Flexible, Multi-platform Middleware for Wireless Sensor and Actuator Networks

Rui Miguel Guerreiro Francisco
Instituto Superior Tecnico
rui.miguel.guerreiro.francisco@ist.utl.pt

***Abstract*—In these days technology is present everywhere, such as in our homes or in our jobs. This presence of technology in our lives can bring some important advantages because the devices can start doing tasks that in the past only humans could do. The devices gain these new characteristics when they become connected to the Internet. This leads to the birth of a new paradigm named Internet of Things. With Internet of Things the devices have access to more information and with it they can make better decisions. With the progress made in the communication between the Internet objects, another paradigm called Wireless Sensor Network emerged. The Wireless Sensor Network is composed by small sensing nodes which perform the acquisition, collection and analysis of data. However, the Wireless Sensor Network has some problems such as energetic consumption and CPU load. The almost infinite capability of storage, the large processing speeds and the rapid elasticity makes Cloud Computing a very good solution to these problems. This work proposes a middleware that seamlessly integrates sensors and actuators on multiple platforms, to efficiently manage the resources of the devices and achieve efficient communication with various platforms (cloud, mobile). The goal is to allow the flow of execution to be transferred between the device and the platform under certain conditions.**

Keywords: Internet of Things, Machine-to-Machine, Wireless Sensor Networks, Cloud Computing, Middleware

I. Introduction

The revolution that has occurred in the past years in sensor and actuator technology is making much easier to build Wireless Sensor and Actuator Networks (WSAN). This WSAN consists in a set of nodes (sensors and actuators) that cooperate among them to achieve the goal of collecting data and make some decisions based on collected data. Autonomous network nodes either have a short range, or their computation power is weak. But when used collectively they are effective over large areas offering higher computational power.

Nowadays WSAN have a more pronounced growth, and it is not unreasonable to expect that in a few years our lives become dependent of WSAN in certain areas such as environmental, medical, transportation entertainment and city management. Although there has been an evolution of the nodes in WSAN these continue to have limited battery, limited computational power, etc. And with these problems the network node can crash due to lack of sufficient resources, and this may lead to jeopardize the smooth operation of the infrastructure. So, in such cases sensor and actuator networks cannot operate as stand-alone networks.

But there must be an efficient way for the captured data to be stored and manipulated. Cloud computing may offer attractive solutions for these issues. Indeed, it allows the reduction of the initial costs associated with the computational infrastructure. Another relevant aspect is that the cloud computing resources are easily and automatically adjustable according to the real infrastructure needs. This way, the computational resources are easily scalable following the growth of the infrastructure. Another important point is related with the fact that the customer only pays for the cloud resources that he actually used, therefore he does not have the problem of paying for resources that were not used. But the most important aspect to this work is that the cloud computing ensures unlimited battery, unlimited storage, unlimited computational power due to its immense of resources.

Now that a possible solution to the problem has been found, is necessary to find a way to know when the node of WSAN does not have the capability to perform some operation, and how to communicate in a transparent manner using the cloud infrastructure. To solve these issues efficiently it is necessary a mediator that manages the resources of the nodes of WSAN efficiently and communicates transparently to users with various platforms. We can call this mediator a middleware.

II. Related word

This section presents and discusses some interesting related work to the Middleware for Wireless Sensor and Actuator Networks. It is necessary to have in mind that robots can be considered a WSAN with extended functionalities. The related work that is going to be described hereafter does not satisfy completely our requirements.

A. ROS

Robot operating System [1] is an open source operating system for robots that was designed to achieve a specific set of challenges taking into account the goal for developing large-scale service robots. The philosophical goals of ROS are: Peer to Peer, Tools-based, Multi-lingual, Thin and Free and open source. These philosophical goals influence the design and implementation of ROS, as described hereafter.

a) Peer to Peer: ROS system consists in a number of different hosts connected at runtime in a peer to peer topology. Peer to Peer connectivity combined with buffering or "fanout" software modules is used to avoid unnecessary traffic flowing across the wireless link that occurs in central server.

b) Multi lingual: because many people have their preferred programming language. For these reasons ROS supports four languages C++, Python, Octave and LISP.

c) Tools-based: ROS has a microkernel instead of a monolithic development and runtime environment. In this microkernel a large number of small tools are used to build and run ROS components.

d) Thin Most: drivers and algorithms could be used in other projects, but some code has become so entangled with the middleware that it is difficult to extract. To solve this problem ROS encourage all drivers and algorithm developers to write standalone libraries without dependencies on ROS. This is achieved by placing virtually all complexity in libraries and only creating small executables.

The fundamental concepts of ROS implementation are: node, messages, topics, and services. Nodes are processes that perform computation. ROS is typically comprised of many nodes. The nodes communicate with each other by passing messages. A message is a typed data structure and can be composed of other messages and array of other messages. A node sends a message by publishing it to a given topic. A node that is interested in a specific date will subscribe. In general publishers and subscribers are not aware of each other. Publish-subscribe mode is a flexible communication paradigm but broadcast routing scheme is not appropriate for synchronous transactions. To treat this issue ROS has services. A service is composed by name and a pair of messages: one for the request and the other for the response.

B. Player/Stage

Player/Stage system is a middleware platform for mobile robotics applications that guarantees an infrastructure, drives and algorithms [3] [4]. The main features of this middleware are the platform-programming language, transport protocol-independence, open source, and modularity.

Main components of this middleware are the player and the stage. The component player is a device repository server where we can find robots sensors and actuators. Each one of these devices have an interface and a driver. The interface is used by the client of this middleware to obtain information collect by the sensor to control the actuator.

The algorithms implemented by the drivers can receive data from other devices, process the received data and then send it back. Other thing the drivers can do is to create arbitrary data when needed. The other component (stage) is a graphical simulator that models devices in a user defined environment.

This system has an architecture with three tiers. In the first tier the clients are software developers for specific robot application. The player which provides common interfaces for different robots and device are the second tier. The third tier is the robots, sensors, and actuators.

Different programming languages like C,C++, Java, and Python are used to access services. Client side libraries are in form of proxy objects. Clients can connect to the Player platform to access data, send commands, or request configuration changes to an existing device in the repository.

C. MARIE (Mobile and Autonomous Robotics Integration Environment)

Integration Environment is a middleware that was made for developing and integrating new and legacy robotic software [5]. MARIE is a flexible middleware, which allows sharing among developers, reuse of code and integration of different robotic software.

The main characteristics of MARIE are interoperability and reusability of robotic application components.

The architecture of the MARIE middleware is divided in three layers which are the following: Core, Component and Application. The core layer is where we can find the services for communication, low-level operating functions and finally the distributed computing functions.

The second layer (component) is the layer that is used to add components that are going to be constantly used by services and to support domain specific concepts.

Finally the application layer has some services and tools that are going to be very useful to build and manage the integrated application. One of the most important aspects of this middleware is its flexibility. This is visible for the middleware to provide some services that allow the adaptation of different communication protocols and applications.

MARIE uses the Adaptive Communication Environment (ACE) communication framework. This framework allows a variety of software components to connect to MARIE using a centralized component. Apart from the centralized component, there exists four functional components that are: application adapters, communication adapters, communication managers, and application managers. The application adapter behaves as proxy between the central component and the application. The goal of communication adapters is to translate the data exchange between application adapters. The connections are created and managed by the communication managers. Finally, application managers instantiate and manage components locally or across distributed processing nodes. MARIE also provides mediator interoperability layers among adapters and managers.

D. RoboEarth Cloud Engine (Rapyuta)

Rapyuta is cloud robotic platform for robots that implements a platform as a Service (PaaS) framework [2]. This framework is open source and is built upon a clone based model.

This clone based model provides secured customizable computing environment (clone) in the cloud. This way the robots can receive help in heavy computation. The robots connect to the Rapyuta and can start the computing environment by their own initiative, launch any computational node uploaded by the developer, and communicate with the launched nodes using the WebSockets protocol. The use of WebSockets protocol provides a full duplex communication channel between the robot and the cloud with predefined messages. The computing environments that are started by the robots have high bandwidth connection to the RoboEarth repository. Thus, the robots are allowed to process their data inside the computational environment in the cloud without the

downloading and local processing. Another aspect of this platform is that the computing environments are interconnected with each other.

The architecture of Rapyuta consists mainly of four elements: the computing environment, the communication protocols, the core tasks and the command data structure. The computing environments are built with Linux Containers. These containers provide isolation of processes and system resources within a single host, and they allow the applications to run at native speed because they do not emulate hardware. Linux containers allow easy configuration of disk, memory limits, I/O rate limits and Central processing unit (CPU) quotas. Thus it is possible to enable one environment to be scaled up to fit the biggest machine instance of the IaaS provider or scaled down to just relay data to the Hadoop backend.

The computing environment has to run any process that is a ROS node. All processes within a single environment communicate between them using ROS interprocess communication. The communication protocols of Rapyuta are divided in three parts: internal communication protocol external communication protocol and the communication between the Rapyuta and the applications that are running inside the Linux container. The internal communication protocol is the protocol that covers all the communication between the processes of Rapyuta. The external module has the goal to define the data sent between the physical robot and the cloud. The core task has four task sets: master, robot, environment and container. The master is the task controller that monitors and maintains the data command structure. The robot is defined by the capabilities necessary to communicate with the computing environment. Finally the container is defined by the capabilities necessary to start/stop computing environment.

Finally Rapyuta is organized in a centralized command data structure with four components. The network is the most complex of the four. These components are used to organize the communication protocols and to provide abstraction to all platform. The next component is the user that is the group of humans that has one or more robots that are going to be connected to the cloud. The authentication of these robots is made by API key that is unique to each user. To manage the robots that are running in the computing environment, there exists the loadBalancer. Finally the distributor has the functionality to distribute incoming connections from robots over the available robots.

All the mentioned middleware solves problems that this work also tries to solve such as: flexibility (MARIE), code reuse, modularity (ROS) and even share information with other devices (player/stage). However, some of these middleware are constrained by the limited capacity of the device (actuators sensors robots) where they are executed. As is the case of the player/Stage middleware that solves some problems that our proposed solution tries to solve, but brings other problems, like when one device sends some data to other device and the device that receives the information does not have the capabilities to run at the moment or even never. This way the device will perform the action even if he has no capacity for such.

Contrary to the other middleware, Rapyuta middleware can solve the problem inherent to the limitations of the hardware by running some algorithms on the cloud platform. But in this the middleware does not exists an algorithm that decides when is necessary to run some code in the cloud or in device (robot, actuator sensor).

III. Solution

This section introduces the distribution of modules over system components and communication architecture.

It is followed by the discussion of system modules. This discussion elaborates on the techniques used to make the system objectives possible.

A. System Components

The system consists in two separate physical components: devices running applications (for example clients) and the cloud that makes data processing and saves data (for example server)(see Fig. 1) . The different logic is distributed among them. There are four types of communication between these two components through the protocols TCP, UDP, SSH and HTTP REST. There is also a publish subscribe model for internal communications in the device. Many of the messages exchanged by these protocols trigger events in system components.

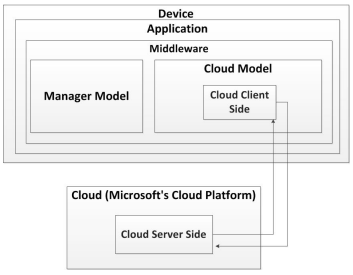

Fig. 1. System Components

Device: Devices are equipments (example cell phones, tablets, and computers) that have a minimum capacity to be able to run applications. Compared to other hardware components, they have limitations such as limited memory and limited battery. Inside of these limited devices there are running applications that were developed by programmers. It is inside of these applications that the management model and cloud client side model are going to be run (see Fig. 1) if the application programmers do not want the hardware components to reach exhaustion because of exhaustive work, ensuring that their application to have a longer life.

Cloud: It is an open and flexible cloud platform that enables to rapidly create, deploy and manage applications. And uses the Platform as a Service model. The main goal of this component is to give to the applications that are running in devices under limiting constraints, more processing power and memory, allowing the device components not to be pushed to the limit.

The cloud server side model runs in this component (see Fig. 1).

B. Manager Module

The management model aims to determine the state of certain hardware components, as well as the state of the Internet connection (depends on how the application programmer defines a good connection or bad connection). The hardware components that will be monitored are the battery, the CPU and the memory. The application programmer defines each component's critical state on a configuration file, before the middleware starts to be used. When one of these components is equal or superior to the critical value and Internet connection is good (the definition of a good Internet connection state is also defined by the application programmer) a certain execution will no longer be run on the device and starts to be executed in the cloud.

Fig. 2 shows the state machine of the management model.

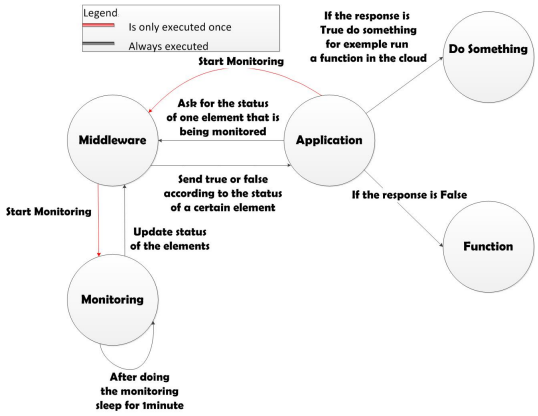

Fig. 2. state machine of the management model

The management model is initialized in the "Middleware" state when the "Application" state sends a command to "start the management middleware" (initialization class monitor). The "Middleware" state contains the conditions for the components to be monitored. These conditions are updated by the state "Monitoring" through a shared queue between the two states.

The "Monitoring" status is initialized when the "Middleware" state receives the command to initialize the management middleware. This state is always checking the conditions of the Wi-Fi signal quality and the conditions of the battery, CPU and memory, through calls to device hardware that runs the application (example robots and actuators) and is integrated with the middleware. The values obtained are compared with the critical values stipulated by the application programmer. The CPU test load is a little bit different from the other tests because it only sends a notification if the read values were three times superior to the critical value. This option avoids reactions to sporadic peaks. If the signal quality of the wireless network is below the critical value stipulated by the application programmer the remaining monitoring tests will not be performed. After each monitoring cycle of the hardware components the "Monitoring" state goes into sleep mode (for example one minute). The sleep mode duration is also defined by the application programmer.

The "Application" state sends requests to the state "Middleware" on the conditions of a certain component (example battery) awaiting for a reply of "True" or "False". If the answer is "False" it means that the state of the component is below the critical status (and hence running with enough resources on the device, so that no action is needed) stipulated earlier by the application programmer. This way the programmer's application can continue to run without any changes, and no event is initiated. This moment is depicted in the state machine through the transition between the "Application" state and "Function" state. The transition between the "Application" state and the "Do Something" state represents the situation where the response sent by the "Application" state is equal to "True". When the answer is equal to "True" it means that the conditions of the component exceed the critical value. In these cases the application programmer has the freedom to choose which action to take after receiving the message. One possible option can be to use the services that a cloud platform provides. For example performing a certain action on a machine provided by the cloud platform. This way some load is removed on the application and on the device that is running the application, and some resources are released (for instance, making available more memory).

C. Cloud Module

The cloud module has the goal of reducing the load on the hardware components. This cloud module is divided into two sub modules: the cloud client and the cloud server. In the following paragraphs both modules will be described. The cloud module state machine is shown in Fig. 3.

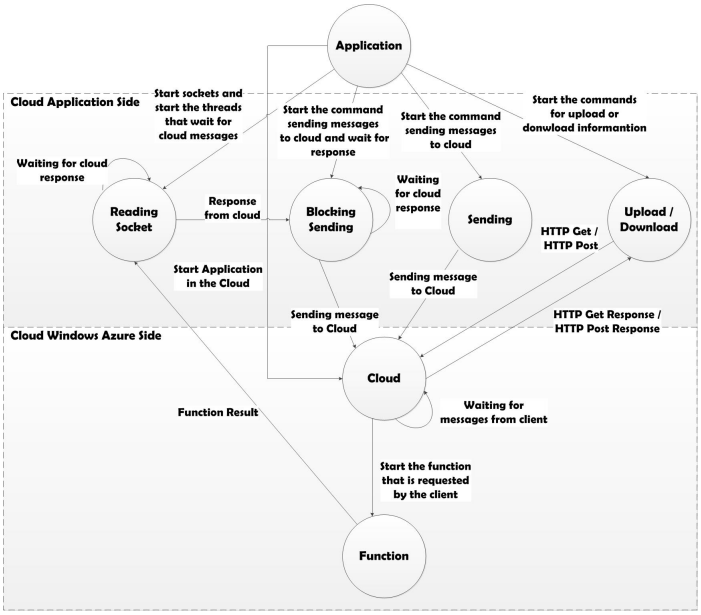

Fig. 3. Cloud module state machine

D. Cloud Client Side

The Cloud Client Side module depicts the communication between the application and the cloud. This communication is initialized when the programmer application makes an initialization call to the middleware. This boot is portrayed in the connections between the ”Application” state and ”Reading Socket” state and between the ”Application” state and the ”Cloud” state. The first step is to boot the server in the cloud via an SSH command. This way it is possible to execute a command remotely. Secondly the TCP socket and the UDP socket from the client side are created. Lastly are also configured the access settings of POST and GET commands of HTTP REST protocol. When the application programmer intends to perform an upload of information in the cloud or download information a POST or GET command to cloud is sent.

The response of the cloud can be one of these four options: either it is a confirmation that the information was successfully saved, or there was an error while performing the operation of storage of information, or the information requested by the GET command, or message for some reason was not been possible to obtain the information requested. This communication is represented in the transitions between the states ”Application”, ”Upload / Download” and ”Cloud”.

In the case of UDP / TCP connections between application and cloud there are two types of connection: the blocking call and the non-blocking call. A blocking call is portrayed through the linking between the states ”Application”, ”Blocking Sending”, ”Cloud”, ”Function” and ”Reading Socket”. In this connection a message is sent to the cloud with the following information: the function name and the arguments that the function needs executed. After the message is sent the state ”Blocking Socket” enters a blocking state and waits for the result of the function that is going to be executed in the cloud. This result will be delivered by the state ”Reading Socket.”

In the non-blocking connection a message equal to the blocking connection is sent. But in this case the state does not stays blocked after the message is sent, the program continues to run and when the response is returned by the cloud it is saved in the device memory until the program needs the response. The states involved in this connection are ”Application”, ”Sending”, ”Cloud”, ”Function” and ”Reading Socket.” The state ”Reading Socket” after being initialized enters in block mode waiting for new entries in the socket that arrive from the state ”Cloud”. When there is a new message on the socket this will be processed in two ways depending on the information of one of the fields of the message. If this field is ”True” the message is from a blocking function and the result of the function is sent to the state ”Sending Blocking”. In the case of field is ”False” the function name and the function result will be stored in the device memory until the program needs the result of this function.

E. Cloud Server Side

This module depicts the communication between the cloud and the application. The component is initialized when the machine where the application server is running receives a SSH connection with the start command. The state ”Cloud” after his startup gets locked waiting to receive messages from the client. To ensure greater efficiency in real-time situation that uses a TCP connection this state contains a mechanism that drops packets. This mechanism aims to solve a possible problem that can occur when the speed of transmission of packets is much higher than the packet reading speed. This can bring an undesirable delay to the application programmer that is using the middleware. Because of that the mechanism is based on the idea that if a new packet arrives and a previous packet was not yet read, the oldest packet is discarded and replaced by the new packet, as this new packet is much more recent than the last. Upon receipt of the message and its decoding it is possible to identify the name of the function intended to be performed and which are its arguments. Knowing what is the name of the function to be performed it is possible to go from state ”Cloud” to ”Function” state. The ”Function” state consists in the execution of the functions that were chosen by the application programmer to be in the cloud. At the end of the execution of a function a message is sent to the client (state ”Reading Socket) with the function name, with the function result and a small information for the state ”Reading Socket” can identify if the response is to the blocking function or to the non-blocking function. The sending of the message is portrayed by the link between the ”Cloud” state and ”Reading Socket” state.

In Fig. 4 it is possible to see in more detail the messages exchanged between the following components: the application, middleware and the cloud.

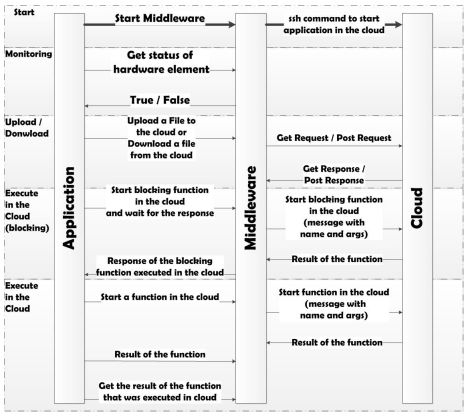

Fig. 4. Messages exchanged between the following components: the application, middleware and the cloud

In Fig. 4 the ”start” section represents the commands sent to initialize the middleware and cloud. The section ”Monitoring” represents the messages sent between the application and the middleware to verify the states of the components: battery, CPU load, memory and Wi-Fi signal quality. The following section represents the messages exchanged to do an upload or

download of information between application and the cloud and the remaining represent the messages that are sent to achieve the distribution of execution flows, differing on the existence of blocking, waiting for a reply and a non-blocking state.

F. Flexibility

To allow the middleware to be as flexible as possible, and simultaneously guaranteeing that it was not necessary to create a solution for every type of language, the following tools were used: IKVM and jython.

IKVM[1] is Open Source software that allows to directly run compiled Java code in C Sharp, transforming a jar in a dll. The IKVM has the following components: a Java Virtual Machine (JVM) implemented in .NET, implementation of Java libraries in .NET, a tool that translates Java byte code to IL NET and tools that enable interoperability between Java and .NET. Jython [7] [9] is the successor of JPython and is an implementation of the Python programming language in Java. The Jython programs can import and use any Java class, with the exception of some standard modules. Furthermore Jython includes almost all modules of the standard Python distribution, with only some of the modules originally implemented in C.

Using these two tools, the middleware becomes more flexible because the core of the middleware can be executed on C Sharp and Python.

IV. Experimental Results Assessment

The objectives of the result assessment is the validation of the implementation , and the determination of adequate system improvements. For achieving these goals, several metrics were firstly defined. This chapter is divided into areas of analysis consisting of sets of testing experiments that cover different analysis aspects. Results are gathered through the execution of a number of experiments that were defined for each area. These results are stored, taking into account the expected value and standard deviation for the set of samples of the targeted metrics. In the end, results analysis are presented in a chart format, so that their interpretation is clearer to the reader. The general assessment methodology focuses on testing, and if possible validating, the different parts of the system individually, and then progressively integrating more complexity. The detailed experimental methodology is described under each test area subsection.

A. Metrics

The following metrics were used to evaluate the potential gains that may be brought by the solution:

- Energy consumed when running the application alone
- Energy consumed when the middleware is integrated with the application
- Time that takes to perform a function in the device
- Time that takes to perform a function when the middleware is integrated
- Delay times when exists migration of execution flow to the cloud platform
- CPU load when application runs stand-alone
- CPU load when running with the help of the cloud.

B. Scenario of Tests

To test the middleware in the worst conditions that an application may be subject an application that makes video processing was chosen. This choice was due to the fact that this kind of applications need a huge CPU load, spending a lot of battery, and requiring rapid responses. The first test scenario consists on the integration of the developed middleware with an Android application (which was being developed in YDreams Robotics). This Android application uses the Computer Vision Library (OpenCV) to do face detection, as well as face tracking after the detection. As this is an application that makes a lot of image processing, it turns out to be a fairly heavy application in terms of CPU processing power, battery consumption and generated traffic. Thus, the device performance gets worse over time. This test intends to

- Check what is the percentage of spent battery
- What are the CPU loads during the application execution
- If the inclusion of middleware brings some significant delays to the application that uses it

In this scenario one BQ tablet with the Android operating system was used, and one virtual machine with one core and 1,75GB of Random-access memory (RAM)for the Microsoft's Cloud Platform.

C. Analysis of Energy Consumed

This application underwent two experimental test setups to check percentage values for the battery energy spent during the execution of the application described above. These experiments lasted 40 minutes and were repeated 5 times. The first experimental setup set the baseline, where the application that detects and tracks the human face was the only one running on the Android operating system and without any interference from the middleware. The second experimental setup consisted of the same application but integrated with the middleware, being part of the algorithm implemented in the cloud. In the graphs of Fig. 5 and Fig. 6, it can be observed that there were no gains with the transferring of certain application execution flows to the cloud. In both figures and in Table I it is possible to see that the results are better when the middleware is not integrated. Although these are almost irrelevant because the difference between the obtained values with the presence of middleware and the values without the presence of the middleware never exceed the 8%. The fact that does not exists any gains in relation to the percentage of battery energy spent with the integration of middleware may be due to excessive use of the camera as it is a hardware component that consumes a lot of battery. But this fact does not explain everything because the test that is conducted without the presence of the middleware also used this component extensively. Another aspect that may have influenced the results to be different from what was expected (the battery values being lower with

[1] http://www.ikvm.net/ (accessed last time on 14 May 2014)

the presence of the middleware) is the constant access to the wireless network because after the connection is made with the cloud, this connection will only be closed when the application is turned off or the application programmer specifies an order of termination of the connection. And this constant access consumes some battery [6] [8].

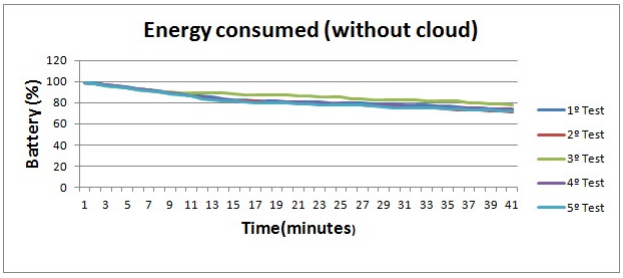

Fig. 5. Energy Consumed without using the cloud

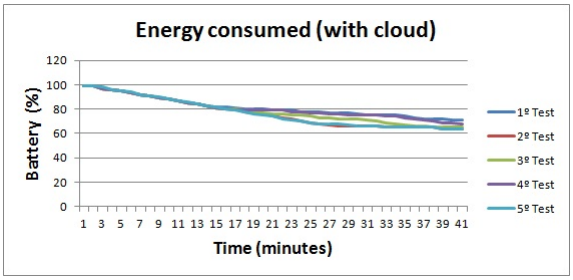

Fig. 6. Energy Consumed using the cloud

TABLE I
ENERGY CONSUMED (PERCENTAGE OF BATTERY CHARGE)

Experimental setup	Without Cloud	With Cloud
Average working battery charge (battery at the end of experiment)	73,6%	66,4%
Standard Deviation	2,70	3,04
Average of the battery charge spent by the five tests	25,2%	32,6%
Standard Deviation	3,11	3,04

D. Analysis of CPU Load

To check whether there are advantages in using this middleware in relation to the CPU load, the "Face Detect/Tracking" application was subjected to the following testing experiments (these tests lasted 20 minutes and were repeated five times):

- In the first experimental setup the application that detects and tracks the human face was the only one running on the Android operating system and without any interference from the middleware.
- The second experimental setup consisted of the same application but integrated with the middleware. The application only sends messages composed by pictures frames from the camera and sends these to the cloud with the analysis of the picture frames made in the cloud.

From figures II, 7 and 8, it is possible to notice that there was a significant gain with the migration of the detection and tracking algorithms to the cloud. This increase is explained by the fact that the heavier work is being done in the cloud, which alleviates the processing in the device's CPU that is running the application, as it only needs to get the frames and sending them. By reducing the CPU load, the lifetime of the battery should increase. As previously explained, this was not the case due to other factors, such as more battery energy required for wireless communications.

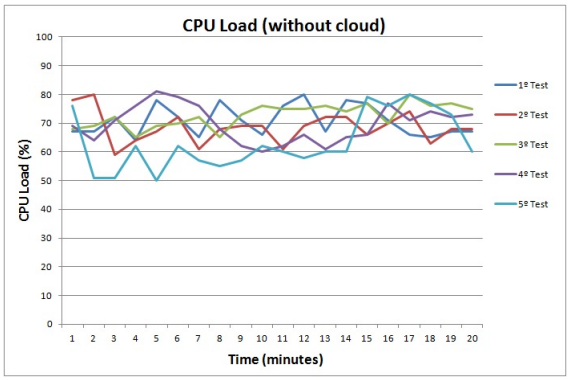

Fig. 7. CPU Load without the presence of the middleware

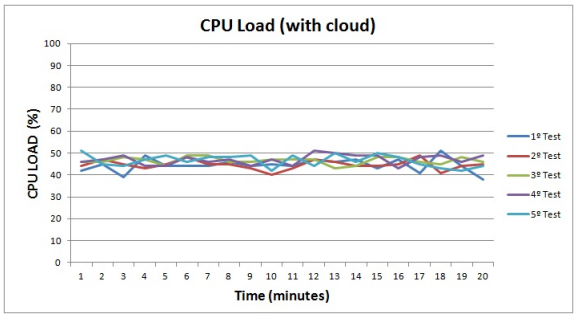

Fig. 8. CPU load with the usage of the middleware

The image processing algorithms requires a lot of CPU to be executed. This situation may preclude those others applications that should run properly (may enter a blocking state) while the image processing applications are also running. The same also happens with image processing application that ceases to have the CPU just for itself, competing for resources such as CPU

TABLE II
CPU LOAD

Experimental setup	without Cloud	With Cloud
Average Of the CPU Load	68,98%	45,84%
Standard Deviation	3,53	1,15

processing time, and this competition may create difficulties to its execution, reducing the framerate, for example less frames per second for the analysis. In order to prove that the utilization of the developed middleware is a good solution to solve this competition problem for limited resources, the following test scenarios were performed: checking the CPU status when an exhaustively analysis of 100 frames is made, and checking the time consumed

- when the analysis is done on the BQ tablet
- when it is done in the cloud.

Through the observation of Table III and Fig. 9, one can verify that when the processing is made in the tablet, the CPU load reaches saturation values. On the other hand, the usage of cloud processing originates a significantly lower CPU load at the tablet.

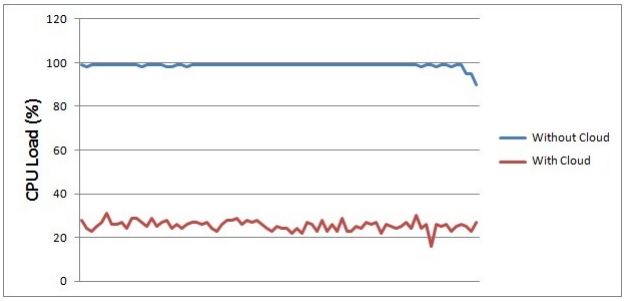

Fig. 9. Load in exhaustive case of heavy consumption of CPU computational resources by applications

It can be concluded that the integration of the middleware is beneficial when we want to run reliably more than one application on a device of limited resources, because with the transferring of execution flows to the cloud much of the processing is done outside the device thus freeing some of the CPU load, so that other device applications can also be executed.

E. Analysis of Time spent

The time delay that can exist to perform certain execution flows in the cloud is addressed hereafter.

As expected the integration of the middleware brought some delay to the execution of the application and this delay may increase when the message size increases.

After the execution of the test (Tables IV, VI and V) it was observed that there is a significant delay when the algorithm

TABLE III
CPU COMPARATIVE LOAD IN EXHAUSTIVE CASE

	Without Cloud	With Cloud
CPU Load average	98,69%	25,53%
CPU Load standard deviation	1,20	2,28
Average of the execution time of the algorithms (ms)	1292	1100
Standard deviation of the execution time of the algorithms	5,60	4,80

is executed in the device. This longer delay is explained by the the network conditions where not being the best in terms of quality. But the factor that most impacts this delay is the utilization of the TCP communication between the application and cloud. Although this means of communication is quite reliable because none of the sent messages is lost, it can also bring much delay because when there is any error in the network the lost message segments will be sent again until message reaches its destination, thereby translating into an extra delay when sending the message. Even with the implementation of a mechanism to discard messages, these are only discarded when the message is fully received

In this test it can also be verify the occurrence of high standard deviation values. This is due to image variations in terms of complexity, since image complexity can cause the increase or decrease of the algorithm execution time.

TABLE IV
TIME (IN MS) SPENT TO DETECT AND TRACK FACE WITH THE PRESENCE OF THE MIDDLEWARE (WITHOUT TRANSMISSION TIME)

Experimental Setup		
Detect	Average (ms)	157,19
	Standard Deviation	4,79
Track	Average(ms)	9,51
	Standard Deviation	25,29

V. CONCLUSIONS

Recent technological advances have presented new concepts applied to the various economic sectors of our society. WSN and M2M are emerging concepts in areas of growing interest. Indeed, the idea of monitoring several types of parameters in various environments has been motivating significant research works in these areas. Concerning some challenges presented

TABLE V
TIME (IN MS) SPENT TO DETECT AND TRACK FACE WITH THE PRESENCE OF THE MIDDLEWARE (WITH TRANSMISSION TIME)

Experimental Setup		
Detect	Average (ms)	575,56
	Standard Deviation	28,18
Track	Average(ms)	423,16
	Standard Deviation	0,934

TABLE VI
TIME (IN MS) SPENT TO DETECT AND TRACK FACE WITHOUT THE PRESENCE OF THE MIDDLEWARE

Experimental Setup		
Detect	Average (ms)	195,06
	Standard Deviation	5,56
Track	Average(ms)	56,95
	Standard Deviation	0,93

by WSN's deployments, Cloud Computing platforms are a prominent element that can respond in a more efficient and powerful way to such issues. This thesis proposes a middleware whose major goal is to solve the problem of lack of resources in devices such as limited memory and battery, with the help of cloud platforms. The proposed architecture's most important models are the management and cloud models. The management model aims to monitor the most problematic hardware components and communicate whenever some of these components are in a critical condition. The cloud uses two sub modules. One is cloud client module and this module communicates with the management module and sends messages to the cloud server module, for instance whenever one of the components is in a critical state. The cloud server module is running in the cloud platform and is where the chosen algorithms are running. These algorithms start to be executed when the cloud server module receives a message from the cloud client module. The performed tests demonstrate that the use of the cloud to solve the lack of resources problem in devices is quite advantageous because it allows the CPU load to be reduced to lower values. This leads to battery with extended autonomy, thereby providing less inconvenience to device users. It also avoids applications entering in a blocking state due to lack of memory, and allows running more applications in simple device that otherwise would exceed the available resources. But most importantly, the decision wether to run an application locally or remotely is done dynamically, according to the status of the available resources as checked through active monitoring.

However, in certain situations it cannot be ensured that all issues are solved with the cloud introduction, because some of problems are typical to hardware components (like the utilization of camera that spends a lot of battery). The solution also presented a small flaw with regard to time lost when performing certain flows in the cloud. This delay is not related with the running time of the algorithm, but instead with the time lost in sending the message between the device (robot or actuator) and the cloud and vice versa. This problem relates to the fact that the middleware uses a TCP protocol for communications which, though a reliable transport means, it introduces latencies due to the error-checking for packets.

This middleware will be beneficial for programmers who want to make the most of the hardware resources available on the devices.

REFERENCES

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R Wheeler, and A. Ng. "ROS: an open-source Robot Operating System." ICRA workshop on open source software. Vol. 3. No. 3.2. 2009.

[2] D. Hunziker, M. Gajamohan, M. Waibel, and R. D'Andrea, . "Rapyuta: The roboearth cloud engine." Robotics and Automation (ICRA), 2013 IEEE International Conference on. IEEE, 2013.

[3] M. Kranz, R. Rusu, A Maldonado, M. Beetz, and A. Schmidt. "A player/stage system for context-aware intelligent environments." Proceedings of UbiSys 6 (2006): 17-21.

[4] R. Rusu, A. Maldonado, M. Beetz, M. Kranz, L. Mösenlechner, P. Holleis, and A Schmidt. "Player/stage as middleware for ubiquitous computing." Proceedings of the 8th Annual Conference on Ubiquitous Computing (Ubicomp 2006)(Sept. 2006). 2006.

[5] C. Cote, Y. Brosseau, D. Letourneau , C. Raïevsky, and F. Michaud . "Robotic Software Integration Using MARIE." International Journal of Advanced Robotic Systems 3.1 (2006).

[6] N. Balasubramanian , A. Balasubramanian, and A. Venkataramani. "Energy consumption in mobile phones: a measurement study and implications for network applications." Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference. ACM, 2009.

[7] S. Pedroni, and N. Rappin. "Jython essentials". O'Reilly Media, Inc. 2002

[8] R. Mayo, and P. Ranganathan. "Energy consumption in mobile devices: why future systems need requirements–aware energy scale-down." Power-Aware Computer Systems. Springer Berlin Heidelberg, 2005. 26-40.

[9] R. Bill Jython for Java programmers. Sams Publishing, 2002.