Intelligent Multi-platform Middleware for Wireless Sensor and Actuator Networks

Rui Francisco[1,3] and Artur Arsenio[2,3](✉)

[1] Computer Science Department, Universidade de Lisboa, Lisbon, Portugal
rui.francisco@ydreamsrobotics.com
[2] Computer Science Department, Universidade da Beira Interior, Covilhã, Portugal
artur.arsenio@ydreamsrobotics.com
[3] YDreams Robotics, Lisbon, SA, Portugal

Abstract. Wireless Sensor and Actuator Networks (WSAN), composed by small sensing nodes for acquisition, collection and analysis of data, are often employed for communication between Internet objects. However the WSAN have some problems such as sensors' energetic consumption and CPU load. The massive storage capacity, large processing speeds and the rapid elasticity makes Cloud Computing a very good solution to these problems. To efficiently manage devices' resources, and achieve efficient communication with various platforms (cloud, mobile), this paper proposes a middleware that allows flows of AI applications' execution to be transferred between a device and the cloud.

Keywords: Internet of things · Wireless sensor and actuator networks · Cloud computing · Middleware · State machines

1 Introduction

It is expect that in a few years our lives become more dependent of internet objects connected by WSAN in 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 computation power, etc. Due to these problems, the network node can crash due to lack of sufficient resources to perform, and jeopardize the smooth operation of the infrastructure. So, especially for demanding AI applications, internet objects using sensors and actuators require specific middleware for integrated operation with networked resources [1, 2].

Cloud computing provides attractive solutions for these issues [3]. 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 uses. Mostly important, cloud computing resources provide almost unlimited battery, storage, and computing power.

So, we need an efficient solution that monitors the WSAN node capability to execute operations, and communicates transparently with the cloud infrastructure.

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2015
R. Giaffreda et al. (Eds.): IoT360 2014, Part I, LNICST 150, pp. 16–22, 2015.
DOI: 10.1007/978-3-319-19656-5_3

2 Solution Architecture

The system consists of devices running applications (egg clients: cell phones, tablets, and computers, as shown in Fig. 1a) and the cloud that makes data processing and saves data. The communication protocols are TCP, UDP, SSH and HTTP Rest, and a publish/subscribe model for internal communications in 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 former 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 (see Fig. 1b).

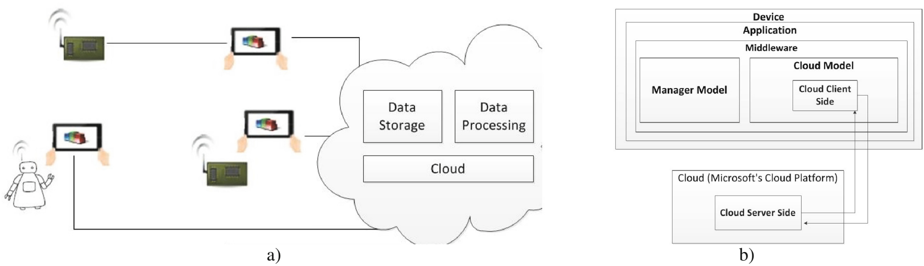


Fig. 1. (a) System Architecture and (b) its Components.

2.1 Manager Module

The management module aims to determine hardware components state (battery, CPU and memory), as well as the Wi-Fi 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 WiFi signal is strong) a certain execution will no longer be run on the device, being transferred to the cloud. Figure 2a 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 Wi-Fi signal 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 one minute.

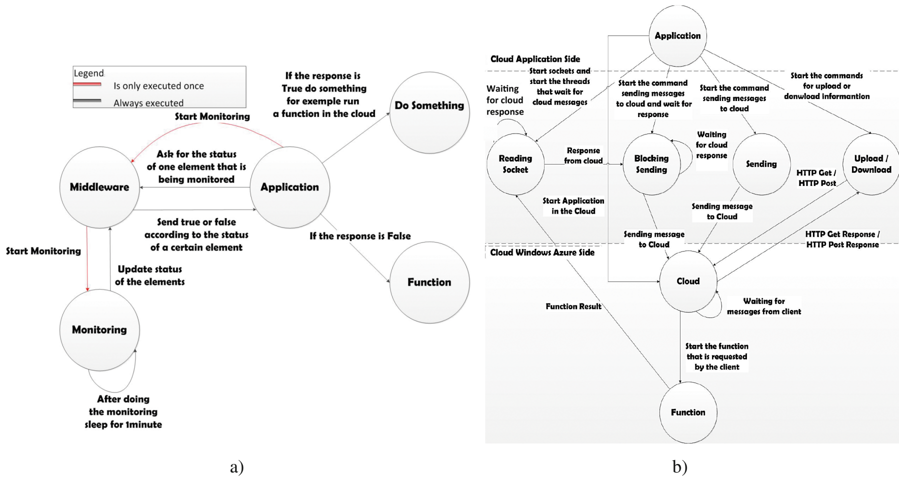


Fig. 2. (a) Management, and (b) Cloud, state machines.

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).

2.2 Cloud Model - Client

The cloud module state machine is shown in Fig. 2b. 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 operation of storage; or (iii) the information as requested by the get command; or (iv) error due to failure on obtaining 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

"Blocking Socket" 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 "Sending Blocking", otherwise the function ID and its result will be stored in the device memory until the program needs the result.

2.3 Cloud Model - Server

The Cloud Server side module 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 or the non-blocking function.

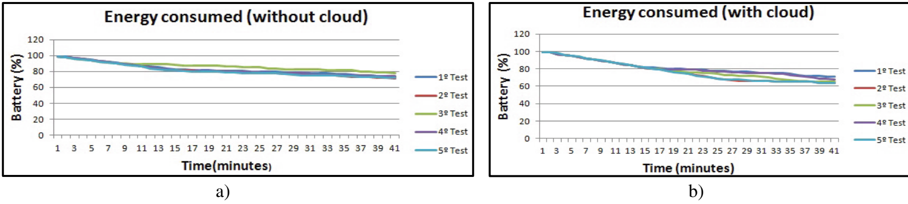
3 Experimental Results Assessment

The metrics were the Percentage of Energy consumed and CPU Load. It was used 1 BQ Edison with Android OS, and one virtual machine with one core and 1,75 GB of RAM. We applied OpenCV to build a face detection and tracking application, heavy in terms of CPU processing power, battery consumption and generated traffic, since it makes significant image processing. Two experimental setups were implemented:

1. The baseline: the application that detects/tracks the human face is the only running on the Android OS (no middleware).
2. Integration with the middleware: the same application only sends messages composed with frames, getting these from the camera, and sending these to the cloud, being the frames' analysis made in the cloud.

3.1 Energy Consumed

The experiments on the 2 aforementioned scenarios lasted 40 min and were repeated 5 times. As shown in Fig. 3. and Table 1, 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 %).



a) b)

Fig. 3. Energy Consumed (a) without using the cloud; and (b) using the cloud.

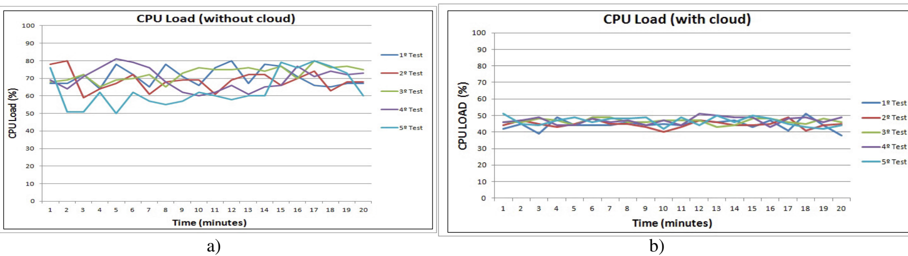
Table 1. Energy Consumed (as percentage of battery charge) and CPU Load

Experimental setup	Without Cloud	With Cloud
Average working battery charge (at the end)	73,6%	66,4%
Standard Deviation	2,70	3,04
Average of battery spent by the five tests	25,2%	32,6%
Standard Deviation	3,11	3,04
Average Of the CPU Load	68,98%	45,84%
Standard Deviation	3,53	1,15

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 [4].

3.2 CPU Load

To check potential middleware advantages in relation to CPU load, the “Face Detect/ Tracking” application was subjected to tests lasting 20 m and repeated five times. In Table 1 and Fig. 4, 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 (just grabs image frames on the device and sends them).



a) b)

Fig. 4. CPU Load (a) without, and (b) with, the presence of the middleware

Hence, in situations where applications compete for CPU time in processing constrained devices, this solution can bring very interesting benefits. The image processing algorithms requires a lot of CPU load to be executed, which may prevent simultaneously other applications to run properly. The same also happens with 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.

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 2, CPU load reaches saturation values for single tablet processing. But usage of cloud processing originates a significantly lower CPU load at the tablet. Hence the integration of the middleware may be beneficial to run reliably multiple applications 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.

Table 2. 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

4 Conclusions

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. 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, because it allows reducing the CPU load. This may lead to battery with extended autonomy. But most importantly, it avoids applications entering in blocking states due to lack of memory, and 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. More recently [5], we have successfully integrated the proposed middleware with a WSAN platform, OpenHAB, for smart home automation.

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

Acknowledgments. Work developed in the scope of the Monarch project: Multi-Robot Cognitive Systems Operating in Hospitals, FP7-ICT-2011-9-601033, supported by EU funds. Artur Arsenio was also supported by CMU-Portuguese program through Fundação para Ciência e Tecnologia, under Augmented Human Assistance project CMUP-ERI/HCI/0046/2013.

References

1. Kranz, M., Rusu, R., Maldonado, A., Beetz, M., Schmidth, A.: A player/stage system for context-aware intelligent environments. In: Proceedings of the System Support for Ubiquitous Computing Workshop (UbiSys), September 2006
2. Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.: ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software, vol. 3. No. 3.2 (2009)
3. Hunziker, D., Gajamohan, M., Waibel, M., D'Andrea, R.: Rapyuta: The roboearth cloud engine. In: IEEE International Conference on Robotics and Automation (ICRA) (2013)
4. Balasubramanian, N., Balasubramanian, A., Venkataramani, A.: Energy consumption in mobile phones: a measurement study and implications for network applications. In: Proceedings 9th ACM SIGCOMM Conference on Internet Measurement Conference (2009)
5. Francisco, R.: Flexible, Multi-platform Middleware for Wireless Sensor and Actuator Networks. MsC Thesis, IST (2014)