# Multi-Tenancy in Decentralised IoT

Sylvain Cherrier*, Zahra Movahedi*, Yacine M. Ghamri-Doudane†
* Université Paris-Est , Laboratoire d'Informatique Gaspard Monge (CNRS : UMR8049)
† L3i Lab, University of La Rochelle, La Rochelle, France.

***Abstract*—Since the Internet of Things (IoT) has become more and more important, new solutions should be proposed in order to adapt the specificities introduced by this interconnection of the physical world (*Sensors* and *Actuators*) and the public networks (The *Internet*). Some of these solutions use a Cloud approach. The amount of data collected by Things rises the interest of the Big-Data community. The main design chosen for the IoT is the centralisation of all data collected and a central treatment of these data. But another approach is to decentralise the data processing, in order to dramatically lighten the network and limit the exchange to a reduced set of semantic messages. This decentralised architecture has assets in term of confidentiality, data ownership and energy saving. But then, how to share things among users, and keep the control? If computing is done on each object, how a user can integrate public objects in its own application, as these objects are used by some other users? How to organise access to the sensors and actuators provided by these objects? This paper proposes an architecture that gives multi-tenant capability to IoT decentralised applications, in which users are using and sharing their objects. A generic architecture is described, and integrated in our IoT platform as an example.**

***Index Terms*—Multi-Tenancy; Internet of Things; Services Oriented Computing; Virtual machine**

## I. Introduction

The Internet of Things is the result of the mix of new capabilities added to everyday objects (*communication*, *computation*), the increase usage of sensors and actuators (*Wireless Sensor and Actuators Network*, WSAN), and the massive take-up and use of the Internet. As soon as they can be connected to the Internet, all these things send the data gathered from their close environment to a central point, where computing is done and actions can be decided. Then, this central point of decision can send an order to an actuator in order to react to the situation.

Some approaches have chosen a "*Service*" point of view. Instead of manipulating data, the Service Oriented Computing (SOC) in IoT represents each Thing (or group of Things) as a service that can be accessible through the Internet [7]. In this approach, two organisation are feasible [12]: A centralised one, in which a unique central point interacts with all the services, and a decentralised one, in which services interact with others, depending on their own requirements.

The first approach is called "Services Orchestration". The central point orchestrates the application, invoking services following the need of its control flow. The second approach is called "Services Choreography". In a Choreography, each node reacts to its environment and to its partners. No one has a complete view of the running application, each stakeholder follows its own control flow.

In any case (Data approach, or Service approach) a centralised IoT raises possible issues such as data confidentiality, application security, data ownership, and also network overload (due to the important number of nodes deployed on WSANs, and the strong constraints of these networks). Choreography reduces in a significant way the transmitted volume of information from and to each node [5]. As data are processed directly on the node, the only messages exchanged are semantic informations describing the result of the data computation. The local processing of data relieves the network usage (and often objects networks are very constrained in terms of bandwidth), the energy consumption (computing cost is cheaper than transmitting) and does not raise the issue of data ownership (data are not transmited). Even if Choreographed IoT programming languages have limited processing expressivity, they are sufficient to meet the needs of IoT applications.

This architecture is able to give users the opportunity to build versatile applications, adapted to theirs needs [7]. But there are an issue when these applications grow, particularly when users want to share objects with others, or use public objects (in a smart city scenario, this could be public display panels, a parking lot reservation system, etc). According to the Choreography paradigm, each user wants his own control flow, his own semantic messages and his own object' reactions, and/or integrate public object within his application (for example, to be informed of one available location in a parking lot, and automatically reserved it). In order to share objects among users and still give them the ability to integrate them in their own decentralised IoT applications, a multi-tenant architecture must be provided.

A multi-tenant Choreographed architecture should solve the following issues:

- How to handle multiple control flows on objects?
- How to define access and rights of each user over the device?
- How to handle conflicts, particularly concerning the actuators usages, in case of receiving contradictory orders?

This paper is organised as follow: Section II presents related works and some background for our solution. Section III describes the foundation of a multi-Tenant architecture. The integration of such an architecture on our IoT platform are given in Section IV. Finally, concluding remarks end this paper in Section V.

## II. Related work and Background

### A. Related work

This paper is about the extension of IoT architectures in order to add a multi-tenancy layer that supports users priority access for shared devices. Our IoT platform is based on a virtual machine, D-LITe [4], that allows constrained devices to realise Decentralized IoT applications. Several virtual machines for resource constrained devices have been implemented in the WSAN literature. In the following part, we review these virtual machines and analyse the support of multi-tenancy in their system.

The *Wukong project* [10] proposes a sensor profile framework based on an object-oriented programming to create virtualized sensor abstractions for low-level physical sensor devices. *WuKong* Applications are designed as program composition using an abstract form of data flow programming and event-driven control flow. In this sense, each device is virtualized by a *WuClass* with a set of properties that describe and allow access to the resource represented by the *WuClass*, and an update() function to implement the class' behaviour. In this approach, even if there is a control access mechanism for allowing/denying the access to an object. However the notion of priority in control access is absent.

*Maté* [8] is the second VM presented here. *Maté* is a domain specific bytecode interpreter running on TinyOS [9] for programming sensor networks. Programs are small scripts containing *Maté* VM instructions (capsules). The basic VM template includes the scheduler, concurrency manager, and capsule store. The scheduler executes runnable contexts in a FIFO round-robin method. The concurrency manager submits contexts to the scheduler based on whether they are ready to run and can safely access the shared resources they require. The capsule store manages capsule storage and loading; it propagates capsules through the network and notifies higher level components when it receives new code. *Maté* does not support priorities.

*Squawk* [13] is a JVM developped by Sun Microsystems that employs Split VM architecture to minimize resource and memory consumption.In this approach, tasks of the VM that require a large of resource consuming (such as class file loading and verification) are performed on the desktop. a preloaded and preverified file is generated and transferred then to the motes. *Squawk* implements a compact garbage collection, named Lisp 2, so that tasks are non-preemptible, which has implication for handling interrupts in a device driver written in Java. *TakaTuka* [2] is another example of JVM that is similar to the *Squawk* reducing the usage of RAM and flash using some optimization methods. The notion of priority is lacking in *Squawk* and *TakaTuka*.

*Darjeeling* [3] is a JVM that provides a rich set of (Java) features including light-weight threads, dynamic memory management (garbage collection), and exception handling, while at the same time being optimised for resource-poor targets. At the moment *Darjeeling* does not support priorities and threads are simply scheduled in a round-robin fashion.

*SwissQM* [11] is an another example of virtual machine for resource constrained devices that uses SQL query-like and XQuery query to program a sensor networks. *SwissQM* is composed of following components: an Operand Stack for the stack-based virtual machine, a Transmission Buffer that is used to store message data and can also serve as temporary storage for programs, a Synopsis (a data structure) that is used for data aggregation and for maintaining state over several invocations of the program and a sensor Interface. *SwissQM* does not provide priority mechanism. The sensor nodes running *SwissQM* form a tree topology rooted at a gateway node that provides access to the sensor network. *SwissQM* is able to execute up to six QM programs concurrently. No priority mechanism is performed in this approach.

*PyMite* [1] is a flyweight Python interpreter written from scratch to execute on 8-bit and larger microcontrollers with resources as limited as 64 KB of program memory (flash) and 4 KB of RAM. *PyMite* supports a subset of the Python 2.5 syntax and can execute a subset of the Python 2.5 bytecodes. *PyMite* can also be compiled, tested and executed on a desktop computer.

### B. D-LITe : a virtual machine for decentralized IoT applications

our own soution for IoT application is base on a virtual machine called *D-LITe* [4]. *D-LITe* is a lightweight RESTful virtual machine deployed on decentralised IoT devices (see Figure 1). *D-LITe* provides a universal access to the functionalities of heterogeneous devices. In *D-LITe*, a set of possible device basic functionalities (called *features*) such as button, switch, timer, led, etc. are firstly defined. Each feature is driven by a set of potential small algorithms that specify the device functionality and control its usage. These algorithms are expressed through "*Transducers*"[1] . A Transducer is a advanced form of Finite State Machines; i.e. each Thing is seen as a component with a current state, inputs, outputs, and transitions. Inputs and outputs are the message events exchanged by nodes through the network, and states are the node's reaction to received messages. A transition links two states. A transition can be triggered by an input. States, transitions, and inputs describes the algorithm to be executed. Transducers add an output to the well-known Finite State Machines description. The output is generated by the Transition when triggered. The transducer representations used in *D-LITe* (and their specificities) are described with SALT [6], a simple description language that limits bandwidth and memory consumption.

*D-LITe* uses exchanged messages between devices in order to compose devices interactions and to create IoT applications. To control the correctness of a composition, typical messages (called "*Interaction Patterns*") are exchanged. In this sense, two devices A and B could connect to each other if and only if the device A' output matches with the device B' input using

[1]These Transducers describe the control flow, the algorithm that defines the object behaviour

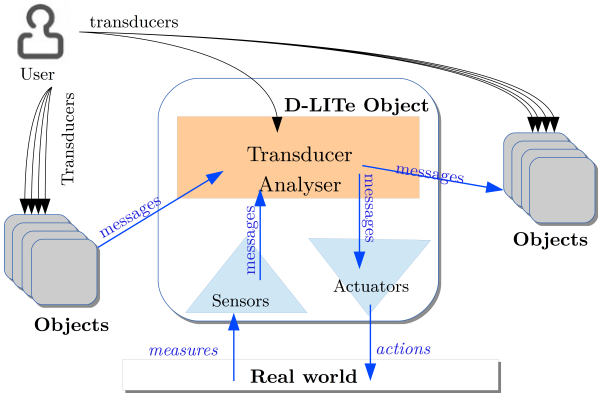


Figure 1. A D-LITe Object, running a Transducer, interacts with the real world and collaborates with other objects. The sensing and collaborating objects send messages to the Transducer Analyser **input**. These messages contain measures or actions to do. These messages trigger reactions from the transducer if a **Transition** is found for the current **state** and the **input**. In that case, the transducer changes its **state**, and sends an **output** message (if defined in the **transition**). This **output** message is for actuating functionality or for collaborating objects.

these *Interaction Patterns*. *D-LITe* allows an end-user to deploy a specific behaviour on each device. Each *D-LITe* enabled node contains a rules analyzer to execute the behaviour. *D-LITe* devices also have a messaging service to interact with each others using XMPP protocol. a standardized protocol for real time communication. This protocol offers instant messaging and presence management. Thus, the discovery of new devices is dynamic and their integration in the global structure is easy. The XMPP-REST (an extension we have created for *D-LITe*, allowing to send REST commands through XMPP) handles behaviours on each device using GET, PUT, DELETE, and POST methods. GET RESTful method helps to discover existing features supported by the device, PUT method deploys a behaviour on a device, DELETE method removes an existing behaviour from the device, and POST method exchanges messages between two device' behaviours. The implementation of REST approach within XMPP allows the use of the presence and chat mechanism offered by the instant messaging protocol, while this extension mimics the calls to a web service with the REST commands.

*D-LITe* follows an event-centric approach. An object (more precisely the behaviour that the virtual machine is currently running) reacts on received messages, as events. There are different kinds of events: **external** events that make an object collaborate with other objects, inside an IoT application; **hardware** events that are used for sensing and actuating the world; and **logical** events that define, alter, and test variables.

The transducer handles the received events and reacts depending the behaviour deployed. the incoming message comes from the **hardware** (a data has been gathered), or an **external** message from another Object (the Choreography). If a transition is triggered by this event (it match the event and the current state), its output is sends it to the **hardware** in order to actuate it, or/and to other nodes (in a case of an **external** message), depending of its type (see [6] for more details, and the use of variables).

## III. Multi-tenant architecture overview

To be able to handle multiples accesses on a single device, the following points should be defined and analysed:

### A. Impacts of multiple control over an IoT device

Since each IoT device can be a sensor or/and an actuator, a multi-Tenant architecture has to cope with these two cases.

Sensing in a multi-tenancy architecture is not a very complex issue to solve. Depending on the access rules, the gathered data are accessible or not. The flow control will read (or not) the data.

Actuating is more complex. In the IoT, every (public, or personal and shared) device must be kept under its owner's control. For example, in smart cities, a public display panel, or a parking lot, are accessible to everyone, but security agent or public services have a priority access (police, hospital, etc). In smart building domain, a fully operational desk (with computer, phone etc) can be shared among several users, but should be reserved to a privileged employee.

The multi-tenant architecture needs to set a shared functionality under an access control mechanism. In this sense, an access rights list for each type user should be provided. Because of IoT devices versatility, the handled functionality must be correctly defined, according to the user's usage. For example, the physical access control to a specific room can be prioritised. In some cases, the usual user of the room is able to allow other employees to access to his room. But he must get a priority access to his own room when he is having confidential meetings. In some other cases, an actuator of the same kind can be used to give an employee the ability to limit accesses to his room, while the security agents have the total control over it. If they need it, they can force the opening of the door.

A multi-tenant solution for the IoT should offer to set which object' functionalities are under control, and exactly which commands have priority for each functionality. For the last example above, we define that "opening" is the under-control action for this door. Then, we can describe the access rules: Security agents have priority on this action, then the employee (owner of the room) has a lower priority, and finally the others.

### B. Impacts of multi-tenant devices over the control flow

Our approach for Multi-Tenancy in the IoT is closed to the access control used in operating systems (such as Linux, Windows, etc). As incoming orders are not predictable, and can be contradictory, the multi-tenancy system uses blocking accesses. On one hand, the blocking mechanism controls the access to a action or a data. On the other hand, when a conflict is detected, the user that has the lower priority is informed that the order has been refused. We propose to send notifications to the device handler, so that a programmer could handle each conflict, and react to an "access denied".

IoT devices have two major capabilities: Sensing the physical world, and/or actuating it. Multi-Tenancy conflicts have different impacts for these two categories:

For the actuating category, there is 3 cases:

- First case, a user wants to trigger an action. The functionality is not under control, or not shared, or the user has the best priority. In that case, his control flow takes control of the functionality, and uses it as long as it needs.
- Second case, a user wants to trigger an action, but this action is already in use by another control flow with a best priority. This is a blocking case. In that case, the control flow is blocked and waits for the functionality to be freed. It is also possible to execute an alternative in that case, in order to bypass the blocking case by doing something else. A specific "*accessDenied*" event is thrown to the control flow, and the programmer can catch it and then describe what to do.
- The last case appears when a control flow has taken the control of a functionality, and then another control flow (with better priority) asks for it. In that case, the prior control flow loses its access, and the access is granted to the second control flow. A specific "*controlLost*" event is thrown to the control flow that has lost the access (because of its lesser priority). It is possible to handle this event, in order to describe an alternative control flow when losing the access to a functionality while processing.

For the sensing category, the control access grants or denies access to the control flow, as defined by its rules. According to the rules, the data is (or not) accessible, and there is no more impact. For example, if the control flow of a given user asks for humidity data but has no access to this sensor, it will be blocked. In our approach, sensing a data is an incoming event. If a transducer has no right to access the data, it means that it will never receive the corresponding event. In facts, if this data is important for the control flow, we can imagine that the chosen control flow should not be executed on this device because of the user's low priority (i.e. in operating system, when a not priority user tries a forbidden action, he can be blocked).

The Figure 2 represents a control flow with alternatives as it tries to lock a door. The different events (*A* and *B*) are the main flow. *Event A* leads to a door locking, then *event B* unlocks it (the second part of each transition is the output message of the transducer. In D-LITe, they are actuating orders for hardware, or messages for other object). But if a priority conflict results in an access denied, an alternative control flow ("case 2") is proposed (triggered by "*accessDenied*" event). This solution gives the opportunity to avoid the lock by giving alternate tasks to do. An access lost lately in the process is handled (see Figure 2) by the "*controlLost*" event catch in the "case 3" branch of the control flow.

### C. Security Layer

In order to realise an access control for sensing and actuating capabilities of the object while it runs several control flows, a security layer is installed between the hardware drivers and the process manager. In the case of our project, we have added it between the operating system and the transducers analyser.

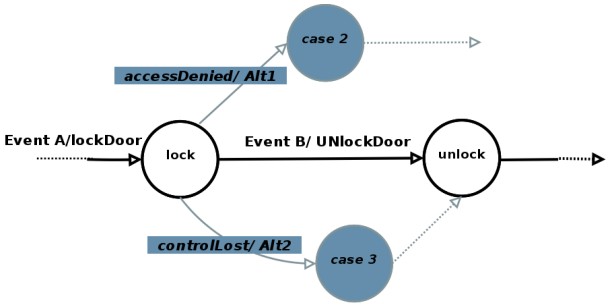


Figure 2. The control flow running in the object tries to lock a door. The main flow receives an *eventA*, then tries to lock the door. If it succeeds, it waits for *eventB* to unlock, and then follow this main control flow. In the case the access to the door lock is not granted, the programmer has described an alternative action (alt1), that will then follow its own logic. If access to the door lock is given by the security manager, but that an higher-priority user request the door usage before our *eventB* frees the access, an alternative is described ("controlLost" event). The other control flow can have its own actions, and then goes back to the main control flow.

The security layer executes access controls each time any transducer sends or receives an event to/from the hardware.

From outside the object, the access to the security layer is cyphered. An object is owned by a user and must be controlled only by him. The owner defines remotely for each functionality the orders' priority. As seen above, for a given functionality, each user may prefer to control a specific order (for the smart building example, *opening the door* if the user aims to prioritise security, or *closing the door* if confidentiality is the main concern).

Priority levels are:

1) private: reserved to the owner
2) priority: from 1 (the higher) to 4 (the lower). The owner can set the priority for each user, each functionality and each order.
3) free: total access to the functionality.

Once the owner has defined his priority access list, all users accessing the device can deploy a transducer (a control flow) on it, and every access to the hardware will be evaluate according to the access list.

## IV. Implementation

This section describes the multi-tenant extension to our platform for the IoT. A Security layer is added to every object, placed between the Transducer analyser and the operating system (see Figure 3).

### A. Security service

The deployment of transducer is unchanged. An authenticated user (D-LITe use XMPP, but this is not mandatory) accesses to an object through the network. He uses the PUT order (XMPP-REST in our case, but this can be modified if needed) to deploy a new transducer. Then, objects exchange events through the XMPP pub-sub mechanism (See Figure 3).

When a new user accesses an object (authorized because the owner has shared this object to him, or because it is a public object such as in the smart city scenario described above),

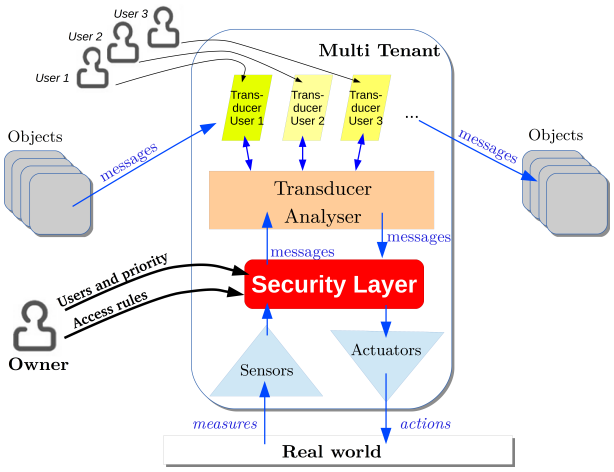


Figure 3. The Security layer controls all access to and from the transducer analyser. The object's owner has a secured access to the Security layer, in order to set rules and users' priority. Then, each time a transducer (installed by any user on this object) tries to access a actuating functionality or wait for an incoming message from a sensor, the security layer checks his priority. It can then grant/deny access and forward/destroy the message.

a new REST endpoint is created for him. When this user DELETE his transducer, the REST endpoint is removed.

Accessing the security service is done through a specific REST endpoint as illustrated in Figure 3, and reserved to the object owner (authenticated by his XMPP credentials). The owner will DELETE rules and PUT new access rules describing his settings.

Table I shows an example of a classification defined by the object's owner. Each user of this Object is categorised and his priority is set. For example, Alice and Carol have an higher priority than Denis. These rules are defined by the object's owner, can be modified at any time, and are used by the security layer in order to grant or deny access to each functionality. The access rules are defined in Table II for object' functionalities. Each authorised order is indicated for each priority level.

According to Table I and Table II, locking the door can be invoked by all users of priority 1 and 2. Carol and Alice can lock and unlock the door while Denis can not. Denis has the right to close the door, but Alice and Carol overtake Denis' order. The door' displayer gives an example of a free access to a functionality. Any user can display a message on the door. Only high priority users (*priority 1*) can clear the displayer, while other users can only add new messages. *Level 3* and *4* users can also display messages, but they have no higher priority than free users: the cells for this functionality and these priority levels (*3* and *4*) are empty, so they can't overtake the access from the free users. On the contrary, *free* users can't use the door lock (their cell is empty), so Denis can do actions (close the door) that *free* users can't do.

Table I
USERS' LIST AND PRIORITY

| Owner | Priority 1 | Priority 2 | Priority 3 | Priority 4 |
|---|---|---|---|---|
| Bob | — | Alice, Carol | Denis | — |

This object is owned by Bob. Bob has given priority 2 to Alice and Carol, and priority 3 to Denis. All others will have no access at all, except for free functionalities.

Table II
ACCESS RULES

| Function | Level 1 | Level 2 | Level 3 | Level 4 | Free |
|---|---|---|---|---|---|
| Door lock | close, lock | close,lock | close | — | — |
| Displayer | clear, display | display | — | — | display |

The door lock is under the control of the security layer. Depending of the transducer's owner, the priority is set for the different actions (*lock* the door, or *display* a message in the door' displayer) described for each level. Users priority is defined in Table I.

### B. SALT

*SALT* [6] is our modified transducer-based language used to describe the control flow running on an object. *SALT* is now version 2 in order to take into account the new events that may occur because of the multi-tenancy extension. Legacy transducers, written in *SALT* version 1 (as defined in previous papers), must still be executable on the new version of our virtual machine *D-LITe*. The orders to access functionalities are still the same. Legacy *SALT* transducers can access orders under the control of the new access control Security Service. But in the case of a blocking order, the legacy transducer stays stuck until the higher-priority user release the functionality.

In *SALT* version 2, the blocking state can be avoided by the introduction of two new events. The "*accessDenied*" event is generated when a transducer invokes an order protected by the access control list and already in use by a higher-priority user. The transducer can describe what to do when receiving this event (see Figure 2), so the control flow will not be blocked in this state. The "*ControlLost*" event is automatically generated by the security layer when an higher-priority user takes control of a functionality that was previously under the control of this transducer. For example, an employee has closed his door, but the fire alarm (with higher priority) starts an evacuation exercise. The "*ControlLost*" event describes the alternative control flow to be executed by the Transducer.

These two new events give our virtual machine an upward compatibility. New transducers handle blocking case *a-priori* (when the access is denied to the control flow). Losing the control of a functionality is proposed as an *a-posteriori* handler, in order to inform the user of that lost, for example. The multi-tenancy extension has a limited impact over the language, as it respects the previous event-centric approach. The addition of a limited number of new events gives a simple solution to integrate new and improved transducers, taking into account the Security layer responses.

### C. Hardware interactions with the Virtual Machine

Extending our Virtual Machine *D-LITe* [4] to cope with Multi-tenant has some impacts over it. First, new users must be able to deploy their transducers in the object. The access control is done through XMPP. In facts, the user get an access to an object through its pub-sub XMPP account. That access

is given by the XMPP server, following the identification of the requester and the friendship with the object's owner. Each time a new user gets an access to the object, a new endpoint is created for him. Then, he can deploy his transducer, and the transducer can be discovered by any other object of his application.

Incoming messages (the *input alphabet* of the transducer) are hardware sensing message or event sent by other objects of the whole application. These other objects send event trough the pub-sub XMPP account, so they arrive on the right endpoint. For hardware incoming messages, they are sent by the security service to all the transducers that are authorized to access them. Depending on the transducer running at the moment, action are executed if a transition for this event has been set for the current state of the transducer.

*Output messages* are generated by the transducer, and are in destination of the hardware (for actuation) or to other objects (to make them react). Hardware messages are controlled by the security layer before their real transmission to the hardware. If they are not authorised (because of the control access rules, or because the functionality is currently used by a higher-priority user), the order is blocked. A "*accessDenied*" event is sent to the input of the transducer, in case it has a transition to handle blocking case. If the access is granted, then the security service stores the user's id and the order currently controlling the functionality. This is useful when the transducer frees the resource, or if another access is requested from another transducer running in the same object. In that case, the rules of the order, the priority of each user are compared, and a decision is taken by the security layer. If our current transducer loses its access, a "*controlLost*" event is sent to its input. The user's id and the order of the higher-priority user are stored in the security layer.

## V. CONCLUSION

In our decentralised architecture, users or organisations can propose an access to their objects and give their friends, citizens or employees the ability to integrate shared things in their own applications. Sharing Things in such an architecture leads to conflicts between all the users that interact with the hardware. In this paper, we propose an architecture that is able to solve these issues. Using the user credentials, a priority definition, and an access list, each object grant or deny access to the different users in regard of the owner definition. In case of conflict, events are sent and help to program alternatives for each user. This paper describes our modified architecture for this purpose, with an upward compatibility with the previous work. In future works, we will concentrate on the different ports of the virtual machine and the impact in term of memory and processing overhead.

## REFERENCES


[1] Python-on-a-chip. https://code.google.com/p/python-on-a-chip/.

[2] F. Aslam, C. Schindelhauer, G. Ernst, D. Spyra, J. Meyer, and M. Zalloom. Introducing takatuka: A java virtualmachine for motes. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 399–400, New York, NY, USA, 2008. ACM.

[3] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 169–182, New York, NY, USA, 2009. ACM.

[4] S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel. D-lite : Distributed logic for internet of things services. In *IEEE International Conferences Internet of Things (iThings 2011)*, pages 16–24. IEEE, 2011.

[5] S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel. Services Collaboration in Wireless Sensor and Actuator Networks: Orchestration versus Choreography. In *17th IEEE Symposium on Computers and Communications (ISCC'12)*, page 8 pp, Cappadocia, Turquie, July 2012.

[6] S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel. SALT: a simple application logic description using transducers for internet of things. In *IEEE International Conference on Communications - Communication Software and Services Symposium (ICC'13 CSS)*, Budapest, Hungary, June 2013.

[7] S. Cherrier and Y. M. Ghamri-Doudane. The "object-as-a-service" paradigm. In *Global Information Infrastructure and Networking Symposium (GIIS), 2014*, pages 1–7. IEEE, 2014.

[8] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *ACM Sigplan Notices*, volume 37, pages 85–95. ACM, 2002.

[9] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[10] K.-J. Lin, N. Reijers, Y.-C. Wang, C.-S. Shih, and J. Y. Hsu. Building smart m2m applications using the wukong profile framework. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 1175–1180. IEEE, 2013.

[11] R. Müller, G. Alonso, and D. Kossmann. A virtual machine for sensor networks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 145–158, New York, NY, USA, 2007. ACM.

[12] C. Peltz. Web services orchestration and choreography. *Computer*, pages 46–52, 2003.

[13] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java™ on the bare metal of wireless sensor devices: The squawk java virtual machine. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 78–88, New York, NY, USA, 2006. ACM.