EAI Endorsed Transactions
on Cognitive Communications

Research Article **ICST.ORG**

Switching Brains: Cloud-based Intelligent Resources Management for the Internet of Cognitive Things

R. Francisco[1] and A.M. Arsenio[2,*]

[1]YDreams Robotics and IST, Edificio A Moagem - Cidade do Engenho e da Artes, Largo da Estação, 6230-311 Fundão, Portugal
[2]YDreams Robotics and Universidade da Beira Interior, Edificio A Moagem - Cidade do Engenho e da Artes, Largo da Estação, 6230-311 Fundão, Portugal

Abstract

Cognitive technologies can bring important benefits to our everyday life, enabling connected devices to do tasks that in the past only humans could do, leading to the Cognitive Internet of Things. Wireless Sensor and Actuator Networks (WSAN) are often employed for communication between Internet objects. However, WSAN face some problems, namely sensors' energy and CPU load consumption, which are common to other networked devices, such as mobile devices or robotic platforms. Additionally, cognitive functionalities often require large processing power, for running machine learning algorithms, computer vision processing, or behavioral and emotional architectures. Cloud massive storage capacity, large processing speeds and elasticity are appropriate to address these problems. This paper proposes a middleware that transfers flows of execution between devices and the cloud for computationally demanding applications (such as those integrating a robotic brain), to efficiently manage devices' resources.

Keywords: Internet of Things, Finite State Machines, Resource Optimization, Wireless Sensor and Actuator Networks, Cloud Computing, Middleware.

5 H F H L Y H G R Q ' H F H P E H U D F F H S W H G R (
& R S \ U L J K W ‹ $ 0 $ U V H Q L R D Q G 5) U D Q
& U H D W L Y H & R P P R Q V $ W W U L E X W L R Q O L F H Q F H K W
U H S U R G X F W L R Q L Q D Q \ P H G L X P V R O R Q J D V W K
G R L F R J F R P H

1. Introduction

Internet changed the way we communicate. Currently, there is an explosive growth on the number of objects connected to the Internet (overpassing by large the number of connected people), giving birth to the Internet of Things (IoT), which refers to a multitude of uniquely identifiable objects (things) connected through the Internet [1]. New paradigms for the Internet of Things are crucial for migrating from nowadays sensor networks into networks of intelligent sensors enabled with actuation mechanisms and cognitive skills. Such future networks will consist of the "Cognitive Internet of Things" (CIoT). This paradigm derives from the need to enable commonplace objects with the ability to comprehend their surroundings and to make decisions autonomously [2].

It is expect that in a few years our lives will become more dependent on internet objects connected by Wireless Sensor and Actuator Networks (WSAN) in areas such as environmental, medical, transportation, entertainment and city management. This WSAN consists of a set of nodes (sensors and actuators) that cooperate among them to achieve the goal of collecting data and make some decisions. Nowadays, sensor networks are becoming a reality, especially for remote monitoring of events in fields such as healthcare, military, forest integrity or prediction of seismic activity in volcanoes. Especially due to cost and energy issues, such sensors are usually simple, with low computational processing capabilities. However, new application requirements, such as energy savings and

RUUHVSRQGLQJ DXWKRU (PDLO DUWUDUVHQLR GL EL SWfl

2.1 WSAN Middleware

IoT technology, supported by wireless network solutions enabling automatic data acquisition, has been applied in Home automation and Ambient Intelligence environments to interconnect objects and materials. We will first give a brief overview of IoT Platforms that can be employed as a starting point for the Internet of Cognitive Things, presenting their comparative evaluation. One of these platforms, OpenHAB, was integrated into the proposed solution.

There are platforms, like Xively (previously Cosm), that enable the creation of CIoT projects and extensions of these, such as Social Internet of Things and Robot as a Service projects, which lets sensors and other equipment post and read data to feeds – much like twitter works for people – which allows them to trade messages and take action. Other Home Automation platforms include openRemote and openHAB. These solutions allow users to connect virtually any device and have the platform take action when a command is given or when a pre-condition is met. Ninja Blocks provides a set of open source hardware parts for people to create their own custom devices that connect to the platform, whereas openHAB and openRemote use more standard hardware platforms.

Platforms like IFTTT and SmartThings are platforms that are more oriented towards defining intelligent behavior from devices, like informing the thermostat that the users are close to their home, through the GPS in their smartphone, and thus turning on the Air Conditioning.

Funf, developed at the MIT Media Lab, is another open source sensing and data processing framework, now a commercial product. Funf consists on software modules (Probes) that act as controllers and data collectors for each sensor's data. It also allows intelligent processing of the personal captured data (e.g. monitoring a person's physical activity by an activity monitor probe that already incorporates motion logic over the accelerometer sensor, and sharing such data on the network). It allows saving data on a remote backend (e.g. on a cloud).

SenseWeb [6] is a large-scale ubiquitous sensing platform, aimed at the global indexing of sensor readings. Its open-source layered modular architecture allows to register sensors or sensor data repositories, using web-based sensing middleware. Table 1 presents a comparative overview of these platforms.

Table 1. Comparative Analysis for Home Automation and Sensing Platforms.

Home automation Platforms	Supported OS	Programming languages	Communication Protocols	Hardware	License
xively	Multiplatform	Javascript, Ruby	HTTP (RESTfull API)	Any	Proprietary
openHAB	Multiplatform (JVM)	Java-OSGi	Any (depends on bindings)	Any	OpenSource (GPLv3)
Ninja Blocks	Multiplatform (the blocks run Ubuntu)	Ruby, Node.js, PHP, Python	HTTP (RESTfull API), Any protocol	NinjaBlocks OpenSource	Proprietary, OpenSource hardware
openRemote	Multiplatform	Java	HTTP (RESTfull API)	Any	OpenSource
SmartThings	iOS, Android	SMART	HTTP (RESTfull API)	Supported hw	Proprietary
ifttt	Web app	Non-programable	HTTP	No hw required	Proprietary
funf	Android	Java	--	Any	OpenSource
SensorMap	Web app	--	HTTP, WebService interface	Any	OpenSource

2.2 Robot Middleware

2.2.1 ROS

Robot operating System [7] is an open source middleware for developing robots. The philosophical goals of ROS are: Peer-to-Peer (P2P), Tools-based, Multi-lingual, Thin and Free and open source. These philosophical goals influence the design and implementation of ROS, as described hereafter:

Multi lingual: ROS supports four languages, namely C++, Python, Octave and LISP.

Peer to Peer: ROS system consists in a number of hosts connected at runtime in a P2P topology. P2P connectivity combined with buffering software modules is used to avoid unnecessary traffic flowing across the wireless link that occurs in central server.

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.

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 executable.

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. A node sends a message by publishing it to a given topic. A node that is interested in a specific topic will subscribe it. In general publishers and subscribers are not aware of each other. Publish-subscribe is a flexible communication paradigm, however not appropriate for synchronous transactions. To address this issue ROS uses services (a service is composed by its name and a pair of messages: one for the request and the other for the response).

2.2.2 YARP

Yet Another Robot Platform [8] is an open-source project to reduce the development effort of robotic software. YARP enables to execute processes that are location independent, and that can run on different machines without any changes in the code. This way it is

possible to move the process between machines that are in a cluster to redistribute the computational load and to recover from hardware failure. The application developer ensures this automatic allocation of processes. The addition of new components can interfere with the existing one in YARP infrastructures. But this problem is alleviated through the inclusion of more processors to the network. YARP also minimizes the dependencies between processes. If process is killed or dies does not require processes to which it connects to be restarted. Furthermore, communication channels between existing processes continue without process restart. For reducing dependencies with the operating system, Communication in YARP uses the Observer design pattern. The state of special Port object is delivered to any number of observers (in any processes). To manage these connections YARP insulates the observed from the observer and the observees from each other. In YARP a port is an active object managing multiple connections. Each connection has state that is changed by external commands. YARP uses many different communication protocols such as Transmission Control Protocol (TCP), User Datagram Protocol (UDP), multicast, etc. The ports can be connected programmatically or at runtime.

2.2.3 PEIS Kernel

Physically Embedded Intelligent Systems (PEIS) Kernel [9] is a middleware that employs the Ecology concept of Physically Embedded Intelligent Systems. PEIS Kernel provides a shared memory model, a simple dynamic model for self-configuration and introspection, and supports heterogeneous devices. The goal of this middleware is to provide a common communication and cooperation model that can be shared among multiple robotic devices. Any robot device that has a software control in the environment is considered a PEIS.

A PEIS is a set of inter-connected software components developed to control sensors or actuators. All devices are connected in PEIS by an uniform communication layer. This layer allows not only the exchange of information between the PEIS devices, but also dynamic joining and leaving. Using a uniform cooperative model allows comparison between all of PEIS devices. Devices that participate in the cooperative model can use functionalities of other PEIS devices to complete their own functionalities.

The organizational structure is divided into three layers: the communication layer, peer-to-peer network layer and Tuple layer. The communication layer at the lowermost abstraction is used to provide communication links and device detection for shared medias. This layer also provides services for initializations, calling functionalities, etc. This communication layer also serves as a bridge, translating message to a more compact protocol suitable over low-bandwidth networks.

On top of the communication layer is the P2P network layer that uses P2P algorithms for optimizing connectivity and performing routing. Finally the Tuple layer is on top of the P2P network layer. Tuple-space is a decentralized version of the shared memory where a number of tuples containing a namespace, key, data as well as a number of meta attributes such as timestamps and expiration date, can be stored and retrieved by any participating process using an abstract tuple.

The database implementation to store tuples assumes that each PEIS component can be used to store all relevant tuples. This database gives some special attention to an abstract tuple, a tuple in which one or more fields have been initialized to a wild-card value while the remaining fields have been given concrete values. This abstract tuple is used for three reasons. The first one is they are used whenever an application is accessing the database to query the current value of a tuple. Additionally, before a PEIS can access tuples at a remote location, a subscription to the corresponding tuples must be made. And finally abstract tuples are used by the event mechanism to setup callbacks when tuples changes value.

2.2.4 Player/Stage

Player/Stage system is a middleware platform for mobile robotics applications [10][11]. 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 has an interface and a driver. The middleware client uses the interface 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 send it back. Drivers can also create arbitrary data when needed. The other component (stage) is a graphical simulator that models devices in a user defined environment.

This system has a three-tier architecture. In the first tier the clients are software developers for a specific robot application. The player, which provides common interfaces for different robots and devices, constitutes the second tier. Robots, sensors, and actuators form the third tier. 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.

2.2.5 Mobile and Autonomous Robotics Integration Environment (MARIE)

Mobile and Autonomous Robotics Integration Environment is a middleware that was made for developing and integrating new and legacy robotic software [12]. MARIE is a flexible middleware, which allows integration of different robotic software. The main characteristics of MARIE are interoperability and reusability of robotic application components. Flexibility is another important aspects of this middleware, which

provides services that allow the adaptation of different communication protocols and applications.

The architecture of the MARIE middleware is divided in three layers: Core, Component and Application. The core layer contains communication services, low-level operating functions and the distributed computing functions. The component layer is used to add components that are going to be constantly used by services. The application layer has services and tools useful for building and managing the integrated application.

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. There are also four functional components: application adapters, communication adapters, communication managers, and application managers. The application adapter behaves as a proxy between the central component and the application. Communication adapters translate the data exchange between application adapters. Connections are created and managed by communication managers. Application managers instantiate and manage components locally or across distributed processing nodes. MARIE also provides mediator interoperability layers among adapters and managers.

2.2.6 RoboEarth Cloud Engine (Rapyuta)

Rapyuta is cloud robotic platform for robots that implements a platform as a Service (PaaS) open source framework [13]. This framework is built upon a clone based model, which provides a secured customizable computing environment (clone) in the cloud. This way the robots can use extra resources for heavy computation. The robots connect to the Rapyuta and can start the computing environment by their own initiative. It allows to launch any computational node as uploaded by the developer, and to 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 backend.

All processes within a single environment communicate with each other using ROS interprocess communication. The communication protocols of Rapyuta are divided in three parts: internal communication protocol, external communication protocol, and the communication between Rapyuta and applications 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 container offers the functionalities required to start/stop the computing environment.

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 platforms. The user is another component, representing the group of humans that have one or more robots to be connected to the cloud. The loadBalancer manages the load from robots running in the computing environment. Finally the distributor has the functionality to distribute incoming connections from robots over the available robots.

2.2.7 Discussion

The related work previously described does not satisfy completely our requirements. They solve indeed problems also addressed on this paper such as: software flexibility (MARIE), code reuse and modularity (ROS), and even information sharing with other devices (player/stage). However, these middleware are still constrained by the limited capacity of the devices (actuators, sensors, or robots) where they are executed. Indeed, the player/Stage middleware will transfer the execution for another device even if the later has no available capacity. Contrary to other solutions, Rapyuta middleware can solve the problem inherent to the limitations of the hardware by running some algorithms on the cloud platform. But still this middleware does not have the intelligence to decide whether it is necessary to run some code on the cloud or on a device. Francisco et al. [14] presented such solution, which is further detailed in this paper.

3. Smart Middleware Architecture

The overall system consists of devices running applications (egg clients: cell phones, tablets, and computers, as shown in Figure 1) and the cloud that performs data processing and storage. The communication protocols employed are TCP, UDP, SSH and HTTP Rest, and a publish/ subscribe model for internal communications on the device. Devices run applications developed by programmers, having constraints such as limited memory and battery (contrary to the cloud). These applications will run the management and cloud client

side modules for programmers to use our middleware. The middleware manager monitors hardware components, and communicates to the cloud client whenever a component reaches a critical condition. The cloud client interchanges application's control messages and data to the cloud server module.

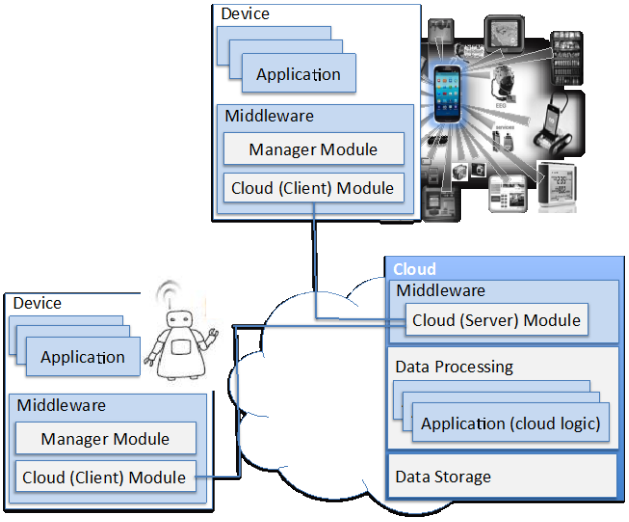

Figure 1. System Architecture and its Components.

3.1 Middleware Requirements

The Middleware runs inside the user developed application, and can be considered as an extension of the operating system, which provides a transparent communication layer between the hardware and the applications. The objective of this middleware is to solve some problems of a modular design, such as interoperability and communication configuration. To achieve these objectives the proposed middleware needs the following resources:

Simplifying the development process: application development is not easy for the robotic environment because each robot manufacturer has its own API. Middleware should simplify the development process by providing higher-level abstractions with simplified interfaces that can be used by developers.

Support communications and interoperability: The robotic and WSAN modules are designed and implemented by different manufacturers. The middleware must provide functions that help to have an efficient communication and simple interoperability mechanisms between these modules.

Provide efficient utilization of available resources: The device (single sensor, single actuator, mobile device, robot) may have single or multiple microprocessors, one or more interconnection networks, and it may need to execute intensive tasks in real time. Therefore efficient resource utilization is required. Middleware supports applications in efficiently using these resources [15][16].

Providing heterogeneity abstraction: The communication and cooperation between hardware and software is very important. To hide complexity at the communication level, as well as heterogeneity of the underlying modules, the middleware is used as a collaborative software layer.

Supporting integration with other systems: Devices often need to interact with other devices for achieving their goals in real time. Therefore the middleware should provide real time interaction services with other systems.

Supporting low resources devices: Devices may have several limitations such as limited power, small memory, limited connectivity, and so the middleware needs to have adequate functionalities to manage these resources.

Providing automatic resource discovery: The devices are dynamic systems due to their mobility. Therefore automatic and dynamic resource discovery and configuration are needed in the middleware.

To provide the abovementioned functionalities the middleware is divided in three layers (see Figure 2): the communication layer, the peer-to-peer network layer and the device manager layer. The communication layer provides communication links with multiple platforms such as the cloud or a mobile device, and it also provides device detection for shared information. This layer provides an heterogeneity abstraction for supporting communications and interoperability. The layer also provides services for system initialization.

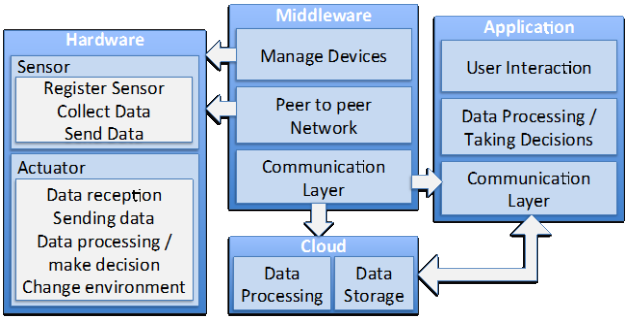

Figure 2. Detailed Architecture.

On top of the communication layer is the peer-to-peer network layer, enabling integration with other systems and automatic device discovery. This layer is responsible the connectivity with routing algorithms.

The manage device layer runs a monitoring algorithm to detect problems, such as if the device is lacking battery power or if the load of CPU is too high. This it way it is possible to support low resource devices and provide an efficient utilization of available resources.

3.2 Applications

Applications will be running on smartphones, tablet, WSAN nodes or in robots, with different operating

systems. The programmer deploying the middleware is not typically the one who develop the applications. Instead, application programmers should integrate the middleware functionalities at development time. The application can have different types of actions:

- Presenting a textual or graphical representation of the information acquired by the device, which may be stored in the device or in the cloud platform.
- Processing information received from the device.
- Performing operations according to the results of information processing.
- Performing specific operations on the device.

The middleware will be integrated with the application thereby ensuring that the application receives information about the status of the hardware components that the middleware is monitoring. Thus the integration between application and the middleware can help to avoid critical situations (for instance excess CPU usage) and prevent device shutdown (for instance due to lack of battery).

3.3 Cloud Platform Services

The cloud-based platform will be in charge of two relevant functions: data processing and data storage. Cloud computing resources can be easily and automatically adjusted according to new application demands or the growth of application's requirements.

Data Processing: The Data Processing component is in charge of processing the data transmitted by the devices. This component executes the heavy work, not possible to be carried out by the device due to lack of resources (such as lack of energy or computational resources to process the collected data). Hence this component allows the device not to crash by executing the work that the device could not do, sending back the processing result to the device.

Data Storage: This component is in charge of storing persistently the collected data from each WSAN node. If devices do not have space to store the collected data because the device memory is full (or above a given threshold), such data can be transmitted to the cloud platform using the middleware, so that data is not lost.

3.4 Manager Module

The management module aims to determine hardware components state (battery, CPU and memory), as well as the wireless connection state. The programmer defines each component's critical state on a configuration file, before the middleware starts to be used. Whenever one of these components achieves a value above a critical value (and wireless signal is strong) a certain execution will no longer be run on the device, being transferred to the cloud. Figure 3 shows the management model's state machine.

The management model is initialized in the "Middleware" state when the "Application" state sends an initialization command. The "Middleware" state contains the monitored component conditions, which are updated by the "Monitoring" state through a shared queue between the two states. The "Middleware" state is always checking the conditions of the wireless signal (Wi-Fi was employed on the scope of this paper) quality and the conditions of the battery, CPU and memory, through calls to device hardware that runs the middleware. The values obtained are compared with the critical values stipulated by the application programmer. The load CPU analysis is a bit different from the other checks, because a notification is only sent if the read values are superior to the critical value for three times in a row (to avoid reactions to sporadic peaks). If the signal quality of the wireless network is below the critical value stipulated by the 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 an interval of time (one minute).

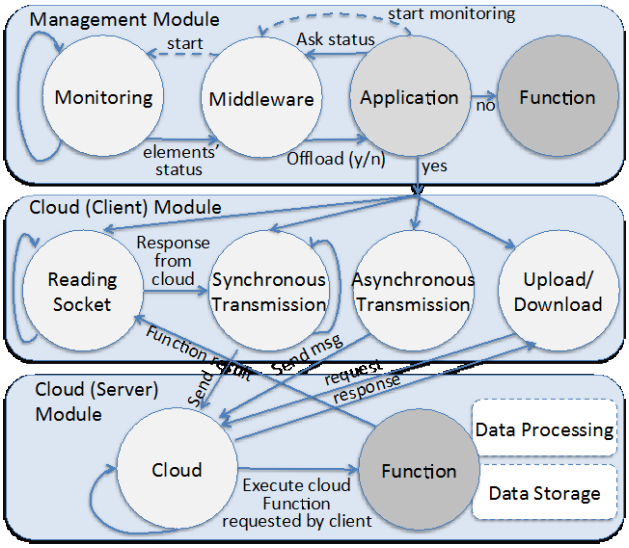

Figure 3. Management and Cloud state machines.

The "Application" state sends requests to the "Middleware" state on the conditions of a component (e.g. battery). If the reply is "False" (transition between the "Application" and "Function" states), it means no action is needed, since the state of the component is below the critical status (and hence running with enough resources on the device). This way the programmer's application can continue to run without any changes, and no event is initiated. In case of a "True" response sent by the "Application" state, the machine transits to the "Do Something" state, meaning the component exceeds the critical value. In this case the programmer chooses the actions to take after receiving the message. One possible option is to use cloud platform provided services for

performing certain actions. This way it is removed some load on the device that is running the application, releasing resources (e.g. memory).

3.5 Cloud (Client) Module

The communication between the application and the cloud is initialized when the programmer application makes an initialization call to the middleware. The first step is for the client to boot the server in the cloud via an SSH command and to create TCP and UDP sockets. The access settings of Post and Get commands of HTTP Rest protocol are also configured, so that whenever the programmer intends an application to perform an upload or download of information in the cloud, it is sent a Post or Get command to the cloud.

The middleware in the cloud responds (transitions between ”Application”, ”Upload / Download” and ”Cloud”) states either: i) with a confirmation that the information was successfully saved; or (ii) there was an error while performing the storage operation; or (iii) the information as requested by the get command; or (iv) error due to failure on getting the requested information.

In a blocking call connection, a message is sent to the cloud with the following information: the function ID and its arguments. After the message is sent the state ”Synchronous transmission” enters in a blocking state and waits for the result of the function that is going to be executed in the cloud, delivered by the state ”Reading Socket”. In the non-blocking case, the state is not blocked after the message is sent (the program continues to run). Once the cloud returns the response, this is saved in the device memory until the program needs to access it. The state ”Reading Socket” after being initialized enters in block mode waiting for new entries in the socket that arrive from the state ”Cloud”. A new message on the socket will be processed differently depending on the information of one message field. If the message is from a blocking function, the result is sent to the state ”Synchronous transmission”, otherwise the function ID and its result will be stored in the device memory until the program needs the result.

3.6 Cloud (Server) Module

The Cloud Server side module (see Figure 3) is initialized at the application server once it receives an SSH connection with the start command, locking the ”Cloud” state, and waiting to receive messages from the client. Upon receipt of the message and its decoding, it is possible to identify the function ID that is intended to be performed and its arguments (going from state ”Cloud” to ”Function” state). The ”Function” state consists of the execution of the functions that were chosen by the programmer to run 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 ID and its result. It is also sent a small packet to identify if the response is to the blocking (synchronous) or the non-blocking (asynchronous) function.

3.7 Transparent Implementation

A key factor in the development methodology was the exploitation of existing frameworks and libraries in the proposed solution implementation process, harnessing the potential of currently available tools.

For a flexible and transparent implementation of the middleware, IKVM (Open Source software that allows to directly run compiled Java code in C#) and jython (an implementation of the Python programming language in Java) were employed, which allow the JAVA core of the middleware to be executed on C# and Python, respectively. Communication interfaces employ JavaScript Object Notation (JSON), since it ensures greater efficiency on message delivery between clients and the cloud. Communications between applications and the Microsoft cloud, when it comes to data storage and downloading of information from the cloud, is ensured by the Microsoft cloud API that uses the Hypertext Transfer Protocol (HTTP) REST protocol.

Aiming to ensure that the configuration of certain points of the middleware is conforming to each application needs, and to set certain aspects of the cloud (such as the login to the machine that was obtained in the cloud), configuration files were used, employing the INI file format (ini4j.jar). These simple files have a pretty basic structure organized in sections and properties. Furthermore this format is quite often used for drivers’ settings and Linux/Unix systems for system configuration.

OpenHAB, a home automation middleware, was integrated into the solution to enable support not only for robot devices and middleware, but also the transparent integration of home automation devices.

4. Experimental Evaluation

The objective of result assessment is the implementation validation, and the determination of adequate system improvements. This section 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. 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.

4.1 Metrics and Experimental Setup

The following metrics were used to evaluate potential solution gains: Energy consumed, Time that takes to perform a function, Delay times and CPU load. Two experimental setups were implemented:

(i) The baseline: application running stand-alone in the device (no middleware, the application that detects/tracks the human face is the only running on the Android OS).
(ii) Middleware integration with the application using cloud resources: the same application solely sends messages containing image frames, getting these from the camera, and sending these to the cloud, being the frames' analysis made in the cloud.

The hardware setup consists of one BQ tablet with Android OS, and one virtual machine with one core and 1,75GB of RAM). The application uses the Computer Vision Library (OpenCV) to do face detection, as well as face tracking after detection. As this application makes significant image processing, it turns out to be a fairly heavy application in terms of CPU processing power, battery consumption and generated network traffic. Thus, the device performance gets worse over time. With these tests it is intended to check the percentage of battery discharged, and CPU load, during the application execution. The second experimental setup consists at the integration of the middleware developed with an Android application, using Microsoft Azure platform as the cloud computing infrastructure.

4.2 Energy Consumption

The application was tested on the two aforementioned scenarios to check percentage values for the battery energy spent. These experiments lasted 40 minutes and were repeated 5 times. As shown in Table 2 and Figure 4, there is no evidence of gains by transferring some application execution flows to the cloud. Results are even slightly better when the middleware was not used (difference never exceeded 8%).

Table 2. Energy Consumption (as percentage of the battery full charge).

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

This similarity may be due to excessive use of the video camera, which consumes a lot of battery power, although this component is also used extensively without the middleware. The constant access to the wireless network should be the largest impact on the results, since the higher transmission rate implies higher energy spending [17].

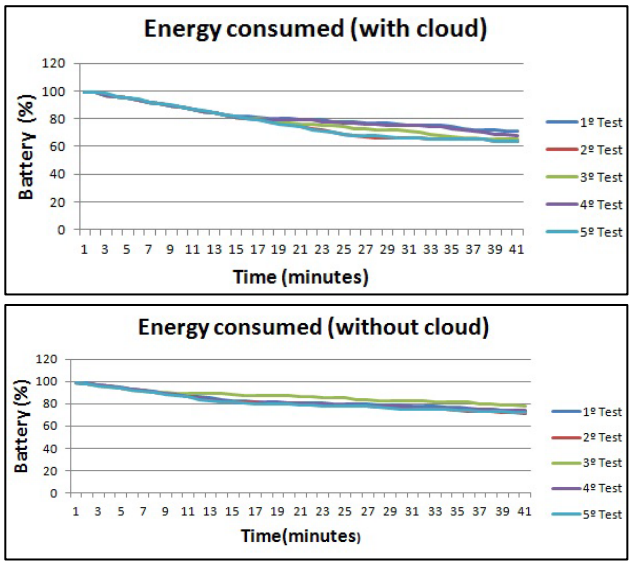

Figure 4. Energy Consumed, with and without cloud.

4.3 CPU Load

To check potential middleware advantages in relation to CPU load, the "Face Detect/Tracking" application was subjected to tests lasting 20 minutes and repeated five times. In Table 3 and Figure 5, there is a significant gain with the migration of the detection and tracking algorithms to the cloud. This gain is due to the heavier work done now in the cloud, which alleviates the processing needs of the device's CPU running the application (the device just grabs image frames on the device and sends 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.

Table 3. CPU Load with and without cloud support.

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

4.4 CPU Load – Heavy Processing

However, in situations where applications compete for CPU time in processing constrained devices, this solution can bring very interesting benefits. The image processing

algorithms require a lot of CPU load to be executed, which may prevent simultaneously other applications to run properly. The same also happens with the image processing application that ceases to have the CPU just for itself, competing for resources such as CPU processing time, and this competition may create difficulties to its execution, such as a smaller frame rate, getting this way less frames per second.

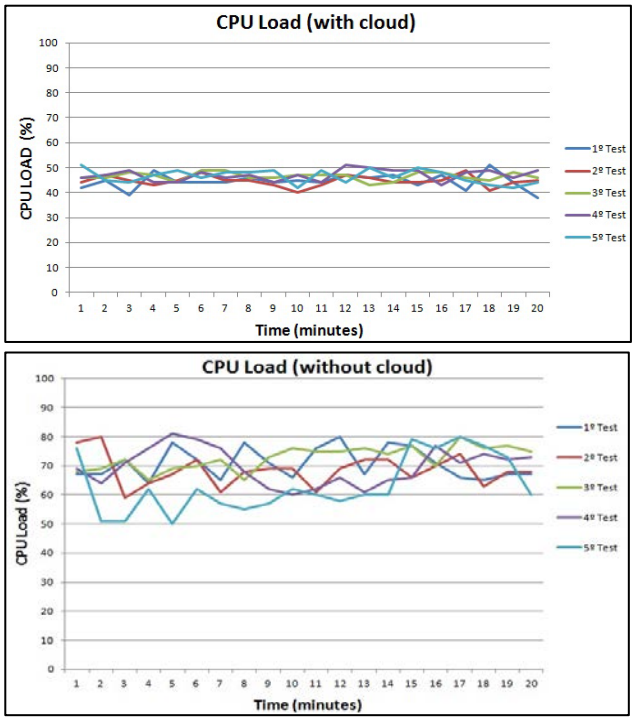

Figure 5. CPU Load with and without cloud.

To check if the middleware solution can solve this competition problem for limited resources, the following test scenarios were performed: checking the CPU status whenever an exhaustive analysis of 100 frames needs to be made, and checking the time consumed for both processing these set of images in the tablet or in the cloud.

According to Table 4 and Figure 6, the CPU load reaches saturation values (100%) for single tablet processing. But usage of cloud processing originates a significantly lower CPU load at the tablet (~25% in average). Hence the integration of the middleware may be beneficial to run reliably multiple applications on a device of limited resources, because by transferring the execution flows to the cloud, much of the processing will be done outside the device, thus freeing some of the CPU resources (decreasing CPU load), so that other device applications can also be executed.

Table 4. 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 (execution time)(ms)	1292	1100
Standard deviation (ms)	5,60	4,80

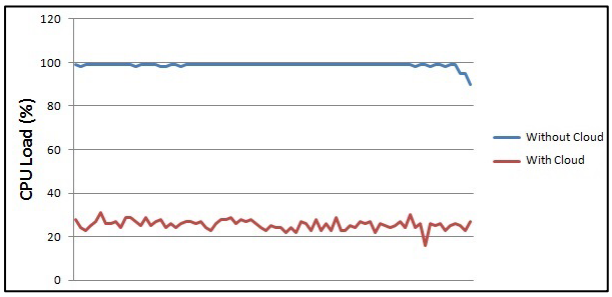

Figure 6. CPU Load in exhaustive case, with and without cloud support.

4.5 Network and Processing Delays

The proposed solution exploits networked cloud services, and so it is of relevance to test time delays associated to performing certain execution flows in the cloud.

As expected the integration of the middleware brought some delay to the execution of the application (see experimental results in Table 5), and this delay may increase whenever the message size increases.

Experimental tests evidenced there is a significant delay comparing with processing time for the baseline test, corresponding to executing the algorithm in the device. This long delay is explained by network conditions, since this network has some restrictions due to its large number of users. But the factor that most impacts this delay is the usage of TCP communications at the transport layer between the application and the cloud. Although TCP is a quite reliable transport protocol, since guarantees message delivery, it can also bring large delays, since network problems result in lost message segments being retransmitted 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 (the use of UDP communication was not feasible to test in this case due to Microsoft Azure platform limitations).

Table 5. Time delays for face detection and tracking.

Experiment	Face Detection / Track Algorithms	Average (ms)	Standard Deviation (ms)
Baseline (no cloud)	Detect	195,06	5,56
	Track	56,95	0,93
Cloud-based (not	Detect	157,19	4,79

accounting transmission time)	Track	25,29	9,51
Cloud-based (including transmission time)	Detect	575,56	28,18
	Track	423,16	0,934

Observation of Table 5 also reveals 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. Another important factor contributing to these high values is the quality of Internet connection, which varies over time due to several factors such as network congestion. Thus the end-to-end execution times of the algorithm may vary significantly.

4.6 Cloud Storage for Wireless Sensors

A second test scenario was prepared consisting of the integration of the middleware with the Android application of BIOPLUX. This Android application connects via Bluetooth to a Plux@ motion bracelet (shown in Figure 7) that contains a motion sensor. This sensor is continuously sending inertial coordinates to the Android application running on a BQ tablet (that saves the coordinates in a file). As the Android device has reduced memory resources it is very likely that the Android memory will eventually be fully allocated after some time. The middleware integration with the Android application solves this issue, since whenever the used memory reaches a certain value the data will be sent to the cloud (using Windows Azure Blob Storage and the cloud storage service). This way the application will not block, and data will not be lost. With this test, whenever the memory arrives to values equal or superior to 50% occupancy and the Internet connection is good (according to the definition of "good" given by the application programmer), a group of files is sent to the cloud. This test scenario will evaluate if after the uploading of the files, the state of the memory will drop below 50% occupancy. And finally it will be checked if the files that were sent to the cloud were successfully saved.

Figure 7. BIOPLUX motion bracelet.

Therefore, this experiment will evaluate whether the hardware management mechanism avoids an application entering a blocking state, or even crashing due to lack of resources. Once the memory of the Android reaches half its capacity, an application sends all files at a specific folder to the cloud and erases them from Android device memory, freeing the memory so that the application can continue to run reliably. After running the experiment, all the files that were in the folder were successfully stored in the cloud, showing that the management mechanism is reacting when the memory reaches is a critical point (Figure 8 shows the upload of one such file on the cloud).

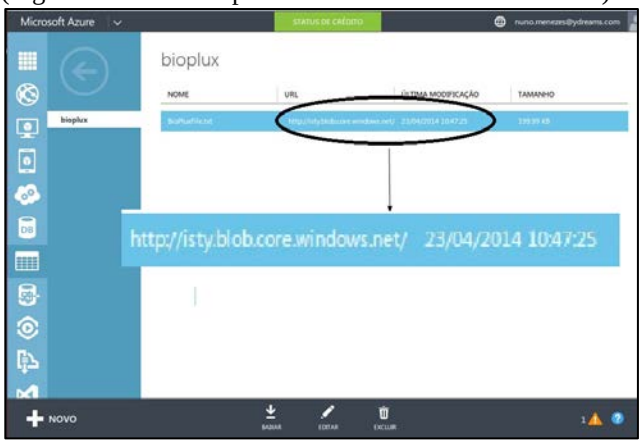

Figure 8. Azure Interface (showing that a file was uploaded to the cloud at specific time instant).

More recently [18], we have successfully integrated the proposed middleware with a WSAN platform, OpenHAB, for smart home automation (see Figure 9). Data storage is transferred between the OpenHAB server device and the cloud according to the monitored conditions.

Figure 9. Experimental setup consists of several sensors and actuators connected to a RaspberryPI running OpenHAB. It is also presented a user interface image showing sensor values in real-time.

4.7 Discussion

The inclusion of this middleware in applications to be deployed on devices with limited hardware resources (e.g.

battery, CPU) brings some advantages for the lifetime of an application and it avoids the application entering in a locking status because of lack of resources. However, in certain applications it is impossible to make gains on all hardware components as seen in aforementioned experimental tests. This is the case for applications that make excessive use of certain hardware components that require a lot of battery. Although the middleware also consumes some device's energy due to higher communication transmissions, this overhead is often small when compared with the energy consumption by one application, or a multitude of them (as it is often the case for complex robotic brains [19]) that has quite high CPU processing needs. The downside of this middleware concerns high transmission times for sending and receiving messages to/from the cloud. However this limitation can be overcome with the implementation of other transport protocols (if the cloud platform allows so). Even with these restrictions the developed middleware proves that it can solve some problems that exists in devices that are limited in terms of resources.

5. Conclusions

Recent technological advances leveraged the introduction of new concepts applied to the various economic sectors of our society. WSAN and cloud robotics are emerging concepts in areas of growing interest. Indeed, the idea of monitoring several types of parameters in various environments has motivated significant research works in these areas. Cloud Computing platforms are a prominent element that can respond in a more efficient and powerful way to current challenges.

This paper proposed a state machine based middleware to manage the transferring of execution flows between terminal devices and the cloud. The main goal was, using cloud technology, to address the problem of a device's lack of resources such as limited memory and battery. On a larger scope, the goal was to address the development of a middleware supporting the flexible, and dynamic, transferring of execution flows on a cognitive robotic brain between on-board devices and the cloud. Experimental evaluation showed that offloading the execution flows into the cloud does not necessarily reduces energy consumption (or increases battery lifetime), because more battery energy may be required for wireless communications. Experiments indicate however that using the cloud to solve the lack of device resources is quite advantageous, due to CPU load reduction. This may lead to battery with extended autonomy. But most importantly, it avoids applications entering in blocking states due to lack of memory. It also allows running more applications in a simple device that otherwise would exceed the available resources. The decision whether to run an application locally or remotely is done dynamically, according to the status of available resources, as checked through active monitoring.

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

References

[1] Woelffle, H., Guillemin, P., Friess, P. and Sylvie, W. (2010) Vision and challenges for releasing the Internet of Things. In Publications Office of the European Union.

[2] Arsenio, A., Serra, H., Francisco, R., Andrade, J., Serrano, E., Nabais, F. (2014) Internet of Intelligent Things – Bringing artificial intelligence approaches for communication networks. In inter-cooperative collective intelligence: techniques and applications, 495, 1-37, Springer.

[3] Iera, A., Floerkemeier, C., Mitsugi, J. and Morabito, G. (2010) The Internet of Things. Guest Editorial. In IEEE Wireless Communications. 17 (6).

[4] Ren, F. (2011) Robotics cloud and robotics school. 7th IEEE International Conference on Natural Language Processing and Knowledge Engineering (NLP-KE).

[5] Guizzo, E. (2011) Robots with their heads in the clouds. Spectrum, IEEE 48 (3), 16-18.

[6] Grosky, W., Kansal, A., Nath, S., Jie, L. and Zhao, F. (2007) SenseWeb: An infrastructure for shared sensing, IEEE MultiMedia, 14 (4), 8-13.

[7] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R. and Ng, A. (2009) ROS: an open-source Robot Operating System. ICRA workshop on open source software. 3(2).

[8] Metta, G., Fitzpatrick, P. and Natale, L. (2006) YARP: Yet Another Robot Platform. International Journal of Advanced Robotic Systems 3(1).

[9] Broxvall, M., Seo, B. and Kwon, W. (2007) The PEIS kernel: A middleware for ubiquitous robotics. In Proc. of the IROS-07 Workshop on Ubiquitous Robotic Space Design and Applications.

[10] Kranz, M., Rusu, R., Maldonado, A., Beetz, M. and Schmidt, A. (2006). A player/stage system for context-aware intelligent environments. In Proc. UbiSys, 6, 17-21.

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

[12] Cote, C., Brosseau, Y., Letourneau, D., Raïevsky, C. and Michaud, F. (2006) Robotic software integration using MARIE. Int. Journal of Advanced Robotic Systems 3(1).

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

[14] Francisco, R. and Arsenio, A. (2014) Intelligent multi-platform middleware for wireless sensor and actuator networks. In Proc. of the 1st International Conference on Cognitive Internet of Things Technologies, Italy.

[15] Mohamed, N., Al-Jaroodi, J. and Jawhar, I. (2009) A review of middleware for networked robots. Int. Journal of Computer Science and Network Security, 9(5), 139-148.

[16] Mohamed, N., Al-Jaroodi, J. and Jawhar, I. (2008) Middleware for robotics: A survey. 2008 IEEE Conference on Robotics, Automation and Mechatronics.

[17] Balasubramanian, N., Balasubramanian, A. and Venkataramani, A. (2009) Energy consumption in mobile phones: a measurement study and implications for network

applications. Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference.
[18] Francisco, R. (2014) Flexible, multiplatform middleware for wireless sensor and actuator networks. MsC Thesis, IST-UTL.
[19] Arsenio, A. (2005) Development of neural mechanisms for machine learning. International journal of neural systems, 15 (1), 41-54.