Towards a formal definition of the Mobile Cloud

Journal:	*Transactions on Cloud Computing*
Manuscript ID	TCC-2016-04-0147
Manuscript Type:	Regular paper
Keywords:	D.3.1 Formal Definitions and Theory < D.3 Programming Languages < D Software/Software Engineering, D.2.7.k Maintenance process < D.2.7 Distribution, Maintenance, and Enhancement < D.2 Software Engineering < D Software/Software, C.2.8 Mobile Computing < C.2 Communication/Networking and Information Technology < C Computer Systems Organization, C.2.8.c Mobile communication systems < C.2.8 Mobile Computing < C.2 Communication/Networking and Information Technology < C Comp, C.2.4 Distributed Systems < C.2 Communication/Networking and Information Technology < C Computer Systems Organization

Towards a formal definition of the Mobile Cloud

Virtual Device Representation

Charif Mahmoudi, Fabrice Mourlin, Abdella Battou
Advanced Network Technologies Division
National Institute of Standards and Technology
Gaithersburg, Maryland, USA
{charif.mahmoudi, fabrice.mourlin, abdella.battou}@nist.gov

Abstract—In the emerging area of mobile cloud computing, two tendencies come together to compose some of the major pillars. On one hand we have the virtualization trends that are affecting the data centers and on the other hand we have mobile devices which proved to be the most effective and convenient tools in human life. This emerging area is then changing the game in terms of mobility of workspaces and the interaction with the connected devices and sensors. This paper provides a formal specification using the π-calculus which defines the virtual device representation in the mobile cloud computing. In addition, we describe a new vision of composite virtual devices that can take advantage of all devices, sensors, and actuators available on the network to build a composite virtual device. We address application offloading and virtual devices networking on mobile clouds. Our architectural model comes from the Cloudlet based system. We will introduce our proposed specifications, architecture, along with case studies showing the structural congruence between a locally executed application and an offloaded version of the same application.

Keywords— formal definition; migration; mobile; mobile cloud computing; offloading; virtualization; virtual device representation

I. Introduction

Mobile devices are increasingly having an essential usage in human life as the most effective and convenient communication tools. The unbounded time and place usage introduced by those devices allows mobile users to accumulate a rich experience of various services and applications. The execution of those services is not limited to the mobile device itself, more and more applications use nowadays remote servers via wireless networks to interact with services. Architectures based on the n-tiers computing have become a powerful trend in the development of IT technology as well as in the commerce and industry fields on mobile computing [1]. Such systems can accept any (finite) number of layers (or tiers). Where each tier like presentation, application processing, and data management functions is physically separated from the others.

However, mobile devices have considerable hardware limitations. Mobile computing faces many challenges in attempting to provide the various applications living on a single device with limited resources such as battery, storage, and bandwidth. Communication challenges like mobility and security arise too. Those challenges motivate the delegation of the resource-consuming application modules to remote servers using the cloud service platforms. Google offers one of the major solutions called AppEngine [2] allowing developpers who don't need to have any previous understanding or knowledge of cloud technology infrastructure to deploy services and use the cloud. This plateform executes the deployed services and expose them as a remote service. This approach is used to delegate massive computation pieces of the mobile software on the cloud infrastructure.

Current mobile cloud architectures are based on cloud computing abstractions (IaaS, PaaS and SaaS) [3]. This architecture adressses the virtualization and distribution of the deployed services, however, the mobility aspect is not designed for the nomadic usage of mobile devices. The lack of specific formalism to address mobile virtualisation contribute to the heterogeineity of the actual solutions. Indeed, the virtualisation of devices and services is following the server architectures that are not suitable for the mobile plateforms because of the heterogeineity of the hardware architectures and the availaibile ressources. Also, the deployed services artefacts are a classical web services. There is no specific representation that makes abstraction of the application offloading and the location management. In such solutions, the remote services implementations depend at the same time of the cloud plateform and the devices capability. In term of development, this constraint implies that the software component devlopped as a remote service cannot be reused in the client side. In addition, interfaces that expose the same services may be differant from an implementation to an other.

Our contribution aims to define an additional abstraction level on the cloud to specify a structure that represents mobile devices It enables a common interface to communicate with differents devices like mobile devices, sensors, and actuators. Communications addressed to the devices are translated to the specific protocols by this representation. And the responses are stored on a cache which is the virtual state of the device. This representation acts also as a "mobile-friendly" platform within the cloud. Indeed, the representation is built on emulation capabilities that offer a compiliant environment with the phisycal device on which the representation is associaeted.

We distinguish three kinds of representations depending on their association (or not) with the phisycal devices. The first type of representation is this associated with simple sensors or actuators. They are the simplest forms for the representation where they act as a cached proxy with a common interface. The second type is the representation associated with the mobile devices. This representation offer offloading capabilities, and keeps in their cache the state of the differents sensors and actuators availaible on the mobile device. We consider this second type of representation as a composition of ressources associated with a mobile device and it is not the exact image of the device, it can be used as an extension to the ressources available locally on the device. The third type of representations has no direct association with a phisycal device. It is a composition of multiple representations. We define this representation as a composition of resources distributed over the network which transform the mobile device into a sort of “super device” by eliminating the physical limitation. So the composite representation adds smartness to the devices by enabling the composition of multiple device in a smooth way.

This paper stresses, in section II, the various faced challenges during virtualization and how they are addressed in related work. We describe the existing cloud techniques that are useful for mobile cloud computing and presents the formalism efforts that are related to the different virtualization aspects. We end this section with an introduction to the π-calculus which is the formalism used for our definition . In section III, we expose our definition of the Virtual Device Representation(VDR) using a formal language and how we address the orchestration and the networking for these VDRs. In section IV, we will expose our proposed architecture for Mobile Cloud Computing (MCC) and the approach that we use in order to have a high performance mobile cloud network. The last section describes a case study for an MCC platform highlighting the structural congruence between the system when a mobile application is running directly on the device and when this same application is offloaded to a mobile cloud.

II. Related Work

A. Mobile Device Challenges

The role of mobile devices has expanded into the modern workplace. Workplaces are not limited to the office and the warehouse anymore. They have expanded to include airport terminals, loading docks and delivery trucks, physician waiting rooms, and even playing fields and family gatherings. Mobile devices have erased workplace boundaries, and as a result, employees can connect with their corporate networks almost anytime, anywhere.

The mobile technology, which is easing access to business data and applications and providing various means of communications, will continue to be embraced by users. For IT, however, mobile technology and the unprecedented pace of change in the mobile domain will generate new IT management challenges. Indeed, as mobile innovation continues, machine-to-machine (M2M) connectivity (or Internet of Things) will further accelerate mobile opportunity [4] and transform how people, enterprises, and governments interact with the many aspects of modern life.

Several trends - and the way companies react to them - will create challenges for IT, as organizations attempt to exercise some control over devices that are not necessarily designed to be secure and manageable. With careful planning and an understanding of best practices and Mobile Device Management (MDM) [5] options, IT can go a long way toward meeting those challenges. With a well-implemented MDM strategy, enterprises can enforce corporate security policies without stifling user productivity.

B. Cloud and Virtualization

The virtualization is used for abstracting the Operating System (OS) and applications from the physical hardware to build a more cost-efficient, agile and simplified server environment. There are two types of virtualization and many major uses of virtualization.

1) Virtualization types

Two kinds of virtualization are used to simulate the machine hardware and allows the execution of a guest OS.

a) Emulation:

Where VM emulates (or simulates) complete hardware if the unmodified guest OS for a different PC cannot be run. There is some hypervisors specialized on “emulation” like Bochs, VirtualPC for Mac and Qemu [6].

b) Full/Native:

Where VM simulates “enough” hardware to allow an unmodified guest OS to be run in isolation. This virtualization type requires that the same hardware CPU if used by the VM and the hypervisor. This type is supported also by many hypervisor like, VMWare Workstation [7] and Microsoft Hyper-V [8].

2) Virtualization usage

We can talk about server virtualization and desktop virtualization. Given that virtualization is used to create a virtual version of a device or resource, many computing devices can be virtualized, including a server, desktop, storage, network, and operating systems. These are the two most common virtualization technologies:

1. Server virtualization: Is defined as the re-composition of a physical server infrastructure into virtual servers.
2. Desktop (or client) virtualization: Is defined as a technology that is used to separate a computer desktop environment from the physical computer.

By using server virtualization, multiple VM instances containing operating systems can run on a single physical server or a single VM can use hardware from multiple physical servers, each with access to the underlying server's computing resources. The virtualization is used to address the resource waste caused by the fact that the host servers operate at less than 15 percent of capacity, leading to server sprawl and complexity. According to VMware statistics [9], virtualization can deliver 80 percent greater utilization of resources on the server and 10:1 or better server consolidation ratio.

Existing efforts aim formalizing the cloud services interactions [10] and orchestration [11]. However, those efforts do not address the virtualization aspect of such cloud systems.

C. Network Virtualization

The objective of this kind of virtualization - considered as a subset of server virtualization - is to provide an abstraction of the networking resources into a logical model that has the same behavior as the physical resources. The virtual networking resources are divided in two categories: first is the physical resources virtualization like vRouter (Router) and vSwitch (Switch), the second is the resources appliances like FWaS (Firewall) and LBaS (Load balancer). This network virtualization approach is called Network Functions Virtualization (NFV) [12] that aims consolidate and deliver the networking components needed to support a fully virtualized infrastructure and shared by multiple tenants in a secure and isolated manner.

In parallel with the pragmatic work on the networking, there are many existing efforts on the definition of formalism dedicated to networking. U. Montanari and M. Sammartino have worked on a proper extension [13] of the π-calculus. The resulting process calculi provide both an interleaving and a concurrent networking oriented semantics. A. Singh et al have also worked on an extension called ω-calculus [14] that formally modeling and reasoning about mobile ad hoc wireless networks. These works focus on the reasoning and the verification of the networking protocols and does not address the virtualization aspect of the networking. This lack of networking virtualization formalism motivates also our high-level definition of network virtualization in the next section.

D. *The π-calculus*

In the mobile cloud, the devices, services, and their compositions are executed in a concurrent and distributed environment. The representations of those devices have to follow a model of computation of such concurrent systems. The π-calculus [15] is a process algebra that offers a syntax for representing processes (like devices representations or services), parallel composition of processes, synchronous communication between processes through channels (or names), creation of fresh channels, replication of processes, and nondeterminism (or choice points). It provides primitives for describing and analyzing a distributed system that are using process migration between peers, process interaction via dynamic channels, and private channel communication.

Table 1 π-calculus constructs

Composition	Denotation
$P\|Q$	A process composed of P and Q running in parallel
$P+Q$	A process that behaves like either P or Q. (nondeterministic choice)
$a(x).P$	A process that waits to read a value x from the channel a and then, having received it, behaves like P.
$\bar{a}\langle x\rangle.P$	A process that first waits to send the value x along the channel a and then, after x has been accepted by some input process, behaves like P.
$(\nu\ a)P$	Ensures that a is a fresh channel (free name) in P. (the ν is pronounced "new.")
$!P$	An infinite number of copies of P, all running in parallel.
$(\lambda\ a)P$	An abstraction a defined on the process P.
$P \frown Q$	A process P chained to a process Q over a binary abstraction. (Chaining Combinator)
$\vec{a}$	A vector of names where magnitude (or size) is noted $\|a\|$
$\emptyset$	A process that does nothing, used to terminate a process.
τ	An internal communication (non-observable operation)

The two central concepts of the π-calculus definition [16] are the processes and channels. Indeed, a process is an abstraction of a thread of control that may represent a device on the cloud, a cloud service like network functions, or business services that may migrate from a mobile device to a mobile device representation on the cloud. A channel is an abstraction of the communication link between two processes. It may represent a web service connection between the mobile device and it virtual representation as a synchronization channel. Channels may also represent the sending and receiving messages interaction between a process local to the mobile device and another remote process on the cloud.

The π-calculus brings the notion of structural congruence that is defined as the least equivalence relation preserved by the process constructs. The structural congruence can be obtained by event reduction semantics or labelled semantics. All concurrent behavior that needs to define a representation of our vision of the mobile cloud would have to be written in terms of constructs Table 1 and rules in Table 2 where P and Q denote processes.

Table 2 π-calculus rules

$P \equiv Q$	A process Q obtained from P by renaming one or more bound names in P. (Structural Congruence)
$P \rightarrow P'$	A process P' obtained after a computation step performed by P. (Reduction Relation)
$P \xrightarrow{\alpha} P'$	A process P' obtained after an action α (eg: $\bar{a}\langle x\rangle, a(x), \tau$) performed by P. (Labelled Transition)

III. Virtual Device Representation

In our approach, the VDR aims to address the mobile cloud computing virtualization paradigm. We use the acronym VDR as a generic label for all types of representations within the mobile cloud. However, there are many differences between the representations of a simple sensor, a mobile device, and a composite device. On one hand, the virtualization capability differs form a VDR type to another. As example, the sensor virtual representation does not have any offloading capability; however, the offloading of application is central to the mobile device representation. On the second hand, the connector to the devices differs also from one VDR type to another. The sensors could be connected to their virtual representation using some specific standards like 802.15.4 [17] (ZigBee, UWA, …) that offers an acceptable transfer rate to synchronize the state of the sensor with its virtual representation. The mobile device, for its part, needs to use technologies with more transfer rate. This allows transferring the needed data application for the application offloading to the mobile cloud. The composite device representation does not have any direct connection to a physical device, this type of representation communicates with the physical devices through their virtual representation using an event bus. The **Error! Reference source not found.** gives an overview of the three different types of VDRs identified.

Because of their differences, the three types of VDRs have specific formal definition defined in (3), (6), and (9). They have also specific roles within the mobile cloud:

1. Sensor VDR (SVDR): it represents a physical sensor or actuator within to the mobile cloud, like a LED or a pressure sensor.
2. “Mobile” Device VDR (DVDR): it represents a physical mobile device within to the mobile cloud, like a smartphone.
3. Composite VDR (CVDR): it represents a composition of SVDR, DVDR, and mobile cloud resources. Like the aggregation of all cameras used in a given conference room.

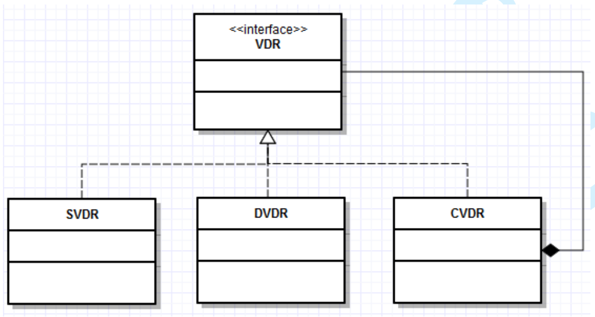

Figure 2 VDR composition relations

The compositions of the VDRs have to respect a pattern as the SVDR and the DVDR cannot be composite. Only the CVDR can be at the same time a composite and a component. The Figure 2 illustrates the composition relations between the different VDRs types and there generalization.

In this section, we present the different aspects turning around the VDR by giving our definition of the VDR, a formal definition using the Higher-Order π-Calculus (HOπC) [18], stressing the orchestration mechanism for the VDRs, and the networking aspect. Our choice for the HOπC is motivated by the need of expressing the mobility of the VDRs in the mobile cloud, also the mobility of mobile applications between the physical devices to its VDRs. In our definition, we do not use the network related extensions of the π-calculus for two reasons: first, those extensions do not address the higher-order paradigm, next, there are designed to express networking protocols not the virtualization-oriented communication.

A. *Definition*

VDR is defined as composition of resources (CPU, RAM, and Storage), mobile devices, and sensors. A software

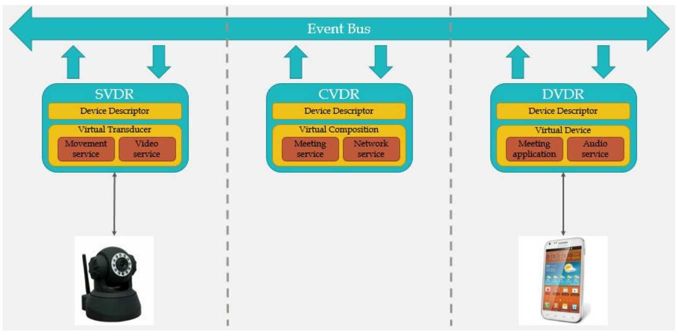

Figure 1 VDRs connections

composite component provides emulation of the behavior of the physical hardware that it represents.

A VDR is a small VM instance used in cloud computing, typically hosting a mobile OS and exposing management services that emulate a display screen and/or a keyboard. As same as the physical handheld computing device that it represents, it can run various types of mobile applications (known as apps) and it has a network connection.

A VDR cloud has two kinds of associations to the physical device. On one hand, there is the direct association to a mobile device or sensor. In this case, a 1..1 cardinality regulates these interactions as is the case for SVDR and DVDR illustrated in Figure 2. We call this first kind “Emulated VDR”. On the other hand, a VDR could not have a direct association to any physical device. In this case, it is called “Native VDR”. CVDR in Figure 2 illustrates a VDR with no direct association with a physical device. However, this kind of VDR could have a 0..* cardinality with VDRs, 0..* cardinality in an indirect association hardware device through their VDRs. This category of VDR is called “Native VDR”.

B. *Formal Specification*

The VDR operates according to an event driven architecture. Every interaction is initiated by a message sent from a driver (further to hardware sensing activity) or a service call. We define an event vector representing all interfaces of a VDR. This event vector contains channels that are used to exchange messages. The event bus is central to our formal model.

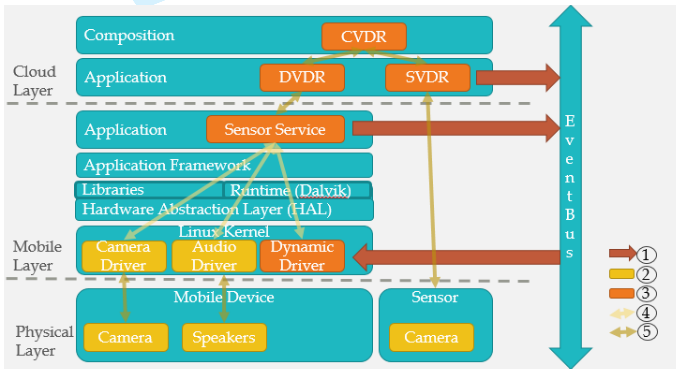

Figure 3 Role of the Event Bus (1: Event propagation, 2: Standards components, 3: Proposed components, 4: Communication via application layers, 5: Direct communication)

As illustrated in Figure 3, the event bus is the component that allows the propagation of the sensing events in a distributed system. Indeed, the mobile devices [19] [20] [21] architectures are not designed to enable a hot integration of remote devices. To enable this kind of integration, we have added two components in the mobile layer celled Sensor Service and Dynamic Driver. These components aim to propagate mobile device embedded sensors events, retrieve the external sensing events, and integrates those events on the mobile Operating System (OS) in the kernel level.

The event vector does not have a static formal definition. It is defined as a vector that contains as many channels as sensors connected to the system. The events are added to the vector on the fly while devices are composed. An example of the event vector is illustrated in (1) where channels are associated to a camera, a microphone, an NFC reader, and a keyboard. All are sources of events.

$$\overrightarrow{evt} \overset{\text{def}}{=} [id, c, camera_1, micro_1, micro_2, nfc_1, keyboard_1, \ldots] \quad (1)$$

The event vector contains by default two element indexed by id and c. The element id is used to retrieve the identifier of the VDR. The element c is used to add a new VDR to an existing composition as illustrated in (12).

The event vector $\overrightarrow{ev}$ is used only for the interactions between VDRs, the interactions between DVDR on one hand SVDR and the physical device on the other hand, are using a service based channel called ws_i that represents a web service based exchange.

$$VDR(\overrightarrow{ws}) \overset{\text{def}}{=} \begin{pmatrix} (\lambda\, \overrightarrow{ev}\, ws_1)SVDR \\ + \\ (\lambda\, \overrightarrow{ev}\, ws_2)DVDR \\ + \\ (\lambda\, \overrightarrow{ev}\, ws_3)CVDR \\ + \\ \emptyset \end{pmatrix} \quad (2)$$

We define the generalization called VDR as a nondeterministic choice between the three concrete types of VDRs as illustrated in (2). This term is used by (16) to instantiate a new specific VDR. The vector $\overrightarrow{ws}$ is composed of a collection of two elements ws_1 and ws_2 that are the web service channels associated respectively with the SVDR and DVDR. The CVDR does not have any web service channel as he has no direct association with a physical device. However, the CVDR has a higher order parameter representing the specific composition definition like the example in (3) passed to (10).

The term $VDR(\overrightarrow{ws})$ have a vector of web services channels as parameter. This vector called $\overrightarrow{ws}$ is shared between the mobile cloud system and the VDRs. The $\overrightarrow{ws}$ channels allow the exchanges with the physical devices. The term VDR creates a new vector, called $\overrightarrow{ev}$ which contains the channels of the VDR interfaces. We benefit in the VDRs definition of the use abstractions where $(\lambda\, \overrightarrow{ev}\, ws_1)SVDR$ is a natural way to write $SVDR(\overrightarrow{ev}, ws)$, and so on for the two other DVDRs. The specific VDR is activated iff the corresponding element in the $\overrightarrow{ws}$ vector is a valid channel and not an empty process $\emptyset$.

The abstraction-based syntax allows the implicit passing of the event vector. Indeed, an administrator (even human or automatic process) creates a composite device. At the creation of the device composition, the event composition vector needs to be shared between the devices representations that are participating to this composition. To do so, the chaining combinator (see Table 1 and [22]) is used to chain the event channels. A composition needs to define a concrete structure of the event vector as illustrated in (1). This concrete event vector will be used by the chaining conbinator to enable the channels needed by the concrete implementation of the virtual devices and sensors.

$$TwoSensors(ws_1, ws_2) \overset{\text{def}}{=} VDR(ws_1\hat{}\emptyset) \frown VDR(ws_2\hat{}\emptyset)$$

$$TwoSensors \equiv (\nu\, \overrightarrow{evt})\; SVDR(ws)\left\{\overrightarrow{evt}/\overrightarrow{ev}\right\} \,\middle|\, SVDR(ws)\left\{\overrightarrow{evt}/\overrightarrow{ev}\right\} \quad (3)$$

The end user has to define in way that is more concrete the behavior of the devices. To illustrate such concrete definition, we give in (4) an example of a Near Field Communication (NFC) sensor [23] that sends the value $sens$ on the nfc_1 channel.

$$Cnfc(\overrightarrow{ev}, ws) \overset{\text{def}}{=} ws(sens).\tau.\overline{ev_{nfc_1}}\langle sens\rangle \quad (4)$$

The SVDR is activated following up the physical sensor connection event. Once connected, the physical sensor sends the identification data to the SVDR through the ws channel. This data is persisted inside the SVDR using the term $DevId$ defined in (7) that gives back the identification data if requested through the right event channel.

$$SVDR(\overrightarrow{ev}, ws) \overset{\text{def}}{=} ws(id).\tau.\begin{pmatrix} DevId(ev_{id}, id) \\ |VirtualSensor(\overrightarrow{ev}, ws) \end{pmatrix} \quad (5)$$

As illustrated in (5), the term $SVDR$ uses the term $VirtualSensor$ defined in (6) to dispatch the data perceived by the physical sensor using the event channel. At this level, we consider the mapping between the sensor and the matching channel as an invisible action represented by τ. Two possible behaviors can be adapted by the term $VirtualSensor$ as illustrated in (6): if a Stop command (17) is received, the process will end; else, the dispatching action is executed. The parallel composition in the term $SVDR$ allows the administrator to retrieve the VDR identifier using the environment channel ev_i. Moreover, this parallel composition avoids the interruption of the execution of the term in order to not influence the execution of the virtual sensor.

$$VirtualSensor(\overrightarrow{ev}, ws) \overset{\text{def}}{=} ws(C).\begin{pmatrix} [C = Stop]\, Stop \\ + \\ C(\overrightarrow{ev}, ws).VirtualSensor(\overrightarrow{ev}, ws) \end{pmatrix} \quad (6)$$

$$DevId(req, id) \overset{\text{def}}{=} req(cb).\overline{cb}\langle id\rangle.DevId(req, id) \quad (7)$$

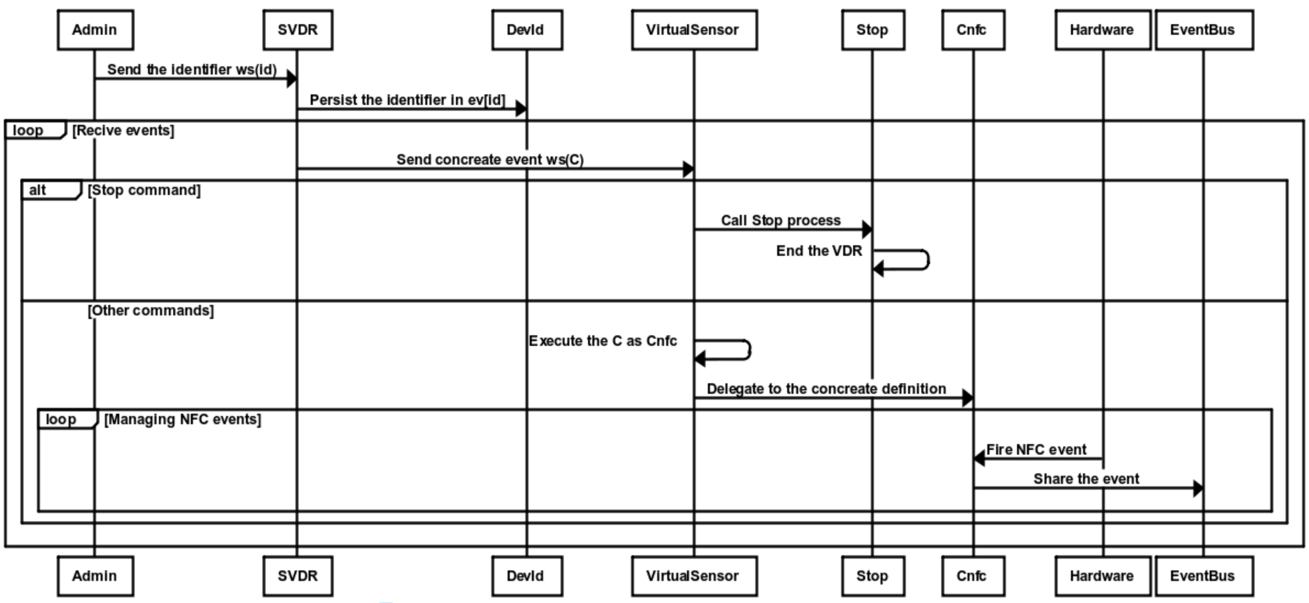

Figure 4 SVDR initialization and execution example

In the sequence diagram Figure 4, we illustrate the initialization of a SVDR. The class represents our π-calculus terms. However, the class "Hardware" is used to design a physical sensor and the class "Event Bus" is used to represent our event bus $\overrightarrow{evt}$ defined in (1). "Hardware" mobile device or the sensor initiates the messages sent through the ws channel. We do not illustrate in the networking infrastructure defined in (21) and (24). This networking infrastructure is used to route the event messages to the target VDR.

The DVDR specification respects the same fundamentals as the SVDR. As illustrated in (8), it uses the term $DevId$ to persist and give back the device identifier. It uses term called $VirtualDevice$ to manage the virtual device behavior. However, the DVDR has the ability to run applications instead of SVDR that only proxy the sensor events.

$$DVDR(\overrightarrow{ev}, ws) \stackrel{\text{def}}{=} ws(id).\tau.\begin{pmatrix} DevId(ev_{id}, id) \\ |VirtualDevice(\overrightarrow{ev}, ws) \end{pmatrix} \qquad (8)$$

We need to dissociate between sensing events sent by the embedded device sensors and the offloaded application requests. To do that, we define a type called App (9) that encapsulates the offloaded application and acts as an application container.

$$App\big(BackEndProc(ws)\big) \stackrel{\text{def}}{=} BackEndProc(ws) \qquad (9)$$

We define in (10) the term $VirtualDevice$ that executes the offloaded application if needed, else, it proxies the sensing data.

$$VirtualDevice(\overrightarrow{ev}, ws) \stackrel{\text{def}}{=} ws(C).\left(\begin{pmatrix} [C = Stop]Stop(\quad) \\ + \\ [C = App(P(x))]P(x) \\ + \\ C(\overrightarrow{ev}, ws) \end{pmatrix} .VirtualDevice(\overrightarrow{ev}, ws)\right) \qquad (10)$$

The main concern of this term is to distinguish between the offloading action represented by $App\big(P(x)\big)$, the stopping action, and sensing actions. To do so, we compare the structure of the name C received through the ws channel. The $VirtualDevice$ runs the higher-order parameter $P(x)$ within the DVDR on the offloading action, calls the term $Stop(\quad)$ on the stopping action, elsewhere, we proxy the message to the corresponding event channel as we do for SVDR using the term . The service channel used for the communication between the physical device and the offloaded application is set as parameter x before the offloading action; this channel is different from the service channel that connects the DVDR and the physical device.

The CVDR in (11) have no direct association with a physical device, its interactions pass through a SVDR or a DVDR. The term $CVDR$ is defined as an aggregation of SVDR and DVDR that are sharing the same events vector. The term $TwoSensors$ defined in (3) is an example of composition definition that may be used as a first approach.

$$CVDR(\overrightarrow{ev}, C) \stackrel{\text{def}}{=} (\nu\ id) \quad DevId(ev_{id}, id)|CompositeDevice(\overrightarrow{ev})\frown C \qquad (11)$$

A new identifier is created the term $CVDR$ and given back if requested using the right event channel ev_{id}.using the term $DevId$.

$$CompositeDevice(\overrightarrow{ev}) \stackrel{\text{def}}{=} ev_c(\vec{e}).\Big(CompositeDevice(\overrightarrow{ev})\left\{{}^{\overrightarrow{ev}\wedge\vec{e}}/{}_{\overrightarrow{ev}}\right\}\Big) \qquad (12)$$

The term $CompositeDevice$ defined in (12) is used to aggregate the events channels. The vector $\overrightarrow{ev}$ is an event channel associated with the actual $CompositeDevice$ while the vector $\vec{e}$ is the event channel associated with the VDR for belonging to this composition. This aggregation is based on two operators: the first one is the renaming operator $\{{}^{x}/{}_{y}\}$ from CCS [24]. The second is the concatenation operator ^. The result of this combination is the use of the concatenation of the two events vectors $\overrightarrow{ev}$^$\vec{e}$ instead of the event vector that was associated with the term $CompositeDevice$.

C. Orchestration

In an MCC context, orchestration is the automation of the management and coordination tasks of the services and components. In addition to the interconnection processes running across heterogeneous systems, the localization of services is an important issue. Processes and VDRs have to cross multiple organizations, systems, and firewalls.

The mobile cloud orchestration aims to automate the configuration, coordination, and management of VDRs and VDRs interactions in such an environment. The process involves automating workflows required for the composition of VDRs and the offloading of mobile Apps. Involved tasks include managing virtualization and emulation in server runtimes, routing the communication flow of apps among VDRs and dealing with exceptions to typical workflows.

In our approach, the orchestrator is composed by three main components as illustrated in (13); we define these three components as common orchestration tasks:

1. Configuration where the cloud orchestrator manage the storage, computation, and networking. In this paper, we do not focus on the resources allocation algorithm (compute and storage); this aspect will be stressed in a future publication. A high-level specification of the networking mechanism is presented in the section III.D.
2. Provisioning where the cloud orchestrator manages the VDRs by providing the run, suspend, and terminate operations.
3. Security where the cloud orchestrator manages the monitoring, and reporting. We have described the details of this aspect also on a separate paper [25] where we describe our implementation and detailed algorithms.

$$Orchestrator(\overrightarrow{api}) \stackrel{\text{def}}{=} Configuration(\overrightarrow{api}) | \text{Provisioning}(\overrightarrow{api}) \quad (13)$$
$$|(\nu\ \overrightarrow{data}) Monitoring(\overrightarrow{api}, \overrightarrow{data})$$

In the term $Configuration$ in (14), we illustrate the use of the configuration api that is used for the allocation of resources, the deallocation (free) of resources, and to suspend the execution. The api is a vector in two-dimensional space. The superscripts indicate the target module (ex: api^c where c stands for $\boldsymbol{C}onfiguration$). The subscripts indicate the service called within the module (ex: api_a where a stands for $\boldsymbol{a}llocate$). The system administrator will use the vector $\overrightarrow{api}$ to configure the deployment environment. We note here that we are not interested in the resource allocation system or free them. At this level, those operations are not observable. However, we associate a new name with each allocation request received from the admin on api_a^c.

$$Configuration(\overrightarrow{api}) \stackrel{\text{def}}{=}$$
$$\begin{pmatrix} api_a^c(allocate).\tau.(\nu\ res)\overline{allocate}\langle res\rangle \\ |api_f^c(free).\tau \\ |api_s^c(suspend).\tau \end{pmatrix} \quad (14)$$
$$.Configuration(\overrightarrow{api})$$

The term Provisioning in (15) uses also an $\overrightarrow{api}$ to ask the configuration module for resource allocation. Once allocated, it receives term, executes the higher-order parameter to enable the creation of the VDR. The parameter is an instance of the term Run defined in (16). We use the abstraction of the resources information res, returned by the term $Configuration$, to communicate this information to the term Run that is preconfigured with the two parameters before its reception through the channel api_r^p.

$$\text{Provisioning}(\overrightarrow{api}) \stackrel{\text{def}}{=}$$
$$\begin{pmatrix} \begin{pmatrix} api_r^p\left(Run(r,\vec{t})\right).(\nu\ allocate)\overline{api_a^c}\langle allocate\rangle \\ |allocate(res).(\lambda\ res)Run(r,\vec{t}) \end{pmatrix} \\ |api_s^p(suspend).\overline{api_s^c}\langle suspend\rangle \\ |\begin{pmatrix} api_t^p(terminate).terminate(ws) \\ .\overline{ws}\langle Stop\rangle.\overline{api_f^c}\langle terminate\rangle \end{pmatrix} \end{pmatrix} \quad (15)$$
$$.\text{Provisioning}(\overrightarrow{api})$$

The suspension is delegated to the term $Configuration$ where it is represented as an non observable action τ. The provisioning sends the term $Stop$ to the VDR to terminate its execution. We use for that the ws channel sent through the channel $terminate$.

The term Run defined in (16) composes a vector depending on the type of the VDR based on requested type using the vector $\overrightarrow{type}$. After the creation of the VDR, it creates and sends a new identifier using the ws channel to start the new created VDR. The emissions on the channel with the new created identifier $\overline{ws}\langle id\rangle$ matches with the equations (5), (8).

$$Run(ws, \overrightarrow{type}) \stackrel{\text{def}}{=}$$
$$[type_v = type_c](\nu\ id)\overline{ws}\langle id\rangle$$
$$\left|\tau.(\nu\ \overrightarrow{ev})\begin{pmatrix} [type_v = type_s]\,VDR(ws\hat{}\emptyset\hat{}\emptyset) \\ |[type_v = type_d]\,VDR(\emptyset\hat{}ws\hat{}\emptyset) \\ |[type_v = type_c]\,VDR(\emptyset\hat{}\emptyset\hat{}ws) \end{pmatrix}\right. \quad (16)$$

$$Stop(\quad) \stackrel{\text{def}}{=} \emptyset \quad (17)$$

To keep our definitions as clear as possible, we didn't integrate the communications between the VDRs and the monitoring module defined in $Monitoring$. We can easily imagine that after each communication on the events vector channel $\overrightarrow{ev}$, an information must be sent to the monitoring module using the api_{put}^m channel. This information is stored in data vector $\overrightarrow{data}$ through the recursive call in (18) to the term $Monitoring$.

$$Monitoring(\overrightarrow{api}, \overrightarrow{data}) \overset{\text{def}}{=} \begin{pmatrix} api^m_{put}(datum).\tau.(\nu\ id)\overline{api^m_{ret}}\langle id\rangle \\ |api^m_{get}(id).api^m_{res}(data_{id}) \end{pmatrix} .Monitoring(\overrightarrow{api}, \overrightarrow{data}\hat{\ }datum) \quad (18)$$

D. *Networking*

On our mobile cloud approach, multiple tenants can use the same physical infrastructure. The network virtualization simplifies the multi-tenancy. The shared infrastructure allows independence of the VDRs regarding the physical host on which it is located. The VDR should be movable between the hosts based on the need. We commit our networking definition to provide access to the VDRs across two different Layer 3 (L3) networks look like they are in the same Layer 2 (L2) domain.

The proposed virtual networking model allows the provisioning module (15) to manage the virtual network component like a VDR and hide the complexity from the user. The model affords also to bypass the 4096 VLAN limit as proposed on VXLAN by the Internet Engineering Task Force (IETF) RFC 7348 [26]. Our model definition is composed from two terms: $vSwitch$ defined in (21) and $vRouter$ defined in (24).

For our network modeling, we define the structure of the packet transiting on the networking infrastructure. The vector $\overrightarrow{ethernet}$ in (19) represents the L2 frame where the names $ethernet_{dst}$ and $ethernet_{src}$ are the channels corresponding to the ws_x used by the VDRs in (2). $ethernet_{ip}$contains the information needed by the $vRouter$ and also the message as $ip_{payload}$. The names that belong to the vectors in (19) and (20) are abbreviations of header fields of the packets as described in the IETF RFC 791.

$$\overrightarrow{ethernet} \overset{\text{def}}{=} [dst, src, tag, type, \overrightarrow{ip}, check] \quad (19)$$

$$\overrightarrow{ip} \overset{\text{def}}{=} \begin{bmatrix} version, ihl, tos, len, id, flag, frag, ttl \\ , proto, check, src, dst, opt, payload \end{bmatrix} \quad (20)$$

Given that our objective is not to stress the networking protocols but to point out the communications between the virtual components, we abstract all network behavior that is not directly related to the virtualization as non-observable operations τ.

$$vSwitch\ (cntl, \overrightarrow{adr}) \overset{\text{def}}{=} Control\ (vSwitch\ (cntl, \overrightarrow{adr}), cntl, \overrightarrow{adr}) \ \Big|\ adr_i(\overrightarrow{ethernet}).\tau.\overline{ethernet_{dst}}\ \langle ethernet_{ip_{payload}}\rangle .vSwitch\ (cntl, \overrightarrow{adr}) \quad (21)$$

The term $vSwitch$ defined in (21) represents the virtualization of the L2 switch. It is modeled as a congruency between the term $Control$ defined in (22) that manages the VDRs connections and a L2 network bridge.

In order to define the terms (22) and (23), we use the notation and the definition of numerals introduced by R. Milner in [22] Section 3.3 as $\underline{n}(x,z) \overset{\text{def}}{=} (\overline{x}.)^n\ \overline{z}$. Milner defined the successor where $(\nu\ xz)\left(\underline{n}(x,z)\middle|Succ(xz, yw)\right) \approx \underline{n+1}(y,w)$. The numerals are used as indexes in order to manage dynamically the control of connection/disconnection of the devices.

$$Control\ (Target, cntl, \overrightarrow{adr}) \overset{\text{def}}{=} cntl_{connect}(link).Target \ \Big|\ cntl_{disconnect}(link).(\nu\ i\ z)\left(Disconnect(Target, \overrightarrow{adr}, (\nu\ \vec{p}), link, \underline{0}(i,z))\right) \quad (22)$$

$$Disconnect(Target, \overrightarrow{old}, \overrightarrow{adr}, port, \underline{n}(i,z)) \overset{\text{def}}{=} \left[\underline{0+n}(i,z) \approx \underline{0+\|\overrightarrow{old}\|}(y,z)\right] (\lambda\ \overrightarrow{adr})Target \ \Big|\ [old_i = port]\ Disconnect\left(Target, \overrightarrow{old}, \overrightarrow{adr}, port, (\nu\ y)\underline{n+1}\,(y,z)\right) \ \Big|\ Disconnect\left(Target, \overrightarrow{old}, \overrightarrow{new}\hat{\ }port, port, (\nu\ y)\underline{n+1(y,z)}\right) \quad (23)$$

The term $Control$ takes three parameters, the first one is higher-order parameter called $Target$ that is used to pass the terms $vSwitch$ and $vRouter$. The second is called $cntl$ and it is used as a channel to control connections of the VDRs. The third parameter is the vector containing connected VDRs channels. The $Target$ parameter is passed also to the term $Disconnect$ defined in (23), we use the abstraction λ to override the addresses vector $\overrightarrow{adr}$ that is still a free name in $Target$ when a device is disconnected.

$$vRouter\ (ipAdr, cntl, \overrightarrow{adr}) \overset{\text{def}}{=} Control\begin{pmatrix} vRouter\ (ipAdr, cntl, \overrightarrow{adr}) \\ , cntl, \overrightarrow{adr} \end{pmatrix} \ \Big|\ adr_i(\overrightarrow{ethernet}).\tau.\begin{pmatrix} [ipAdr = ethernet_{ip_{dest}}] \\ \overline{ethernet_{dst}}\langle\overrightarrow{ethernet}\rangle \\ + \\ \overline{ethernet_{ip_{dest}}}\langle\overrightarrow{ethernet}\rangle \end{pmatrix} .vRouter\ (cntl, \overrightarrow{adr}\hat{\ }link) \quad (24)$$

The term $vRouter$ defined in (24) represents the virtualization of the L3 routing. It is modeled as a concurrency between a control (22) that manages the virtual switches or VDRs connections and a L3 network bridge. In this model, we do not illustrate some features like IP forwarding to keep our definition clear.

The management of the networking infrastructure is exposed as a part of the provisioning API. To do so, we illustrate in (25) an extension of the term $Provisioning$ defined initially in (15). We add in this extension the possibility to connect and disconnect devices. Moreover, this extension offers the possibility to create a new L2 switch.

$$\text{Provisioning}(\overrightarrow{api}) \overset{\text{def}}{=} \left(\begin{array}{c} \dots \\ \dots \\ \dots \\ \left| \begin{array}{l} api^{p}_{vsCreate}(ret).(\nu\ cntl) \\ \quad \left(\begin{array}{l} \tau.(\nu\ \overrightarrow{adr})vSwitch\,(cntl, \overrightarrow{adr}\,) \\ \qquad \mid \overline{ret}\langle cntl \rangle \end{array} \right) \end{array} \right. \\ |api^{p}_{vsConnect}(cntl, adr).\tau.cntl_{connect}(adr) \\ |api^{p}_{vsDisconnect}(cntl, adr).\tau.cntl_{disconnect}(adr) \\ \dots \end{array} \right) .Provisioning(\overrightarrow{api}) \quad (25)$$

In (14), we describe the Switch related provisioning API, the channel $api^{p}_{vsCreate}$ is used to create the virtual switch and return the control channel *cntl* to the initiator of the request. The Router provisioning API is similar to the Switch's API, the two differences are that the api^{p}_{vs*} channels are defined as api^{p}_{vr*} and the $api^{p}_{vrCreate}$. They are used to create a virtual router, to keep our definition clear, we omit this part of the definition.

Our previous formal model aims to define a new architecture of the Cloudlet based approach. The definitions are useful not only for this current work but also for all software researcher in cloud computing domain that could use it as a framework to proof properties related on Cloudlet based systems.

IV. Architecture

The definition presented in the previous section is based on the state-of-art regarding the MCC research [27] [28] that converge into the Cloudlet-based MCC. The cloudlets are defined as trusted and resource-rich network of computers. They offer bridging capabilities to the Internet. They are available for using by nearby mobile devices through a direct and reliable connection. In this section, we describe our Cloudlet-based architecture by illustrating some of the technical aspects described in an abstract level in the formal definition of Section III. We also link the technical implementation (or implementation model) with the corresponding formal model. In addition, we introduce our contribution to the migration pattern and stress the projection of the ACID (Atomicity, Consistency, Isolation, and Durability) properties from the formal model to the implementation model.

A. *Cloudlet-based MCC*

In our approach, we have identified the need of a set of rules and regulations, as a protocol, which determines how data and processes are transmitted between the different components of the MCC. The Figure 5 illustrates our vision of the MCC that is composed of three layers: first is the Device Layer (DL) composed by physical sensor (e.g.: camera, thermostat …) and mobile devices. The second is the Cloudlet Layer (CL) that is composed from the network of Cloudlets; each Cloudlet could contains the VDRs, Virtual Service Representation (VSR), and local services. The third layer is the Internet Layer (IL) composed by the central Cloud. The Central Cloud contains services, needed registries, and Internet services (e.g.: the media sensors).

In the CL, we define a networking infrastructure based on the NFV. As illustrated in Figure 6, the device is connected to the VDR through a vRouter defined in (26) and a vSwitch defined in (23). The networking infrastructure is managed using the cloud orchestrator API, in our implementation model representing the pragmatic implementation of our system; we use OpenStack [29] that contains a powerful networking module called Neutron. This module is based on Open vSwitch [30]. This implementation provides a REST [31] API for the creation and the management of the provided virtual networking infrastructure.

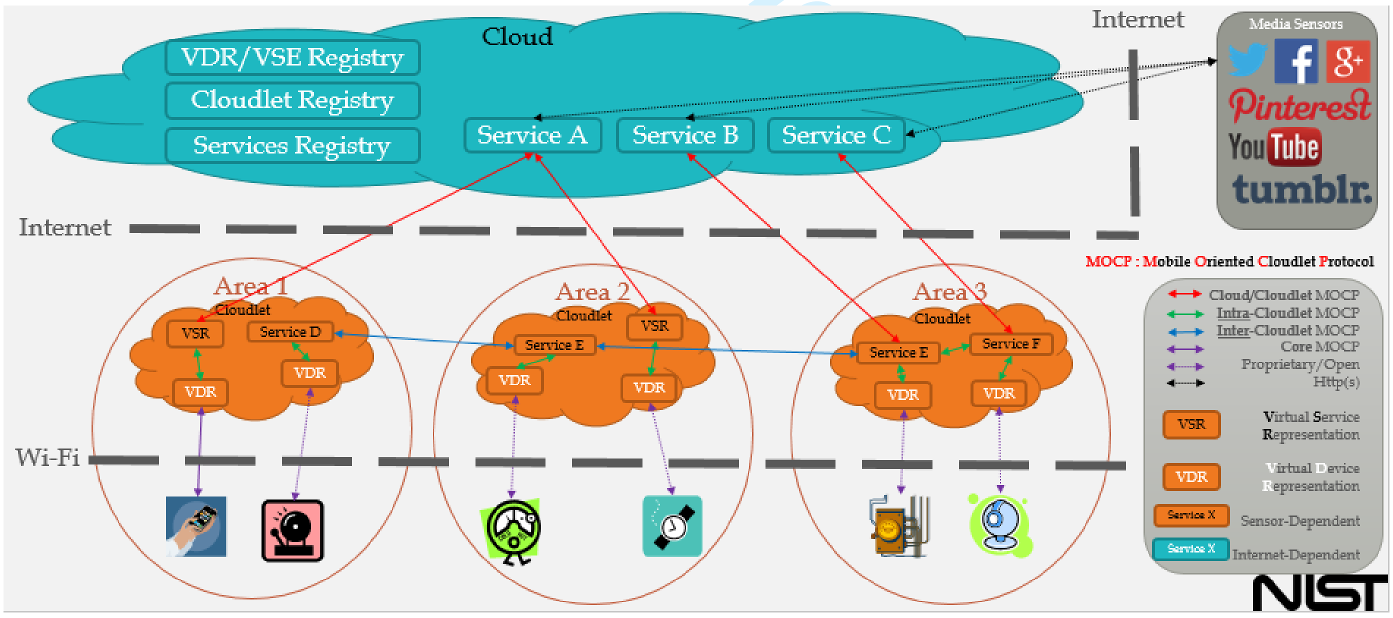

Figure 5 Global architecture

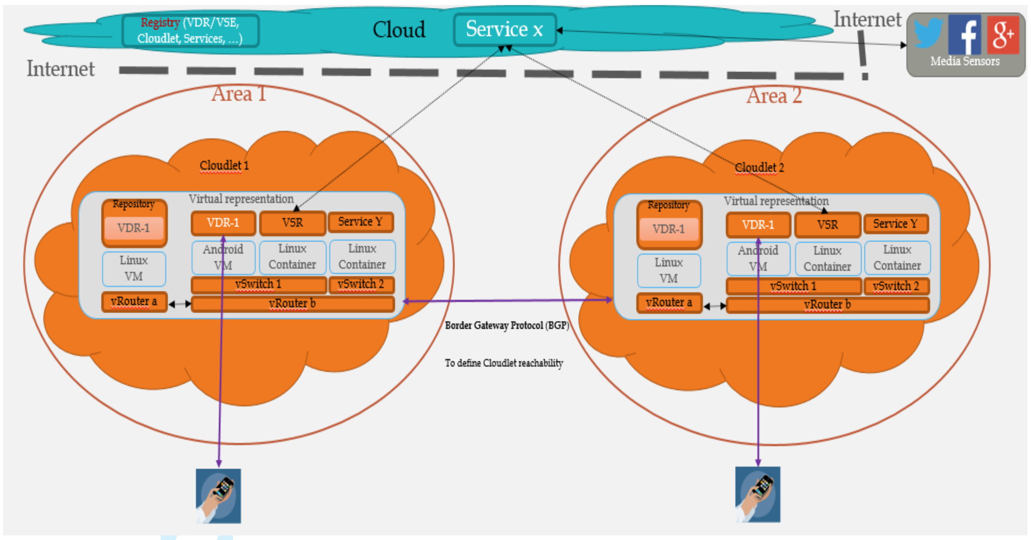

Figure 6 Cloudlet structure and networking

As a part of our vision of the MCC system, the Mobile Oriented Cloudlet Protocol (MOCP) is a central protocol that aims to connect each piece of such systems. The four main parts of this protocol are illustrated in Figure 5 and each part is used as a framework for the communication between two types of components:

- **Cloud/Cloudlet MOCP** is used for the communication between **Cloud Services** and **VSR/Local services**
- **Intra-Cloudlet MOCP** is used for the communication between **VSR/Local services** and **VDR**
- **Inter-Cloudlet MOCP** is used for the communication between Cloudlets **Local services**
- **Core MOCP** is used for the communication between the **VDR** and the **Mobile Devices**

The formal definition presented in this paper focus on the communications used by the Core MOCP for the migration of the mobile applications (called Apps) from the physical device to the VDRs. Our implementation model, as illustrated in Figure 7, extends the formal definition in (8) by adding technical details. The two main components of the VDR are the *Device Descriptor* that is modeled by the $DevId$ in (7) and the *Virtual device* is modelled in (10). The *Backend app* is modeled as the higher-order parameter $BackEndProc$ in (9). The *OSGi* [32] *container* operations are considered as non-observable operations.

Our offloading approach differs from the actual overlays oriented [33] approaches that execute a dynamic profiling to partition applications automatically. As illustrated in Figure 7, our offloading approach is based on a component called *Backend app*. This component contains the business logic of the mobile application. We consider the *Backend app* as an ACID service that can migrate from one host to another one using the term defined in (9). Our definition of the DVDR in (10) allows the ACID properties by isolating the *Backend app* in an atomic process. The execution of the *Backend app* makes a durable impact on the target VDR. These properties are extended to the implementation model by using the OSGi framework that isolates the class loading inside the JVM and guarantees a strict lifecycle of the *Backend app* bundle. This lifecycle management guarantees the consistency of the changes performed during the execution of such process.

The Apache Felix [34] OSGi implementation is used in our architecture due to an Android porting effort from the Apache Software Foundation (ASF) that is supporting an Android distribution since the version 1.3. This mechanism works with stateless Backend services that provides responses to the requests from Frontend Cloudlet Android Application Package (CAPK), and then requires no further interactions. Subsequent Frontend CAPK requests depend on the result of the first request. Regarding the stateful Backend service, we face more difficulties to manage such issue. This is because single action typically involves more than one request. We need thus another isolation level in top of the OSGi.

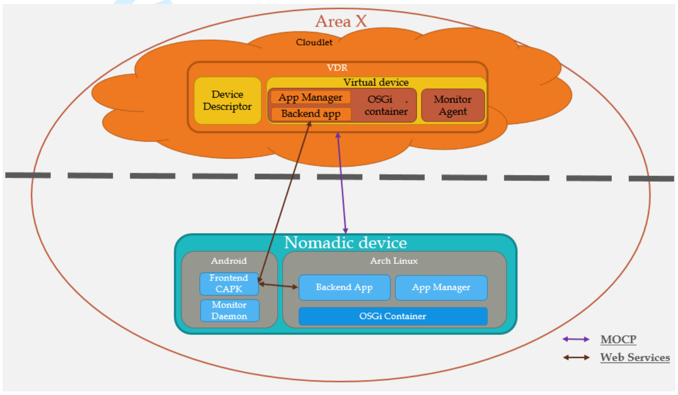

Figure 7 DVDR implementation model

To address the issue of the state management, we use actually a *chroot* [35]of ArchLinux that provides an additional layer of abstraction using the Docker package available with this distribution. We are working on the integration of Docker on

Android to bypass the need of a *chroot* and to allow a native isolation support on Android.

V. Case Studies

Our case study aims to show the structural congruence between a *Backend app* offloaded in a VDR and the same *Backend app* running in the mobile device. Our objective is to illustrate that a *Backend app* (9) that runs in a VDR is identical up to structure in particular parallel composition to the *Backend app* which runs in a mobile device. This result is obtained after the reduction of both systems to an identical system.

A. *Mobile device*

We define first the terms $FrontEnd$ which represents the Frontend CAPK and $BackEnd$ which represents the Backend app used in our study. Those terms are composing the mobile devices defined in (28) and (29).

The term $FrontEnd$, defined in (26), is a model of a "web view" which sends messages to the Backend using the channel ws, once the response received by the Backend, the Frontend executes another iteration as a recursion. This term has also the $touch$ channel as parameter to communicate with the user defined in (31).

$$FrontEnd(touch, ws) \stackrel{\text{def}}{=} (\nu\, cb)$$
$$touch(event).\tau.\overline{ws}\langle event, cb\rangle \tag{26}$$
$$|cb(res).FrontEnd(touch, ws)$$

The term $BackEnd$, defined in (27), react to the message sent by the Frontend. If the abstraction $intra$ binds to the same channel as the parameter ws, the Backend app is executed locally to the mobile device. Otherwise, the Backend send a message containing a copy of itself to the corresponding VDR and terminates the local execution. The execution continues into the VDR after the offloading.

$$BackEnd(ws) \stackrel{\text{def}}{=} (\lambda\, intra)(\nu\, end)$$
$$ws(event, cb).\tau$$
$$.\begin{pmatrix} [intra = ws].\overline{intra}\langle\ \ \rangle \\ +\,\overline{ws}\langle App\big((\lambda\, ws)BackEnd(ws)\big)\rangle.\overline{end}\langle\ \ \rangle \end{pmatrix} \tag{27}$$
$$|\overline{end}\langle\ \ \rangle.\emptyset + intra(\ \ ).\tau.(\nu\, res)\overline{cb}\langle res\rangle$$

We define two parallel compositions as models for the mobile devices. The first mobile device is defined in (28) as the parallel execution of a Frontend and a locally executing Backend. The second mobile device is defined in (29) as the parallel execution of a Frontend and a Backend which is configured to be offloaded to the VDR.

$$Devicelocal(ws, touch) \stackrel{\text{def}}{=}$$
$$FrontEnd(touch, ws)|(\lambda\, ws)BackEnd(ws) \tag{28}$$
$$DeviceRemote(ws, touch) \stackrel{\text{def}}{=} (\nu\, local)$$
$$(FrontEnd(touch, local)|(\lambda\, local)BackEnd(ws)) \tag{29}$$

To keep the clarity of our specification, we omit the details of the definition of the term $Admin$, we define just the signature in (29). It is important to note that this term sends all needed messages using the vector $\overrightarrow{api}$. It starts the networking infrastructure and the VDRs.

$$Admin(ws, \overrightarrow{api}) \stackrel{\text{def}}{=} \cdots \tag{30}$$

The term $user$ defined in (31) represents a device user executing a single action by sending an event to the Frontend through the channel $touch$ that represents the device's touch screen. We have defined a simple action for the user because we want a system that can be reduced manually by a human in a reasonable time slot.

$$User(touch) \stackrel{\text{def}}{=} (\nu\, event)\overline{touch}\langle event\rangle \tag{31}$$

B. *Systems*

To be able to verify the structural congruence, we define two systems as parallel composition of the mobile user, mobile device, administrator, and the orchestrator. The term $SystemMig$ defined in (32) represents the system that will lead raise to a Backend offloading after some reductions.

$$SystemMig \stackrel{\text{def}}{=} (\nu\, ws)$$
$$\begin{pmatrix} (\nu\, touch)\begin{pmatrix} user(touch) \\ |DeviceRemote(ws, touch) \end{pmatrix} \\ |(\nu\, \overrightarrow{api})\begin{pmatrix} Admin(ws, \overrightarrow{api}) \\ |Orchestrator(\overrightarrow{api}) \end{pmatrix} \end{pmatrix} \tag{32}$$

The term $SystemLocal$ defined in (33) represents the system that will allow raising to a Backend executed locally after some reductions.

$$SystemLocal \stackrel{\text{def}}{=} (\nu\, ws)$$
$$\begin{pmatrix} (\nu\, touch)\begin{pmatrix} user(touch) \\ |Devicelocal(ws, touch) \end{pmatrix} \\ |(\nu\, \overrightarrow{api})\begin{pmatrix} Admin(ws, \overrightarrow{api}) \\ |Orchestrator(\overrightarrow{api}) \end{pmatrix} \end{pmatrix} \tag{33}$$

The systems (32) and (33) used in our case study are illustrated in Figure 8 where we can see the similarities between the two systems. In addition, this Figure illustrates the offloading specific connections and the instantiated components. The structural congruence that we stress in the next section is based on assumption that after the execution of the two systems, we will find a two reduced systems $SystemMig'$ and $SystemLocal'$ that are structurally congruent.

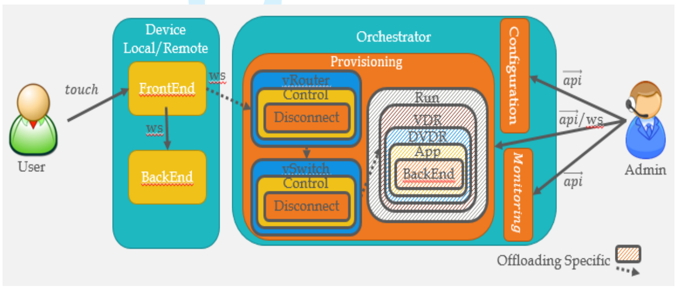

Figure 8 The system used in our case study

C. *Structural congruence*

We have performed some computations steps to fully reach a stable system starting from $SystemMig$. The computation starts with sending a touch event using $\overline{touch}\langle event\rangle$ from the $User$ (31) and ends with the reception of the stopping signal $\overline{ws}\langle Stop\rangle$on the $VirtualDevice$ (11). We call this reached stable state after those reductions $SystemMig'$ where $SystemMig \xrightarrow{\overline{touch}\langle event\rangle,\ldots} SystemMig'$.

We have applied the operation to the $SystemLocal$. However, the reduction of this system is simpler by dint of no offloading related reductions. Also, we obtain $SystemLocal'$ where $SystemLocal \xrightarrow{\overline{touch}\langle event\rangle,\ldots} SystemLocal'$.

The structural congruence is defined [22] to be a congruence relation over a process respecting the following laws:

- Agents (processes) are identified if they only differ by a change of bound names
- $(\mathcal{N}/\equiv,+,\emptyset)$ is a symmetric monoid
- $(\mathcal{P}/\equiv,|,\emptyset)$ is a symmetric monoid
- $!\,P \equiv P \mid !\,P$
- $(\nu\ x)\emptyset \equiv \emptyset, (\nu\ x)(\nu\ y)P \equiv (\nu\ y)(\nu\ x)P$
- If $x \notin fn(P)$ then $(\nu\ x)(P|Q) \equiv\ P \mid (\nu\ x)Q$

Where $fn(P)$ is defined as the free names of the process P.

After the reduction of $SystemMig$ and $SystemLocal$, we observed that only some bound names and non-observable actions compos the difference between the two reduced systems. We can thus claim the structural equivalence of the two systems as $SystemMig'\ \equiv SystemLocal'$.

By definition, the structural definition is commutative and associative. We can then write:

$$\begin{array}{ll} \text{given that} & SystemMig\ \equiv SystemMig' \\ \text{and} & SystemLocal\ \equiv SystemLocal' \\ \text{and} & SystemMig'\ \equiv SystemLocal' \\ \text{then} & SystemMig\ \equiv SystemLocal \end{array} \tag{34}$$

VI. Conclision and Future Works

In this paper, we present our vision of a MCC system. We provide some equations of our formal definition of the MCC to express our architecture of such system. All the equations presented in this paper focuses on the communication interactions between the MCC system components. We present also some aspects of our architecture dedicated to the realization of a MCC solution. Our case study presents two systems where the first one implies the offloading of an application and the second one implies the local execution of the same application. We demonstrate in this case study the structural congruence between offloading and local execution of a mobile application. We consider that as a proof of the transparency of the offloading in our MCC system.

On our future work, we will focus on two aspects of the MCC. The first one is a formal definition of a metric to define a unit to measure the applications migration. The second aspect is the definition of the data collection and algorithm to calculate the application-offloading cost. Also, how to analyze collected data to define when the offloading can be an optimization in the MCC context.

In parallel with our Cloudlet work, we investigate the possible applications of such system especially to boost the intelligence of smart-buildings and to help on the composition of the devices within the Internet of Things (IoT).

References

[1] C. Kyung-Yong, J. Yoo and K. Kim J., "Recent trends on mobile computing and future networks.," Personal and Ubiquitous Computing, vol. 18, no. 3, pp. 489-491, 2014.

[2] D. Sanderson, Programming google app engine: build and run scalable web apps on google's infrastructure., O'Reilly Media, Inc., 2009.

[3] M. Armbrust, "A view of cloud computing," Communications of the ACM, vol. 53, no. 4, pp. 50-58, 2010.

[4] W. Geng, S. Talwar, K. Johnsson, N. Himayat and K. D. Johnson, "M2M: From mobile to embedded internet.," IEEE Communications Magazine, vol. 49, no. 4 , pp. 36-43, 2011.

[5] L. Liu, R. Moulic and D. Shea, "Cloud service portal for mobile device management," IEEE 7th International Conference on e-Business Engineering (ICEBE), pp. 474--478, 2010.

[6] D. Bartholomew, "Qemu a multihost multitarget emulator," Linux Journal, no. 145, p. 3, 2006.

[7] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman and E. Y. Wang, "Bringing virtualization to the x86 architecture with the original vmware workstation," ACM Transactions on Computer Systems (TOCS), vol. 30, no. 4, p. 12, 2012.

[8] A. Velte and T. Velte, Microsoft virtualization with Hyper-V, McGraw-Hill, Inc., 2009.

[9] B. Walters, "VMware virtual platform," Linux journal, vol. 63, p. 6, 1999.

[10] J. Jiulei, L. Jiajin, H. Feng, W. Yan and S. Jie, "Formalizing Cloud Service Interactions," Journal of Convergence Information Technology, vol. 7, no. 13, 2012.

[11] C. Mahmoudi, Orchestration d'agents mobiles en communauté, Universite Paris-Est Creteil, 2014.

[12] N. M. K. Chowdhuryr and R. Boutaba, "A survey of network virtualization," Computer Networks, vol. 54, no. 5, pp. 862-876, 2010.

[13] M. Ugo and S. Matteo, Network conscious pi-calculus, Pisa: Universita di Pisa, 2012.

[14] A. Singh, C. Ramakrishnan and S. A. Smolka, "A process calculus for mobile ad hoc networks," Science of Computer Programming, vol. 75, no. 6, pp. 440-469, 2010.

[15] D. Sangiorgi and D. Walker, The pi-calculus: a Theory of Mobile Processes, Cambridge university press, 2003.

[16] R. Milner, Communicating and mobile systems: the pi calculus, Cambridge university press, 1999.

[17] R. Milner, P. Joachim and W. David, "A calculus of mobile processes," Information and computation , vol. 100, no. 1, pp. 1-40, 1992.

[18] R. {Napier and M. Kumar, iOS 7 Programming Pushing the Limits: Develop Advance Applications for Apple iPhone, iPad, and iPod Touch, John Wiley & Sons, 2014.

[19] J. Annuzzi Jr, L. Darcey and S. Conder, Advanced Android Application Development, Addison-Wesley Professional, 2014.

[20] J. Wingfield, Developing a Windows Phone Application, Cardiff Metropolitan University, 2014.

[21] R. Milner, The polyadic π-calculus: a tutorial, Berlin Heidelberg: Springer, 1993.

[22] G. Milette and A. Stroud, Professional Android sensor programming, John Wiley & Sons, 2012.

[23] R. Milner, A calculus of communicating systems, Springer, 1980.

[24] C. :. M. F. Mahmoudi, "Monitoring Architecture for Cloudlet Based Mobile Cloud Computing," Submitted.

[25] T. Sridhar, L. Kreeger, D. Dutt, C. Wright, M. Bursell, M. Mahalingam, P. Agarwal and K. Duda, Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks, IETF, 2014.

[26] H. T. Dinh, C. Lee, D. Niyato and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches.," Wireless communications and mobile computing, vol. 13, no. 18, pp. 1587-1611, 2013.

[27] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra and P. Bahl, "MAUI: making smartphones last longer with code offload," Proceedings of the 8th international conference on Mobile systems, applications, and services, pp. 49-62, 2010.

[28] A. Corradi, M. Fanelli and L. Foschini, "VM consolidation: A real case based on OpenStack Cloud," Future Generation Computer Systems, vol. 32, pp. 118-127, 2014.

[29] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen and S. Shenker, Extending Networking into the Virtualization Layer., Hotnets, 2009.

[30] R. Fielding, "Representational state transfer," Architectural Styles and the Design of Netowork-based Software Architecture, pp. 76-85, 2000.

[31] Alliance, OSGi, Osgi service platform, release 3, IOS Press, Inc., 2003.

[32] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," Proceedings of the sixth conference on Computer systems, pp. 301-314, 2011.

[33] Felix, Apache, "Apache Felix-welcome," Apache Software Fundation, 12 03 2015. [Online]. Available: http://felix.apache.org. [Accessed 26 03 2015].

[34] P. Lessard, Linux process containment: A practical look at chroot and user mode Linux, 2003.

[35] Yaser Jararweh, T. Loai, A. Fadi and D. Fahd, "Resource efficient mobile computing using cloudlet infrastructure," IEEE Ninth International Conference on Mobile Ad-hoc and Sensor Networks (MSN), pp. 373-377, 2013 .