Dissertation
Master in Computer Engineering – Mobile Computing

Framework for supporting JavaScript-Based Mobile Agents

Carlos Alfredo Silva Villafuerte

Leiria, *June 2018*

Dissertation
Master in Computer Engineering – Mobile Computing

Framework for supporting JavaScript-Based Mobile Agents

Carlos Alfredo Silva Villafuerte

Dissertation developed under the supervision of Professor *Nuno Alexandre Ribeiro Costa, teacher* at the School of Technology and Management of the Polytechnic Institute of Leiria, co-supervision of Professor *Carlos Fernando de Almeida Grilo*, teacher at the School of Technology and Management of the Polytechnic Institute of Leiria and co-supervision of Professor Jorge Luis Veloz Zambrano, teacher at School of Computer Science of the Universidad Técnica de Manabí.

Leiria, *June 2018*

This page was intentionally left blank

Acknowledgements

A special thanks to Professor Nuno Costa for his dedication, guidance, motivation and valuable professional constructive suggestions during the arrangement and execution of this dissertation work. Furthermore, I am grateful for the patient guidance and judgment given by Professor Carlos Grilo and I am especially indebted with Professor Jorge Veloz for his useful critiques and advices provided during this work. All of them have actively worked to provide me with the help to pursue my goals. It has been a privilege to have their guidance in a work of this magnitude.

Thanks to all authorities of the Universidad Técnica de Manabí and the Departamento de Relaciones Internacionales, Convenios y Becas for giving me the opportunity to go to specialize academically abroad.

Thanks to the Secretaría de Educación Superior, Ciencia, Tecnología e Innovación de Ecuador (SENESCYT) and the Instituto de Fomento al Talento Humano, this research work would not have been possible without it financial support.

Nobody has been more important to me in the pursuit of this goal than my family. I wish to thank my loving and supportive wife, Lady Cedeño, who was always there in difficult times and especially to my wonderful children, Keyla S., Brigitte S. and Carlos S., whom provide unending inspiration and happiness to my life during all time.

Finally, I wish to thank my parents for their support and encouragement throughout my formative education. I wish express my very profound gratitude to Andrea Alcívar and Emilio Cedeño, who always offered their unconditional help.

This page was intentionally left blank

Abstract

The evolution of technology in interconnection solutions, such as networks or the Internet, and the emergence both of wireless sensors networks and distributed systems allowed many communication architectures to appear, being the Client-server architecture the most common. Here, we present a dissertation work about the mobile agents computing paradigm. A middleware and a mobile agent framework have been developed using the JavaScript language that allows the development, execution and the ability to move JavaScript mobile agents through the local network and Internet using Node.js for desktop operating systems and React Native for mobile operating systems, such as Android and iOS. This initiative arose as a way of dealing with problems raised by the considerable amount of existing Java based mobile agents platforms, which force the installation of the Java Virtual Machine on the devices, making complicated its execution in operating systems like macOS, iOS and others operating systems not compatible with Java.

Keywords: Mobile agents, Middleware, Framework, JavaScript, Node.js, React Native.

This page was intentionally left blank

Resumo

A evolução da tecnología relativamente a soluções de comunicação, tais como as redes ou a Internet, e o surgimento das redes de sensores e sistemas distribuídos permitiram o aparecimento de várias arquiteturas de comunicação, sendo a mais comum a arquitetura Cliente-servidor. Nesta dissertação é apresentado um trabalho sobre o pardigma dos agentes móveis. Mais concretamente, neste trabalho foi desenvolvido um middleware e uma framework para agentes móveis utilizando a linguagem de programação JavaScript que permitem o desenvolvimento e execução de agentes com a capacidade de se moverem através de uma rede local e da Internet, utilizando Node.js em sistemas desktop e React Native em sistemas operativos móveis, tais como Android ou iOS. Este trabalho surgiu como uma forma de resolver as lacunas deixadas pela quantidade considerável de plataformas de agentes móveis baseadas em Java, que forçam a instalação de uma Java Virtual Machine nos dispositivos, dificultando a sua execução em sistemas operativos como macOS, iOS e outros sistemas operativos não compatíveis com Java.

Palavras-chave: Agentes móveis, Middleware, Framework, JavaScript, Node.js, React Native.

This page was intentionally left blank

Table of Contents

This page was intentionally left blank

List of figures

List of tables

List of acronyms

API	Application Programing Interface
ASDK	Aglets Software Development Kit
AWB	Aglets WorkBench
CSS	Cascading Style Sheets
DSN	Distributed Sensor Network
FIPA	Foundation for Intelligent Physical Agents
FTP	File Transfer Protocol
GPS	Global Positioning System
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext transfer protocol
IBM	International Business Machines Corporation
IoT	Internet of Things
IP	Internet Protocol
JADE	Java Agent DEvelopment Framework
JS	JavaScript
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LGPL	Lesser General Public License
MQTT	Message Queuing Telemetry Transport
NAT	Network address translation
NPM	Node Package Manager
OS	Operating System
QPL	Qt Public License
REV	Remote Evaluation
RPC	Remote Procedure Calls
SMTP	Simple Mail Transfer Protocol
TCL	Tool Command Language
TCP/IP	Transmission Control Protocol/Internet Protocol
WebRTC	Web Real-Time Communication
WSN	Wireless Sensor Network
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

This page was intentionally left blank

1 Introduction

Traditionally, applications in distributed systems have been structured using the client-server paradigm (request/response model), in which client and server processes communicate either through message passing (asynchronous) or Remote Procedure Calls (RPC) (synchronous). When using the synchronous approach, the client suspends itself after sending a request to the server, waiting for the results of the call. An alternative architecture is called Remote Evaluation (REV). In REV, the client, instead of invoking a remote procedure, sends its own procedure code to a server and requests the server to execute it and return the results. The mobile agents paradigm is an evolution of these antecedents.

Software agents can be used for information searching, filtering and retrieval, or for electronic commerce on the Web, thus acting as personal assistants for their owners [26]. Software agents can also be used in low-level network maintenance, testing, fault diagnosis, and for dynamically upgrading the capabilities of existing services. For example, some researchers propose a mobile agent-based solution for solving energy sink-hole problems [1], due to fact that repeated and continuous transmission of data to the sink leads to energy loss in all the nodes in the case of a flat Wireless Sensor Network.

1.1. Technical contextualization

Although networking offers many potential benefits, it can raise just as many challenges when existing centralized systems must be retrofitted for communication and distribution. Tying together two systems quickly reveals incompatibilities in command languages and data formats and, as more systems are added, the incompatibilities multiply. Mobile agents can help solving these kind of problems by serving as a *lingua franca* that many systems can share. Because they include code, agents can be used to map between complex interfaces on heterogeneous systems. The one requirement is for the agent language primitives to be executable by each type of system [3].

This idea is not new. In 1968, the original Arpanet designers were already considering the problem of emulating the terminals for one timesharing system on the terminals of another. Their solution was the Decode–Encode Language (DEL). On connecting to a remote system, the appropriate terminal emulator would be sent back to the client as DEL code. Unfortunately, this innovative work was set aside before it could be fully developed [3].

Mobile agents are code-containing objects that may be transmitted between communicating participants in a distributed system. As opposed to systems that only allow the exchange of nonexecutable data, systems incorporating mobile agents can achieve significant gains in performance and functionalities [3].

There are many advantages for the mobile agents paradigm. Some of them include the reduction of the used network bandwidth, distribution of processing and loading through the hosts of the network, support for a more flexible peer-to-peer model, scalability and control decentralization [4].

The mobile agent concept is illustrated in Figure 1. A client computer consists of an application environment, for example, OS/2 or Microsoft Windows, which contains one or more applications for interaction with a remote server. These applications may include information searching and retrieval, transaction front-ends, or email clients. These applications are bound to an execution environment for mobile agents [5].

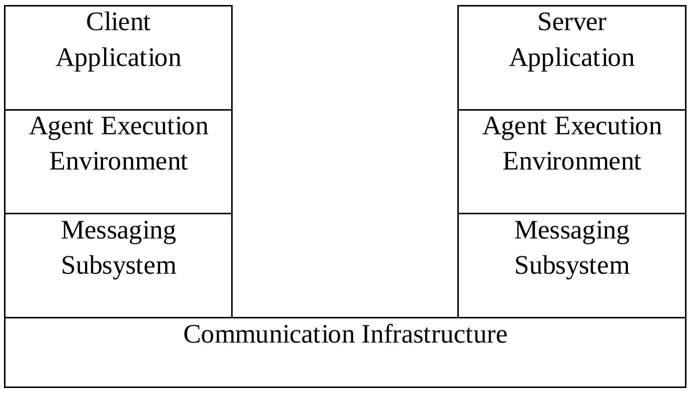

Client Application
Agent Execution Environment
Messaging Subsystem
Server Application
Agent Execution Environment
Messaging Subsystem
Communication Infrastructure

Figure 1. Conceptual model for mobile agent computing.

Related work in this area is treated in Chapter 2 and reveals that there have been middleware and frameworks developed for mobile agents support, where middleware packages are developed in Java and some in other programming languages that have no compatibility with iPhone Operating System (iOS) and macOS. Currently, some of them have no updates and have limitations for their use and execution in the large number of devices that exist today, including single-board computers, such as Raspberry Pi. Many of these operating system have different versions, which are modified types of open source or proprietary software.

1.2. Motivation

The mobile agents paradigm has been given little attention and has been little estimated as a technology solution for some scenarios where software applications could be implemented. Furthermore, it should be mentioned that using the mobile agents paradigm implies significant changes in the communication architecture and applications source code.

There is a large amount of programming languages in which applications can be developed to be executed in different operating systems. A programming language for mobile agents must be able to express their construction, transmission, receipt, and subsequent execution. Its implementation must address such practical problems such as handling architectural heterogeneity between communicating machines and providing sufficient performance for applications based on agents [3]. From these, we have chosen JavaScript as the main programing language, using Node.js runtime [6] for desktop Operating Systems (OS) and React Native Facebook framework [7] for mobile OS. The main reasons for our choice are the following: JavaScript offers code manipulation through object-oriented programming (the code-containing objects that represent mobile agents), it is a strong typing language and there is a large number of libraries available on the Internet.

1.3. Objectives

To fulfill the main aim of this dissertation work, the following objectives were stablished:

- To develop a JavaScript middleware for JavaScript Mobile Agents support that allows mobile agents to move through the network both desktop computers, with different OS, such as Windows, Linux and macOS and mobile devices, with Android and iOS;
- To develop a mobile agent framework for desktop OS in order to help programmers developing, executing, tracking mobile agents through the network and gathering host information where the middleware is running;
- To perform functionality tests in real devices with Android and iOS OS with different release versions.

1.4. Dissertation structure

The rest of this dissertation is organized follow:

In Chapter 2, the state of the art is presented and discussed. In Chapter 3, we describe the proposed solution specification. In Chapter 4, the implementation is presented with real devices. Chapter 5 describes the evaluation environment and test's results. Finally, conclusions and future work are presented in Chapter 6.

This page was intentionally left blank

2 State of the art

One of the goals in the area of distributed systems is to allow communication between devices with the least amount of bandwidth consumption [34]. The appearance of Wireless Sensor Networks (WSN) represented a challenge for distributed systems applications. In distributed systems. Reducing the unnecessary communication not only contributes to a lower bandwidth consumption of bandwidth, but also to a decrease in device energy consumption, thereby increasing the useful life of batteries.

With the appearance of the World Wide Web, some computational paradigms emerged to optimize the use of resources, both computational and network resources. For example, the emergence of remote evaluation (REV), code on demand and the mobile agents paradigm appeared as a solution for clients that have specially bandwidth limitation and other limitations [35].

The client-server model is commonly used to develop applications (some are presented in Section 2.1.1), executing the process on server side, lightening the load to the client (web browsers or client applications); even some services offered by large companies of the Internet, e.g. Facebook and Google, offer to the developers community the possibility of consuming services through Application Programming Interfaces (API). For example, obtaining personal information, instant messaging services (push messaging), Global Positioning System (GPS) navigation services, and others. These services are used and developed following the client-server paradigm. In the last couple of years, the approach to reduce the burden of processes on the client's side has been resumed again with the appearance of platforms for application development executed on the server side. For example, the possibility of developing web applications using the Java programming language through the Vaadin framework. Another case are web applications developed with JavaScript using tools like Node.js runtime with Angular development framework or React framework.

The raise in both the development of platforms for mobile agents support, as well as the specification and development of programming languages [3] that have the necessary characteristics for the development of mobile agents platforms started a long time ago, with the World Wide Web. Actually, some of them have been abandoned [16] [38] [37] or had the last update some years ago [25] [37] [43], but others have survived over the years [15] [43]. Some projects are still maintained and offering updates support [15] [43] thanks to the developers community. Some are open-source, while others have remained with a proprietary license [16] [43]. Some features and limitations of this agent frameworks are detailed in Section 2.2.

2.1. The computational paradigm

There are different computational paradigms proposed by the research community. The four main paradigms are:

- Client-Server paradigm;
- Remote Evaluation paradigm;
- Code on demand paradigm;
- Mobile agents paradigm.

The following sections describe each one of these paradigms.

2.1.1. Client-server paradigm

In the client-server paradigm the server is limited to the execution of some procedures and data storage [29]. This paradigm is the most used in the development of distributed applications for desktop or web applications. In distributed desktop applications it is usually specified that the server offers the storage service through a database engine, such as MySql, Postgres, Oracle or MariaDB or others (see Figure 2). In web applications, the server offers storage services accompanied with a web server that offers all the resources to the client software (web browsers) to download the necessary code for its execution and do the necessary connections.

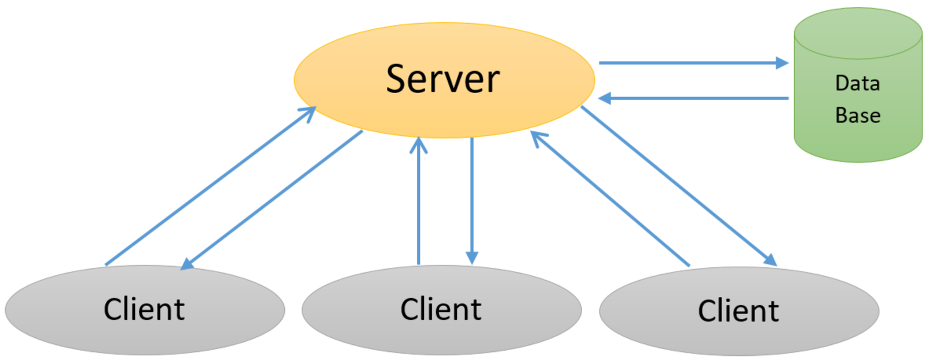


Figure 2. Client-server architecture.

There are a lot of applications that use the client-server paradigm, such as, File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP) and Hypertext Transfer Protocol (HTTP) [27].

2.1.2. Remote evaluation paradigm

The remote evaluation (REV) paradigm implies that the client sends requests to the server in the form of code to be executed. The server performs the compilation and execution of the code sent by the client and, after its execution, returns the results to the client [29]. In this paradigm the sending of the code is done from the client to the server, which is important since REV is used to define any technology that involves the transmission of code from a client host to a server host. This paradigm belongs to the mobile code family.

2.1.3. Code on demand paradigm

The code on demand paradigm refers to any technology in which a client needs to execute a task, but in which part of the code is missing. A host within the network must provide the necessary code to the client, which receives the code and complies it with the execution of the task [29].

This paradigm is currently ceasing to be used due to great security incentives when implementing this technology. For example, the use of Java applets, which are downloaded from the server and executed by the client. Another example are applications developed with Adobe ActionScript executed on platforms such as flash player and JavaScript (client side). Commonly, they are used in web applications, in which a client makes the request to the web page, the server returns the web page (code) that contains a link to the code, which is then executed in the web browser and locally on the client computer. This is usually used to perform operations and interact directly with devices that are located on the client side, such as printers, biometric equipment, among others.

2.1.4. Mobile agents paradigm

The mobile agents paradigm refers to an approach where the code has a great difference as far as the paradigms mentioned above are concerned. In those paradigms the code is inactive until its execution [29]. While in the mobile agents paradigm approach, the code (agent) is executed and remains in an active form, adding autonomy and mobility characteristics among all the hosts of the network.

The mobile agents paradigm is based on the principle that a software or a code part, called **agent**, has the ability to move in the current state to another device and continues with its execution. (see Figure 3). However, an agent based platform must provide a set of capabilities to the agent. Chapter 3 details the basic features that an agent-based platform should provide.

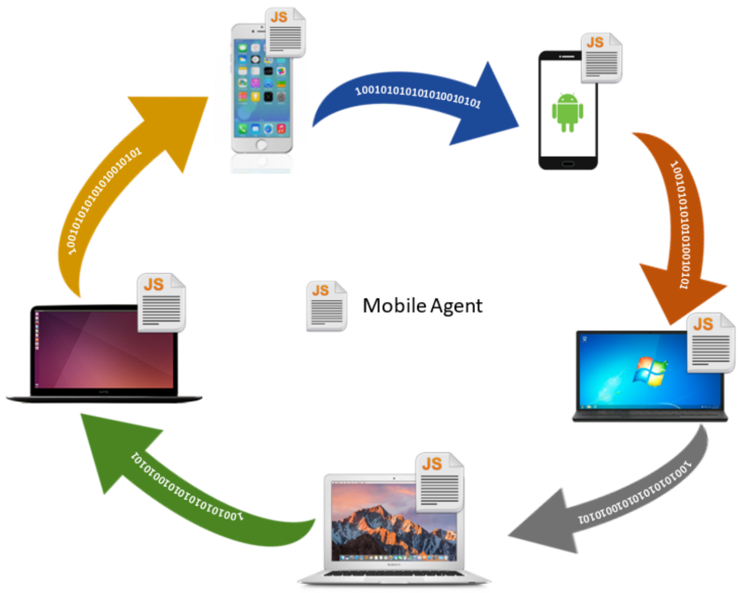


Figure 3. Mobile agents paradigm.

The Mobile Agent, the Remote Evaluation and the Code on Demand paradigm are part of the code mobility family, but they work and have very different approaches. For example, in REV, a client sends the code to a server for its execution. Nonetheless, in code on demand paradigm, a client downloads the code from the server to execute it locally. While, in mobile agents paradigm, the agents are code objects with the ability to migrate (move) autonomously among the available devices. However, the client-server paradigm is the most used and chosen during the architectural design of system (applications) today, among the main reasons for using this paradigm is the ease of maintaining applications when deploying updates and new services.

2.2. Mobile Agents based middleware and frameworks

In the past decade, many mobile agent platforms have been developed. Some of the well-known mobile agent platforms include Mole [8], Aglets [9], Concordia [10], D'Agents [11], Ara [12], TACOMA [13], JADE [15], Ajanta [16], Tryllian's agent development kit [17], Fipa-os [18], Grasshopper [19], JACK Intelligent Agent [43] and Zeus [21]. However, most of them, such as Mole, Aglets, Concordia, JACK, Ajanta and JADE, were developed to support only Java mobile agents. D'Agents supports mobile agents written in Tcl, Scheme and Java, whereas Ara supports those written in Tcl, C, C++ and Java. TACOMA was originally developed to support Tcl mobile agents, and it was subsequently extended to support mobile agents written in multiple languages including C, Tcl, Perl, Python and Scheme.

In the research work [22] the authors performed some evaluations of platforms for mobile agents. The used criteria were the following:

- Communication: support for inter-platform messaging;
- Agent mobility: strong (ability of the system to migrate code and execution state of executing unit), weak (migration of code only);
- Clean, efficient method of migrating, threads needed to be recreated/restarted by an awaiting daemon;
- Usability and documentation: user and developer level of acceptance.

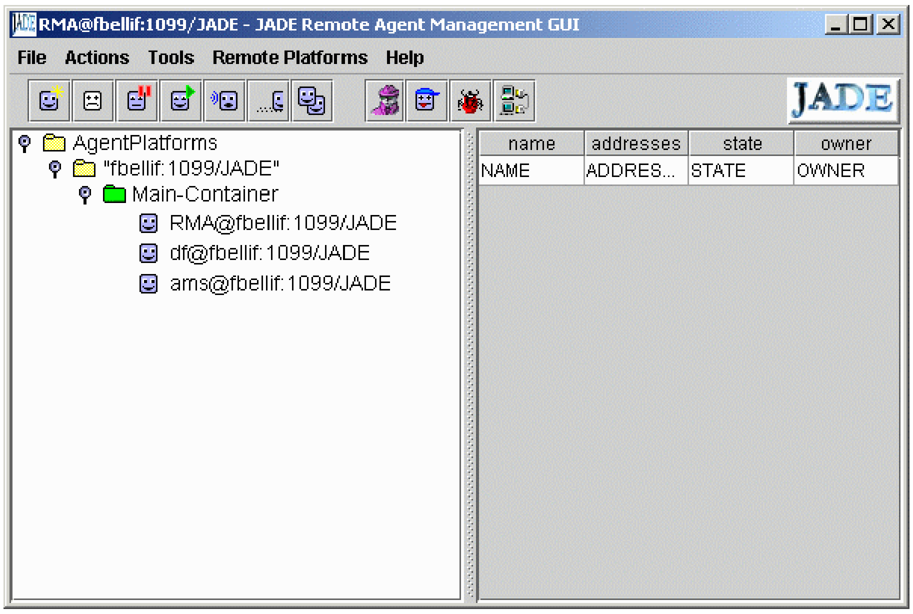
The authors concluded that, according to the evaluation criteria, among the best frameworks for mobile agents are: JADE and Aglets.

An agent-based platform must provide some features, for example: mobility, autonomy, intelligence, security, communication, and others. Some agent-based platforms are described below.

2.2.1. Java Agent DEvelopment Framework

The Java Agent DEvelopment Framework (JADE) is a software framework fully developed in Java to aid the development of peer-to-peer agent applications in compliance with the Foundation for Intelligent Physical Agents (FIPA) [2] specifications for interoperable intelligent multi-agent systems. JADE is an Open Source project, and the complete system can be downloaded from the JADE home page [15]. Furthermore, it provides a peer to peer agent communication based on the asynchronous messages passing paradigm.

JADE offers a graphical tool for the depuration and implementation of solutions based on mobile agents. The remote Graphical User Interface (GUI) allows the remote management, monitoring and controlling of agents status (see Figure 4). The minimal system requirement is the Java version 5.



name	addresses	state	owner
NAME	ADDRES...	STATE	OWNER

Figure 4. JADE Remote agent management GUI [23].

JADE gives developers the necessary libraries (i.e. the Java classes) to develop agents, and a run-time environment called container, which must be active on the device before doing the execution of the agents. The communications between the agents is done through asynchronous messages, the most used in distributed systems. The agent identification is done through their name and agents can send communication messages to agents that do not exist (or not yet exist) [24].

One of the limitations of this framework is that it only supports agents developed in Java, limiting its execution in iOS devices, such as iPhone, Apple Watch and others. JADE is a free software, distributed by Telecom Italia [36]. It is open source under the terms and conditions of the LGPL (Lesser General Public License Version 2) license. The last available version is JADE 4.5.0; it was released in July 2017.

It currently offers support for mobile devices with the Android Operating System thanks to the contribution of the LEAP project. Additionally, it offers support for executing agents in networks with personalized configurations, such as Network address translation (NAT) and firewall, in addition to events of IP address changes. JADE allows the execution of more than one agent at the same time within the same host, that is, there is one thread per agent. Furthermore, it offers a Sniffer Agent, which allows to track messages exchanged in a JADE agent platform (see Figure 5). The development of JADE is still continuing. Further improvements, enhancements, and implementations have already been planned, most of them in collaboration with interested users of the JADE community.

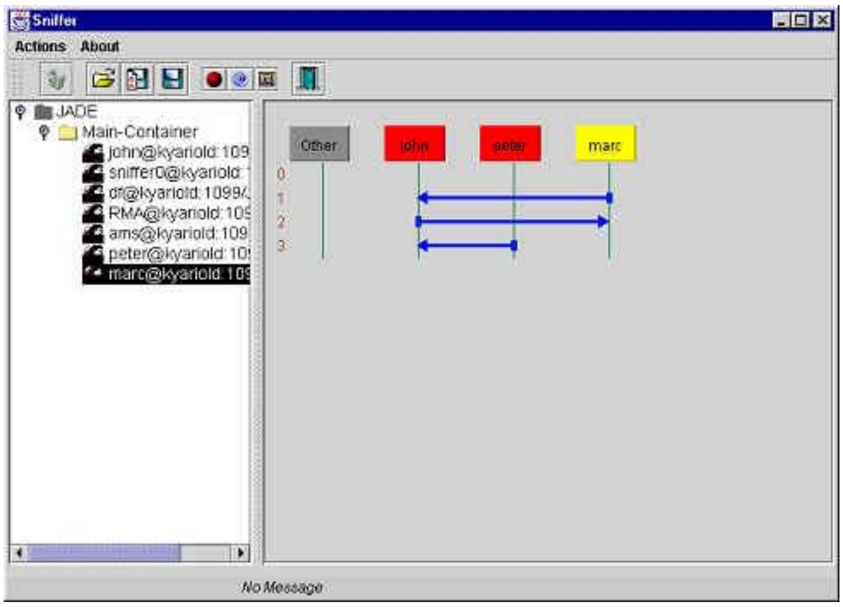


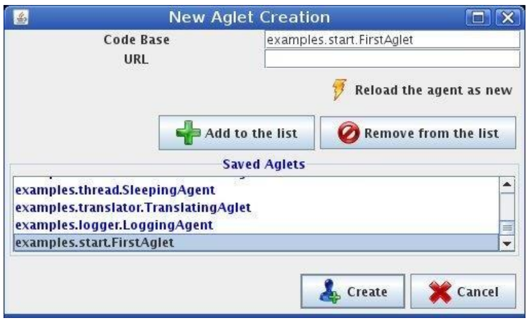
Figure 5. JADE sniffer agent GUI [23].

2.2.2. Aglets software development kit

The aglets software development kit (ASDK) is a platform for the development of Java applications based in mobile agents. This platform offers the ability for an agent (called aglet) to autonomously and spontaneously move from one host to another. Aglets is not Foundation for Intelligent Physical Agents (FIPA) [2] compliant.

Aglets includes both a stand-alone server called "*Thaili*", and a library that allows developers to build mobile agents. Aglets is completely made in Java and distributed under the IBM Public License. Aglets was originally developed at the IBM Tokio Research Laboratory. The original project name was AWB, which stands for Aglets WorkBench, and was latter changed to Aglets. It is currently hosted at sourceforge.net as an open source project [25]. The last stable release version of Aglets is 2.0.2 published in 2004, but there is a file last update on May 2013 at sourceforge.net [25].

Aglet offers a network configuration interface which allows the configuration of HTTP proxy options. Figure 6 shows the graphical interface for the creation of an agent and the list of agents already created.

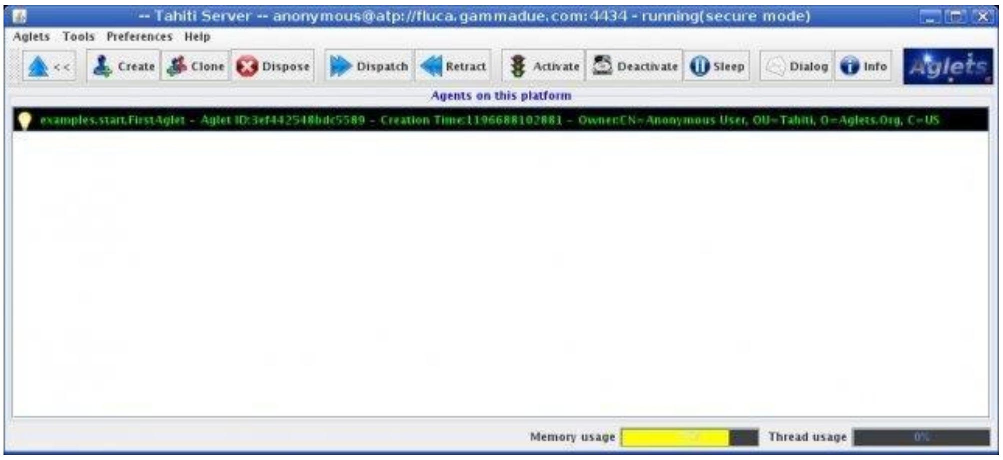


Code Base: examples.start.FirstAglet
URL:

Saved Aglets
examples.thread.SleepingAgent
examples.translator.TranslatingAglet
examples.logger.LoggingAgent
examples.start.FirstAglet

Figure 6. Aglet agent creation interface [41].

Aglets offers an interface for the creation and execution of agents. To use it, it is necessary to first authenticate. In Figure 7, it can be seen the graphic interface of the platform. It allows to control the agent life cycle, that is, to start an agent and the possibility to stop it and it also provides a unique identifier to each agent [41]. An important feature of agents in Aglets is that they can communicate with a Java Servlet.



examples.start.FirstAglet - Aglet ID:3ef442548bdc5589 - Creation Time:1196688102881 - Owner:CN=Anonymous User, OU=Tahiti, O=Aglets.Org, C=US

Figure 7. Aglet GUI manage agent [41].

2.2.3. JACK™ intelligent agent

The JACK™ Intelligent Agent is an agent development environment in the Java programming language [20]. JACK implements the FIPA [2] standards and it needs a DCI network for communication similar to TCP/IP and it needs one process running as a name-server. JACK does not offer mobility. JACK software is a proprietary license, but it is possible to access a trial version for 60 days, after filling out and sending a request to the group Autonomous Decision-Mark software (AOS) [43]. AOS and their partners have developed some applications, for example, Realistic Virtual Actors, Rules of Engagement, Oil & Gas, Intelligent PHM and Human Behaviour Representation [43], in areas such as unmanned aerial vehicles, surveillance, air traffic management, real-time scheduling, and virtual actors.

The JACK™ Development Environment (JDE) is a cross-platform graphical editor suite developed with Java for developing JACK™ agents. Currently, this platform is available for Windows, Linux and the macOS Operating System. JACK™ Intelligent Agents supports two agent reasoning models: a basic agent model and a teams agent model. Figure 8 shows the JACK workspace.

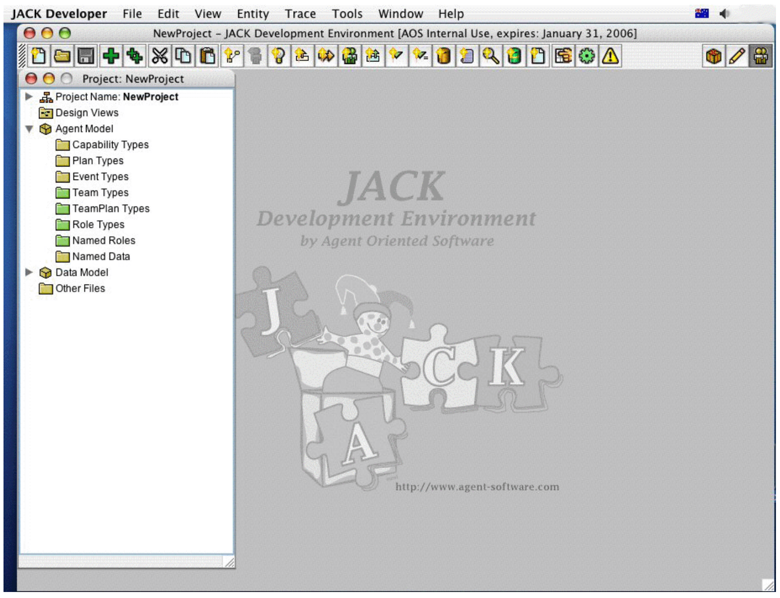
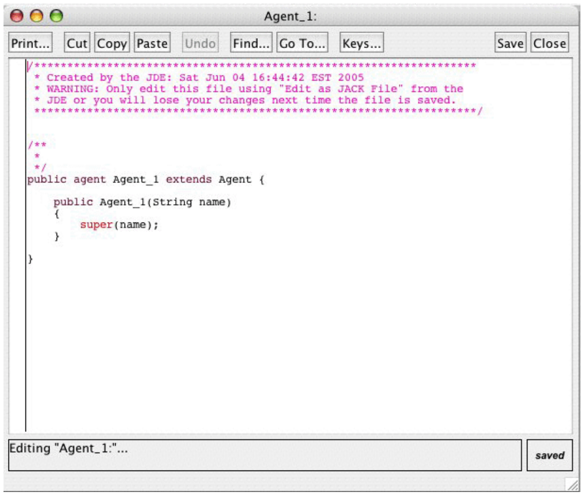


Figure 8. JACK development environment GUI [43].

The JDE supports the generation and execution of code, as well as aspects of the software development life cycle. Within a JDE session only one project can be active at a time. It also integrates an internal code editor (see Figure 9). The last published version was in July 2015, JACK 5.6c and it is compatible with Java v1.8.



```
/*********************************************************************
 * Created by the JDE: Sat Jun 04 16:44:42 EST 2005
 * WARNING: Only edit this file using "Edit as JACK File" from the
 * JDE or you will lose your changes next time the file is saved.
 *********************************************************************/


/**
 *
 */
public agent Agent_1 extends Agent {

    public Agent_1(String name)
    {
        super(name);
    }

}
```

```
Editing "Agent_1:"...
```

Figure 9. JACK code editor GUI [43].

2.2.4. Ajanta

Ajanta is a Java-based framework for programming mobile agent-based applications on the Internet [16].

Ajanta was a product of the Department of Computer Science (DCS) of the University of Minnesota developed during 1997-2002. Currently, DCS focuses in the distributed systems research. In the project official website they mention availability for Research Assistantship Positions for PhD level graduates students and the possibility to win Funds for supporting participation of undergraduate students under the REU awards [26]. There is the possibility of the Ajanta project still has support for its specification and development. It is a remote possibility that in the coming years we will see new versions available from this platform.

The binary is free but available upon request and Ajanta does not implement any standards. For communication, it uses the Java method invocation, authenticated RMI, ATP, agent collaboration Extensible Markup Language (XML) and offers weak mobility through Java serialization (byte code).

Ajanta is a system for programming agent based applications over the Internet. The main focus of the Ajanta design is on mechanisms for secure and robust execution of mobile agents in open systems. Agents in this system are active mobile objects, which encapsulate code and execution context along with data [26].

Ajanta uses Java facilities, such as object serialization, remote method invocation and security model. Ajanta provides a base *AgentServer* class, allows agent migration to/from other servers through agent transfer protocol, secure agent control and monitoring functions for agent developers and secure access to server resources for agents [44]. It uses proxies for protecting server resources from malicious agents. Figure 10 shows an agent server with an resident agent on it. An agent consists in some items and each item is encrypted with the public key of the target server. The procedure to send an agent from one server to another is as follows: the first server sends a request to the target server providing the agent's credentials, agent owner's signatures and other parameters. The target server verifies the credentials against the owner’s signature, thus allowing the target server to decide whether to permit the transfer based on the agent’s owner [16].

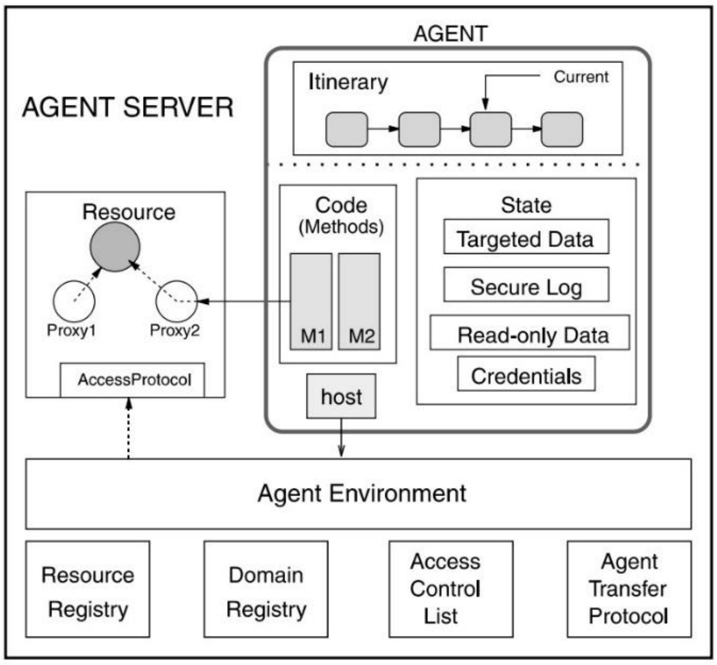


Figure 10. Ajanta agent server [16].

2.2.5. Tacoma network agents

Currently, Tacoma (Tromsø And COrnell Moving Agents) is a project on operating systems support for agents, but started a pure mobile code platform. It offers a generic toolkit for building close distributed communication infrastructures [37]. The last software upload has been TOS in April 2002. It is a lightweight distributed computing middleware platform, under Qt Public License (QPL).

The Tacoma project has a series of distributed systems where agents can be moved through the Internet and a secure message passing mechanism (application specific message handlers) developed through a framework.

One of the important characteristics of the Tacoma platform (before TOS), it was the support for a variety of programming languages. For example, C and the Tool Command Language (TCL), Python and Scheme, which is a very powerful and easy to learn programming language. It offers the possibility of developing applications such as desktop, web application, networking applications, administration, among others [45]. The mobile entities or agents in Tacoma, called carrier, can be developed in any programming language supported by the platform. When an agent moves, it is transported by an operator that contains [37]:

- The path - It is an ordered list of the targets in which the carrier is going to be moved, and;
- The data - It is a hashtable structure where serializable objects are stored.

The last update of Tacoma was in 1999, with the version Tacoma v2.0, in its beta edition, supporting binary data, which allows mobile agents to communicate and synchronize with all connected hosts.

A TOS kernel is a Java virtual machine that accepts carriers. Figure 11 shows a kernel with four extensions: The first one implements a service that sends messages to other instances, the second allows to execute programs outside the kernel, the third one allows to unload a file. Last but not less important, it allows to accept serialized versions of carriers and starts running them in the kernel [46].

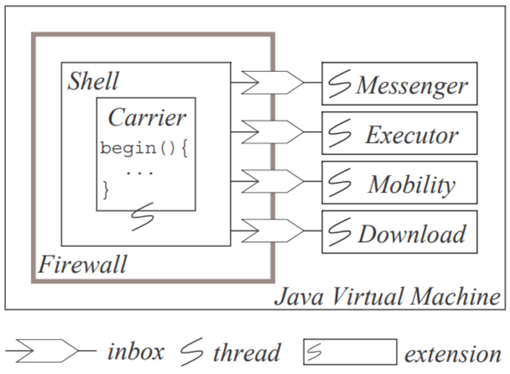


Figure 11. TOS kernel structure [46].

2.2.6. Tryllian's agent development kit (ADK)

The Tryllian's Agent Development Kit is a software toolkit that allows developers to create and implement solutions based on mobile agents in distributed systems. It supports agents developed in Java and the platform is based on mobile components [38].

Currently, there is no evidence that the project is still active. The official page is shown in white but, through the Internet Archive web page, it is possible to visualize the web page history of this project [39]. Tryllian started like a young and dynamic start-up company specializing in mobile software agents in 1999. According to the main web page stored in the Internet Archive, some years after its appearance, it became a Java Mobile Agent Portal, offering publications and articles about applications within the mobile agent area.

2.2.7. Concordia

Concordia is a framework for the development and execution of mobile agents through a middleware infrastructure. Concordia has been implemented in the Java language, with the aim of eliminating the existing short interoperability between different operating systems, providing support for agent persistence and guaranteeing the transmission of agents through the network. Concordia offers two types of security: a) protects the agent from improper handling and b) protects server services from unauthorized access [10].

An important feature is the communication between agents offered by Concordia, which is based on the paradigm of asynchronous distributed events and collaboration. The Concordia paradigm of collaboration offer benefits, such as a simple programming interface, asynchronous management of district events, support for agent mobility, persistence and transparent failure recovery [10].

The Concordia platform infrastructure consists of a set of Java classes for the server execution and the agents development with their respective activation. Each node within a Concordia system is composed of some server components, which can be executed on Java Virtual Machines (JVM) (see Figure 12).

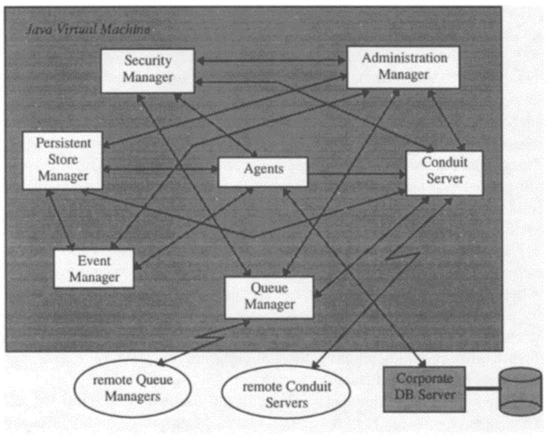


Figure 12. Concordia system data flow diagram [10].

2.3. Middleware and frameworks

An important aspect to take into consideration when selecting a platform for the development of systems based on mobile agents is the type of license that the development and implementation platform presents. Table 1 shows license type and agent programming language support.

Table 1. Middleware/Framework list.

Middleware/Framework	Agent programing language	License
Java Agent DEvelopment Framework	Java	LGPL Version 2
Aglets software development kit	Java	IBM Public License
JACK ^{TM} intelligent agent	Java	Proprietary
Ajanta	Java	Proprietary
Tacoma network agent	C, Tcl/Tk, Perl, Python, and others	Qt Public License (QPL)
Tryllian's agent development kit	Java	Not available today
Concordia	Java	Unknown

As mentioned above, the "boom" of this technology was during the appearance of the Internet. There are projects that are still running based on this technology, and even work about programming languages specification [3] for agents-based platform development.

There are platforms developed to support agents based on Java. For example, in Section 2.2.1 it is mentioned that JADE currently allows its execution on Android devices, but, usually, smart devices have limited JVMs, which makes it more difficult to install and execute the necessary tools for its operation.

2.4. Applications and solutions based on mobile agents

The mobile agents technology is rapidly emerging as a powerful computing paradigm. A lot of agent-based systems have been proposed and multi-agent systems have been studied as solutions for current problems in different areas. This section reviews some agent-based systems and applications targeted for specific scenarios.

A mobile virtual butler to bridge the gap between users and ambient assisted living

In [31], the authors developed a virtual butler that provides the interface between the elderly and a smart home infrastructure. The virtual butler is receptive to user questions and it is also capable of interacting with the user whenever it senses that something has gone wrong, notifying next of kin and/or medical services, etc.

The virtual butler resorts to the “follow me” approach of mobile agents. For this purpose, it is receptive to user voice commands, informing and alerting users by voice synthesis. It can also collect state and ‘emotions’ in order to enrich the facts database, even when the users are outside their instrumented smart home. The middleware platform used in this work was JADE and the mobile agents were developed using the Java language. However, the mobile agent that runs on the mobile platform has severe limitations and it is not the same one that runs in the in-home computers.

An agent-based solution to energy sink-hole problem in flat wireless sensor networks

In flat Wireless Sensor Network (WSN) there is a problem conventionally known as energy sink-hole, which causes early failure of the network. That is due to energy loss in all the nodes when repeated and continuous transmission of data to the sink occurs. Especially, depletion of energy is highly acute in case of nodes that are near to the sink. The authors of [32] propose a mobile agent-based solution for solving the energy sink-hole problem, aiming to extend the network life by reducing redundant data being passed to the nodes near to the sink and, thereby, reducing the load and saving battery life. The algorithm was implemented using aglets and the analytical results showed significant improvement in the network lifetime.

Integrating the mobile agents technology with multi-agent systems for distributed traffic detection and management systems

A number of agent-based traffic control and management systems have been proposed and the multi-agent systems have been studied. In [33], the authors propose to integrate the mobile agent technology with multi-agent systems to enhance the ability of traffic management systems to deal with the uncertainty in a dynamic environment. They have developed an IEEE FIPA [2] compliant mobile agent system called Mobile-C and designed an agent-based real-time traffic detection and management system (ABRTTDMS). The system based on Mobile-C takes advantages of both stationary agents and mobile agents. The use of mobile agents allows ABRTTDMS to dynamically deploy new control algorithms and operations to respond unforeseen events and conditions. Mobility also reduces incident response times and data transmission over the network. The simulation of using mobile agents for dynamic algorithm and operation deployment demonstrates that the mobile agent approach offers great flexibility in managing dynamics in complex systems.

Mobile-agent-based collaborative signal and information processing in sensor networks

In work [28], the authors developed an energy-efficient, fault-tolerant approach for collaborative signal processing information (CSIP) among multiple sensor nodes using a mobile-agent-based computing model, with the aim of increasing energetic efficiency of the equipment and improve fault tolerance. Since each node no longer needs to send the local information to a data processing center, habitually using the client-server model, the code moves to the sensor nodes through mobile agents. The evaluation of the implementation of this solution in moving agents allowed them to conclude that there is a superior performance when using this paradigm in relation to the client-server model, however the energy efficiency assessment was carried out through simulation.

Multiresolution data integration using mobile agents in distributed sensor networks

In work [30], the authors propose the use of mobile agents for the design of an infrastructure for the data integration in the network (DSN), to decrease the consumption of bandwidth. They also develop an improved version of an enhanced multiresolution integration (MRI) algorithm where multiresolution analysis is applied at local nodes before accumulating the overlap function by mobile agents [30]. According to the evaluations done, the authors were able to conclude that the new infrastructure based on mobile agents in their most optimal performance, can save more than 98% of the execution time, due the less data transfer time spent.

The works described show the implementation of the mobile agents paradigm in applications solution of various types of problems. It is worth mentioning that the development of these applications can be limited or advantageous according to the features offered by the chosen agent platform (middleware and framework).

3 Proposed solution

This chapter describes the solution's functional and architecture specification, starting by presenting the system architecture and all macro interactions among different developed software modules. Then, each software tool is detailed in terms of communication architecture and behavior.

3.1. System architecture

The most fundamental part of the agent-based platform is the software that supports the agent mobility through devices, called **middleware**. The agent development can be done using any text editor, and the agent execution can be done from the middleware or using the mobile agent framework. In a larger network, with lots of running middlewares, it is very difficult the execution of some agents and to know how many middlewares have an agent running or the agent deployment, which could be frustrating. The development of a software tool, called **mobile agent framework** was idealized to build agents and to facilitate the management and control of both agents and middlewares.

The platform also consists in a stand-alone server called **registry server**, it works as a communication intermediary between middlewares and mobile agent frameworks, similar to the functions of a router device in a network.

The specified agent-based platform allows the agent execution, moving it through different local networks and through the Internet. Figure 13 shows a scenario, with different devices running a middleware instance. The devices are in different private networks with custom Internet Service Providers (ISP) and all middlewares communicate with others through the registry server.

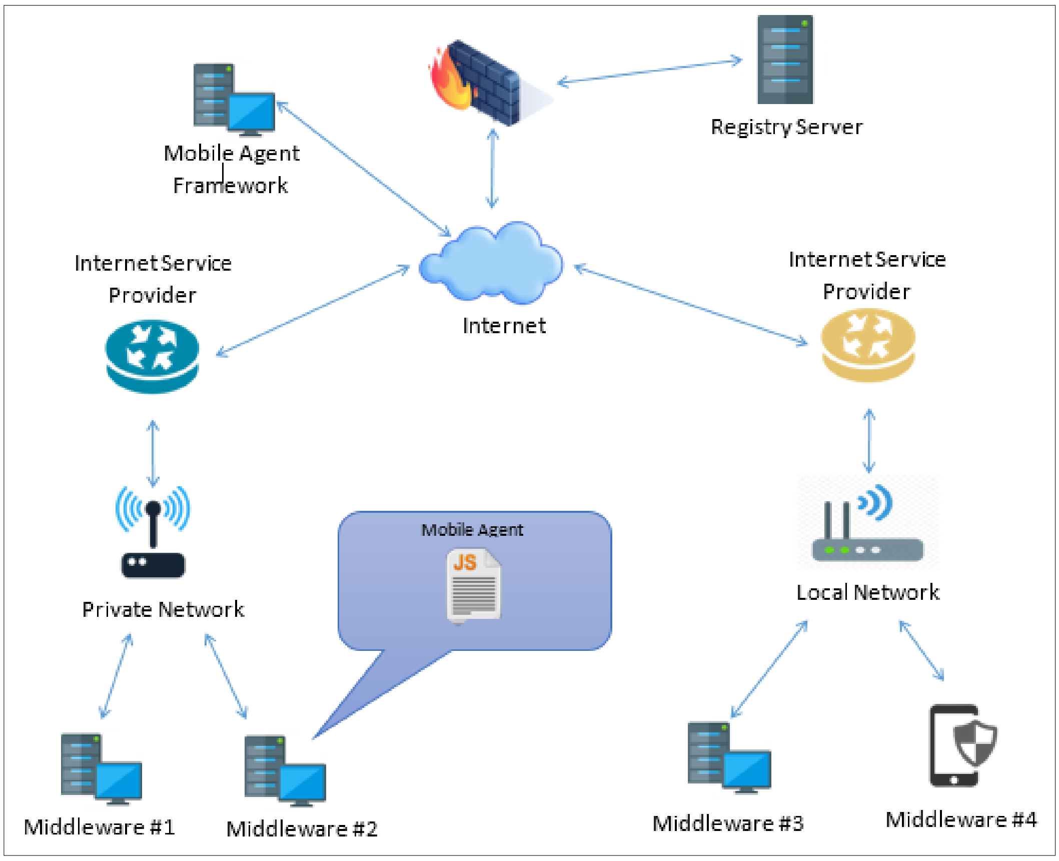


Figure 13. Agent-based platform system architecture.

In order to implement the agent-based platform, the following steps are needed:

- Run the registry server in a central computer to receive connections from client devices (using middleware software). In Figure 13, the registry server has been executed on a computer with a public IP address assigned. The user must ensure that the assigned ports are not being used by other programs or services in both middleware and registry server. Furthermore, the relevant firewall settings to allow connections and communication through the TCP / IP protocol are done. Additionally, if a developer wants to use another port (no the defaults one), the ports configurations need to be specified before run the registry server;
- Execute the JavaScript middleware in each device in the network, including both desktop computers and mobile devices. It is mandatory to specify the registry server IP address in each middleware (see Section 4.2). In case the registry server has been assigned with custom ports for listening services (different from the default ones specified in Section 4.1) it is necessary to configure the port assigned to the registry server (Section 4.1 specifies the different configurations of these parameters) in all middlewares and mobile agent frameworks.

Figure 13 also shows a device running the JavaScript framework. In this case, it is also necessary to configure the custom IP address assigned to the registry server (Section 4.4 shows the available ways to make these configurations).

3.2. Solution architecture

The middleware operations are transparent to the user (including developers). When the operating system starts, it initiates a middleware instance. The middleware allows the execution of a maximum of three instances per host (except for mobile devices, where only one instance can be executed) in order to easy the solution evaluation in a single host (anyone interested in this solution can try it without stablishing a full network, by evaluating it in single node). The middleware creates a listener service (socket server) which is used for communication between the API and the middleware. Figure 14 shows two hosts executing three middleware instances, where each middleware has an independent connection with the registry server

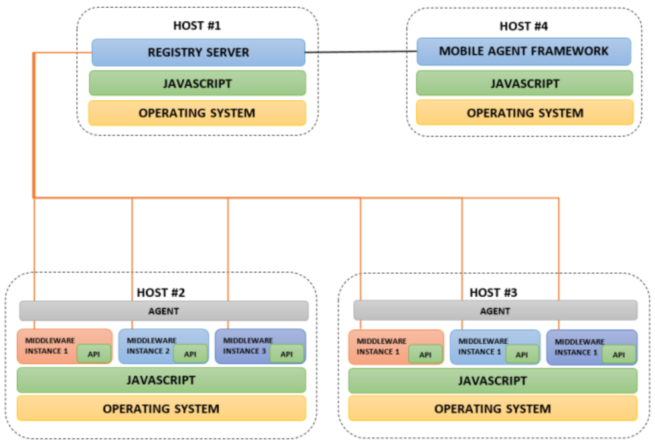


Figure 14. Solution architecture.

Each listener service is created through a communication port. Currently, three ports are specified into a developed middleware, which specifies the number of instances to be executed in a device.

3.3. Criteria evaluation

For the specification and development of an agent-based platform, the following criteria were considered:

- The programming language (PL) to build both agents and software tool of the developed platform (middleware and mobile agent framework);
- The communication protocol system for message passing between middleware;
- The communication architecture.

3.3.1. The programming language (PL)

Although belonging to the implementation stage, the programming language used for agent based platform development plays an extremely important role and that is why it is addressed in this chapter. According to the state of the art chapter, traditional agent-based platforms have chosen programming languages that limit the final solution for one or more operating systems but no one reach all of them (at least the most used ones today). For example, the Java programming language and the software tools developed in Java can be executed on any devices that have a running Java Virtual Machine (JVM). However, there are widely used operating systems that are not enough compatible to run the JVM, including all applications developed in this environment, for example: macOS and iOS Operating System, to name just two.

The choice of the programming language took place at the beginning of the project and it was carried on according to the following requirements:

- Be object-oriented (agents can be considered object structures);
- Be robust (for error handling during importation, execution agent and events not controlled);
- Support for object serialization (convert code executed to a string of bytes);
- Be flexible (simple variable declaration and function support);
- Be compliant with traditional operating systems, including the mobiles ones.

According to those requirements, two programming languages were selected for evaluation: a) Python and b) the JavaScript language. The Java programming language was not considered because the results could have the same limitations as the solutions mentioned in Chapter 2, despite the fact that Java is one of the programming languages most used for the development of applications. Furthermore, both Python and JavaScript are “booming” nowadays thanks to the appearance of development frameworks and runtime servers that facilitate the development and deployment of applications.

A first software prototype tool was developed in order to send serialized objects (agents) from one host to another, using the Python programming language. Python offers a wide range of libraries with built-in modules (written in C) that provide access to system functionality (e.g. the file system) and modules written in Python [40] that provide standardized solutions to many programming problems. The first prototype was developed using the Python libraries (see Figure 15).

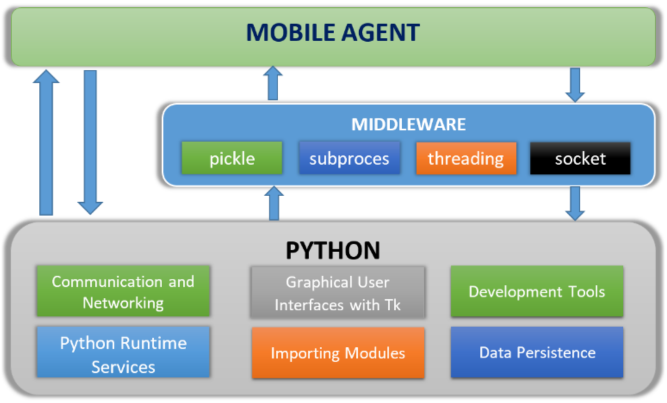


Figure 15. Python prototype software stack.

However, the following limitations were identified:

- For the correct execution of both the prototype and the agents, it was required the same version of Python;
- The prototype was tested on operating systems, such as Android and iOS, where it is necessary to install a third-party application (Python code compiler). However, only in some compilers versions it is possible to install additional Python libraries. The lack of those libraries limited not only the development of new functionalities for the platform, it causes limitations for agents execution;
- The developed prototype only allowed to move efficiently agents that do not exceed the 8.94 MB data size, the library used to send objects (agents) had this limitation. Several configurations were attempted, but there were no satisfactory results to mitigate this limitation;
- It is necessary to be very careful with blank spaces, line breaks and other aspects, because Python code will not run if these specifications are not met.

A JavaScript based prototype was also developed for which the following criteria were evaluated:

- Objects creation and execution;
- Serialize and unserialize objects process;
- Establish communication between two nodes through TCP / IP;
- Exchange objects between two devices;
- Execution of the developed application on different devices, including mobile devices.

After verifying all criteria, it was concluded that JavaScript fits mobile agents programming needs.

Another important aspect is that it is possible to develop native mobile applications (Android and iOS) thanks to the React Native framework [7] developed by Facebook Inc, while for desktop applications development is possible to use the Node.js runtime [6]. For the agent-based framework development, the Electron development framework [53] can be used, allowing to build desktop applications using the JavaScript programming language.

JavaScript has the following remarkable characteristics [47]:

- It is an object-based language. In case of accessing a non-existent object property, the results is an undefined value. But, there is a difference between an object that does not have a specific property and a property that has been assigned as undefined;
- The types of the declared variables can change according to the assigned value, except when null or undefined values have been assigned;
- Variables can be created without the need to declare what type of data they will store;

- It has a function called "***eval***" that allows a text string to be interpreted as a program;
- It supports callback functions, which allow listeners to listen to the results of an operation and errors during execution. A callback is a function called at the completion of some task, which prevents any blocking and allows other code to be run in the meantime.

3.3.2. The Communication protocol system

It must be possible to move an agent from one host to another. In order to achieve that, the following elements are necessary:

- A communication model;
- A communication protocol;
- An application message based format.

The communication infrastructure is IP based and each middleware needs to have a network interface (ethernet or Wi-Fi adapter). The communication protocol is based on TCP/IP v4 using the WebSocket API [55], which allows bidirectional communication between a client and a server based on the TCP/IP layer. The WebSocket API is asynchronous, which allows sending both agents and messages without the need to stop the current process to wait for a response.

Some solutions detailed in Chapter 2 use the Foundation for Intelligent Physical Agents (FIPA) [2], which uses a communication language (called ACLs). The solutions developed in this work use different messages types specified with keywords using JavaScript Object Notation (JSON) format.

3.3.3. The communication architecture

The most suitable communication architectures available to build an agent-based platform are the peer-to-peer and client-server architectures.

In the peer-to-peer model, all computers within the network have direct communication. On its turn, the client-server model consists of a central host in a network and several other computers as clients. Both these approaches are promising and provide an agent-based platform with different capabilities and limitations, which are summarized next.

In the initial phase of this work, a functional prototype has been developed with the objective of evaluating the most appropriate communication architecture for an agent-based platform. The first one to be evaluated was the peer-to-peer model.

Each middleware possessed the following features:

- It has the ability to perform a horizontal IP scan (current subnet) within the network to search new available middlewares running;
- It provides to mobile agents being executed a list of available hosts within the network that have a middleware running;
- It is identified through the device IP address.

The developed prototype following the peer-to-peer model allowed the identification of some limitation, such as:

- Only middlewares running on devices within the same network segment were detected;
- The operations of sending/moving agents through the Internet are limited. The devices need to have assigned a public IP address or be within a custom computer network, using technologies as Network Address Translation (NAT) with port forwarding, Virtual Private Network (VPN), and others;
- The search for new middleware running within the network is very time consuming, causing that the middleware spends a lot of time before and during its execution.

A second version of this middleware was developed, taking as reference certain functions of the previous one version, adding the "**leader's choice**" approach.

The "leader's choice" feature means that, within the network, one middleware took the leader role, while the others were followers (clients). To better understand this feature, let us consider the scenario illustrated in Figure 16 where two middleware (hosts) have been connected in sequence (one after another).

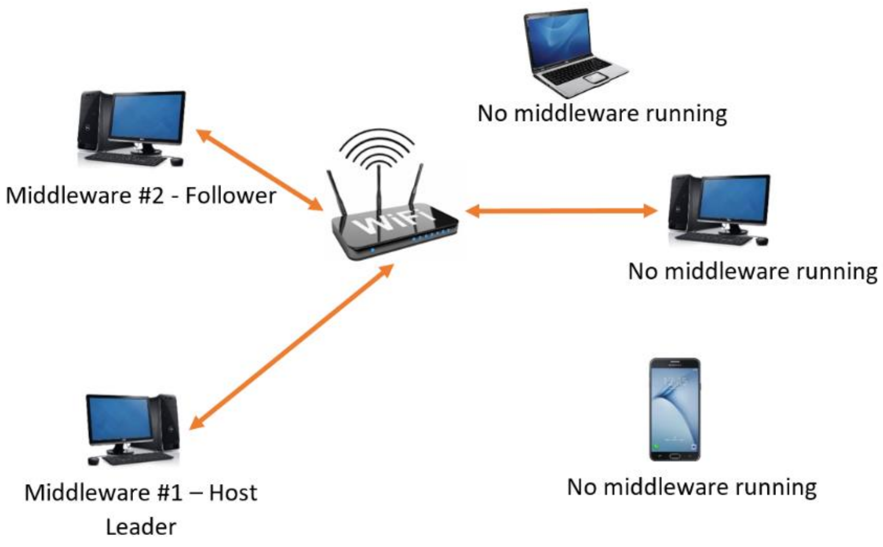


Figure 16. Connection follower to host leader.

Within a custom network, when a Middleware #1 is executed, it starts a middleware listener (Server socket). After that, it performs a IP address scan in the network to search for a leader host. If a host leader is not found, the middleware (let us call it Middleware #1) assumes itself the role of leader (running a Server socket). Figure 17 shows the middleware scan function ANSI/ISO flowchart.

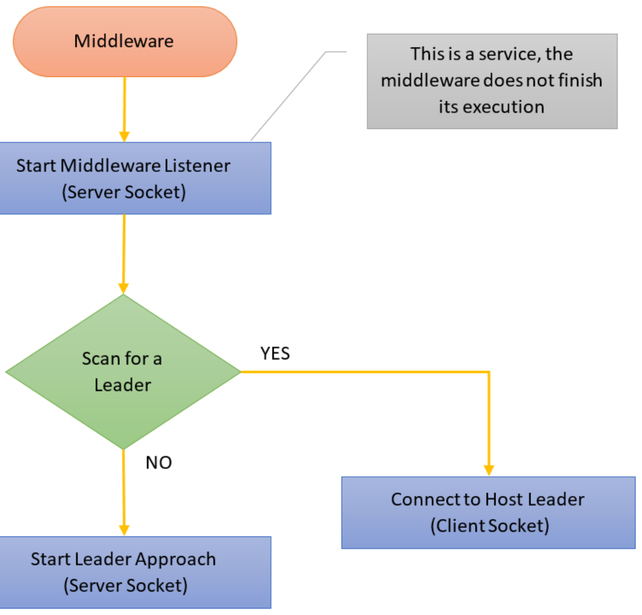

Middleware
This is a service, the middleware does not finish its execution
Start Middleware Listener (Server Socket)
Scan for a Leader
YES
NO
Connect to Host Leader (Client Socket)
Start Leader Approach (Server Socket)

Figure 17. Middleware scan function flowchart.

Later, when a second middleware enters the network (let us call it Middleware #2), it will undertake the same scanning process and it will be able to find a host leader running in the network (Middleware #1). After this, Middleware #2 connects to Middleware #1 through a socket client connection. The purpose of a host leader approach, is that in the connection of new middlewares, the leader has an updated list of all devices in the network that have a middleware running, and this list is shared with each host when it connects to the leader host.

When a new middleware connects to the leader host, it notifies all the already connected middlewares that a new host (e.g. Middleware #3) joined the platform network (see Figure 18). All middlewares are also notified when one of the connected middlewares closes the connection with the leader host or when the device stops the middleware execution.

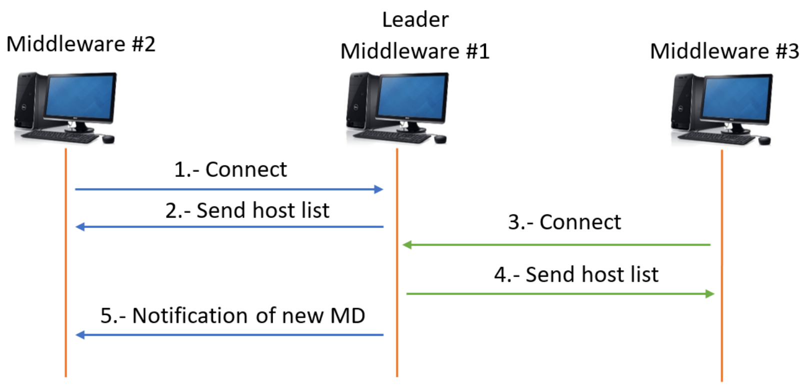


Figure 18. Middleware notification from leader host.

This approach minimizes the spent time in searching for middlewares running in devices within the network when compared to the first version. Other cases were also considered. For example, if something goes wrong with the leader host (hardware failure, middleware stopping, and others), the first host connected to it (Middleware #2, in the above scenario), immediately starts searching for a leader host. After finishing the search process and having no results, the middleware assumes the leader role. To avoid having more than one leader host in the network, each follower middleware disconnected from the leader host (Middleware #1) performs the host leader search at different waiting times. This waiting time is defined automatically according to the host number assigned with the last leader host connection (Middleware #1), the host number assigned is multiplied with a default value established within the middleware source code (by default, each middleware has the same assigned time value). This guarantees that Middleware #3 cannot be a leader before Middleware #2. That is, Middleware #3 only assumes the leader role if the predecessor (Middleware #2) fails to assume this role. The formula used to define the waiting time is the following:

$$T = HN * C,$$

where T represents the waiting time, HN represents the host number assigned and C represents the number of seconds.

When a connection with a new leader host is established, the host number of each middleware is reassigned (according to the order of connection with the new leader host). In this way, the process is autonomous and there is no need of human interaction (final user or developers) when a middleware fails, even if it is the leader host.

Using the notion of a leader host did not completely eliminate the excess time used in the search for new middleware within the network, when compared to the original peer-to-peer model. However, it represents an improvement.

Subsequently, the client-server model was evaluated, developing a prototype that allowed the sending of an object from one host to another host and the results were the following:

- Thanks to the existence of a centralized host, the search function of new middlewares running or the searching for a leader host within the network is not necessary. The central host would be known by all the software tools, diminishing the time of execution and lightening the tool by removing these features;
- The central host maintains a connection with all the client software tools. When the connection is established, the central host generates a list of all the running middlewares and has the ability to share it with each connected middlewares;
- The middlewares do not need to be inside the same local network. The centralized host is configured with a public IP that allows access to any middleware from any place that has an Internet connection. Furthermore, the clients can have the same IP address in the same segment or be in different local networks. This does not cause problems with the identification of the host clients;

- In comparison with the peer-to-peer model, it is not necessary to perform the additional configuration in network devices, such as NAT or VPN. It is only required that the centralized host allows the communication in the ports used.

The results obtained show that the client-server model is the most appropriate for an agent-based platform.

3.4. Middleware architecture

The developed middleware works as a distributed software abstraction layer, which operates between the application layers (agents) and the operating system and network layers. Figure 19 shows the complete software stack, including the hardware layer.

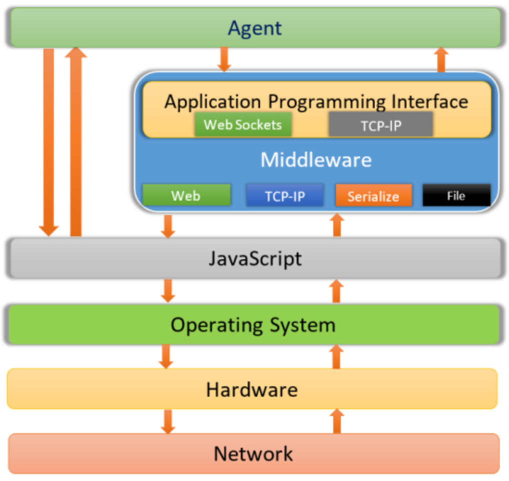


Figure 19.The complete software stack.

The developed JavaScript middleware provides:

- An environment where JavaScript mobile agents are executed;
- A structure template for mobile agents;
- A unique agent identifier for each agent, which generated by the middleware at the time of agent execution.

The middleware provides the agents with an easier way to access the services through the use an Application Programing Interface (API). All communication between the agents and the middleware is abstracted by the API layer.

The API must provide the following functionalities:

- It allows agents to communicate and gather information with the middleware;
- It provides an easy interface to access information, such as the middleware list running within the network;
- It provides agents an easy of moving to any host with a middleware running within the network.

3.4.1. Functional and non-functional requirements

In order to allow mobility, the platform should provide a set of services, including the support for mobile agent migration. As such, the requirements were defined as follows:

Middleware functional requirements

- The middleware must offer agents an easy way to access and use the middleware functionalities;
- The middleware must guarantee the transmission (agent migration) of agents through the local network and the Internet.

Middleware non-functional requirements

- The middleware should offer support for mobile agents persistence in order to overcome software or hardware failures. That is, it should be able to store a copy of the agents, with their corresponding status, in a non-volatile medium.

3.5. Mobile agent architecture

The life-cycle of a JavaScript mobile agent goes through the following states:

1. Initiated - The mobile agent has been restored or imported;
2. Executed - The mobile agent has been executed (this state has a persistence functionality);
3. Deleted - The mobile agent has been deleted by the developer or as a consequence of a move operation;
4. Transit - The mobile agent is moving to a new location.

The next code snippet shows the structure of a mobile agent ('helloworld.js'):

```
exports.runAgent = function() {
    console.log('Example 001 with API');
};

```

3.6. Mobile agent framework architecture

The developed mobile agent framework is a Graphical User interface (GUI) application type, in order to help developers to create agents in an easy way. It allows sending the agents to execute in a specific host (only hosts with a running middleware), to obtain information from the connected (host) equipment (both middleware and frameworks) and allows to perform the tracking of the agents that are currently executing and moving within the entire network. Figure 20 illustrates the interaction between the framework layers.

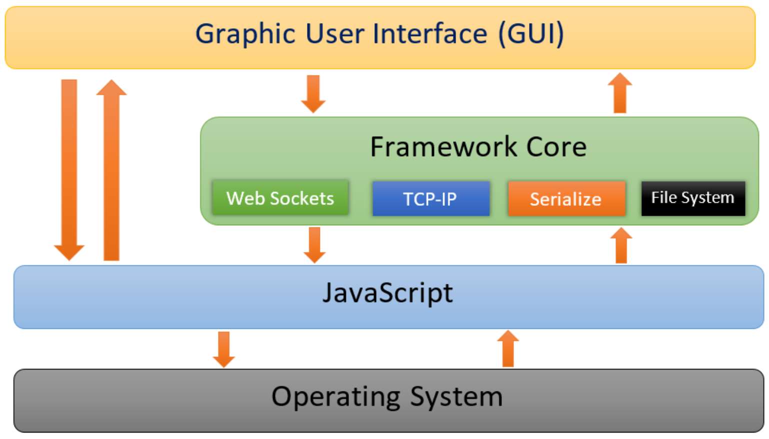


Figure 20. Framework stack.

The framework's GUI should provide the following operations:

- Creation and modification of agents;
- Get the connected hosts list within the middleware network;
- Gathering information about if a middleware has a running mobile agent;
- Send an agent to run in a specific host;
- Request information about the devices and the operating system of a specific host;
- Track the movement of agents.

3.6.1. Functional and non-functional requirements

In order to facilitate the development and administration of agents and hosts, the framework provides a set of services. The defined requirements were the following:

Mobile agent framework functional requirements

- The framework must facilitate the development of agent;
- The tool must offer developers the possibility of executing a custom agent on any connected host (middleware);
- The framework must allow the tracking of mobile agents and to know if a middleware (host) has an agent and its status.

Mobile agent framework non-functional requirements

- The framework should allow getting device information of all connected hosts. This information is useful to identify the type and version of the host's operating system, version of Node.js runtime, npm version, among others;
- The framework must show the middleware and framework list running within the network.

Figure 21 shows the main window sketch of the mobile agent framework. This window must contain a text editor that allows the mobile agents development easily. It must allow saving and reading agents stored in the file system and sending an agent to a specific host, for which it contains a drop-down list of the hosts with a middleware running.

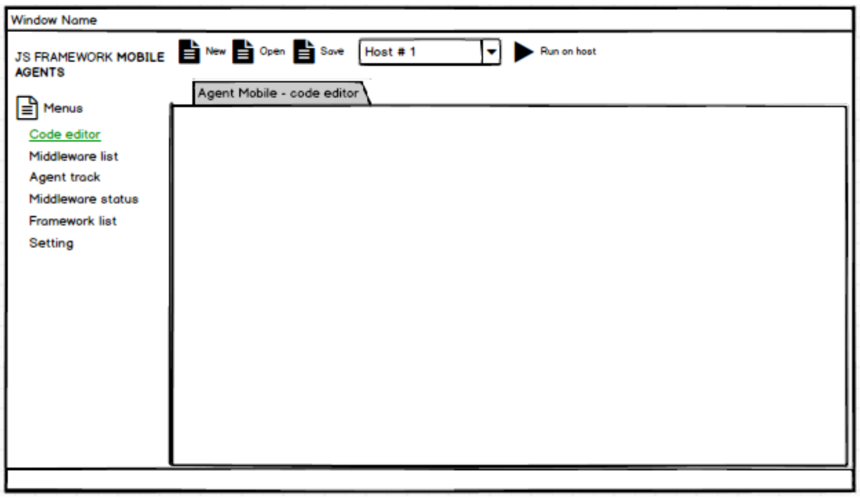
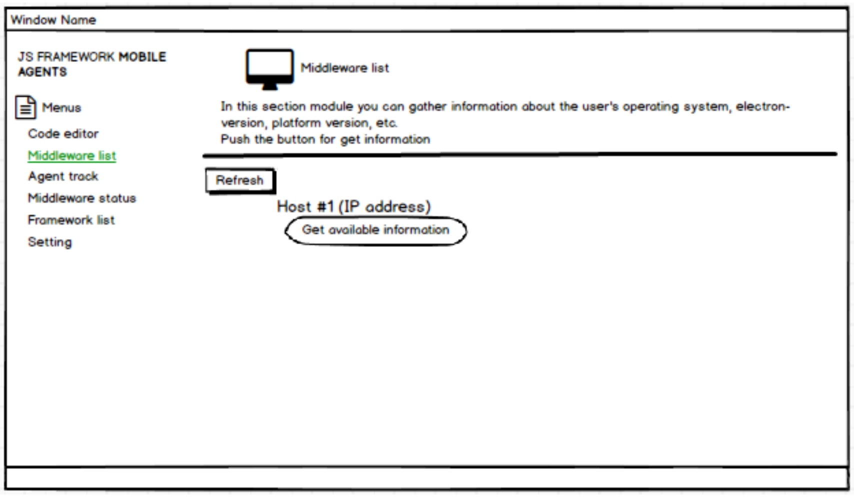


Figure 21. Framework prototype - Interface 1.

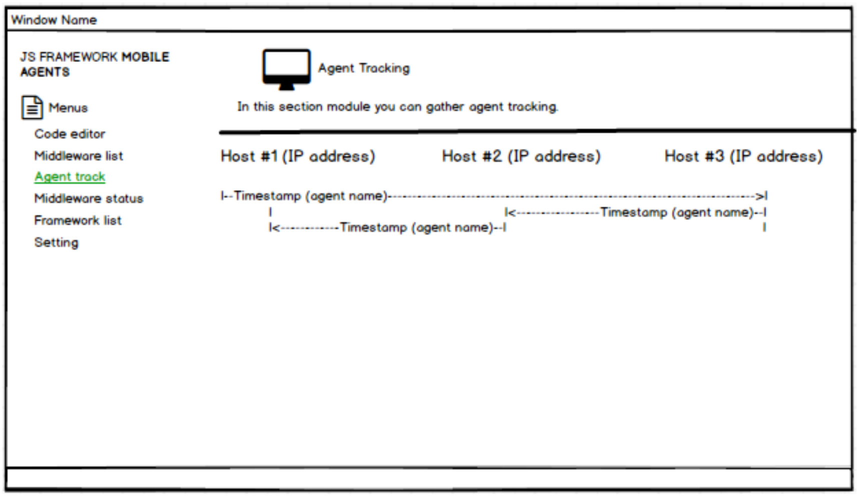
Figure 22 shows a simple window that allows to visualize all connected and active hosts with a middleware running and gathering information about hosts.



In this section module you can gather information about the user's operating system, electron-version, platform version, etc.
Push the button for get information

Figure 22. Framework prototype - Interface 2.

Figure 23 shows a window prototype for agent tracking. The mobile agent framework must show all hosts with middleware running and the agents movement.



Window Name

JS FRAMEWORK **MOBILE AGENTS**

Menus

- Code editor
- Middleware list
- Agent track
- Middleware status
- Framework list
- Setting

Agent Tracking

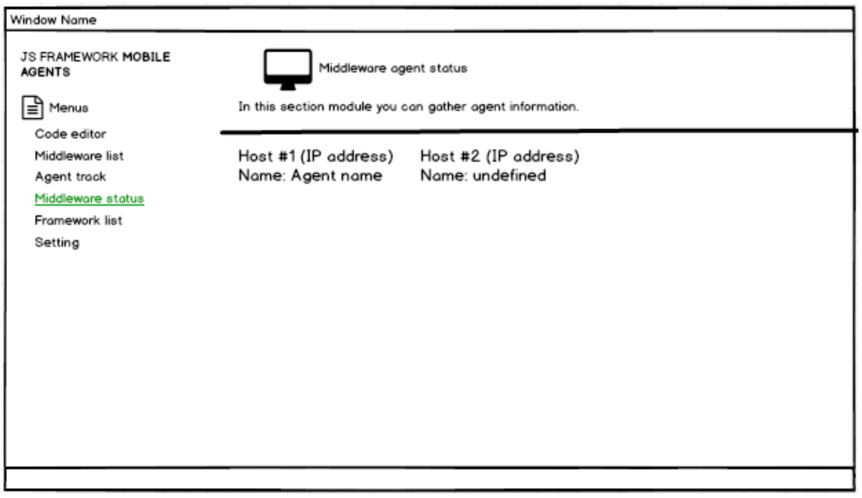
In this section module you can gather agent tracking.

```
Host #1 (IP address)          Host #2 (IP address)          Host #3 (IP address)

I--Timestamp (agent name)---------------------------------------------------------->I
      I                                I<----------------Timestamp (agent name)--I
      I<------------Timestamp (agent name)--I                                     I
```

Figure 23. Framework prototype - Interface 3.

Figure 24 shows a window prototype where the mobile agent framework allows gathering information about middlewares in relation with agents running in it.



Window Name

JS FRAMEWORK **MOBILE AGENTS**

Menus

- Code editor
- Middleware list
- Agent track
- Middleware status
- Framework list
- Setting

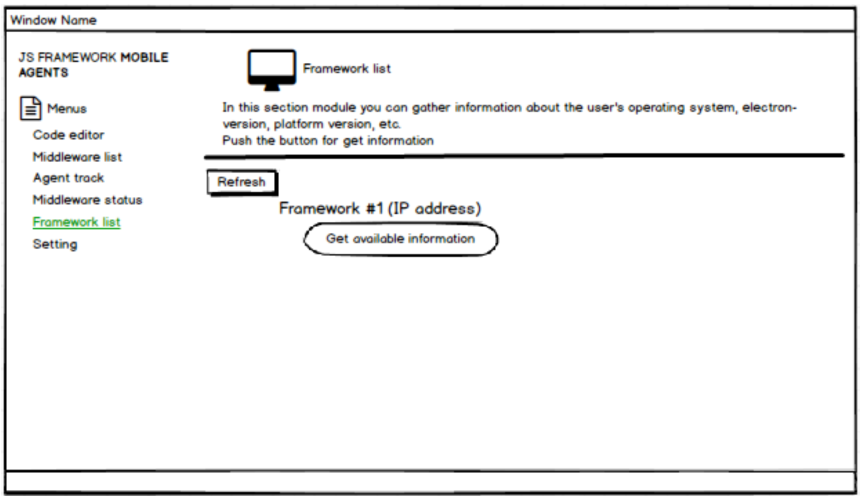
Middleware agent status

In this section module you can gather agent information.

Host #1 (IP address)	Host #2 (IP address)
Name: Agent name	Name: undefined

Figure 24. Framework prototype - Interface 4.

Figure 25 shows a window prototype, which allows to know the mobile agent framework list connected to the registry server. Furthermore, it allows gathering information about hosts.

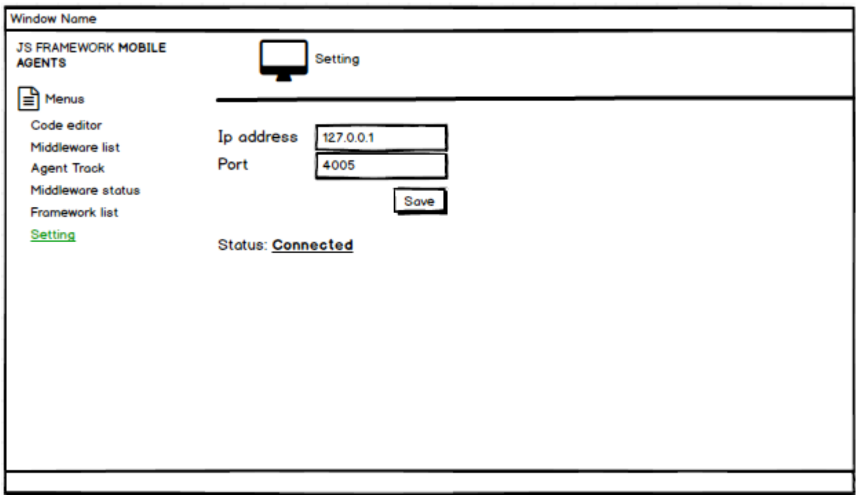


In this section module you can gather information about the user's operating system, electron-version, platform version, etc.
Push the button for get information

Framework #1 (IP address)

Figure 25. Framework prototype - Interface 5.

Figure 26 shows the setting window used to input the IP address and port used by the registry server.



Ip address 127.0.0.1
Port 4005

Status: Connected

Figure 26. Framework prototype - Interface 6.

The mobile agent framework is an additional tool in the agent-based platform. It provides facilities to manage agents, middlewares and frameworks connected in extensive networks, where there is a massive amounts of devices.

4 Implementation

This chapter details the developed JavaScript mobile agent platform, including communication, process flow and decisions made. The middleware follows the client-server communication model based on the asynchronous message passing paradigm using JavaScript Object Notation (JSON) format to represent messages. This architecture allows both middleware and framework to connect to a central host, called registry server, making it easier the communication and information gathering. For example, the connected hosts list (middleware and mobile agent framework running), mobile agents tracking, and other functionalities are detailed below.

The programming language for both agents and platform tools was JavaScript using asynchronous communication with WebSocket [55].

According to Figure 14 from Chapter 3, the proposed mobile agent platform solution is composed of:

- Registry server - It is a software tool developed to route communication between all software tools of the developed agent-based platform;
- Middleware - It is a software tool to run on host boot, and supports running agents;
- Middleware Application Programming Interface (API) - It is a JavaScript module that allows communication between agents and the underlying middleware;
- Agent - It is a program developed using JavaScript, that represents the user application in the context of the proposed mobile agent platform;
- Mobile agent framework - It is a GUI tool for agents development, management, tracking and information gathering.

For the development of these modules, the following JavaScript libraries were used:

- WebSocket (ws) [60] - It is a JavaScript library to create server and client socket communication. Furthermore, it offers some additional advantages over the "net" library [48] for JavaScript;
- Node-serialize [61] - It is a JavaScript module for object serialization including its functions into a JSON object. It is used to perform serialization operations on communication messages and agents (serialized objects), including in all tools offered by the platform;
- File System (fs) [62] - It is a JavaScript module that allows to do operations with the operating system file system;
- Util [42] - It is a JavaScript module, where only the method Util.inspect() was used. This method returns a string object representation which is used for debugging and to handle messages/communication objects more easily;

- Colors [23] - It is a JavaScript module used to assign different colors to each console messages:
  - Red color - For error messages;
  - Orange color - For warning messages;
  - Green color - For successful messages;
  - Black color - For general messages.

The developed software tools, e.g. registry server, middleware, framework and middleware API have internal variables that allow these messages to be printed through the console. By default, these variables are assigned with `true` value. To deactivate the printing of these messages, it is necessary to change the value to `false`.

4.1. Registry server

The registry server works as an intermediary between middlewares and frameworks. It works as a connection and routing server allowing to send agents through the Internet.

The registry server is based on events, which means that the middleware and the mobile agents framework do not need to do a request to the registry server for information gathering. Instead, the registry server has the ability to send information to all clients connected at any time. Furthermore, the messages are sent in an asynchronous way.

The registry server is composed by the following main components:

- Middleware service: For middleware connection handling;
- Framework service: For the mobile agents framework connection handling;
- Middleware handling: For keeping information about existing middlewares:
  - Middleware list: It is a list name of all currently connected middlewares;
  - Middleware agent container: It contains the current agent (backup) and the assigned middleware hostname;
- Framework handling: For keeping information about the framework:
  - Framework list: It is a list of connected mobile agent frameworks;
  - Framework agent container: It contains an agent copy of all connected frameworks.

The registry server follows a linear execution (line by line) with the exception that the software does not finish its execution in the last line. Due to the listening services, it continues executing and performs operations based on events. This means that the communication messages are received through the listening services and then the suitable function is invoked according to the message type. Figure 27 shows the process ANSI/ISO flowchart performed by the registry server.

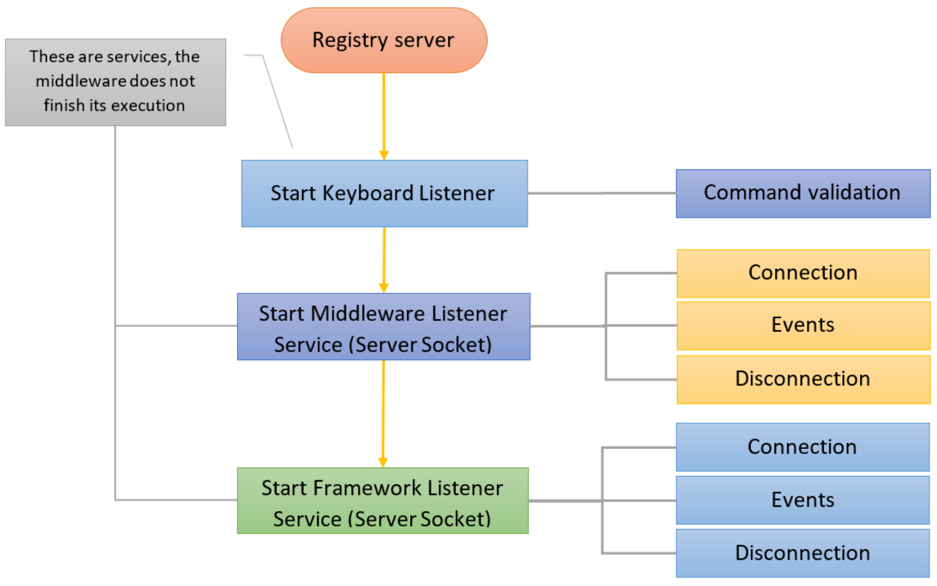

These are services, the middleware does not finish its execution
Registry server
Start Keyboard Listener
Command validation
Start Middleware Listener Service (Server Socket)
Connection
Events
Disconnection
Start Framework Listener Service (Server Socket)
Connection
Events
Disconnection

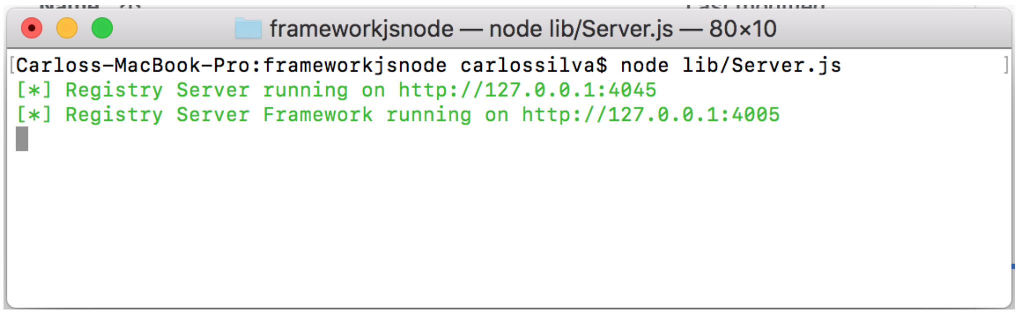
Figure 27. Registry server process flowchart.

The registry server does the following main operations:

- It assigns to each middleware and mobile agent framework an unique name specified during the connection with the registry server;
- It notifies all middlewares when a new middleware connects or when one is disconnecting from the registry server;
- It sends information to all frameworks about moving agents, when a new mobile agent framework connects to the registry server and when frameworks lose connection with the registry server.

When the registry server is executed, it follows the following process:

1. It initiates a key listener service, which allows invoking functions within the middleware through command sending by consoles;
2. The registry server initiates two listening services, the first one for stablishing a connection with the middleware and another one for the framework;
3. It outputs information through console messages with information about services running correctly (see Figure 28).



```
Carloss-MacBook-Pro:frameworkjsnode carlossilva$ node lib/Server.js
[*] Registry Server running on http://127.0.0.1:4045
[*] Registry Server Framework running on http://127.0.0.1:4005
```

Figure 28. Registry server running on console.

The registry server performs operations based on events. For example, when a middleware stablishes a new connection, the registry server assigns an unique name to the connected middleware, called middleware hostname. Then, it adds information about the new middleware to the local "middleware list" repository and sends it information, such as, assigned hostname, connection number and the middleware list. Later, the registry server notifies all middlewares and frameworks about the new connected middleware. Finally, it prints to the console the message about the new middleware that has been connected.

The registry server reacts when an agent movement occurs between middlewares. It notifies all frameworks about agent movement, sending information such as agent name, source middleware hostname and target hostname. Subsequently, it sends the agent to its target host using asynchronous messages.

When the registry server receives information about an agent that has been successfully sent to the target host, it sends the confirmation message of agent reception to the respective source (middleware or framework). Furthermore, the registry server reacts when it receives information about a middleware that has a mobile agent running. In that situation, it registers the information in the local repository "Middleware agent container" and notifies all connected frameworks about the existence of an agent in the specific host.

When a middleware loses connection with the registry server, the registry server removes all information about the disconnected middleware from its repositories and notifies all middlewares and frameworks about the disconnected device.

The registry server starts a framework service listener which performs operations when some events occurs. For example, when a mobile agent framework establishes a connection for the first time, the registry server assigns a framework hostname and send it to the connected framework. Later, it sends the connected middleware and frameworks lists to the new framework that has established the recent connection and notifies all connected frameworks about the new connected framework.

When a framework loses connection, the registry server removes all information about the disconnected framework from its repositories and notifies all frameworks about the disconnected framework. There are additional events that are not described here because in these events the registry server works as a router without performing additional operations, that is, it redirects the messages to the target host, both for middleware or mobile agent framework.

The registry server execution is carried out by host console through the following command:

```
node Server.js
```

By default, the ports used to run the listening service are the following:

```
Middleware:        4045 TCP
Framework:         4005 TCP
```

It is possible to change these ports and there are two ways to do it:

1. From the source code - the Server.js file can be edited with any text editor;
2. Passing values through the console at the moment of starting the service. For example:

   ```
   -middleware-port [param]
   -framework-port [param]
   ```

   The next command line is used to launch the registry server and to pass information about ports:

   ```
   node Server.js –middleware-port 8008 –framework-port 8080
   ```

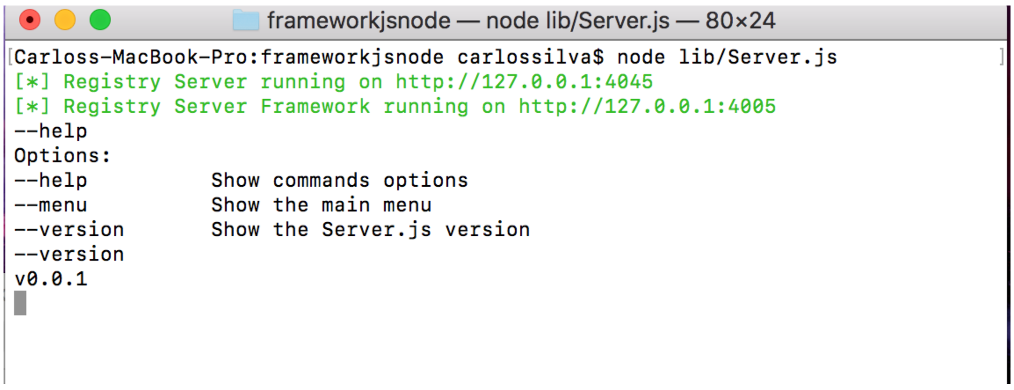
   In case the listening service ports have been changed in the registry server, it is necessary to do the corresponding change also in each middleware and mobile agent framework.

The registry server has the following options:

```
  --help          Shows commands options.
  --menu          Shows the main menu.
  --version       Shows the Middleware.js version.
```

The `-menu` option shows the following available options and the results are shown in the message console (see Figure 29):

```
[1] Show Middleware connected.
[2] Show Framework connected.
```



```
Carloss-MacBook-Pro:frameworkjsnode carlossilva$ node lib/Server.js
[*] Registry Server running on http://127.0.0.1:4045
[*] Registry Server Framework running on http://127.0.0.1:4005
--help
Options:
--help          Show commands options
--menu          Show the main menu
--version       Show the Server.js version
--version
v0.0.1
```

Figure 29. Registry server console menu.

4.2. Middleware

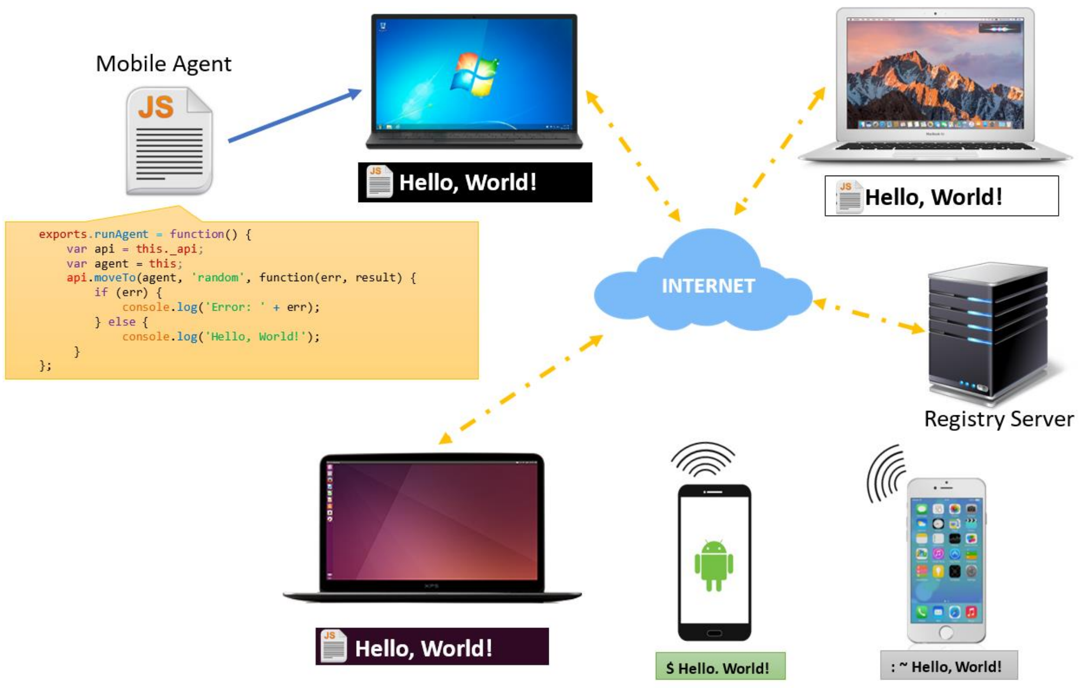
The middleware is the core component and allows the execution and mobility of agents. It needs to be installed on each of the network hosts, including mobile ones. It has an API that allows agents to access middleware functions in an easy way. The API could be considered as a thin layer (JavaScript file) that allows mobile agents to communicate with the middleware.

The developed mobile agent solution includes two different middleware versions with the same functions and capabilities: a version for desktop operating systems, called desktop middleware (for Windows, Linux and macOS Operating systems) and another version for mobile operating systems, called mobile middleware, which supports Android and iOS operating systems. The desktop middleware is supported by the Node.js [6] runtime, while the mobile middleware version is supported by the React Native framework [7], allowing the creation of the mobile middleware as a native application. The desktop middleware uses different libraries than the mobile one. For communication support, the WebSocket protocol was chosen due to its simplicity and support for JavaScript using Node.js runtime and React Native framework. WebSocket allows asynchronous communication, which increases asynchrony in client server interactions.

Both JavaScript middlewares have the following features:

- Data persistence - The module 'fs' for Node.js and ‘react-native-fs’ for React native are used to perform operations on the device file system, e.g, save a serialized copy of the mobile agent in a non-volatile memory;
- Asynchronous messaging communication - The module 'Web Socket' is used to communicate and send object/data between middlewares and the registry server;
- System image - The module 'node-serialize' for both Node.js and React Native is used to save and resume the current execution state of mobile agents through serialization and unserialization operations.

The developed JavaScript middleware allows the execution of mobile agents and the ability to move them through the network. The middleware establishes a connection with the registry server. Figure 30 shows a simple scenario where there is a mobile agent was deployed to move randomly within the middlewares connected in the network The agent was defined only to print a message to the console: "Hello, World!". Moving through the Internet is done using the registry server, which means that it can run on each device that have a stable connection to the registry server.



```
exports.runAgent = function() {
    var api = this._api;
    var agent = this;
    api.moveTo(agent, 'random', function(err, result) {
        if (err) {
            console.log('Error: ' + err);
        } else {
            console.log('Hello, World!');
      }
};
```

Figure 30. A simple scenario where the mobile agent moves across the Internet.

Figure 31 shows the connection process carried out between the middleware and the registry server and all operations performed by the registry server during this process. For example, the storage operations in the new connected middleware are performed in a synchronized mode. Once a new middleware has been connected, the registry server asynchronously notifies all the connected middlewares about the new middleware attached to the platform network, too. Furthermore, the figure also shows when the registry server notifies the framework using asynchronous messages about the connection of a new middleware in the network. An important point to mention is that, as the messages are asynchronous, the server does not wait for the answer. This means that the described operations do not happen in a linear way or sequential mode.

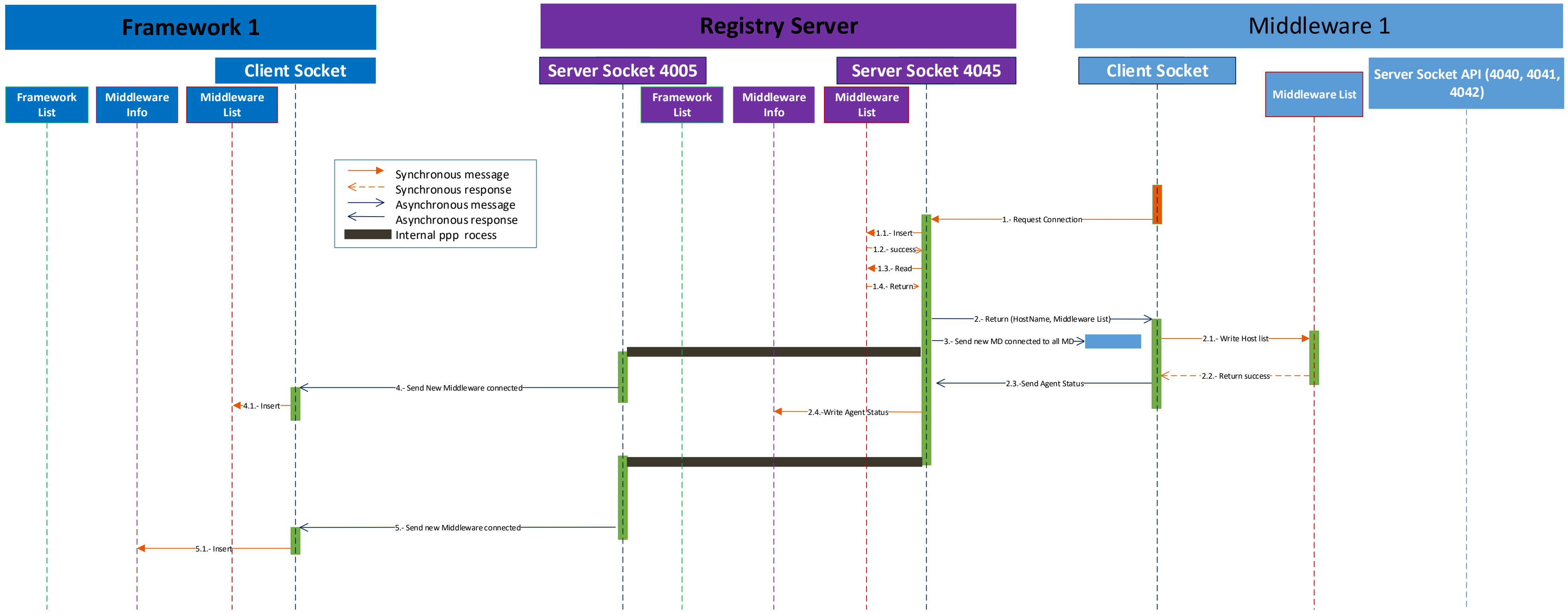


Figure 31. Middleware connection data flow (Basic UML sequence of Microsoft Visio).

4.2.1. Desktop middleware

Node.js [6] runtime was used for the development of the desktop middleware. Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine [6]. One of its important features is that Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Thanks to this runtime, it is possible to create very sophisticated applications using ready-to-use libraries managed by npm [51]. The latest version of Node.js available is 4.0.

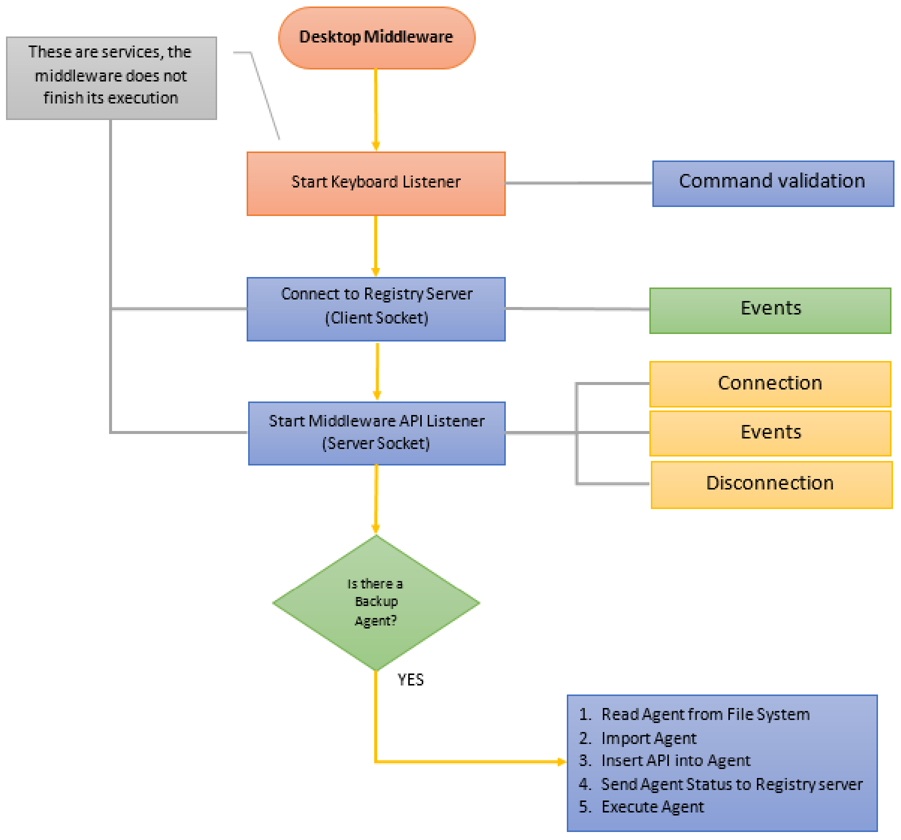
Each host is able to run a maximum of three instances of desktop middlewares, while mobile devices are able to run just one instance of the mobile middleware. Each desktop and mobile middleware instance can only support the execution of one agent at a time.

The desktop middleware version has the following components:

- A client connection handler (client socket);
- A listener service (socket server):
  - API Service Listener: For middleware API connection.
- Two internal repositories:
  - Middleware host list: It is a list of all currently connected middlewares;
  - Middleware agent container: Contains the current agent (backup).

When the middleware is started, it does the following process (see Figure 32):

1. It initiates a key listener service for command input;
2. It establishes a connection with the registry server;
3. It initiates the listener service to accept connections from the middleware API;
4. It verifies if there is an agent backup in the file system; in case there is one, the middleware performs the import, notifies the registry server and executes the agent. The existence of an agent backup means that there was a software or hardware failure in the recent past, hence, the agent is recovered using the following actions:
   a. Import the agent from the file system;
   b. Insert the middleware API inside the agent;
   c. Notifies the registry server with agent information;
   d. Execute the agent.



Desktop Middleware
These are services, the middleware does not finish its execution
Start Keyboard Listener
Command validation
Connect to Registry Server (Client Socket)
Events
Start Middleware API Listener (Server Socket)
Connection
Events
Disconnection
Is there a Backup Agent?
YES
1. Read Agent from File System
2. Import Agent
3. Insert API into Agent
4. Send Agent Status to Registry server
5. Execute Agent

Figure 32. Desktop middleware flow process.

When the desktop middleware establishes a connection with the registry server, the middleware prints to the console a message saying that it has been connected to the registry server. Then, the middleware stores the hostname assigned by the registry server, updates the local middlewares list using the host list received from the registry server. Finally, it sends information to the registry server about the agent.

When the middleware receives information about a new middleware that has been connected to the registry server, the middleware stores the new connected middleware hostname in the internal list. Nonetheless, when another middleware is disconnected or loses connection to the registry server, the middleware removes the disconnected middleware from the internal list.

When the middleware sends an agent and after it receives the confirmation message that the agent has been received successfully by the target host, it deletes the mobile agent from the memory; it then sends information to the registry server saying that the agent has been deleted. Finally, it deletes the mobile agent from the file system.

When the middleware receives a framework request to send information about the current host, it extracts information from the device and sends the obtained information to the framework that required it, through the connection with the registry server.

In addition, the middleware has a listening service for the middleware API, which is used by the agents to access the functionalities and data of the middleware. When the middleware receives the request to send an agent from the API, with the serialized agent inside the message body, it converts the agent into a serialized object in order to facilitate its manipulation. Then, it erases the middleware API from the agent before sending it, it notifies the registry server about the agent and sends the agent to the destiny host using the registry server connection. Finally, the middleware notifies the API informing that the agent was sent to the registry server. The confirmation message that the agent was received by the destiny middleware is received by the sent middleware, not by the API.

When none of these events occur, the middleware prints a message to the console saying that the occurred event is not supported or defined.

The middleware could be launched using the following command:

```
node Middleware.js
```

By default, the ports used to open the listening service are the following:

```
Registry server:         4045 TCP
Middleware API:          [4040, 4041, 4042] TCP
```

When the registry server has been executed with other port, all middlewares need to change the registry server port used. There are two ways of changing the registry port in the middleware:

1. From the source code, the Middleware.js file can be edited using any text editor;
2. Sending the value at the moment of starting the service, for which the following option is enabled. For example:

   ```
   -registry_server-port [param]
   ```

   The next command line shows the middleware execution command when the registry server uses a custom port (not default) for middleware listening services:

   ```
   node Middleware.js –registry_server-port 8008
   ```

   Remember, this port has to be the same assigned to the registry server. Otherwise, it is not necessary to modify the port, the middleware uses the default ports.

The desktop middleware has the following options (see Figure 33):

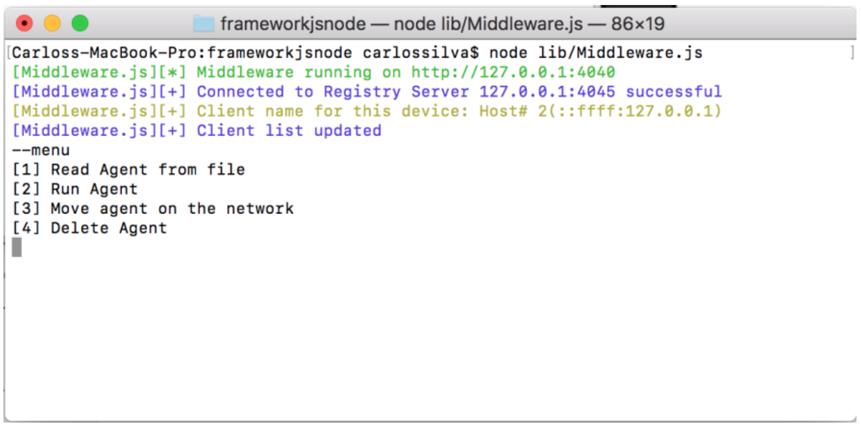
```
-a [agent.js]   Loads and runs the mobile agent.
-h [hostName]   Send the import agent to specific host.
--help          Shows commands options.
--menu          Shows the main menu.
--status        Shows status Middleware.
--version       Shows the Middleware.js version.
```

The `-menu` option shows the following available options and the results are shown by message console:

```
[1] Read Agent from file.
[2] Run Agent.
[3] Move agent on the network.
[4] Delete Agent.
```

The `-status` option prints the follow values to the console:

```
Hostname
Agent status
```



```
Carloss-MacBook-Pro:frameworkjsnode carlossilva$ node lib/Middleware.js
[Middleware.js][*] Middleware running on http://127.0.0.1:4040
[Middleware.js][+] Connected to Registry Server 127.0.0.1:4045 successful
[Middleware.js][+] Client name for this device: Host# 2(::ffff:127.0.0.1)
[Middleware.js][+] Client list updated
--menu
[1] Read Agent from file
[2] Run Agent
[3] Move agent on the network
[4] Delete Agent
```

Figure 33.Desktop middleware running.

It is possible to run the middleware and execute an agent with only a command using the attribute: `-a`; the command is the following:

```
node Middleware.js –a agent1.js
```

In case the user wants to send an agent to a specific host, without executing it in the current host, he/she can uses the attribute –h. It is mandatory to do the agent import too. The command is the following:

```
node Middleware.js –a agent1.js –h ‘Host #2’
```

4.2.1.1. Desktop middleware API

To facilitate the communication and interaction between the mobile agents and the middleware, an API has been developed. Each agent can reach the API through the WebSocket technology.

The API for desktop middleware is composed by the following components:

- A client socket - For establishing a client connection with the middleware using WebSocket;
- Middleware handling - For keeping information about middlewares:

    - Middleware host list: It is a list of all currently connected middlewares;
- Public functions - Each function supports a reference for a **callback function**. The callback function is called at task completion, which means that it adds a result listener function.

Both the desktop middleware API and mobile middleware API offer the following functions:

- MoveTo ([params]) - It allows the agent to move from one host to another. The needed parameters are the following:
    - Agent (object);
    - Host_destiny (specify the target middleware hostname);
    - Callback function (it is a callback function that returns the message confirmation about the agent received from target host or errors occurred during the send).
- GetAvailableHosts ([param]) - It returns the middlewares hostname list. The parameter is the following:
    - Callback function (it is a callback function that returns the hostname list or errors during the information gathering).
- GetHostName([param]) - It returns the middleware host name assigned by the registry server. The param is the following:
    - Callback function (it is a callback function that returns the middleware hostname or an error message about the operation).
- GetRandomHost ([param]) - It returns a random host from the list of available hosts. The parameter needed is the following:
    - Callback (it is a callback function that returns a random middleware hostname as result or an error if there is a not expected result).

The client socket service reacts when the connection with the middleware is established by printing to the console a message stating that it has connected with the desktop middleware. After, it stores the name assigned by the middleware, updates the local middleware list sent by the middleware and sends the custom request (generated when each API public function are called).

When the API receives the message confirmation that the agent was sent successful, it terminates the client connection with the middleware and returns the results through the callback function.

When the API receives the middleware hostname list from current middleware, the API terminates the client connection with the middleware, updates the local middleware list and returns the results through the callback function.

When the API receives the current middleware hostname, the API terminates the client connection with the middleware, stores the middleware name assigned by the registry server and returns the results through a callback function. In case the middleware notifies a not supported events by the API, it terminates the client connection with the middleware and returns an error through the callback function about the event.

In order to access the middleware API, agents have to import the API like this:

```
var api = require('path-to-API-file');
```

By default, the API uses the following ports:

```
Middleware:       [4040, 4041, 4042] TCP
```

To facilitate the use of the API, the middleware embeds the API inside the agent after agent importation. There are two ways of using the API from the agent source code:

```
1) this._api.callFuncion([params…]);
2) var api = this._api; api.callFunction([params…]);
```

The two commands above offer different capabilities, but allow the use of the public API functions. It is recommended to use the form `var api = this._api`, as it allows to store the API within a variable and allows it to be used in any part of the agent code, even within other functions declared within the agent.

The first way only works when it is declared and used in the main agent structure (not works when it called inside another function), because the function's `this` keyword refers to the global object.

The next source code shows an agent example that calls the `moveTo([params…])` API public function:

```
exports.runAgent = function() {
    var api = this._api;
    var host = 'Host #2';
    var agent = this;
    api.moveTo(agent, host, function(err, result) {
        if (err) {
            console.log('Error: ' + err);
        } else {
            console.log('Agent sent successful');
     }
};
```

The next agent example shows the use of the `getAvailableHost([param…])` API public function:

```
exports.runAgent = function() {
    var api = this._api;
    api.getAvailableHost(function(err, result) {
        if (err) {
            console.log('Error: ' + err);
        } else {
            console.log('First middleware list: ' + result[0]);
     }
};
```

The next example shows an agent invoking the `getHostName([param…])` API public function:

```
exports.runAgent = function() {
    var api = this._api;
    api.getHostName(function(err, result) {
        if (err) {
            console.log('Error: ' + err);
        } else {
            console.log('Middleware host name: ' + result);
      }
};
```

The next source code shows an agent example using the `getRandomHost([param…])` API public function:

```
exports.runAgent = function() {
    var api = this._api;
    api.getRandomHost(function(err, result) {
        if (err) {
            console.log('Error: ' + err);
        } else {
            console.log('Random middleware hostname received: ' +
  result);
      }
};
```

4.2.2. Mobile middleware

The middleware for mobile devices was also developed in JavaScript, but is supported by React Native, allowing it to be compiled in native code for mobile operating systems, such as Android and iOS. React Native [7] is a framework for the development of mobile applications using JavaScript. React Native allows the creation of real mobile applications, not "mobile web app" for mobile operating systems, indistinguishable from applications built using Objective-C [49], Swift [50] or the Java.

One of the advantages of developing applications using React Native is that it is not necessary to install any additional application on the device because the developed application can be installed on Android and iOS as any application developed in the native languages, and it can be published in the respective applications stores (Google Play and Apple Store).

The mobile middleware module provides:

- Graphic Interfaces 1 - It is a window with text input to interact with the menu;
- Graphic Interfaces 2 - It is a setting interface to configure IP address and port.

Additionally, the mobile application provides a floating panel where the console messages are printed. Figure 34 shows the application main graphic interface of the application on both Android and iOS.

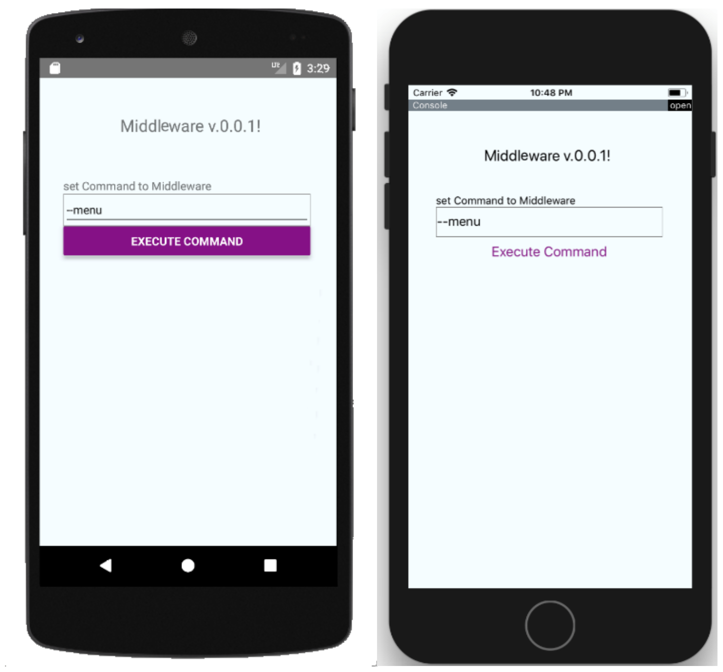


Figure 34. Mobile middleware interface – Main window.

The mobile application offers a configuration interface, which allows the input of the IP address and the communication port of the registry server. Furthermore, the window shows an information message about middleware connection. Figure 35 shows the setting interface on both Android and iOS.

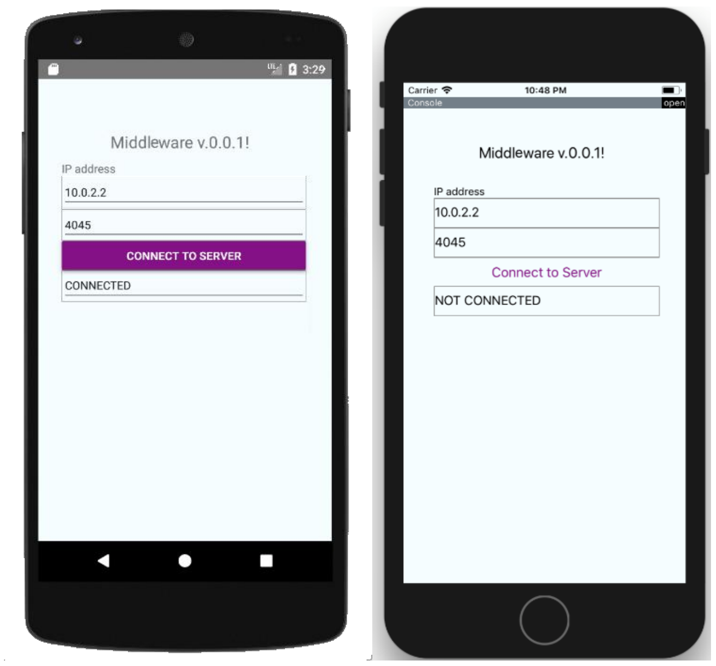


Figure 35. Mobile middleware interface- Setting window.

The mobile middleware supports the following options:

```
--help          Shows commands options.
--menu          Shows the main menu.
--status        Shows status Middleware.
--version       Shows the Middleware.js version.
```

The `-menu` shows the available options:

```
[1] Run Agent
[2] Move agent on the network
[3] Delete Agent
```

The `-status` option returns the follow values:

```
Host name
Agent status
```

4.2.2.1. Mobile middleware API

The public functions offered by the mobile middleware API are the same as those specified in the API for desktop middleware, including the same number of parameters.

The public API functions `moveTo()`, `getAvailableHost()`, `getHostName()`, `getRandomHost()` are used in the same way as the desktop API functions mentioned in Section 4.2.1.1.

4.2.3. Agents

This section describes the agent structure, which is mandatory for the successful execution of the agent in the platform.

4.2.4. JavaScript agent structure

The mobile agent must follow the next structure in order to be executed into the agent-based platform:

```
exports.runAgent = function() {
    //Custom code here.
    var api = this._api;
};
```

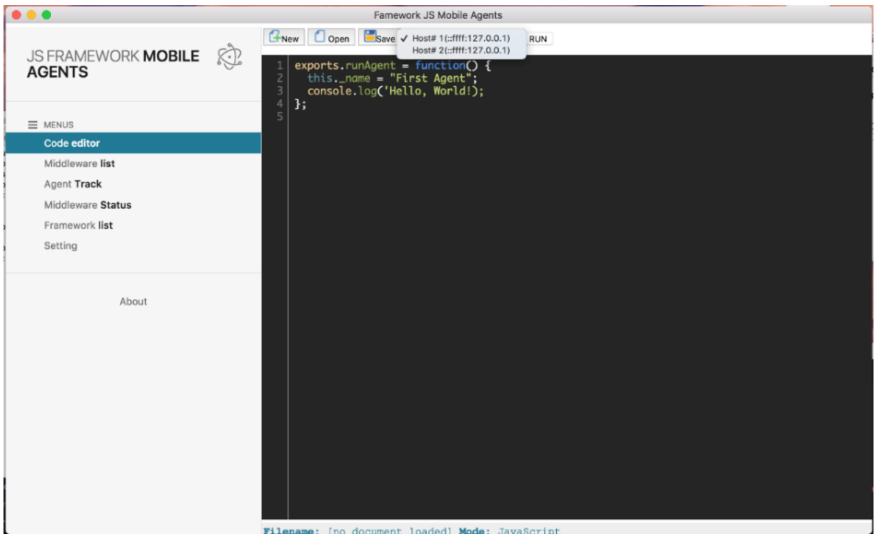
Within that structure the developer can insert custom JavaScript code and perform the import of the API to use the public functions mentioned in Section 4.2.1.1. The middleware embeds the API inside the agent during agent importation and it removes the API from agent before it sent the agent through the network. This agent structure has been optimized, providing simplicity for the development and understanding of the source code of an agent.

4.2.5. Developing agents

There are two ways of running a mobile agent: one is through the framework and the other is through the middleware:

Using the mobile agent framework

The graphic interface framework provides a very useful code editor (uses colors to easily identify functions, reserved words of the JavaScript language, and others). When creating a new agent, the framework creates a document with the basic structure of an agent. At the top, there is a list (ComboBox) of all the hosts running the middleware (see Figure 36).

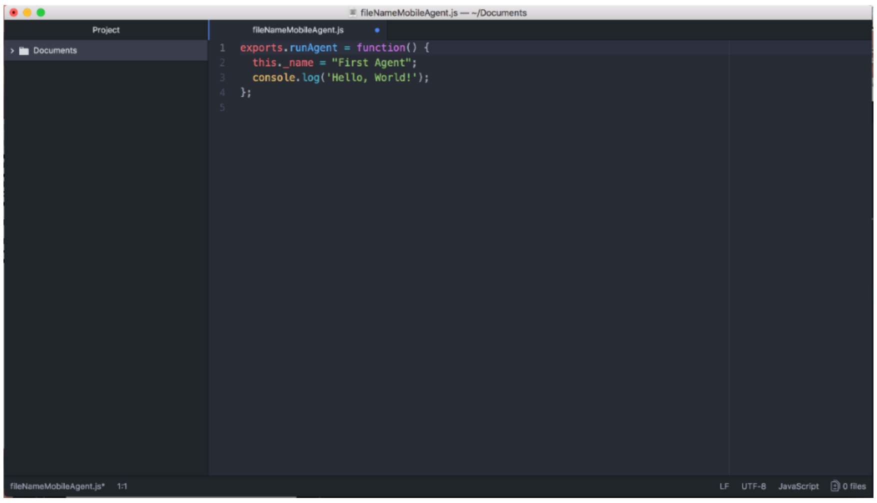

Famework JS Mobile Agents
JS FRAMEWORK MOBILE AGENTS
New
Open
Save
✓ Host# 1(::ffff:127.0.0.1)
Host# 2(::ffff:127.0.0.1)
RUN
MENUS
Code editor
Middleware list
Agent Track
Middleware Status
Framework list
Setting
About
Filename: [no document loaded] Mode: JavaScript

```
exports.runAgent = function() {
  this._name = "First Agent";
  console.log('Hello, World!);
};
```

Figure 36. Creating agent from Framework GUI.

Using middleware command line

An agent can be developed using any text editor available for the operating system (see Figure 37).


fileNameMobileAgent.js — ~/Documents
Project
Documents
fileNameMobileAgent.js
fileNameMobileAgent.js* 1:1
LF UTF-8 JavaScript 0 files

```
exports.runAgent = function() {
  this._name = "First Agent";
  console.log('Hello, World!');
};
```

Figure 37. Creating agent from text editor.

Once created, the file is saved by adding the .js extension. For example, **`fileNameMobileAgent`**.js. To load the mobile agent into memory and execute it in the device, it is necessary to execute the following:

```
node Middleware.js -a fileNameMobileAgent.js
```

Section 4.2.1 shows all the options and functions available from the middleware in relation to agent execution.

4.3. Mobile agent framework

One of the additional tools developed to facilitate the development and administration of agents, is the mobile agent framework. It has a connection to the registry server using WebSocket and, through this connection, the framework can send executing agents to any middleware and extracting information from the device.

The mobile agent framework has been developed with the JavaScript programming language using the Electron Framework, allowing cross platform desktop apps built with JavaScript, HTML and CSS. The framework has the following components (see Figure 38):

- A client connection (client socket) with the registry server;
- Internal repositories:
  - Middleware list - It is a list of all currently connected middlewares;
  - Framework list - It is a list of all connected frameworks.

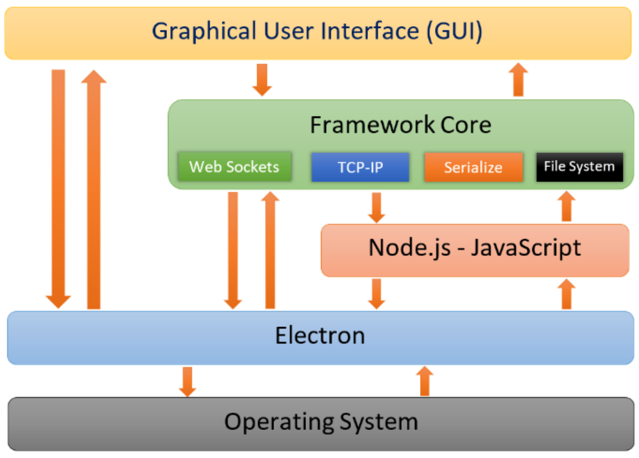
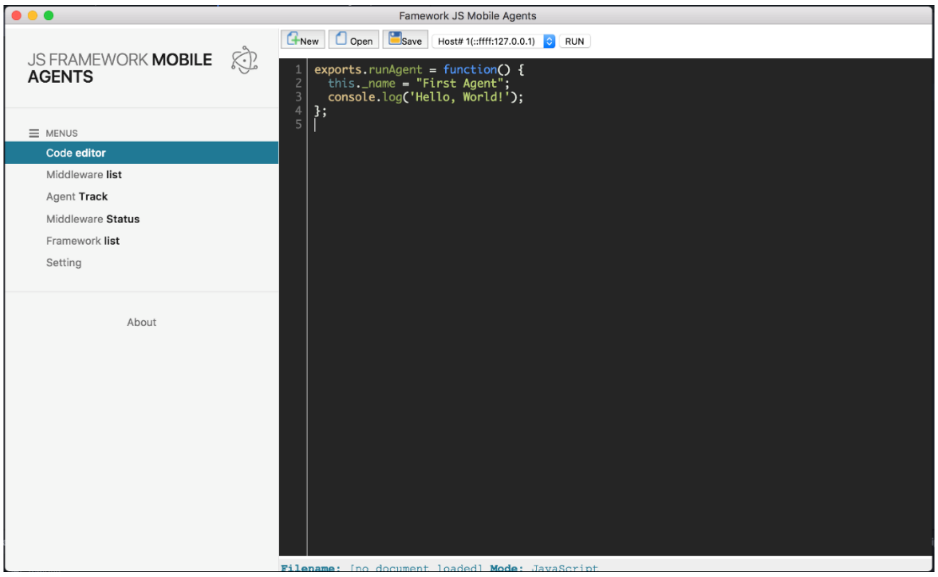


Figure 38. Framework architecture stack.

Electron is an open-source software developed by GitHub. It allows to build GUI desktop applications using components originally developed for web applications. It uses the Node.js runtime [6] for the backend and uses chromium [54] for the frontend. There are many applications that were developed under the Electron platform. For example, Skype IM & video calls, GitHub Desktop, WebTorrent, WordPress.com, Slack, Atom [54], among others.

The mobile agent framework provides a graphic interface that exposes six different windows:

Window #1 is a text editor for mobile agents programming with features such as, new, open and save agent code in the file system. Furthermore, it allows to send the agent to run on a specific host through a drop-down list, which contains a middleware hostname list. Figure 39 shows the window with basic agent structure.

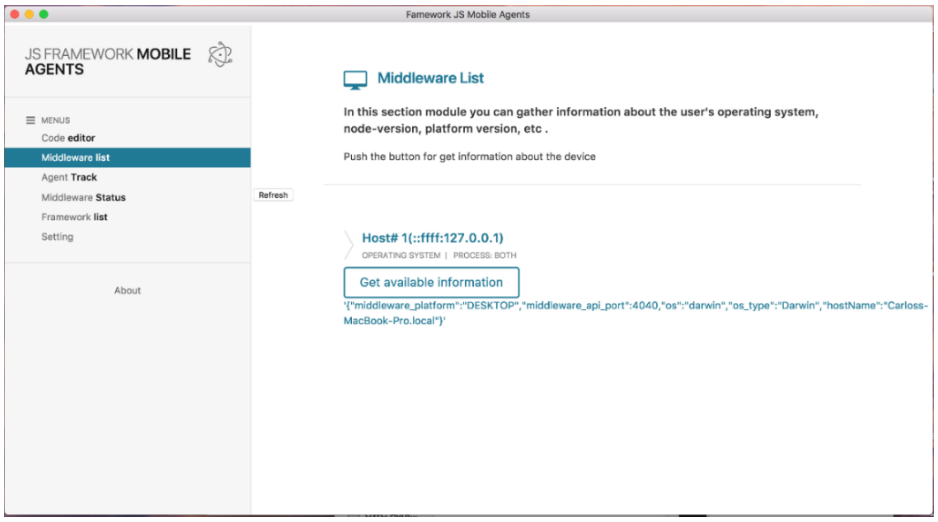


```
exports.runAgent = function() {
  this._name = "First Agent";
  console.log('Hello, World!');
};
```

Filename: [no document loaded] Mode: JavaScript

Figure 39. Mobile agent framework - Code editor.

Window #2 shows all middlewares hostnames connected to the registry server and it allows information gathering about hardware and operating systems (see Figure 40).


Famework JS Mobile Agents
JS FRAMEWORK MOBILE AGENTS
MENUS
Code editor
Middleware list
Agent Track
Middleware Status
Framework list
Setting
About
Refresh
Middleware List
Host# 1(::ffff:127.0.0.1)
OPERATING SYSTEM | PROCESS: BOTH
Get available information

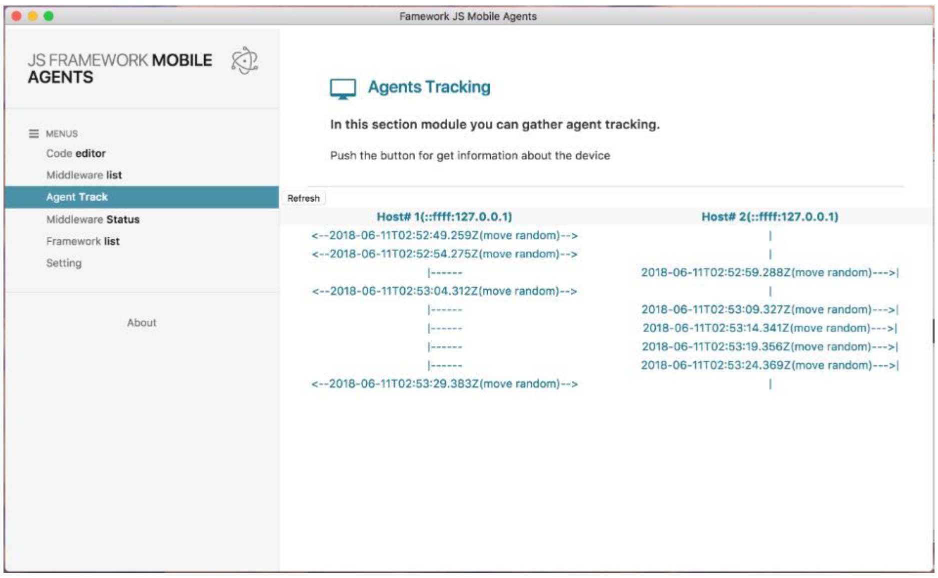
In this section module you can gather information about the user's operating system, node-version, platform version, etc .

Push the button for get information about the device

```
'{"middleware_platform":"DESKTOP","middleware_api_port":4040,"os":"darwin","os_type":"Darwin","hostName":"Carloss-MacBook-Pro.local"}'
```

Figure 40. Mobile agent framework – Middleware connected information.

Window #3 is a graphic interface for monitoring (tracking) the mobile agent movement between middlewares. The window shows the agent name and the movement timestamp (see Figure 41);



Agents Tracking

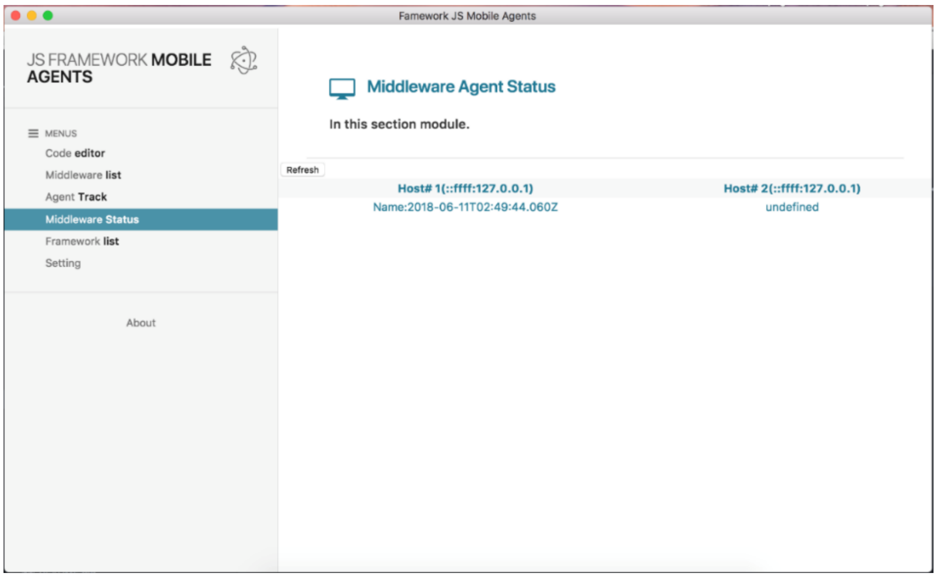
In this section module you can gather agent tracking.

Push the button for get information about the device

Host# 1(::ffff:127.0.0.1)	Host# 2(::ffff:127.0.0.1)
<--2018-06-11T02:52:49.259Z(move random)-->	|
<--2018-06-11T02:52:54.275Z(move random)-->	|
|------	2018-06-11T02:52:59.288Z(move random)--->|
<--2018-06-11T02:53:04.312Z(move random)-->	|
|------	2018-06-11T02:53:09.327Z(move random)--->|
|------	2018-06-11T02:53:14.341Z(move random)--->|
|------	2018-06-11T02:53:19.356Z(move random)--->|
|------	2018-06-11T02:53:24.369Z(move random)--->|
<--2018-06-11T02:53:29.383Z(move random)-->	|

Figure 41. Mobile agent framework - Agent tracking interface.

Figure 42 shows Window #4, which shows the middleware hostname list with running agents names.



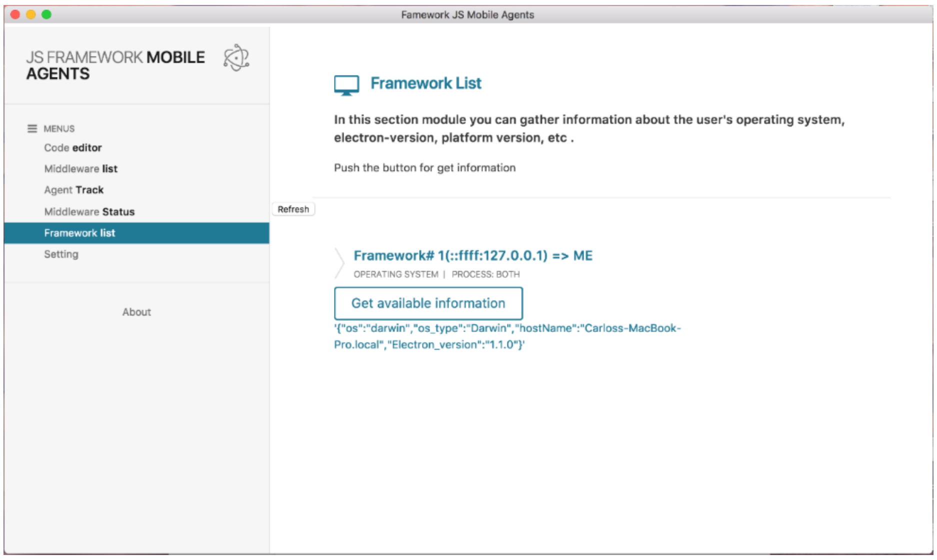
Middleware Agent Status

In this section module.

Host# 1(::ffff:127.0.0.1)	Host# 2(::ffff:127.0.0.1)
Name:2018-06-11T02:49:44.060Z	undefined

Figure 42. Mobile agent framework - Middleware agent information.

Window #5 shows the list of current connected frameworks, highlighting the IP address and allows gathering information about hosts (see Figure 43);



In this section module you can gather information about the user's operating system, electron-version, platform version, etc .

Push the button for get information

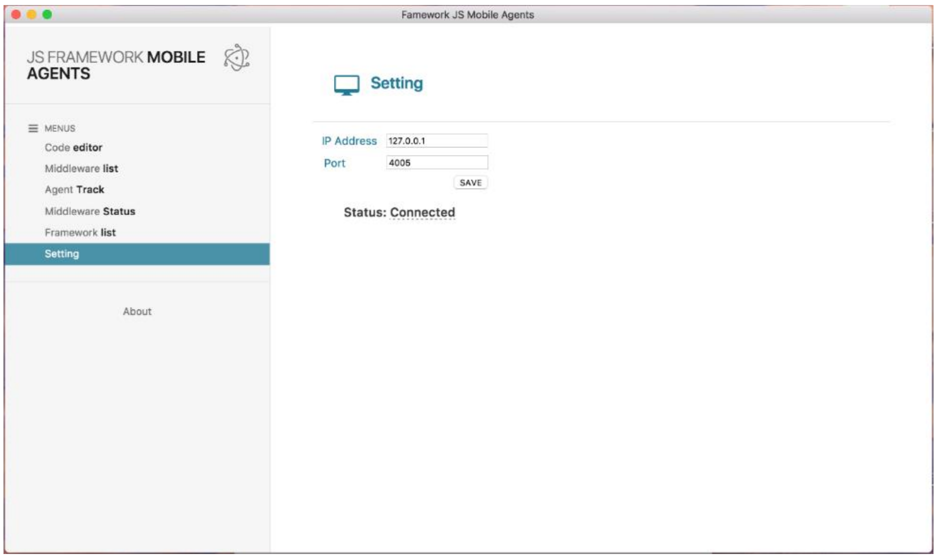
Framework# 1(::ffff:127.0.0.1) => ME

OPERATING SYSTEM | PROCESS: BOTH

'{"os":"darwin","os_type":"Darwin","hostName":"Carloss-MacBook-Pro.local","Electron_version":"1.1.0"}'

Figure 43. Mobile agent framework – Framework connected information.

Figure 44 shows Windows #6, which allows IP address and port configuration and shows the connection status.



IP Address 127.0.0.1

Port 4005

Status: Connected

Figure 44. Mobile agent framework - Configuration interface.

The mobile agent framework performs actions according to the following events:

When the framework establishes a connection with the registry server, it stores the hostname assigned by the registry server and updates the internal middleware and framework hostname list with the list sent by the registry server of connected middleware and sends information about the framework state.

When a framework receives information about a new framework that has been connected to the registry server, it registers the new connected framework in the local framework list and updates the window (see Figure 43) showing in real time the new connected framework. Nonetheless, when it receives the messages that a framework that has been disconnected from the registry server, it removes the disconnected framework from the internal framework list and updates the window (see Figure 43), by removing the disconnected host from the window.

When the framework receives information that a middleware that has been connected to the registry server, it registers the new middleware in the internal repository and updates the Window #2, adding the new connected middleware hostname (see Figure 40). Additionally, when it receives a message that a middleware that has been disconnected from the registry server, it removes the middleware from the internal middleware list and removes all information that is linked to the disconnected middleware.

When the framework receives host information, both for middleware and framework, it updates the corresponding window. However, when the event corresponds to the gathering, it sends the information to the framework that requests it, the framework gets device information and sends it to the framework that requests this information, using the registry server connection.

For mobile agent framework execution is necessary to run the following command in a terminal inside project folder:

```
Electron .
```

By default, the ports used to open the listening service are the following:

```
Registry server:   4005 TCP
```

It is possible to change the default port and there are two ways to do it:

1. From the source code, the *config.data* file can be edited from any text editor;
2. Using the mobile agent framework configuration interface (see Figure 44).

Figure 45 shows the process carried out during the connection establishment between the framework and the registry server. Once the connection to the registry server is established, the framework can react to events. Those events occur in the registry server in relation to the connected middleware, framework and agents. The events are notified automatically to each software tool without the need to perform periodic requests by the framework or middleware in order to obtain current information about events. These are notified through asynchronous messages from the registry server to all connected frameworks and middlewares. Additionally, the image shows the framework connection process after the registry server notifies all connected frameworks about a new framework that has been recently connected to the network.

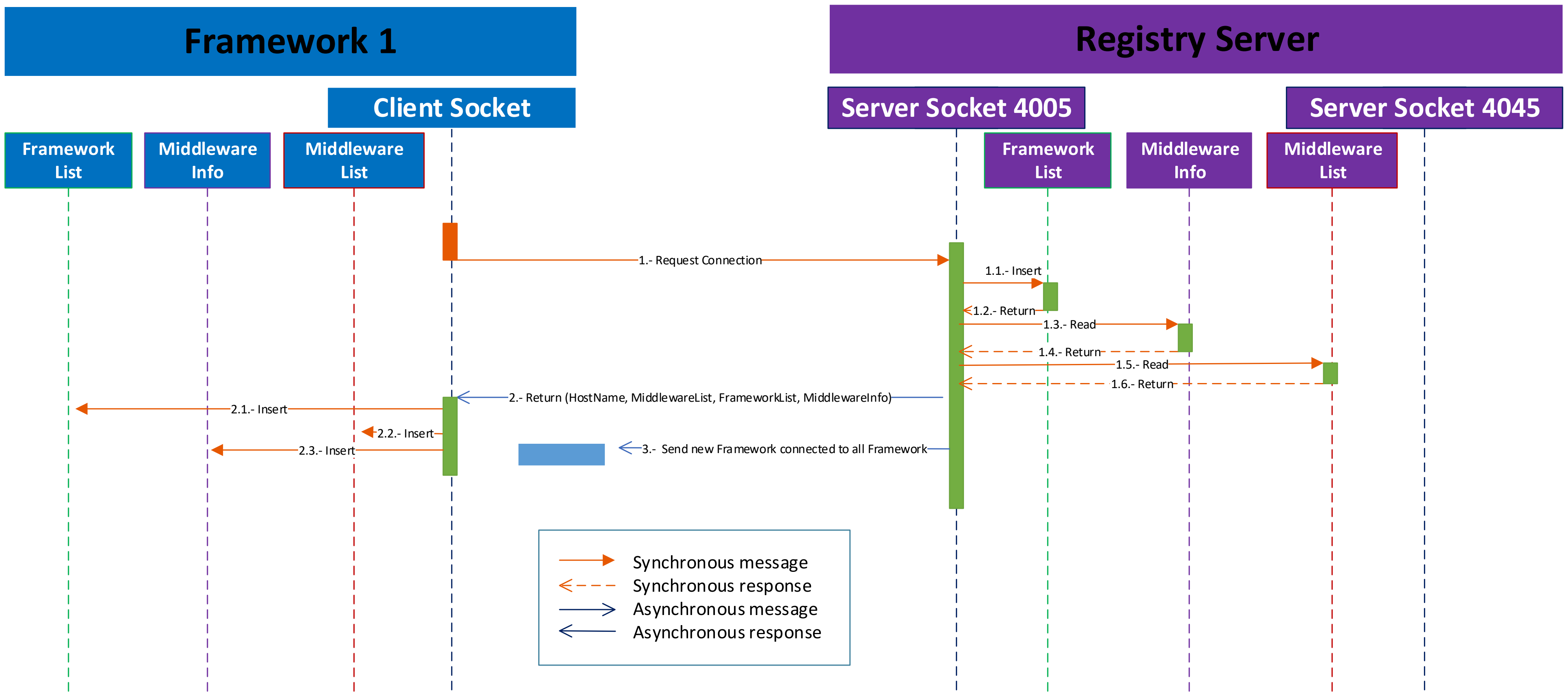


Figure 45. Framework connection data flow (Basic UML sequence of Microsoft Visio).

4.4. Summary

Figure 46 shows a controlled scenario, where an agent is sent from a Framework #1 to execute in Middleware #2. The custom agent prints a message to the console, after moving to Middleware #1 and finishes its execution.

The first interaction is the agent sending from the developed framework. To achieve that, the framework uses the registry server as an intermediary for sending the agent to the target middleware (Middleware #2). Once the agent has been received by Middleware #2, the registry server receives the delivery confirmation message and notifies the framework about the successful delivery. In addition, the Middleware #2, in a second message, notifies the registry server that it has an agent in memory. These messages are sent asynchronously, so Middleware #2, which has received the agent, continues with the process of executing the agent and it is not stopped when sending the notifications to the registry server.

The agent is executed (prints a message to the console) and proceeds to move to Middleware #1 using the API. The agent interacts with Middleware #2 through the API to get the list of connected hosts. In this case, it receives Middleware #1 as a result. After the agent moves from the Middleware #2 to Middleware #1 (using the registry server as an intermediary) it notifies all the connected frameworks about the agent's movement. This process occurs in when the agent is still moving to Middleware #1.

Once the agent arrives to Middleware #1, it sends the confirmation message to the source middleware (Middleware #2). When Middleware #2 receives the message confirmation, it removes the agent from memory and the backup from the file system. Finally, Middleware #1 notifies the registry server about the existence of an agent inside and resume agent execution.

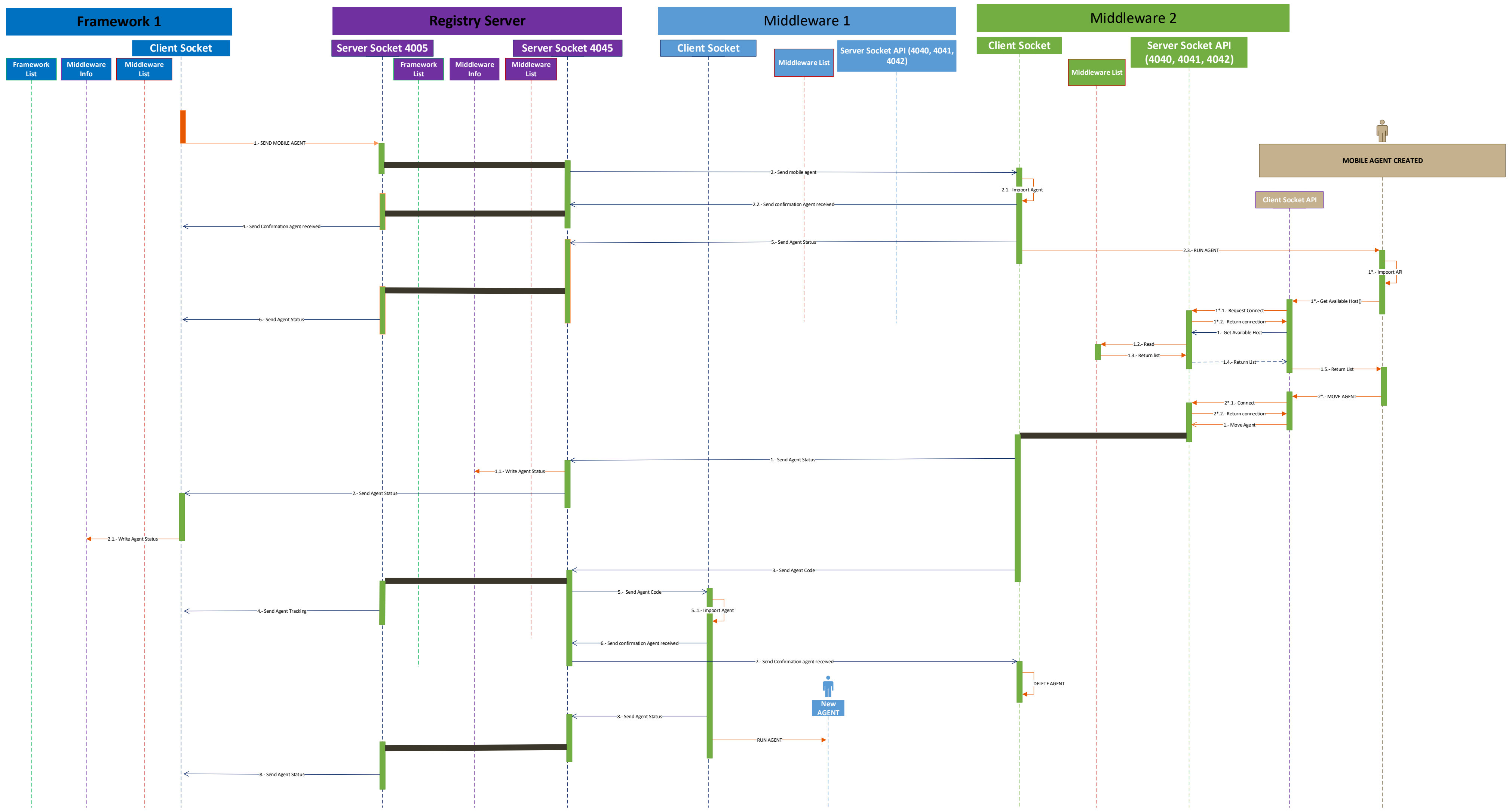


Figure 46. Agent execution data flow (Basic UML sequence of Microsoft Visio).

5 Test and evaluation

This chapter details the tests performed to the developed solution in two different environments: a) LAN and b) WAN. Three different scenarios were tested in each environment in order to evaluate all the developed software tools and verify their success functioning. The environments were evaluated using some computers and mobile devices, running different operating systems for both desktop and mobile operating systems. Table 2 shows the devices list, operating systems versions and the software versions (Node.js runtime and the React Native versions) used.

The requirements for the successful desktop middleware execution for desktop operating systems are the following:

- Node.js runtime environment, version 4.2.6 or higher;
- Package manager for JavaScript (npm), version 3.10 or higher.

Mobile devices it is only needed to have installed the mobile application developed with React Native [24] that allows the development of native mobile applications in the JavaScript language, for both Android and iOS operating systems.

Table 2. Devices used in test and evaluation environment.

Identifier	Device Type	Device Information	Operating System	Version	Device Image
DC-01-MAC	Desktop Computer	MacBook Pro (Retina, 13-inch, Early 2015)	macOS High Sierra 10.13.4	Node.js v6.11.0 Npm 4.6.1	
DC-02-WINDOWS	Desktop Computer	Dell System Inspiron N4110	Windows 10.0 (64-bit, Build 14393)	Node.js v8.9.4 Npm (MiKTeX 2.9.6630 64-bit	
DC-03-UBUNTU	Desktop Computer	Intel Desktop Board	Ubuntu 16.04.2 LTS	Node.js 4.2.6 Npm 3.5.2	
MD-01-IOS	Mobile device	iPhone 6s	iOS 11.3.1 (15E302) MODEL:MKQJ2QL/A	React native 0.46.11	
MD-02-ANDROID	Mobile device	Samsung S4 mini GT-I9192	Android 4.4.2 Kernel 3.4.0-4545479 Compilation: KOT49H.I9192UBSCQB1	React native 0.46.11	
SB-01-RASPBIAN	Single-Board computer	Raspberry PI 3 Model B V1.2 2015	2018-04-18-raspbian-stretch Debian 9.4	Node.js 4.8.2 Npm 1.4.21	

Table 3 shows the hardware specification for each used device. The developed platform can work successfully with devices with lower processing power.

Table 3. Hardware devices specification used in test envirommet.

Identifier	Processor	RAM	Network support
DC-01-MAC	2.7 GHz Intel Core i5	8 GB 1867 MHz DDR3	IEEE 802.11a/b/g/n
DC-02-WINDOWS	Intel(R) Core(TM) i5-2430M CPU @2.40GHz 64bits	8 GB DDR3	LAN: Realtek PCIe FE Family 10/100 WI-FI: Intel(R) Centerino(R) Wireless-N 1030
DC-03-UBUNTU	Intel Core 2 Duo 2.4GHz	4 GB	LAN 10/100
MD-01-IOS	Apple A9 64-bit ARM-based system-on-chip (SoC)	2GB	802.11ac Wi-Fi
MD-02-ANDROID	32bits - Qualcomm Snapdragon 400 MSM8930	1.5 GB, 533MHz	Wi-Fi: a, b, g, n
SB-01-RASPBIAN	Broadcom BCM2837RIFBG; TN1703 P11; 849-14 N3 a	1 GB RAM	BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board

The agent-based platform was evaluated in two different environments. The first one is a controlled local network. Figure 47 shows the evaluated communication architecture and the equipment involved, with additional information such as IP address and middleware connection order.

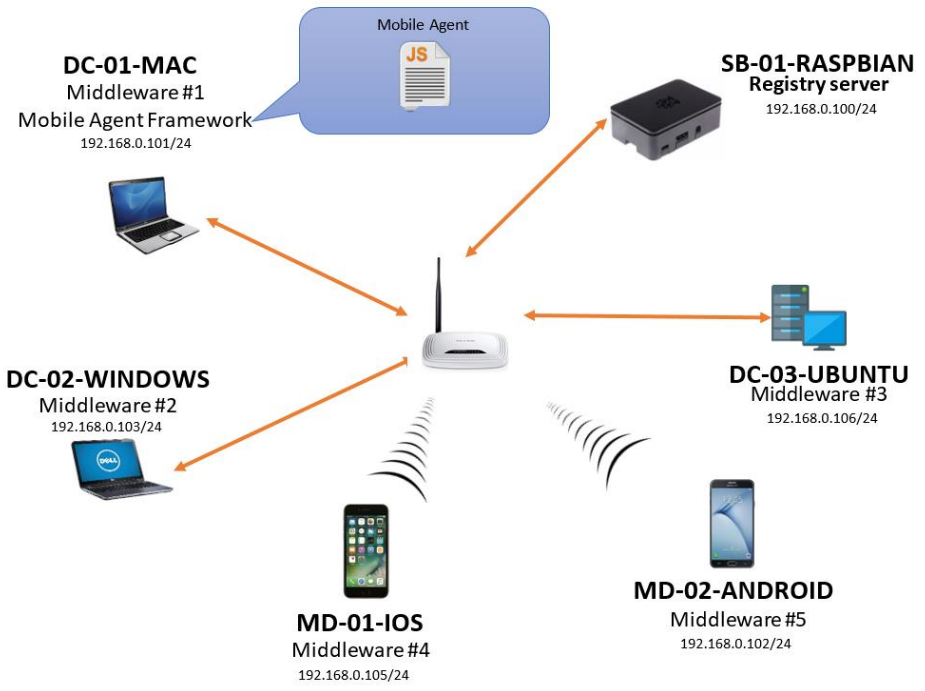


Figure 47. LAN Test environment.

The second environment includes the integration of two local networks and the Internet. It was necessary a virtual machine allocated and configured in Amazon Web Services (AWS), with a public IP address. Figure 48 show the communication architecture.

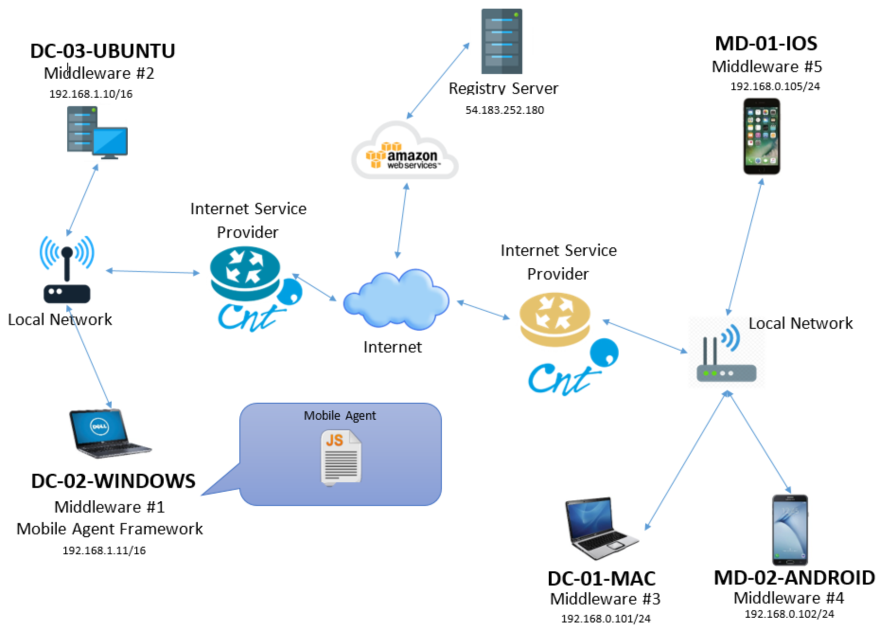


Figure 48. WAN test environment.

In each environment mentioned above the following three scenarios were evaluated:

Scenario 1 - Hello world agent with sequential movement

In this scenario, a mobile agent was developed with the aim of printing the message "Hello, World!" to the console on each device it moves to. The agent movement pattern is sequential, which means, it is executed on Middleware #1, Middleware #2 ... to Middleware #n. Figure 49 shows the process ANSI/ISO flowchart followed by the mobile agent.

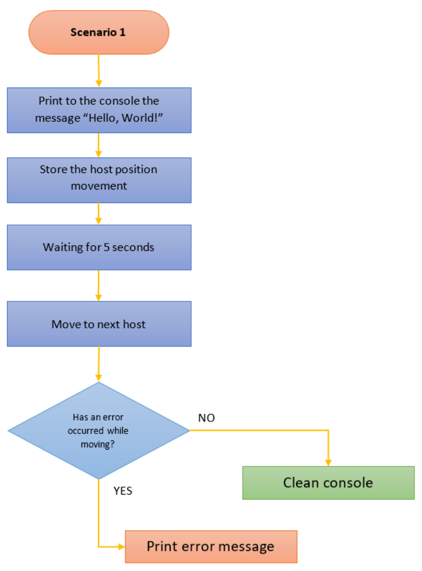
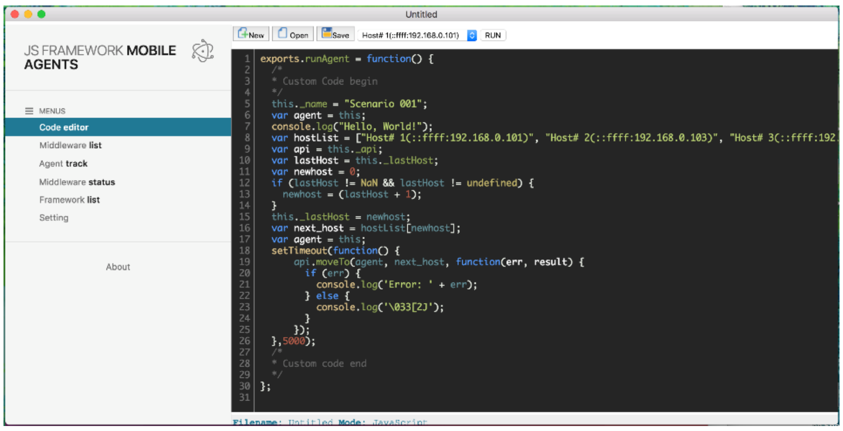

Scenario 1
Print to the console the message "Hello, World!"
Store the host position movement
Waiting for 5 seconds
Move to next host
Has an error occurred while moving?
NO
Clean console
YES
Print error message

Figure 49. Test scenario agent 1- Flowchart

The agent was deployed from the framework and sent to Middleware #1. Then, it moves to the next target host, so it is necessary for the agent to save the position of the host to which it has moved to. Figure 50 shows the agent source code.



```
exports.runAgent = function() {
  /*
  * Custom Code begin
  */
  this._name = "Scenario 001";
  var agent = this;
  console.log("Hello, World!");
  var hostList = ["Host# 1(::ffff:192.168.0.101)", "Host# 2(::ffff:192.168.0.103)", "Host# 3(::ffff:192
  var api = this._api;
  var lastHost = this._lastHost;
  var newhost = 0;
  if (lastHost != NaN && lastHost != undefined) {
    newhost = (lastHost + 1);
  }
  this._lastHost = newhost;
  var next_host = hostList[newhost];
  var agent = this;
  setTimeout(function() {
      api.moveTo(agent, next_host, function(err, result) {
        if (err) {
          console.log('Error: ' + err);
        } else {
          console.log('\033[2J');
        }
      });
  },5000);
  /*
  * Custom code end
  */
};
```

Filename: Untitled Mode: JavaScript

Figure 50. Test scenario agent 1 - Agent source code.

This scenario allows to verify the execution of a mobile agent developed in the framework and sent to a specific host to start its journey. The expected result is a message printed in the console of each device.

Scenario 2 – Hello world agent with random movement

In this scenario an agent was developed to print the "Hello, World!" message in the console, with a random movement pattern. There are three ways to develop an agent that moves in a random way: a) through the `getAvailableHost()` function, which returns a middleware hostname list, b) using the `getRandomHost()` function, which returns a middleware hostname randomly, and c) using the reserved word `"random"`. This last approach makes the task simpler and easier. Figure 51 shows the mobile agent process ANSI/ISO flowchart.

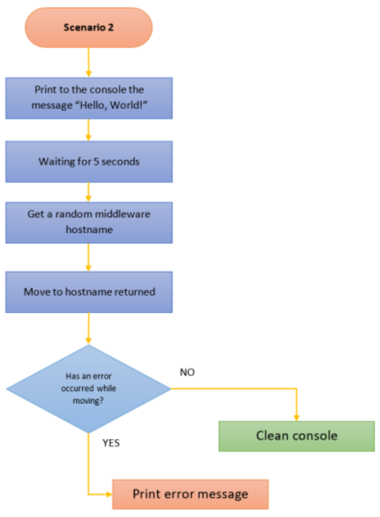
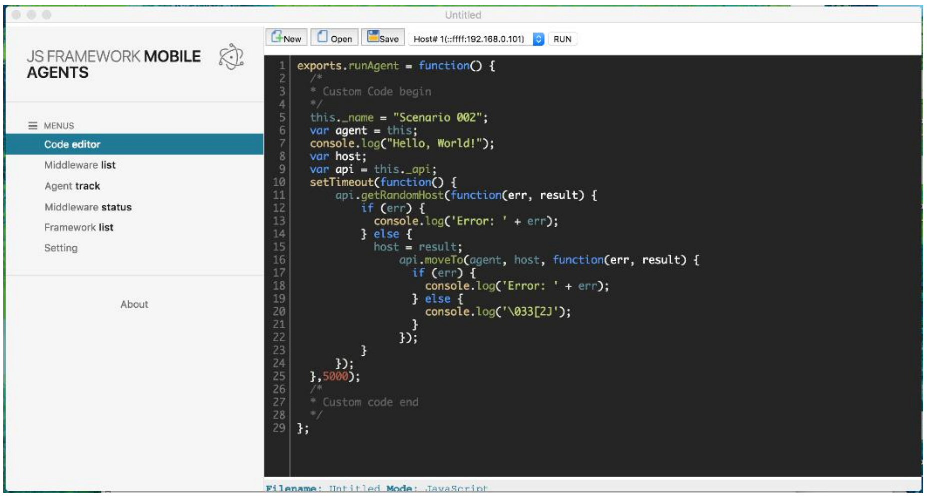

Scenario 2
Print to the console the message "Hello, World!"
Waiting for 5 seconds
Get a random middleware hostname
Move to hostname returned
Has an error occurred while moving?
NO
YES
Clean console
Print error message

Figure 51. Test scenario agent 2 - Flowchart.

The agent was developed using the mobile agent framework. Figure 52 shows the agent source code.



```
exports.runAgent = function() {
    /*
    * Custom Code begin
    */
    this._name = "Scenario 002";
    var agent = this;
    console.log("Hello, World!");
    var host;
    var api = this._api;
    setTimeout(function() {
        api.getRandomHost(function(err, result) {
            if (err) {
              console.log('Error: ' + err);
            } else {
              host = result;
                  api.moveTo(agent, host, function(err, result) {
                    if (err) {
                      console.log('Error: ' + err);
                    } else {
                      console.log('\033[2J');
                    }
                  });
            }
        });
    },5000);
    /*
    * Custom code end
    */
};
```

```
Filename: Untitled Mode: JavaScript
```

Figure 52. Test scenario agent 2 - Agent source code.

Scenario 2 allows to verify the agent execution using two public functions offered by the middleware API: a) the `getRandomHost()` function and b) the `moveTo()` function. It is expected to see in each host the message printed to the console.

Scenario 3- MAC address gathering with sequential movement

This scenario includes the test and evaluation of an agent with the aim of gathering the MAC address for each host it executes on, following a sequential movement pattern. The collected MAC addresses list is printed to the console when the agent returns to the starting host. Figure 53 shows the agent process ANSI/ISO flowchart.

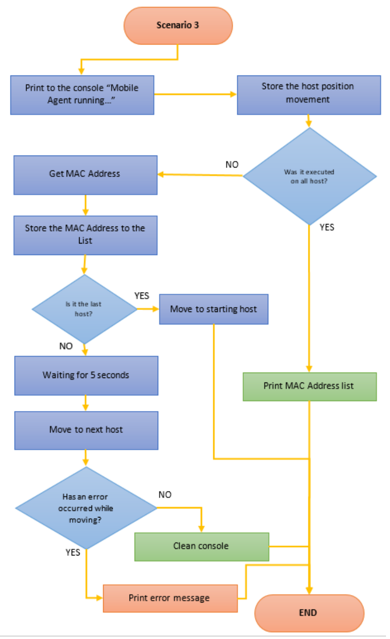
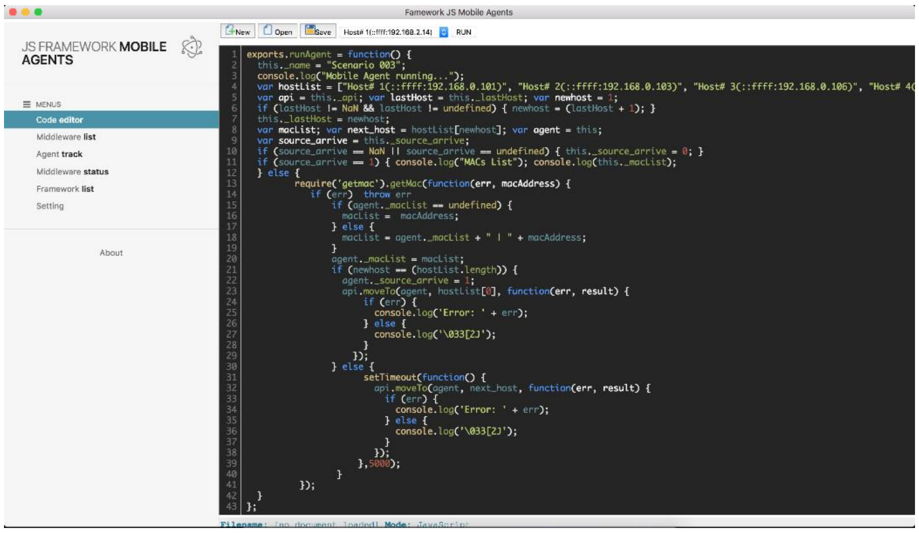


Figure 53. Test scenario agent 3 - Flowchart.

To obtain the MAC address from the host operating system it is necessary to have the "*get mac*" JavaScript module installed in each host. By default, the middleware incorporates the libraries mentioned in Section 4.2. Furthermore, it incorporates the `os` and `getmac` libraries installed through npm. Figure 54 shows the agent source code.


Famework JS Mobile Agents
JS FRAMEWORK MOBILE AGENTS
MENUS
Code editor
Middleware list
Agent track
Middleware status
Framework list
Setting
About
New
Open
Save
Host# 1(::ffff:192.168.2.14)
RUN
Filename: [no document loaded] Mode: JavaScript

```javascript
exports.runAgent = function() {
  this._name = "Scenario 003";
  console.log("Mobile Agent running...");
  var hostList = ["Host# 1(::ffff:192.168.0.101)", "Host# 2(::ffff:192.168.0.103)", "Host# 3(::ffff:192.168.0.106)", "Host# 4(
  var api = this._api; var lastHost = this._lastHost; var newhost = 1;
  if (lastHost != NaN && lastHost != undefined) { newhost = (lastHost + 1); }
  this._lastHost = newhost;
  var macList; var next_host = hostList[newhost]; var agent = this;
  var source_arrive = this._source_arrive;
  if (source_arrive == NaN || source_arrive == undefined) { this._source_arrive = 0; }
  if (source_arrive == 1) { console.log("MACs List"); console.log(this._macList);
  } else {
        require('getmac').getMac(function(err, macAddress) {
           if (err)  throw err
               if (agent._macList == undefined) {
                 macList =  macAddress;
               } else {
                 macList = agent._macList + " | " + macAddress;
               }
               agent._macList = macList;
               if (newhost == (hostList.length)) {
                 agent._source_arrive = 1;
                 api.moveTo(agent, hostList[0], function(err, result) {
                     if (err) {
                       console.log('Error: ' + err);
                     } else {
                       console.log('\033[2J');
                     }
                   });
               } else {
                     setTimeout(function() {
                       api.moveTo(agent, next_host, function(err, result) {
                         if (err) {
                           console.log('Error: ' + err);
                         } else {
                           console.log('\033[2J');
                         }
                       });
                     },5000);
                }
        });
  }
};
```

Figure 54. Test scenario agent 3 - Agent source code

Scenario 3 allows to verify the ability of a mobile agent to store data (in this case, storing MAC addresses) and to move with the data through the connected host network.

5.1. LAN test bed

The scenarios specified above were tested within a controlled local network. The relevant configurations were made in each device. For example, allowing the port traffic in the firewall and antivirus access control configurations for the successful operation of each device.

Desktop computers are connected to the router with a network wire of three feet, while the mobile devices are connected using Wi-Fi. In this environment, additional equipment was used together with the computers and mobile devices mentioned above.

One Router Tplink 150M Wireless Lite N

Model No. TL-WR741N / TL-WR741ND

Four Network wire

NEXXT UTP CAT5.e E178558 CM rated EIA/TIA-568B.2 ETL Verified

Each scenario was tested in this environment twice. The first and second agent scenarios returned the expected results while the third scenario was successfully executed on all desktop middleware, but an error occurred in the mobile middleware during its execution. After analyzing the error, it was identified that the problem occurred during the use of the `getmac` library [52]. Currently it only has support for desktop computers or any operating system where the underlying commands are present: On Windows, the command *getmac* is used and for Unix/Linux/Mac, *ifconfig* is used if present. Other environments are not supported.

5.2. WAN test bed

All devices need to have an Internet connection for connecting with the registry server and some additional devices were used in this test:

1 Computer EC2

Server t2.micro with Ubuntu server 16.04 LTS (HVM)

2 Modem ADSL2 (RTSA04N)

Internet Service Provider (ISP) “Coorporación Nacional de Telecomunicaciones”.

In this test environment all scenarios described above were evaluated and were executing twice. The results obtained were successful. During the tests in the WAN test bed it could be noticed a difference in agents execution in relation with time spend. Due to the agents movement operations, more time was spent in test bed due to the distance position of the registry server (on the Internet).

After evaluating the three scenarios in the test environments, the results obtained were as expected. Hence, each agent execution was successful. The execution is granted where the agents do not have syntax errors and the agent structure is followed. The tests revealed that when a mobile agent uses a custom or external library in their source code or some functions not supported by Node.js or React Native (it depends on the installed middleware), the successful agent execution will not be possible, so the call to external libraries must be kept in mind during agent development.

6 Conclusions and future work

The evaluation of the developed agent-based platform has provided some results that allows to establish the following conclusions and future work to improve the developed solution.

6.1. Conclusions

This dissertation work resumes the mobile agents paradigm and related work developed in this area. There are already agent-based platforms that allow the deployment of agent-based solutions, but some do not have enough support for mobile operating systems today, for example, Android and iOS. To accomplish the aim of this research, a middleware was developed for desktop and mobile devices using the JavaScript language. Additionally, the registry server was developed to allow the agents movement through the Internet. Furthermore, to facilitate the development, deployment and agent tracking, a framework was idealized and developed, and it can be seen as a dashboard that allows to visualize and gather information on both middlewares and frameworks running within the network.

The platform allows mobile agents to be moved in local networks and through the Internet. This is possible through the process of object serialization, that allows to store a running software on a string of bytes. Agents can be sent through the network and be resume with their last state using the unserialization process.

In Node.js runtime there is the *eval* JavaScript function, which allows to execute JavaScript code. In the first phase, the *eval* function was used for the agent execution, but that presented some limitations. For example, it is difficult to encapsulate the executed code (agent) into an object and send it through the network. To solve this limitation, it was specified that the agent must follow a defined structure and be handled as an "object" with the aim of being possible to store the code (agent) within an object using the “required" function for agent importation. This provides the ability to move the agent through the network and facilitates the manipulation by the middleware.

The JavaScript programming language was designed mainly for web applications development. Initially, JavaScript only supported client-side execution in web browsers. Currently, it has the ability to run on the server side too. For example, with Node.js runtime it is possible to deploy a web application running on the server side. Furthermore, Node.js runtime allows to create desktop applications using JavaScript and React Native framework can be used to create mobile applications for mobile devices using JavaScript. This is one of the features that supported the decision to choose it as the programming language for the development of the proposed solution.

The developed API provides to agents an easy way for accessing the services offered by the middleware, making easy and simple the development of mobile agents. The middleware APIs for desktop and mobile devices have different operating approaches, but they provide the same functions and capabilities to the agent and the public functions are used in the same way. This means that the API and the middleware functionality is transparent for the user or programmer. The middleware API is imported and inserted into the agent each time it is imported and it is removed before sending the agent to another middleware. Thanks to this process, agents are smaller and lighter to move between nodes inside the connected host network.

All developed software tools platform allow the development and deployment of solutions based on mobile agents easily. The middleware offers support for Windows, Linux and macOS and mobile devices platforms such as Android and iOS. This research work is in its initial stages but the results obtained are promising for future research.

6.2. Future work

There is a lot of work to be done that can provide more capabilities and functionalities to the agent-based platform. The most important directions that were identified are the following:

To work in a communication standard specification between all the software tools of the developed platform. Currently, all the communication is managed through a list type of asynchronous messages that allows to identify the operation to be performed, source and target request. Messages are sent through WebSocket using JSON. It is very important to develop a message structure for future updates of the platform.

An important aspect to improve in the developed platform is in the security area, since the security models in the WebSocket protocol [55] are based on the source. To mitigate security problems involved in the operation, the middleware must provide enough security to prevent computer attacks within the network of connected nodes. For example, this may help to mitigate *Man In The Middle* (MITM) attacks, where a third-party software can send a malicious agent to any host or modify an agent during transit from a trusted source to a specific middleware. One solution is the use of an authenticated SSL connection between software tools. There are other solutions that can be used to mitigate security issues during agent mobility and communication messages between the server registry, framework and middleware connected to the agent-based platform, such as message/agent encryption using private and public keys, use virtual private network for accessing to registry server and others.

Currently, each middleware only supports a single agent execution. When a middleware receives a second agent, during the execution of an agent, the second agent executes after the first one. As future work, the middleware can be modified with an agent container for providing a runtime environment for agent execution and to allow more than one agent to execute concurrently on the same host through the creation of asynchronous I/O.

The communication API used in the developed platform was WebSocket [55]. As future work, other communication approaches and APIs can be analyzed and compared. For example, MQTT [56], WebRTC [57], Firebase [58], XMPP [59], and others. Each one offers important functions that can provide the agent-based platform with features that help optimize and improve the performance of processes. For example, MQTT could be a good option, since it is a lightweight protocol used in Wireless Sensor Networks communications and it provides a centralized broker server with different levels of message delivery.

For the developed platform, it is necessary to use a standard for communication messages, analyze the development of a communication standard or the use of an existing one, such as Foundation for Intelligent and Physical Agents (FIPA) [2], which uses a communication language (called ACLs) or MASIF (Mobile Agent Systems Intercommunication Facility) to name just a two.

One feature to be incorporated into the agent-based platform is the capacity of agents to establish communication with other agents that are running on other hosts. Finally, but not less important, the platform could be endowed with the ability to run developed agents in other programming languages, such as LUA, TCL, C, among others, for which it is necessary to add the support of more compilers and allow the installation of additional libraries that may be required for the successful execution of a mobile agent.

This page was intentionally left blank

References

[1] Yadav, M., Sethi, P., Juneja, D., & Chauhan, N. (2018). An Agent-Based Solution to Energy Sink-Hole Problem in Flat Wireless Sensor Networks. In Next-Generation Networks (pp. 255-262). Springer, Singapore.

[2] FIPA. Retrieve from June 2018, of http://www.fipa.org/.

[3] Knabe, F. C. (1995). Language support for mobile agents (Doctoral dissertation, Carnegie Mellon University).

[4] Silva Filho, R. S. (1998). The Mobile Agents Paradigm. ICS221-Software Engineering Final Paper (Winter 2001), UC Irvine.

[5] Chess, D., Harrison, C., & Kershenbaum, A. (1996, July). Mobile agents: Are they a good idea?. In International Workshop on Mobile Object Systems (pp. 25-45). Springer, Berlin, Heidelberg.

[6] Node.js. Retrieved from April 2018, of https://nodejs.org/en/.

[7] React Native. Retrieved from April 2018, of https://github.com/facebook/react-native/.

[8] J. Baumann, F. Hohl, K. Rothermel, M. Strasser, W. Theilmann. (2002). MOLE: a mobile agent system, Software-Practice and Experience 32 (6) 575–603.

[9] D.B. Lange, M. Oshima. (1998). Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley, MA.

[10] Wong, D., Paciorek, N., Walsh, T., DiCelie, J., Young, M., & Peet, B. (1997, April). Concordia: An infrastructure for collaborating mobile agents. In International workshop on mobile agents (pp. 86-97). Springer, Berlin, Heidelberg.

[11] R.S. Gray, G. Cybenko, D. Kotz, R.A. Peterson, D. Rus. (2002). D’Agents: applications and performance of a mobile–agent system, Software-Practice and Experience 32 (6) 543–573.

[12] H. Peine. (2002). Application and programming experience with the ara mobile agent system, Software-Practice and Experience 32 (6) 515–541.

[13] D. Johnansen, K.J. Lauvset, R. van Renesse, F.B. Schneider, N.P. Sudmann, K. Jacobsen. (2002). A TACOMA retrospective, Software-Practice and Experience 32 (6) 605–619.

[14] F. Bellifemine, G. Caire, A. Poggi, G. Rimassa. (2008). JADE: a software framework for developing multi-agent applications. Lessons learned, Information and Software Technology 50 (1–2) 10–21.

[15] JADE - Java Agent Development Framework. Retrieved from April 2018, of http://jade.tilab.com/.

[16] Tripathi, A. R., Karnik, N. M., Ahmed, T., Singh, R. D., Prakash, A., Kakani, V., ... & Pathak, M. (2002). Design of the Ajanta system for mobile agent programming. Journal of Systems and Software, 62(2), 123-140.

[17] Tryllian’s. Retrieved from April 2018, of http://www.tryllian.com/.

[18] Emorphia. Retrieved from April 2018, of http://fipa-os.sourceforge.net/index.htm.

[19] Baumer, C., Breugst, M., Choy, S., & Magedanz, T. Grasshopper. (1999, October). A universal agent platform based on OMG MASIF and FIPA standards. In First International Workshop on Mobile Agents for Telecommunication Applications (MATA’99) (pp. 1-18). Sn.

[20] Howden, N., Rönnquist, R., Hodgson, A., & Lucas, A. (2001, May). JACK intelligent agents- summary of an agent infrastructure. In 5th International conference on autonomous agents.

[21] Nwana, H. S., Ndumu, D. T., Lee, L. C., & Collis, J. C. (1999, April). ZEUS: a toolkit and approach for building distributed multi-agent systems. In Proceedings of the third annual conference on Autonomous Agents (pp. 360-361). ACM.

[22] Nguyen, G., Dang, T. T., Hluchy, L., Balogh, Z., Laclavik, M., & Budinska I. (2002). Agent platform evaluation and comparison. Rapport technique, Institute of Informatics, Bratislava, Slovakia.

[23] Colors. Retrieve from May 2018, of https://github.com/Marak/colors.js.

[24] Bellifemine, F., Caire, G., Poggi, A., & Rimassa, G. (2008). JADE: A software framework for developing multi-agent applications. Lessons learned. Information and Software Technology, 50(1), 10-21.

[25] Aglet community. Retrieved from April 2018, of http://aglets.sourceforge.net/.

[26] Ajanta. Retrieved from April 2018, of http://ajanta.cs.umn.edu/.

[27] Oluwatosin, H. S. (2014). Client-server model. IOSRJ Comput. Eng, 16(1), 2278-8727.

[28] Qi, H., Xu, Y., & Wang, X. (2003). Mobile-agent-based collaborative signal and information processing in sensor networks. Proceedings of the IEEE, 91(8), 1172-1183.

[29] Cao, L. (Ed.). (2009). Data mining and multi-agent integration. Springer Science & Business Media.

[30] Qi, H., Iyengar, S., & Chakrabarty, K. (2001). Multiresolution data integration using mobile agents in distributed sensor networks. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 31(3), 383-391.

[31] Costa, N., Domingues, P., Fdez-Riverola, F., & Pereira, A. (2014). A mobile virtual butler to bridge the gap between users and ambient assisted living: a smart home case study. Sensors, 14(8), 14302-14329.

[32] Yadav, M., Sethi, P., Juneja, D., & Chauhan, N. (2018). An Agent-Based Solution to Energy Sink-Hole Problem in Flat Wireless Sensor Networks. In Next-Generation Networks (pp. 255-262). Springer, Singapore.

[33] Chen, B., Cheng, H. H., & Palen, J. (2009). Integrating mobile agent technology with multi-agent systems for distributed traffic detection and management systems. Transportation Research Part C: Emerging Technologies, 17(1), 1-10.

[34] Tanenbaum, A. S., & Van Steen, M. (2007). Distributed systems: principles and paradigms. Prentice-Hall.

[35] Lange, D. B. (1998, July). Mobile objects and mobile agents: The future of distributed computing?. In European conference on object-oriented programming (pp. 1-12). Springer, Berlin, Heidelberg.

[36] Telecom Italia Group. Retrieved from May 2018, of http://www.telecomitalia.com/tit/en.html.

[37] Tacoma Network agents. Retrieve from May 2018, of http://www.tacoma.cs.uit.no/software.html.

[38] Tryllian. Retrieve from May 2018, of http://www.isvworld.com/isvs/tryllian.
[39] Internet Archive - Tryllian. Retrieve from May 2018, of http://web.archive.org/web/20050511090700/http://www.tryllian.com:80/.
[40] Python. Retrieve from May 2018, of https://docs.python.org/3/library/.
[41] Aglets. Retrieve from May 2018, of https://sourceforge.net/projects/aglets/files/User_s%20Manual/March%202009/.
[42] Util. Retrieve from May 2018, of https://nodejs.org/api/util.html.
[43] JACK. Retrieve from May 2018, of http://www.aosgrp.com/.
[44] Tripathi, A. R., Karnik, N. M., Vora, M. K., Ahmed, T., & Singh, R. D. (1999). Mobile agent programming in Ajanta. In Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on (pp. 190-197). IEEE.
[45] Tcl Developer Xchange. Retrieve from May 218, of https://www.tcl.tk/.
[46] Marzullo, K., Lauvset, K. J., & Johansen, D. (2000). TOS: A Kernel of a Distributed Systems Management System.
[47] Jensen, S. H., Møller, A., & Thiemann, P. (2009, August). Type analysis for JavaScript. In International Static Analysis Symposium (pp. 238-255). Springer, Berlin, Heidelberg.
[48] Networking. Retrieve from June 2018, of https://facebook.github.io/react-native/docs/network.html.
[49] About Objective-C. Retrieve from May 2018, of https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html.
[50] Swift. Retrieve from May 2018, of https://developer.apple.com/swift/.
[51] Npm packager manager. Retrieve from May 2018, of https://www.npmjs.com/.
[52] GetMac. Retrieve from June 2018, of https://www.npmjs.com/package/getmac.
[53] ELECTRON. Retrieve from May 2015, of https://electronjs.org/.
[54] ATOM. Retrieve form May 2018, of https://atom.io/.
[55] The WebSocket Protocol. Retrieve from May 2018, of https://tools.ietf.org/html/rfc6455
[56] MQTT. Retrieve from May 2018, of http://mqtt.org/.
[57] WebRTC. Retrieve from May 2018, of https://webrtc.org/.
[58] Firebase. Retrieve from May 2018, of https://firebase.google.com/.
[59] XMPP. Retrieve from May 2018, of https://xmpp.org/.
[60] WebSocket/ws. Retrieve from May 2018, of https://github.com/websockets/ws.
[61] Node-serialize. Retrieve from May 2018, of https://github.com/luin/serialize.
[62] File System. Retrieve from May 2018, of https://nodejs.org/api/fs.html.

Appendices

Repositories

Registry server, desktop middleware and API repository
git clone https://casv6721@bitbucket.org/teamframeworkjs/frameworkjsnode.git

Mobile middleware repository
git clone https://casv6721@bitbucket.org/teamframeworkjs/frameworkjsreactnative.git

Mobile agent framework repository
git clone https://casv6721@bitbucket.org/teamframeworkjs/frameworkjselectron.git

Scenario 01 – Agent source code

```
exports.runAgent = function() {
  /*
  * Custom Code begin
  */
  this._name = "Scenario 001";
  var agent = this;
  console.log("Hello, World!");
  var hostList = ["Host# 1(::ffff:192.168.0.101)", "Host# 2(::ffff:192.168.0.103)", "Host# 3(::ffff:192.168.0.106)", "Host# 4(::ffff:192.168.0.105)", "Host# 5(::ffff:192.168.0.102)"];
  var host = "Host# 2(::ffff:192.168.0.101)";
  var api = this._api;
  var lastHost = this._lastHost;
  var newhost = 0;
  if (lastHost != NaN && lastHost != undefined) {
    newhost = (lastHost + 1);
  }
  this._lastHost = newhost;
  var next_host = hostList[newhost];
  var agent = this;
  setTimeout(function() {
      api.moveTo(agent, next_host, function(err, result) {
        if (err) {
          console.log('Error: ' + err);
        } else {
          console.log('\033[2J');
        }
      });
  },5000);
  /*
  * Custom code end
  */
};
```

Scenario 02 – Agent source code

```
exports.runAgent = function() {
  /*
  * Custom Code begin
  */
  this._name = "Scenario 002";
  var agent = this;
  console.log("Hello, World!");
  var host;
  var api = this._api;
  setTimeout(function() {
    api.getRandomHost(function(err, result) {
      if (err) {
        console.log('Error: ' + err);
      } else {
        host = result;
          api.moveTo(agent, host, function(err, result) {
            if (err) {
              console.log('Error: ' + err);
            } else {
              console.log('\033[2J');
            }
          });
      }
    });
  },5000);
  /*
  * Custom code end
  */
};
```

Scenario 03 – Agent source code

```
exports.runAgent = function() {
  this._name = "Scenario 003";
  console.log("Mobile Agent running...");
  var hostList = ["Host# 1(::ffff:192.168.0.101)", "Host# 2(::ffff:192.168.0.103)", "Host#
3(::ffff:192.168.0.106)", "Host# 4(::ffff:192.168.0.105)", "Host# 5(::ffff:192.168.0.102)"];
  var api = this._api; var lastHost = this._lastHost; var newhost = 1;
  if (lastHost != NaN && lastHost != undefined) { newhost = (lastHost + 1); }
  this._lastHost = newhost;
  var macList; var next_host = hostList[newhost]; var agent = this;
  var source_arrive = this._source_arrive;
  if (source_arrive == NaN || source_arrive == undefined) { this._source_arrive = 0; }
  if (source_arrive == 1) { console.log("MACs List"); console.log(this._macList);
  } else {

    if (err)  throw err
    if (agent._macList == undefined) {
      macList =  macAddress;
```

```
        } else {
          macList = agent._macList + " | " + macAddress;
        }
        agent._macList = macList;
        if (newhost == (hostList.length)) {
          agent._source_arrive = 1;
          api.moveTo(agent, hostList[0], function(err, result) {
              if (err) {
                console.log('Error: ' + err);
              } else {
                console.log('\033[2J');
              }
            });
        } else {
              setTimeout(function() {
                api.moveTo(agent, next_host, function(err, result) {
                  if (err) {
                    console.log('Error: ' + err);
                  } else {
                    console.log('\033[2J');
                  }
                });
             },5000);

      });
      }
    };
```

This page was intentionally left blank

Glossary

Atom (Software)	Atom is a free and open-source text and source code editor for macOS, Linux, and Microsoft Windows with support for plug-ins written in Node.js, and embedded Git Control, developed by GitHub.
Code on demand	In distributed computing, it is any technology that sends executable software code from a server computer to a client computer upon request from the client's software. Some well-known examples of the code on demand paradigm on the web are Java applets, Adobe's ActionScript language for the Flash player, and JavaScript.
Electron framework	Electron is a framework for creating native applications with web technologies like JavaScript, HTML, and CSS.
Firebase	Firebase [58] is a platform developed by Firebase, Inc. in 2011, and acquired by Google in 2014, firebase offers the Cloud Messaging service, allows to send notification as custom messages to client applications.
IoT	Internet of Things is the network of physical devices, vehicles, home appliances, and other items embedded with electronics, software, sensors, actuators, and connectivity which enables these things to connect and exchange data.
Middleware	Middleware is computer software that provides services to software applications beyond those available from the operating system.
MQTT	MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium.
Node.js	It is a JavaScript runtime built on Chrome's V8 JavaScript engine.
Npm	It is a package manager for the JavaScript programming language.
React Native	It is a framework for building native apps using React, it is developed by Facebook company.
Remote Evaluation	In computer science, remote evaluation is a general term for any technology that involve the transmission of executable software form a client computer to a server computer for subsequent execution at the server.
Serialize	It is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, across a network connection link) and reconstructed later (possibly in a different computer environment).
System image	It is a serialized copy of the entire state of a computer system stored in some non-volatile form such as a file.
WebRTC	Web Real-Time Communication [57] allows web browsers and mobile applications the ability to write rich, real-time multimedia applications (i.e. video chat) on the web, without requiring plugins, downloads or installs.

XMPP	XMPP [59] is a communications protocol for message-oriented Middleware based on Extensible Markup Language (XML). Originally named Jabber, XMPP allows the exchange of data almost in real time.