Hands-On Tutorial Guide*. Real Time Engineers Ltd, 2016.

[56] FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions.

[57] FIPA Architecture Board. FIPA ACL Message Representation in Bit-efficient Encoding Specification.

[58] &#xc9 de Oliveira, Lcio Jeronimo, Ijar Milagre da Fonseca, H&#xe9 Koiti Kuga, and lio. Fault Detection and Isolation in Inertial Measurement Units Based on -CUSUM and Wavelet Packet, 2013. DOI: 10.1155/2013/869293.

[59] A. M. Sabatini. Quaternion-based extended Kalman filter for determining orientation by inertial and magnetic sensing. *IEEE Transactions on Biomedical Engineering*, 53(7):1346–1356, July 2006.

[60] Leonard A. Mcgee, Stanley F. Schmidt, Leonard A. Mcgee, and Stanley F. Sc. Discovery of the kalman filter as a practical tool for aerospace and. Technical report, Industry, National Aeronautics and Space Administration, Ames Research, 1985.

[61] Texas Instruments. BOOSTXL-SENSORS Sensors BoosterPack Plug-in Module.

[62] R.J. Freund and W.J. Wilson. *Statistical Methods*. Academic Press, 2003.

[63] Ronald L. Wasserstein and Nicole A. Lazar. The asa's statement on p-values: Context, process, and purpose. *The American Statistician*, 70(2):129–133, 2016.

[64] E. J. Lefferts, F. L. Markley, and M. D. Shuster. Kalman Filtering for Spacecraft Attitude Estimation. *Journal of Guidance Control Dynamics*, 5:417–429, September 1982.

[65] J. Hidalgo-Carri, S. Arnold, and P. Poulakis. On the Design of Attitude-Heading Reference Systems Using the Allan Variance. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 63(4):656–665, April 2016.

[66] M.A. Hopcroft. allan - File Exchange - MATLAB Central.

# Appendix A

# Reference Application

This section describes the algorithm that is based on the work of [59].

Any rigid body orientation in space can be determined when the axis orientation of a coordinated frame $B$ attached to the body itself is specified with respect to an absolute coordinate system: the navigation frame $N$. The transformation between representations is expressed as:

$$\overrightarrow{\boldsymbol{x}}^b(t) = \boldsymbol{C}_n^b[\boldsymbol{q}(t)]\,\overrightarrow{\boldsymbol{x}}^n(t) \tag{A.1}$$

The expression described in A.1 represents the transformation of a 3x1 column-vector relative to the navigation frame $N$ to a body frame $B$ by using the direction cosine matrix (DCM). The DCM is given in terms of the orientation quaternion $\mathbf{q}=[\overrightarrow{e},q_4]^T$, where $\overrightarrow{e}=[q_1,q_2,q_3]^T$ is the vector part and $q_4$ is the scalar part of the quaternion.

The DCM matrix is described as:

$$\boldsymbol{C}_n^b(\boldsymbol{q}) = \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & -q_1^2 + q_2^2 - q_3^2 + q_4^2 & 2(q_2q_3 + q_4q_1) \\ 2(q_1q_3 + q_2q_4) & 2(q_2q_3 - q_4q_1) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix} \tag{A.2}$$

The rigid body angular motion is given by the vector differential equation:

$$\frac{d}{dt}\boldsymbol{q} = \boldsymbol{\Omega}[\overrightarrow{\boldsymbol{\omega}}]\boldsymbol{q} \tag{A.3}$$

where:

$$\boldsymbol{\Omega}[\overrightarrow{\boldsymbol{\omega}}] = \begin{bmatrix} 0 & \omega_3 & -\omega_2 & \omega_1 \\ -\omega_3 & 0 & \omega_1 & \omega_2 \\ \omega_2 & -\omega_1 & 0 & \omega_3 \\ -\omega_1 & -\omega_2 & -\omega_3 & 0 \end{bmatrix} \tag{A.4}$$

and $\overrightarrow{\boldsymbol{\omega}}$(t)=[$\omega_1$,$\omega_2$,$\omega_3$]$^T$ corresponds to the angular velocity of $B$ relative to $N$.

The discrete model of A.3 is given by:

$$\begin{cases} \boldsymbol{q}_{k+1} = exp(\boldsymbol{\Omega}_k T_s)\boldsymbol{q}_k \quad , \quad k = 0, 1, ... \\ \boldsymbol{q}_0 = \boldsymbol{q}(0) \end{cases} \tag{A.5}$$

Where $T_s$ is the algorithm's process integration step and the quaternion is determined at time instants $kT_s$ with known initial conditions $q_0$.

## A.1 Filter Design

The aim of this Extended Kalman Filter algorithm is to determine the orientation of a rigid body by using a gyro and aided by an accelerometer and a magnetometer. The gyro measurements values are used in the prediction part, while the aiding sensors are used in the correction part of the algorithm.

The state vector of the algorithm is composed by the rotation quaternion, the tri-axis accelerometer and magnetometer bias as a result of a 10x1 column vector.

$$\overrightarrow{\boldsymbol{x}}_{k+1} = \begin{bmatrix} \boldsymbol{q}_{k+1} \\ {}^a\overrightarrow{\boldsymbol{b}}_{\boldsymbol{k+1}} \\ {}^m\overrightarrow{\boldsymbol{b}}_{\boldsymbol{k+1}} \end{bmatrix} = \Phi(Ts, \overrightarrow{\boldsymbol{\omega}})\overrightarrow{\boldsymbol{x}}_k + \overrightarrow{\boldsymbol{w}}_k = \begin{bmatrix} exp(\boldsymbol{\Omega}_k T_s) & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{q}_k \\ {}^a\overrightarrow{\boldsymbol{b}}_{\boldsymbol{k}} \\ {}^m\overrightarrow{\boldsymbol{b}}_{\boldsymbol{k}} \end{bmatrix} + \begin{bmatrix} {}^q\boldsymbol{w}_k \\ {}^a\boldsymbol{w}_k \\ {}^m\boldsymbol{w}_k \end{bmatrix} \tag{A.6}$$

Where the $\overrightarrow{\boldsymbol{w}_k}$ is the random walk. Since it is assumed that the random walk vectors are not correlated, the process covariance matrix $\boldsymbol{Q}_k$ has the following expression:

$$\boldsymbol{Q}_k = \begin{bmatrix} (T_s/2)^2 \Xi_k \sigma_g^2 \boldsymbol{I} \Xi_k^T & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & {}^a\sigma_w^2 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & {}^m\sigma_w^2 \end{bmatrix} \tag{A.7}$$

$\sigma_g$ corresponds to the white noise variance of the gyroscope, while ${}^a\sigma_w$ and ${}^m\sigma_w$ correspond to the random walk variance of the accelerometer and magnetometer, respectively. $\Xi$ expression is given by:

$$\Xi = \begin{bmatrix} q_4 & -q_3 & q_2 \\ q_3 & q_4 & -q_1 \\ -q_2 & q_1 & q_4 \\ -q_1 & -q_2 & -q_3 \end{bmatrix} \tag{A.8}$$

The measurement model is given by the following expression:

$$\overrightarrow{\boldsymbol{z}}_{k+1} = \begin{bmatrix} \overrightarrow{\boldsymbol{a}}_{k+1} \\ \overrightarrow{\boldsymbol{m}}_{k+1} \end{bmatrix} = \boldsymbol{f}[\overrightarrow{\boldsymbol{x}}_{k+1}] + \overrightarrow{\boldsymbol{v}}_{k+1} = \begin{bmatrix} \boldsymbol{C}_n^b(\boldsymbol{q}) & \mathbf{0} \\ \mathbf{0} & \boldsymbol{C}_n^b(\boldsymbol{q}) \end{bmatrix} \begin{bmatrix} \overrightarrow{\boldsymbol{g}} \\ \overrightarrow{\boldsymbol{h}} \end{bmatrix} + \begin{bmatrix} {}^a\overrightarrow{\boldsymbol{b}}_{\boldsymbol{k+1}} \\ {}^m\overrightarrow{\boldsymbol{b}}_{\boldsymbol{k+1}} + \end{bmatrix} + \begin{bmatrix} {}^a\overrightarrow{\boldsymbol{v}}_{\boldsymbol{k+1}} \\ {}^m\overrightarrow{\boldsymbol{v}}_{\boldsymbol{k+1}} + \end{bmatrix} \quad \text{(A.9)}$$

where $\overrightarrow{\boldsymbol{g}}$ and $\overrightarrow{\boldsymbol{h}}$ are the gravitational and magnetic field, respectively, obtained when the body is not rotated.

On the other hand, since the accelerometer and magnetometer measurement noise ${}^a\overrightarrow{\boldsymbol{v}}_{\boldsymbol{k+1}}$ and ${}^m\overrightarrow{\boldsymbol{v}}_{\boldsymbol{k+1}}$ are not correlated, the covariance matrix of the measurement model $\boldsymbol{R}_{k+1}$ is:

$$\boldsymbol{R}_{k+1} = \begin{bmatrix} \sigma_a^2 \boldsymbol{I} & \mathbf{0} \\ \mathbf{0} & \sigma_m^2 \boldsymbol{I} \end{bmatrix} \quad \text{(A.10)}$$

where $\sigma_a$ and $\sigma_m$ are white noise variance values of the accelerometer and magnetometer, respectively.

Also, as seen in A.9, the expression is not linear. Therefore, the Jacobian matrix needs to be computed:

$$\boldsymbol{F}_{k+1} = \frac{\partial}{\partial \overrightarrow{\boldsymbol{x}}_{k+1}} \overrightarrow{\boldsymbol{z}}_{k+1} \Big|_{\overrightarrow{\boldsymbol{x}}_{k+1} = \overrightarrow{\boldsymbol{x}}^-_{k+1}} = \begin{bmatrix} \frac{\partial \boldsymbol{C}_n^b(\boldsymbol{q})}{\partial \boldsymbol{q}} \overrightarrow{\boldsymbol{g}} & \boldsymbol{I_3} & \mathbf{0_3} \\ \frac{\partial \boldsymbol{C}_n^b(\boldsymbol{q})}{\partial \boldsymbol{q}} \overrightarrow{\boldsymbol{h}} & \mathbf{0_3} & \boldsymbol{I_3} \end{bmatrix} \quad \text{(A.11)}$$

Once the matrices are defined, the Extended Kalman Filter equations are summarized below:

1. *A priori* state estimate computation

$$\overrightarrow{\boldsymbol{x}}^-_{k+1} = \Phi(T_s, \overrightarrow{\boldsymbol{\omega_k}})\, \overrightarrow{\boldsymbol{x}}_k \quad \text{(A.12)}$$

2. *A priori* error covariance matrix computation

$$\boldsymbol{P}^-_{k+1} = \Phi(T_s, \overrightarrow{\boldsymbol{\omega_k}}) \boldsymbol{P_k} \Phi(T_s, \overrightarrow{\boldsymbol{\omega_k}})^T + \boldsymbol{Q_k} \quad \text{(A.13)}$$

3. *Kalman Gain* computation

$$\boldsymbol{K_{k+1}} = \boldsymbol{P}^-_{k+1} \boldsymbol{F}^{\boldsymbol{T}}_{\boldsymbol{k+1}} (\boldsymbol{F_{k+1}} \boldsymbol{P}^-_{k+1} \boldsymbol{F}^{\boldsymbol{T}}_{\boldsymbol{k+1}} + \boldsymbol{R_{k+1}})^{-1} \quad \text{(A.14)}$$

4. *A posteriori* state estimate computation

$$\overrightarrow{\boldsymbol{x}}_{k+1} = \overrightarrow{\boldsymbol{x}}^-_{k+1} + \boldsymbol{K_{k+1}} [\overrightarrow{\boldsymbol{z}}_{k+1} - \boldsymbol{f}(\overrightarrow{\boldsymbol{x}}^-_{k+1})] \quad \text{(A.15)}$$

5. *A posteriori* error covariance matrix computation

$$\boldsymbol{P}_{k+1} = \boldsymbol{P}^-_{k+1} - \boldsymbol{K_{k+1}} \boldsymbol{F_{k+1}} \boldsymbol{P_{k+1}}^- \quad \text{(A.16)}$$

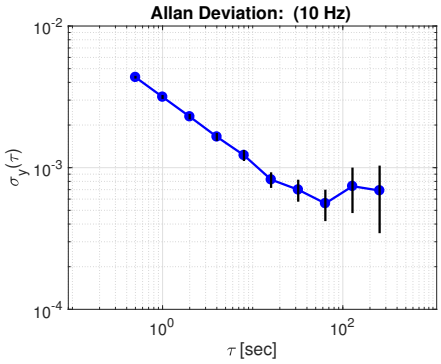


Figure A.1: Allan Deviation graph for x-axis accelerometer

Table A.1: Sensors' white noise and random walk parameter

| | $\sigma_g$ (deg/s) | $\sigma_a$ (m/$s^2$) | $\sigma_m$ ($\mu$T) | ${}^a\sigma_w$ (m/$s^2$) | ${}^m\sigma_w$ ($\mu$T) |
|---|---|---|---|---|---|
| x-axis | 0.0168 | 0.0032 | 0.1634 | 0.0018 | 0.1047 |
| y-axis | 0.0135 | 0.0035 | 0.1705 | 0.0022 | 0.1169 |
| z-axis | 0.0135 | 0.0039 | 0.1831 | 0.0024 | 0.1119 |

## A.2 Algorithm's initial conditions

In order to work properly, it is required to set specific initial conditions in the algorithm. First, the error covariance matrix is initialized 10000 times of the process noise covariance matrix Q. Then, the gravitiy vector $\overrightarrow{\mathbf{g}}$ is set to [0 0 -9.80665] and the magnetic vector $\overrightarrow{\mathbf{m}}$ is set to [22.61 1.17 25.57]. On the other hand, the process integration step $T_s$ in practice is smaller than the sensors sampling period [64]. By choosing a smaller value than the sensor's sampling period ensures a smooth transition between the internal states. In this implementation, this value is set to be 100 times of the sensors' sampling rate. Once these values were set, the sensors' white noise and random walk values have to be determined. These values are usually obtained from the datasheet, however, as those values are not available, the method discussed by the Hidalgo-Carri et al. [65] is applied in order to find them. These methods use the Allan Deviation to determine the white noise and the random walk. In order to determine those values for the system, 10000 samples acquired at 10Hz were obtained while the sensors were held still. Then, a `MATLAB` script obtained from [66] was used in order to obtain the white noise and random walk. An example of an Allan deviation graph is shown in Figure A.1:

The white noise parameter are obtained from the Allan Variance graph for $\tau(1)$, while the random walk is obtained for $\tau(3)$. The used values for the EKF are shown in Table A.1.

# Appendix B

# The MAES' API

## B.1 Agent Class

- `Agent Constructor`:Creates an instance of Agent Class. The user passes as parameters: the agent's name, the agent's priority, the agent's stack pointer and the agent's stack size.

- `Agent_AID`: Returns the AID of the object.

## B.2 Agent Platform Class

- `Agent Platform Constructor`: This constructor creates an instance of the class and sets the name of the platform. Similar as an Agent object, an Agent Platform object needs to be declared in a global location. There are two version of the constructor: one **without** the user conditions and the other **with** the user conditions.

- `void agent_init()`: This method creates the mailbox instance and the task instance associated with each agent. This method can only be called in the `main()` function or by the AMS Agent. The method requires parameters such: an Agent object and the wrapper function that encapsulates the agent's behaviours.

  This method is defined twice as one of the methods receives additionally two user-defined arguments that can be passed as parameters to the wrapper function.

- `bool boot()`: This method boots the Agent Platform. Can be only called from `main()`.

- `bool agent_search(Agent_AID aid)`: This method searches an agent's AID within the Agent Platform. Returns `TRUE` if found.

- `void agent_wait(Uint32 ticks)`: The caller agent is set to `inactive` state for a time specified by the `ticks` variable.

- `void agent_yield()`: The caller agent releases control of the processor so another agent of the same priority can take over. No effect is perceived if there is no other same-priority `active` agent.

- `Agent_AID get_running_agent()`: Obtains the AID of the `running` agent.

- `int get_state(Agent_AID aid)`: Gets the state of the agent's AID specified in the argument.

- `Agent_info get_Agent_description(Agent_AID aid)`:
  Gets the Agent_info description of the agent's AID specified in the argument.

- `AP_Description get_AP_description()`: Gets the Agent Platform description.

- `ERROR_CODE register_agent(Agent_AID aid)`/
  `deregister_agent(Agent_AID aid)`: Can be only performed by the AMS agent. Registers/Deregisters agent of the platform. When the registration is successful, the agent's priority changes from -1 to the priority set by the user. When an agent is deregistered, the priority is set to -1 and the `AP` variable is set to NULL.

- `ERROR_CODE kill_agent(Agent_AID aid)`: Can be only performed by the AMS agent. Terminates the execution of the agent. Cannot be invoked again.

- `ERROR_CODE suspend_agent(Agent_AID aid)`/
  `resume_agent(Agent_AID aid)`: Can be only performed by the AMS agent. Suspends/Resumes execution of an agent. Suspending the execution of the agent will set the agent's priority to -1, while resuming the execution returns the agent's priority.

- `ERROR_CODE restart(Agent_AID aid)`: Can be only performed by the AMS agent. Restart the agent execution by killing and creating the agent.

Some of the class methods return an `ERROR_CODE` type, which is an alias of the integer value. The possible return values are: `NO_ERROR, FOUND, HANDLE_NULL, LIST_FULL, DUPLICATED, NOT_FOUND, TIMEOUT, INVALID, NOT_REGISTERED`.

## B.3 Agent Organization Class

- `Agent_Organization(ORG_TYPE organization_type)` constructor: This constructor creates an instance of the class. This method requires the organization type as parameter.

- `ERROR_CODE create()`: Creates the organization. This method needs to be called from an Agent's behaviour. Consequently, the `Agent_AID owner` from `org_info` (see Listing 4) is assigned to the caller of this method. If the organization is already created and another agent calls this method, an error is returned. The `org` variable of the creator is pointing to this organization.

- `ERROR_CODE destroy()`: Clears the members and banned list of the organization. Also, this method clears the "affiliation", "role" and "org" spot of each member of the organization.

- `ERROR_CODE isMember(Agent_AID aid)`: Checks if an agent is a member of the organization.

- `ERROR_CODE isBanned(Agent_AID aid)`: Checks if an agent is banned from the organization.

- `ERROR_CODE change_owner(Agent_AID aid)`: Changes organization's owner. The method is performed by the owner.

- `ERROR_CODE set_moderator(Agent_AID aid)`: Sets organization's moderator.The method is performed by the owner. Changes the `role` variable of the agent.

- `ERROR_CODE set_admin(Agent_AID aid)`: Sets organization's administrator. The method is performed by the owner. Changes the `affiliation` variable of the agent.

- `ERROR_CODE add_agent(Agent_AID aid)`: Adds an agent to the organization. The method is performed by the owner or administrator of the organization. The `org` variable of the added agent is set to point to this organization.

- `ERROR_CODE kick_agent(Agent_AID aid)`: Removes agent from the organization. The method is performed by the owner or administrator of the organization. The `org` variable of the removed agent is set to `NULL`.

- `ERROR_CODE ban_agent(Agent_AID aid)`: Ban agent to the organization. The method is performed by the owner or administrator of the organization.

- `ERROR_CODE remove_ban(Agent_AID aid)`: Removes the ban of an agent. The method is performed by the owner or administrator of the organization.

- `void clear_ban_list()`: Clears the banned agent list. The method is performed by the owner or administrator of the organization.

- `ERROR_CODE set_participant(Agent_AID aid)`: Allows a member to participate in the conversation. The method is performed by the owner or moderator of the organization.

- `ERROR_CODE set_visitor(Agent_AID aid)`: Sets a member as only listener in the conversation. The method is performed by the owner or moderator of the organization.

- `int get_org_type()`: Gets the organization type.

- `org_info get_info()`: Gets the organization information.

- `int get_size()`: Gets the number of members of the organization.

## B.4 Agent Message Class

- `Agent Message` constructor: An instance of this class **must** be created within the wrapper function, so the object can be associated with the caller agent. The numbers of subscribers of the Agent Msg object is set to zero and the receivers list is cleared.

- `ERROR_CODE add_receiver(Agent_AID aid_receiver)`: Adds an agent into the list. Returns an error when the agent's AID is `NULL`, the receivers list is full or the agent is not found in the platform.

- `ERROR_CODE remove_receiver(Agent_AID aid_receiver)`: Removes an agent from the list. Returns an error when the agent is not found in the platform.

- `void clear_all_receivers()`: Clears the list.

- `void refresh_list()`: Updates the list. The method also removes any de-registered receivers and receivers that are not located in the same agent organization.

- `MSG_TYPE receive(Uint32 timeout)`: Waits the time specified by `timeout` for any incoming message. When there is a timeout, the method returns `NO_RESPONSE`. Otherwise returns the message type specified by FIPA69.

- `ERROR_CODE send(Agent_AID aid_receiver, int timeout)`: Sends message to the specific agent. The `timeout` value specifies the time that the sender waits for an available spot in the target's agent mailbox. Returns an error when the time expires, the agent is not registered in the platform or the message is not allowed due to specific Agent Organization communication restrictions.

- `ERROR_CODE send()`: Multicast the message to the list of recipients. The number of iterations of this method is equal to the list size and on each iteration the method `ERROR_CODE send(Agent_AID aid_receiver, int timeout)` is called. Returns the last error.

- `set_msg_* methods`: Set of methods used to set the message type and the message content.

- `get_* methods`: Set of methods used to get the values of the message object members.

- `AMS request methods`: Used by the agent to send a request to the AMS. If the agent belongs to an organization, the request might be valid depending of the organization communication restriction. Returns error code.