A Communication Protocol Design aimed at a Multi-Agent System Framework for Miniaturized Satellite Systems

Diseño de un protocolo de comunicación para la plataforma de desarrollo software basado en agentes utilizados en sistemas satelitales miniaturizados[1]

Samantha Interiano-Valverde[2], Davide Scazzoli[3], Carmen Chan- Zheng[4], Johan Carvajal-Godínez[5], Maurizio Magarini[6]

Fecha de recepción: 8 de agosto de 2019
Fecha de aprobación: 27 de noviembre de 2019

Interiano-Valverde, S; Scazzoli, D; Chang-Zheng, C; Carvajal-Godínez, J; Magarini, M. A Communication Protocol Design aimed at a Multi-Agent System Framework for Miniaturized Satellite Systems. *Tecnología en Marcha*. Vol. 33-4. Octubre-Diciembre 2020. Pág 62-76.

https://doi.org/10.18845/tm.v33i4.4505

1 Work based on the Thesis of [13]
2 Politecnico di Milano, Milano, Italia. Correo electrónico: samantha.interiano@mail.polimi.it.
3 Politecnico di Milano, Milano, Italia. Correo electrónico: davide.scazzoli@polimi.it.
4 Universidad de Costa Rica. Puntarenas, Costa Rica. Correo electrónico: carmen.chan@ucr.ac.cr.
5 Space Systems Laboratory. Tecnológico de Costa Rica, Cartago, Costa Rica. Correo electrónico: johcarvajal@tec.ac.cr.
6 Politecnico di Milano. Milano, Italia. Correo electrónico: maurizzio.magarini@polimi.it.

Keywords

MAES framework; embedded systems; real-time; agent; inter-platform; I^{2}C.

Abstract

In recent years space-science and exploration have become more accessible due to the popularization of the concept of CubeSats. CubeSats are being used as a result of their convenient size and weight requirements, allowing for target missions to be designed, developed, and launched with a significant reduction of costs compared to traditional space missions. Furthermore, the development of target missions has become more intricate, forcing a shift in the traditional notion of using a centralized control architecture to a distributed architecture. A distributed architecture tackles the problem of a possible functionality loss over the control unit. This paper describes the extension of capabilities of the Multi-Agent Systems Framework for Embedded Systems (MAES). This extension provides MAES framework with the ability to perform inter-platform communication, so now the control unit architecture can be broadened allowing agents from different platforms to interact and perform cooperatively different routines designed by the developer, so it is not limited to the capabilities of just one platform. Moreover, this paper shows the results of the experimental setup showing the precision of inter-platform message exchange and the relationship between the delay of the exchange of inter-platform messages and the number of agents that are simultaneously running on each platform.

Palabras clave

Marco MAES; sistemas embebidos; tiempo real; agente; interplataforma; I^{2}C.

Resumen

En los últimos años, la ciencia espacial y la exploración se han vuelto más accesibles debido a la popularización del concepto de CubeSats. Los CubeSats se está utilizando debido a sus convenientes requisitos de tamaño y peso, lo que permite que misiones especificas se diseñen, desarrollen y lancen con una reducción significativa de los costos en comparación con las misiones espaciales tradicionales. Además, el desarrollo de misiones específicas se ha vuelto más complejo, forzando un cambio en la noción tradicional de usar una arquitectura de control centralizada para una arquitectura distribuida. Una arquitectura distribuida resuelve el problema de una posible pérdida de funcionalidad en la unidad de control. Este documento describe la extensión de las capacidades del *framework* de sistemas multiagentes para sistemas embebidos (MAES). Esta extensión proporciona al *framework* MAES la capacidad de realizar comunicaciones entre plataformas, de modo que ahora la arquitectura de la unidad de control puede ampliarse, permitiendo a los agentes de diferentes plataformas interactuar y realizar de forma cooperativa diferentes rutinas diseñadas por el desarrollador, por lo que no se está limitado a las capacidades de solo una plataforma. Asimismo, este documento muestra los resultados de la configuración experimental que muestra la precisión del intercambio de mensajes entre plataformas y la relación entre el retraso del intercambio de mensajes entre plataformas y la cantidad de agentes que se ejecutan simultáneamente en cada plataforma.

Introduction

Space science and exploration have become more accessible in recent years due to the popularization of the concept of CubeSats. Twenty years ago, CubeSats were developed as a concept model that intended to involve universities to participate in space missions [1]. It rapidly caught the attention of researchers, space agencies, governments, and companies because of the specifics on size and weight requirements meant that targeted missions could be designed, developed and launched with a significant reduction of costs. The development of more intricate target missions has forced a shift in the traditional notion of using a centralized control architecture to an architecture that is distributed since once the CubeSat is launched there is no way of providing maintenance to it. Hence, if the control unit is affected, then the entire system may lose functionality. One of the ways the space industry has been doing this shift is using multi-agent systems.

A Multi-Agent System (MAS) can be defined as a dynamic distributed system that envelops autonomous agents/nodes, who cohabit inside a society where they can work individually on a specific task, or they can interact with others and work together towards achieving a common goal [2]. The architecture of MAS is modular, making it scalable, therefore allowing agents to be linearly aggregated. The system as a whole is robust since the architecture allows the system to tolerate failures of one or more agents, as the responsibilities and control are shared between the agents. [3]

According to Coulouris *et al.* [4], the advantages of adopting a distributed architecture are given by the capacity of concurrency program executions, no need of centralized control, scalability, and independent failure, meaning that if a component of the system fails the other ones can still run independently, and they may never notice that another component has failed. As a result, the Multi-Agent Systems Framework for Embedded Systems (MAES) was developed to provide the developer with a tool to implement real-time MAS based applications in embedded environments. This capability is something that the market was lacking, especially for the development of real-time applications in the space industry using distributed architectures.

The design and implementation of the MAES framework were carried out by Chan-Zheng *et al.* [5]. The development started as a result of the need in the science community to have a tool that fills the gap between real-time operating systems and multi-agent systems used for satellite applications. Since MAES is still in the initial stage of development, for it to be released and used by developers in the design of future satellite applications the guidelines given by the Foundation for Intelligent Physical Agents (FIPA) are used. FIPA is a foundation that advocates for the standardization of distributed agent-based applications and services through collaboration with international companies, universities, and the IEEE Computer Society [6] [7].

MAES framework conveniently reduces the time and effort of developers to implement a MAS application because it already implements the services and the primitive communication structure with the intra-platform agents for a highly constraint embedded platform. This paper focuses on the description of the upgrade done to the MAES framework in order to allow for inter-platform communication. Thus, the control unit architecture can be enhanced to allow agents from different platforms to interact and perform cooperatively different routines designed by the developer, so it is not limited to the capabilities of just one platform.

The current state of MAES framework development so far complies with some FIPA specifications, which are the minimum requirements for intra-platform operation (see table 1).

One of the purposes of this work is to add more capabilities to MAES's framework so it can allow communication between an assortment of platforms, all running MAES in a given communication topology.

Table 1. Current minimum FIPA specifications implemented on MAES

FIPA component
Agent
Agent Platform
Agent Management Service
Message Transport Service (Intra-platform)

The outline of the paper is as follows. First the article describes the design of the inter-platform design, extending on the communication protocol selection, message strategy selection, and its implementation. Then is described the verification of the operation of inter-platform communication as to observe the implementation's operation. Finally, it draws the main conclusions and discusses future work developments.

Inter-Platform Protocol Design

This Section presents the design choices for the communication protocol, following the FIPA guidelines for Multi-Agent Systems, which addresses the question: "What is the most efficient architecture that best manages the resources for communication applications in multi-platform systems running MAES?"

Communication Protocol Selection.

The communication protocol chosen for the implementation of the inter-platform message transport service of MAES is I^2C (Inter-Integrated Circuit) because of its usage in previous research related to Cubesats design for the communication between the on-board computer (OBC) and peripheral devices [8] [9]. The small footprint and other advantages of I^2C have contributed to its popularity that, in turn, has driven most microcontroller manufacturers to include I^2C among the standard BUS interfaces provided [10].

Message Routing Strategy.

In order to extend the communication of agents and give them the ability to communicate between agents outside its platform, the inter-platform messages shall act in compliance with the FIPA specifications. FIPA specifications do not explicitly specify what protocol must be used when inter-platform communication is intended. FIPA specifications mention only how the packet must be composed in term of its contents.

In an abstract form, the message can be expressed in two parts: message envelope and message payload. The message envelope comprises a collection of parameters expressed as a key-value tuple (name/value pair). The envelope must contain at least the *to-from*, *date,* and *acl representation*. The message payload embodies the encoded Agent Communication Language (ACL) message. Therefore, this paper describes the design choices of a static approach.

The static approach begins with the assumption that the sender agent knows to whom it wants to communicate with, much as in other types of communication systems, the recipient is always known, the only thing that both approaches handle is how they deliver the messages. The message packet is comprised by the header (Sender I^2C address, Receiver I^2C address, and message type) and the payload (Sender AID, Receiver AID, message type and content).

Static Approach

The static approach works as follows. At start-up, each platform knows the other platforms' ID, which then correlates to their I^{2}C addresses, so all platforms have the information of each other before any message exchange is done. Afterward, when the *sender agent* sends a message to a *receiver agent,* the local Message Transport System (MTS) has first to check if the *receiver agent* is local. If it is, it proceeds as it is already implemented by *MAES Framework* on the contrary when the *receiver agent* is not registered to the same platform as the *sender agent* the next step is to search by the agents' AID. The agent AID is a concatenation of the platform's ID and its ID. So, when the search is done in the table previously shared, the MTS takes from the agent AID, the portion of the platform's ID and determines the I^{2}C address of the platform. That enables the MTS to compose the data message's header and the payload (Packet), which is sent to the platform where the *receiver agent* is registered.

When the message is received at the destination platform, the MTS looks in the payload for the receiver's *agent* AID and confirms that the agent is registered to the Agent Management Service (AMS). So, if the case occurs where the agent is de-registered (killed) from its AMS, the receiver platform will send a NON-UNDERSTOOD message to the *sender agent,* and that packet is discarded.

The static approach fits best into the CubeSat environment because the setting is contained enough to consider it a small network. The advantage of using the static approach is that it demands less CPU/Memory overhead for storing each *foreign agent's* AID and corresponding I^{2}C address of the platform it belongs to and does not add processing time for the inquire of the *foreign agent's* location. Furthermore, there is no bandwidth overhead because updates are not shared between the platforms, thus providing more control over the traffic that is routed.

Inter-platform Communication Implementation

The hardware that was used for the experimental set-up is the *MSP432P401R SimpleLink Microcontroller LaunchPad,* which uses Texas Instruments proprietary real-time operating system called *TI RTOS SYS/BIOS.* In such operating system applications are organized by a collection of threads, meaning that each application's function is encapsulated in a stream of instructions that are executed by the processor. The real-time operating system supports four different types of threads: Hardware interrupts (Hwi), Software interrupts (Swi), tasks, and the idle thread. *MAES Framework* architecture is based on tasks, as each task have its stack, apart from Hwi and Swi that share the same stack (system stack). This mechanism allows the task to take over a resource, and it makes other tasks wait for the release of the resource (block). Tasks are executed based on their priority level. There are 16 priority levels where level 0 is the lowest priority, and 15 corresponds to the highest priority, while level -1 means that the task is inactive. The priority level is defined by the developer; thus, it gives the developer the flexibility to decide how the tasks are executed.

As discussed before, *MAES' framework* architecture is based on tasks, meaning that each agent is equivalent to a single task within the framework. Therefore, the agent that has the highest priority is the one who is going to be running first and can request actions to another agent.

Each interaction between agents is done through the exchange of messages. TI-RTOS has a set of modules that allow tasks to communicate with each other, which are: queues, events, semaphores, and mailboxes. *MAES Framework* MTS is mapped to the usage of mailboxes; each agent has its mailbox, so, for example, a *sender agent* can signal when a message is posted to the *receiver agent* (notifies of an incoming message), for which the *receiver agent's* mailbox prepares to receive the data from the sender.

Agent Platform and Agent message classes extension

The base version of the *MAES Framework* contains four main classes: *Agent class*, *Agent Message class, Agent Platform class*, and *Behaviour-related classes,* as described by Chang-Zheng [5]. Thus, in order to bring out the capabilities of inter-platform agent communication, this section describes the extension of the Agent Message and Agent Platform classes, and it adds the *Communication class* to the already existing MAES library.

The extension of the *Agent Platform* class includes the methods needed to add the Agent Communication Channel (ACC). The ACC is the service on MAS who is in charge of determining if the messages are local or they need to be encapsulated into the payload of the ACL message to be delivered to a different platform (figure 1). The ACC is where the foreign messages arrive at the platform and where they are de-capsulated to be forwarded to the receiver agent.

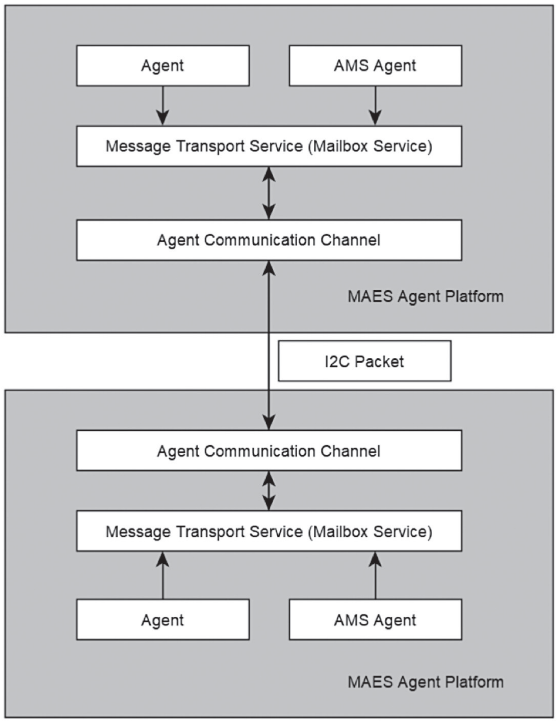

Agent
AMS Agent
Message Transport Service (Mailbox Service)
Agent Communication Channel
MAES Agent Platform
I2C Packet
Agent Communication Channel
Message Transport Service (Mailbox Service)
Agent
AMS Agent
MAES Agent Platform

Figure 1. Block description of the communication method used for MAES inter-platform communication.

On MAES the ACC is mapped as the I^{2}C Master, Slave, and Postman agents, see figure 2. These new agents are all initialized like a regular agent when each method runs it creates its unique task and handle (Agent_AID) and its corresponding mailbox and mailbox handle. Besides, when the intent is to use the inter-platform communication, the developer must know that for this case the agent priority allocation of MAES changes, which means is that the I^{2}C Slave agent now becomes the agent with the highest priority. As a result, the incoming messages from the other platforms are addressed as soon as possible, since they are considered as an external interruption that must take precedence over local message addressing. Then, the I^{2}C Slave's "*Postman,"* who is the agent that oversees the unpacking of the packets received by the I^{2}C Slave, becomes the agent with the second-highest priority. Next, the I^{2}C Master agent, who is the agent that proceeds with the encapsulation of the message contents into the I2C Packet, gets the third-highest priority as a result of the possible immediate need to send a response to an external message that arrived from a *foreign agent*. Lastly, the AMS agent becomes the agent with the fourth highest priority. The remaining *local agents* that are created follow the already established priority description from the base MAES Framework, which is defined by the developer.

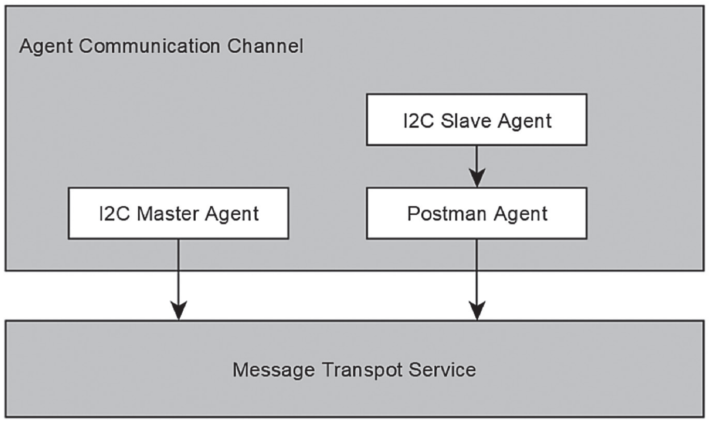
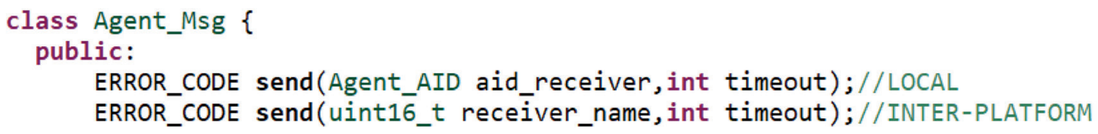


Figure 2. Block description of the Agent Communication Channel

Regarding the Agent Message class, the extension takes advantage of the polymorphism of the functions, which allows for a member function to act differently according to the type of object that invokes it. This property is used for the *send* function (see figure 3), where depending on the intended recipient, the message is addressed to either the *local agent* or to a *foreign agent*, the distinction lies in how the receiver agent is called. That is an important distinction because when an agent is registered to a platform and therefore becomes a member of the platform's AMS, the agent is given a unique name inside the platform it belongs. That is called the *Agent_ID*, and it is the name by which the other local agents refer to when they choose to send a message to it. Concerning the inter-platform communication, each agent has what can be called an "Alias" by which agents from different platforms can refer to it. The agent's "Alias" is an unsigned 16-bit integer (uint16_t), that concatenates the eight most significant bits (MSB), the number of the Platform's (uint8_t) identifier, and the eight least significant bits (LSB) the agent's (uint8_t) identifier. The agent identifier is a unique integer from 1 to 62 defined by the developer. This range is a convention defined by the MAES framework, in which the maximum agents a platform can host is 62.

Each Agent Alias represents the global name by which the agent is referred to inside the whole MAES Framework. That means that even if the agent's identifier is the same on more than one platform, there will not be a conflict of Aliases because the identification of the platform ID is the distinction between the Aliases. Also, it is not possible that inside a platform there is more than one agent with the same agent identifier because during compilation an error will be prompted to the user warning that the agent identifier is already being used.



```
class Agent_Msg {
  public:
      ERROR_CODE send(Agent_AID aid_receiver,int timeout);//LOCAL
      ERROR_CODE send(uint16_t receiver_name,int timeout);//INTER-PLATFORM
```

Figure 3. Snippet of class Agent_Msg regarding the send function.

Now, both *send* functions construct the ACL message with the message parameters: *sender, receiver, performative,* and *content* and post the message to the mailbox of the intended recipient. So, what happens when the intended recipient belongs to another platform? Which mailbox does the message is posted? The answer is that the ACL message is posted into the I^{2}C Master's mailbox, which is the agent that takes the recently posted ACL message (which is now considered the payload) and adds the envelope that contains the parameters: *to-from* and *acl-message,* which is described more in detail in the next subsection.

Agent Communication Channel Class

The Communication class encases the usage of the I^{2}C protocol, where each platform can act as an I^{2}C Master when a *local agent* chooses to send a message to a *foreign agent*, or act as Slave when an incoming message arrives with the destination of a *local agent.* The Master and Slave each have their task (each of them is an individual agent) and consequently their stack allowing for incoming and outgoing messages to be allocated in different queues. Therefore, there are no conflicts on the utilization of the I^{2}C channel, which means that the Master and Slave each use one out of the four *MSP432P401R Launchpad* modules.

After a message about the intention of a *local agent* to communicate with a *foreign agent* is posted in the I^{2}C Master's mailbox. The I^{2}C Master agent receives the message and proceeds with the encapsulation of the contents into the I^{2}C Packet. The ACL message with the *sender Alias*, *receiver Alias, performative,* and *content* are added to the payload structure of the packet. Finally, the *Platform receiver address* and, *Platform sender address* are placed in the header structure of the packet, as shown in figure 4.

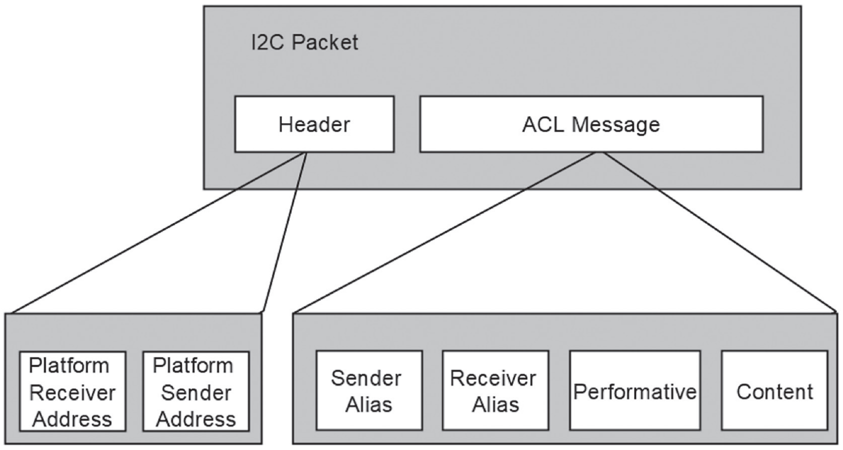


Figure 4. I^{2}C Packet encapsulation of the inter-platform message exchange

The I^{2}C Master determines the *Platform receiver address* looking at the *receiver Alias* field and correlates the MSB 8bits of the agent's (uint16t) Alias with the table of I^{2}C platform addresses that are already registered at start-up. When the 8_bit I^{2}C address is determined, it is placed in the *Platform receiver address* field. The *Platform sender address* parameter is filled by copying the local platform's I^{2}C address.

The I^{2}C packet refers to a uint8 variable that encapsulates the header and the ACL message, that is passed by value to the I^{2}C Master driver API, where a parameter is a uint8 variable previously mentioned. Other filled parameters are the 8_bit I^{2}C receiver address, the packet size and the intention of writing information to the I^{2}C Slave of the receiver platform.

During the reception process, the I²C Slave agent receives the I²C packet and forwards it to the "Postman," who first checks the *receiver Alias* field and compares the 8 MSB with the platform's ID to determine that the message is intended for the said platform. Then, the 8 LSB of the *receiver Alias* are compared to find the match. The "Postman" does an inquiry to the platform's AMS using the *receiver Alias* and asks for the Agent's_AID. Once the "Postman" has the Agent's_AID, it proceeds to post the ACL message to the newfound *local agent's* mailbox. After the *local agent* receives the message, it proceeds with the instructions given by the *performative* and *content* of the ACL message that was already defined by MAES intra-platform methods. In case there is no correspondence between the platform address or the agent "Alias," the "Postman" constructs an I²C packet with a message reply to the *sender Alias* with the *performative* NOT_UNDERSTOOD.

Multi-Agent System Inter-Platform Communication Implementation and Testing

This section is devoted to the operating performance of the inter-platform communication. For these experiments, three applications are used to show the verification of the inter-platform communication protocol execution, the precision of the message round trip time and the relation between the delay and the number of agents accessing simultaneously to the I²C bus.

Hardware Setup

The hardware setup used to perform measurements is described in what follows. The Launchpad used is the Texas Instruments MSP432P401R SimpleLink Microcontroller LaunchPad, who features 64KB RAM, 256KB Flash memory, a low power ARM 32-bit Cortex-M4F microcontroller (MCU), up to 48 MHz system clock. Additionally, it has an onboard probe for programming and debugging, two buttons and two LEDs for user interaction, backchannel UART through USB to PC, lastly four I²C modules and up to eight SPI modules. [11]

In order to carry out the measurements to verify the behavior of the implementation, two *MSP432P401R Launchpads* were used. They were connected between each other using the pinout described in table 2 for the I²C Master and Slave connection. That is because of the need to use two libraries (I2CMaster.h and I2CSlave.h) from Texas Instruments that run with the Real-Time Operating System (RTOS). Here, the bitrate is 100 kb/s, and the value of the two pull-up resistors is 10 kΩ, with a voltage supplied of V_{cc}=3.3 V.

Table 2. Pin set-up for one Launchpad

Master Pins	Pull-up resistor	Slave pins
UCB1SDA (P6.5)	10 kΩ	UCB0SDA (P1.6)
UCB1SCL (P6.4)	10 kΩ	UCB0SCL (P1.7)
GND	NONE	GND

Inter-platform communication protocol verification

This experiment aims at verifying the implementation of the inter-platform communication protocol. The scenario consists of having an agent from one Launchpad send a message to a *foreign agent* from another Launchpad and expect an answer from the *foreign agent*. All measurements are performed using the Texas Instruments proprietary Code Composer Studio (CCS) version 8.2.0.00007 and the RTOS Analyzer's Execution Analysis tool, to examine how the agents execute and interact within the MAES Framework.

Therefore, two *MSP432P401R Launchpads* were connected with each other, and the MAES Framework updated library was installed on both of them. In this scenario, the size of the mailboxes for the I²C Master and I²C Slave is of 3 messages each, and both Launchpads allocate a total of 5 agents that are listed in table 3 for Launchpad 1 and in table 4 for Launchpad 2.

Table 3. Agents on Launchpad 1.

Name	Priority
I²C Slave_Task	15
Postman_Task	14
I²C Master_Task	13
AMS_Task	12
Write_Task	1

Table 4. Agents on Launchpad 2.

Name	Priority
I²C Slave_Task	15
Postman_Task	14
I²C Master_Task	13
AMS_Task	12
Read_Task	1

All measurements were executed using launchpad 1 as reference considering it houses agent "Write" Therefore, the results reported below correspond to Launchpad 1's viewpoint. First, upon start-up, all the agents are initialized before they can perform any of their tasks. The initialization process takes place in compliance with the design of the MAES framework, which first sets up sequentially the agents from the highest to the lowest priority. This is shown in figure 5, where the I²C Slave, the Postman, and the I²C Master agents are the ones that initialize first. These are followed by the AMS agent, who then allows agent "Write" to initialize to be ready for operation and, immediately after the agent "Write" sends a message with the *performative: REQUEST* to agent "Read" located on Launchpad 2.

As expected, right after the initialization of all the agents, agent "Write" takes over the execution and composes the ACL message with the *performative: REQUEST, sender, receiver and content* and posts the message to the I²C Master's mailbox, as is shown on mark 1 from figure 6. Mark 2 shows the I²C Master preempting agent "write" to receive the message from its mailbox, compose the packet with the header and payload (ACL message) and, initializing the I²C channel after it returns execution to agent "write"(mark 3) for it to terminate the routine of message composition and sending. Marks 4 & 5 show the I²C Master agent sending the message through the I²C bus to Launchpad 2.

Finally, the "idle loop" task, which is part of the TI RTOS, runs when no other task is scheduled to run. For the case shown in figure 7 it runs because it is waiting for the arrival answer message from Launchpad's agent "Read" (mark 1) after mark 2 shows how the I²C Slave executes when it has received the answer message from Launchpad 2. On mark 3, the Postman takes over execution, where the header of the packet gets read, and the Postman determines that the message is intended for agent "Write" who is registered to the platform's AMS, so it posts the packet's payload (ACL message) to agent "Write's" mailbox. Mark 4 shows agent "Write" executing, meaning it receives the confirmation message from agent "Read" thus marking the end of the send/receive cycle. Afterwards, the cycle starts again for the next message.

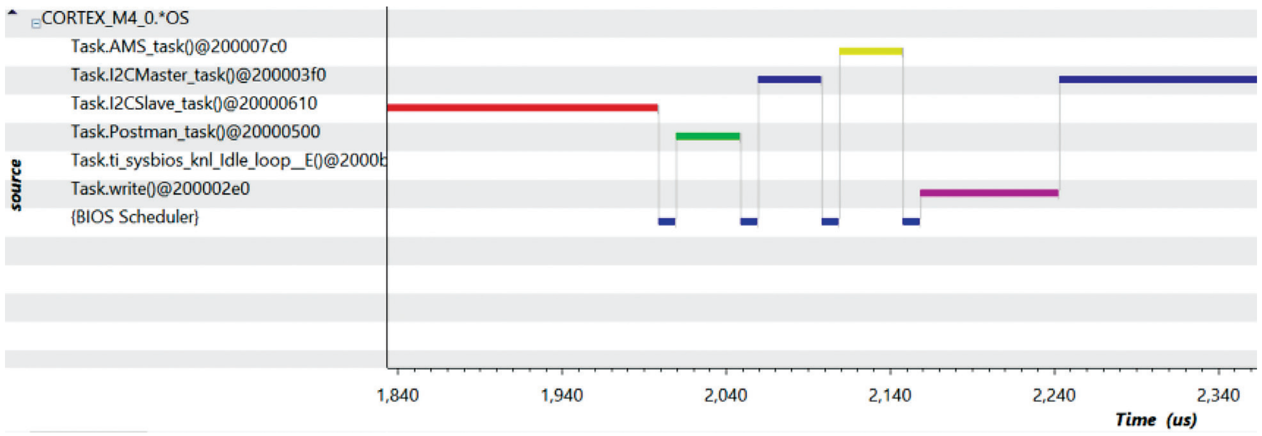


Figure 5. Agent Initialization on Launchpad 1.

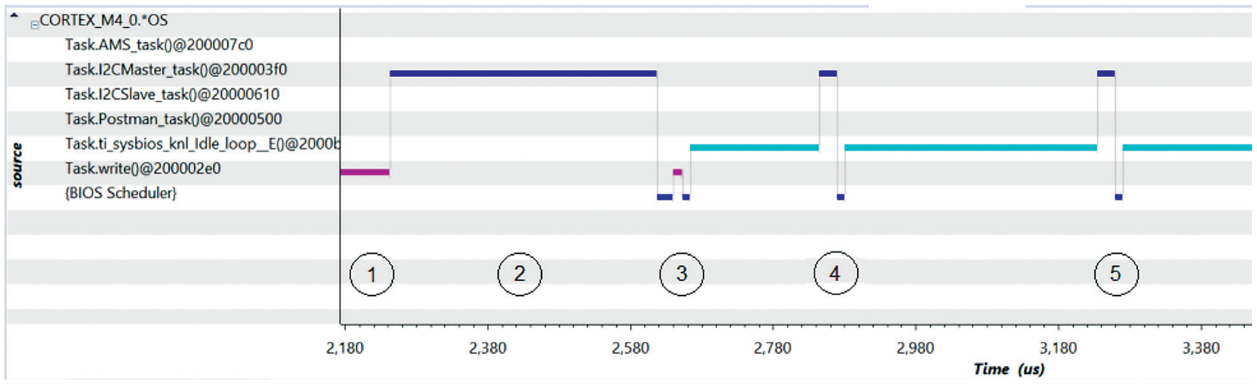


Figure 6. Message trace of a sent message on Launchpad 1.

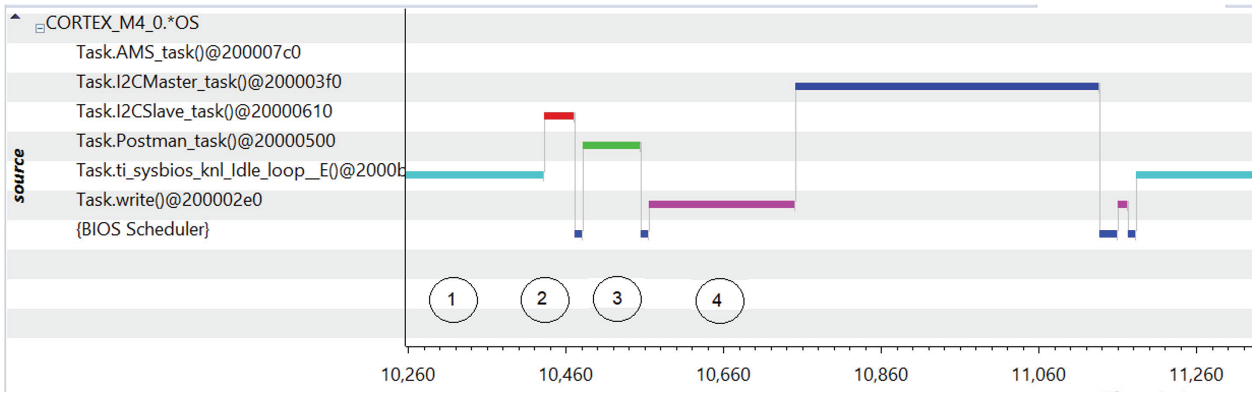


Figure 7. Message trace of a received message on Launchpad 1.

Precision of the Inter-platform communication protocol

This experiment is intended to determine the precision of the I²C bus and MAES Framework while transmitting and receiving inter-platform messages. The scenario is the same as the one described before. A sample of 300 messages was collected, where the time that is taken by agent "Write" to send the request message and to receive the answer from agent "Read" was measured. Figure 8 shows the histogram obtained from measuring the round-trip time (RTT). It can be observed that the probability of having an RTT between 5.82 ms and 5.84 is around 45%, which can be considered as the minimum time an agent has to wait for a response after sending a message to a *foreign agent* when the I²C bus is working with perfect conditions.

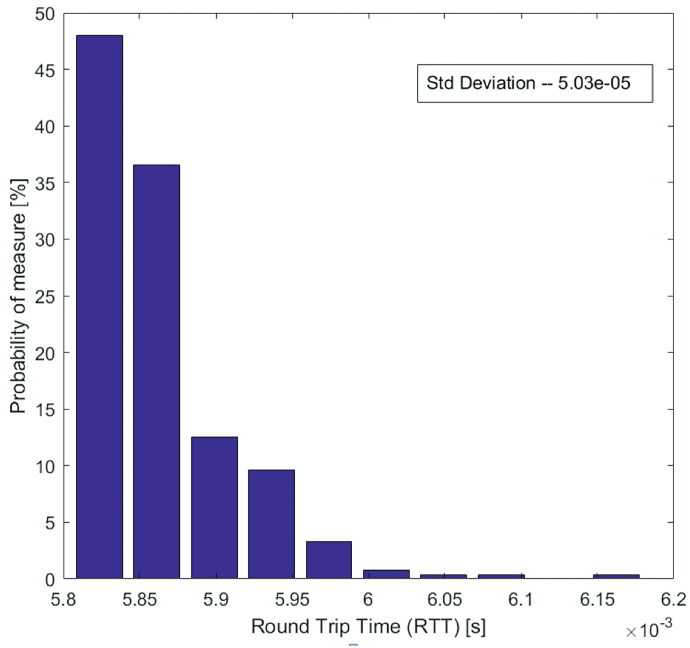


Figure 8. Histogram showing the round-trip time of 300 samples sent from Launchpad 1 to Launchpad 2.

Relationship between the number of agents and the delay

The purpose of this experiment is to determine the effect on the round-trip time on an agent when the number of agents transmitting at the same time on the I²C bus increases. To obtain the results, the same setup, was used. This time the number of agents on each Launchpad was increased simultaneously, first starting with one agent (agent "Write" on Launchpad 1 and agent "Read" on Launchpad 2) until reaching 14 agents operating on each Launchpad, counting for a total of 14 trails. On each trial, all agents were triggered to start sending a message with the *performative: REQUEST* at the same time, thus measuring the round-trip time for an agent "Write." A number of 300 samples were considered for each trial. The measurements are taken from Launchpad 1's viewpoint. Figure 9 shows a Box Plot, where it can be observed that the precision of the samples holds independently of the number of agents running on the Launchpad, as seen before. Moreover, the Box Plot shows that when the number of agents accessing simultaneously the I2C bus increases, the round-trip time for the agent "Write" increases as well. This result indicates that the relationship between the number of agents on a Launchpad and the delay increases linearly.

This linear relationship resembles the results obtained for an analysis conducted to characterize the bus performance of satellite missions. In this case, CAN bus was used as the communication channel between the attitude and orbit control subsystem (AOCS), that is based on a multi-agent system application, and a distributed architecture of sensors. According to Carvajal-Godinez [12], the results show that the bus utilization increases linearly, in the function of the number of nodes in the network as long as the bus saturation is not reached. For MAES Framework maximum capacity of the I^{2}C was not reached because the Launchpad ran out of RAM. That forced the experiment to be stopped at 14 agents. However, both results coincide on the fact that for Multi-Agent based systems, the delay of the messages hold a linear relationship with the number of agents/nodes for I^{2}C and CAN buses, as long as the buses are not working at maximum capacity.

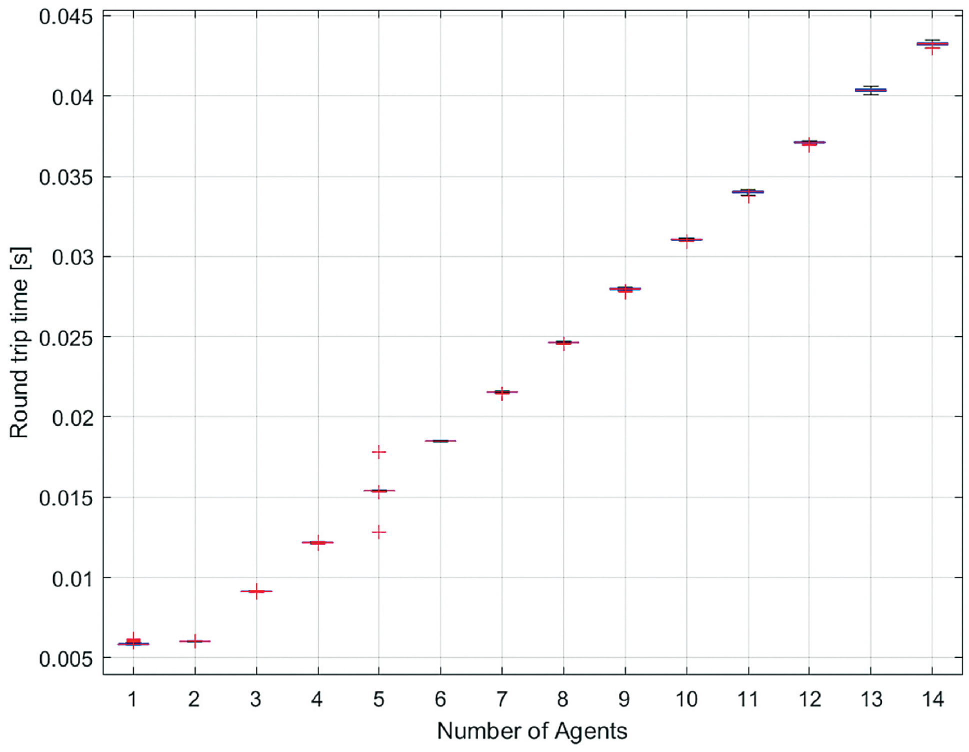


Figure 9. Box Plot of the round-trip time of agent "Write" vs. the number of agents running

Conclusions and Future Work

This paper reports the design of a communication architecture that allows platforms, which are running Multi-Agent framework for Embedded Systems (MAES), the ability for their local agents to communicate with foreign agents thus extending the distributed control architecture that serves as a tool for developers to design embedded real-time applications for future CubeSat target missions.

It was determined that I^2C fits the design requirements for the implementation of inter-platform communication using MAES framework because of its scalability, ability to arbitrate messages, and its low implementation complexity.

A static message routing strategy was selected for the implementation because the CubeSat setting is contained enough not to demand a complex execution in order to handle inter-platform messages.

The extension of MAES capabilities framework was done by adding the Agent Communication Channel (ACC), who is the service on MAS that is in charge of determining if the messages are local or need them to be encapsulated into the payload of the ACL message and deliver them to a different platform. FIPA's ACC is mapped as the I^2C Master, Slave and Postman agents. The experiments show that the round-trip time of a message is below 5.84 ms in around 50% of the cases when there is no congestion. Further experiments show that the relationship between the delay of the exchange of inter-platform messages increases linearly with the number of agents that are simultaneously exchanging messages on the I^2C bus while saturation is not reached.

The inter-platform communication protocol gives the MAES framework extended capacities to perform inter-platform communication. Thus, the control unit architecture can be broadened, allowing agents from different platforms to interact and perform cooperatively routines designed by the developer, so it is not limited to the capabilities of just one platform.

Future work should consider the integration of mobility capabilities into MAES framework, which allows agents to move between platforms promoting a flexible inter-operability to the framework, so it can be arranged as it finds most convenient. Likewise, in future work, there can be an extension of the Communication Class to allow wireless communication technologies to be the channel between platforms. That could give the developer the ability to design its application with more flexibility. Also, the addition of Wireless Communication can provide redundancy based on a unique technology to the inter-platform communication, thus increasing the reliability of MAES Framework.

References

[1] M. Swartwout, "The First One Hundred CubeSats A Statistical Look," *Journal of small Satellites,* vol. 2, no. 2, pp. 213-233, 2013.

[2] P. Maes, "Artificial Life Meets Entertainment: Lifelike Autonomous Agents," *Communications of the AC,* vol. 38, no. 11, pp. 108-114, 1995.

[3] D. M. Glavic, "Agents and Multi-Agent Systems: A Short Introduction for Power Engineers," Montefriore Institute, 2006. [Online]. Available: http://www.montefiore.ulg.ac.be/~glavic/MAS-Intro_Tech_report.pdf. [Accessed 19 3 2018].

[4] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, Distributed Systems Concepts and Design, Addison-Wesley, 2012.

[5] C. Chan-Zheng and J. Carvajal-Godínez, "A Multi-Agent System Framework for Miniaturized Satellite," *Tecnología en Marcha,* vol. 32, no. 1, pp. 54-67, 2018.

[6] F. Bellifemine, G. Caire and D. Greenwood, Developing multi-agent systems with JADE, John Wiley and Sons,Ltd, 2007.

[7] F. f. I. P. Agents, "Foundation for Intelligent Physical Agents," FIPA, 2018. [Online]. Available: http://www.fipa.org/. [Accessed 19 3 2018].

[8] J. D. Jensen and D. C. M. Swenson, "Command and Data Handling Subsystem Design for the Ionospheric Observation Nanosatellite Formation (ION-F)," *SSC00,* vol. VII, no. 3, 2002.

[9] C. Nagarajan, R. G. D'souza, S. Karumuri and K. Kinger, "Desing of a Cubesat Computer Architecture using COTS Hardware for Terrestrial Thermal Imaging," in *2014 IEEE International Conference on Aerospace Electronics and Remote Sensing Technology*, 2014.

[10] S. Sabri, S. Yuhaniz and K.Kamardin, "Designing a low Cost CubeSat's Command and Data Handling Subsystem kit," *ARPN Journal of Engineering and Applied Sciences,* vol. 11, no. 10, pp. 6259-6264, 2016.

[11] Texas Instruments, "MSP432P401R SimpleLink Microcontroller LaunchPad," Texas Instruments, 2018.

[12] J. Carvajal-Godinez, J. Guo and E. K. A. Gill, "Effects of Saturation for High-Throughput Satellite Buses," in IEEE Transactions on Aerospace and Electronic Systems, vol. 56, no. 2, pp. 1014-1025, April 2020, doi: 10.1109/TAES.2019.2940341.

[13] S. Interiano, "A Communication Protocol Design for Multi-Agent Systems Framework for Embedded Systems (MAES)-based platforms," Politecnico di Milano, Milano,Italy, 2018.