Formal Definition of Edge Computing: An Emphasis on Mobile Cloud and IoT Composition

Charif Mahmoudi[†‡], Fabrice Mourlin[†] and Abdella Battou[‡]
[†]Advanced Network Technologies Division, National Institute of Standards and Technology
[‡]Algorithmic, Complexity and Logic Laboratory, University of Paris-Est Créteil
charif.mahmoudi@nist.gov, fabrice.mourlin@u-pec.fr, abdella.battoug@nist.gov

Abstract—Within the Edge computing umbrella, mobile cloud computing is an emerging area where two trends come together to compose its major pillars. On one hand, the virtualization affecting the data centers hypervisors. On the other hand, device's mobility, especially Smart Phones, 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 of the Mobile cloud component using the π-calculus. The proposed model defines the mobile cloud component, the virtual device representation, and interaction that leads to application offloading and device composition. This paper describe our contribution that enables the composition of virtual devices from physical devices, sensors, and actuators available on the network. Moreover, we present a model of application offloading and virtual devices networking on mobile clouds. Our architectural model is inspired from the Cloudlet based system. In addition to the formal specifications and architecture this paper presents a case studies showing the structural congruence between a locally executed application and an offloaded version of that same application.

Keywords— formal definition; migration; mobile; mobile cloud computing; offloading; virtualization; virtual device representation, fog, internet of things

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 a 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 resources-consuming application modules to remote servers using the cloud service platforms. Google offers one of the major solutions called AppEngine [2]. Such a solution is allowing developers without previous understanding or knowledge of cloud technology infrastructure to deploy services and use the cloud. These platforms execute the deployed services and expose them as a remote service. That enables delegation of massive computation pieces of the mobile software to the cloud infrastructure.

As one component of the Edge computing, current mobile cloud architectures are based on cloud computing abstractions (IaaS, PaaS and SaaS) [3] and adapt this concepts for a deployement at the edge of the network. This architecture addresses 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 heterogeneity of the actual solutions. Indeed, the virtualisation of devices and services is following the server architectures that are not suitable for the mobile platforms. This is due to the heterogeneity of the hardware architectures and the available resources. Another limitation is the lack of specific representation of the mobile devices on the cloud. The deployed services artefacts are a classical web service. There is no specific representation that makes abstraction for the application offloading and the location management. Moreover, using a generic representation makes the remote services implementations dependant of both the cloud platform and the devices capability. In term of development, this constraint implies that the software component developed as a remote cannot be reused in the client side. In addition, interfaces that exposes the same services may be deffirent 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 act also as a "mobile-friendly" platform within the cloud. Indeed, the representation is built on emulation capabilities that offer a compliant environment with the physical device on which the representation is associated.

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

This paper describes, in Section II, the various challenges faced during virtualization and how they are addressed in work 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 differents 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.

With the emergence of Fog, the network connections between edge devices and the cloud are reconsidered as part of the computational processes being done close to the edge devices called edge computing. Mobile Cloud is one of the implementations of the Edge to incorporate the network needed to get processed data to its destination. The mobile technology, which is easing access to business data and applications is also providing various means of communications. These features continue to be embraced by users. For IT point of view, mobile technology and the unprecedented pace of change in the mobile arena 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 allow the execution of a guest OS. First is emulation where VM emulates (or simulates) complete hardware if the unmodified guest OS for a different PC cannot be run. There are some hypervisors specialized on "emulation" like Bochs, VirtualPC for Mac and Qemu [6]. Second is 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 hypervisors like, VMWare Workstation [7] and Microsoft Hyper-V [8].

2) Virtualization usage

By using 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 addresses the resources 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.

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 have 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 LBaaS (Load balancer). This network virtualization approach is called Network Functions Virtualization (NFV) [12]. It aims to consolidate and deliver the networking components needed to support a fully virtualized infrastructure and shared by multiple tenants in a secure and isolated manner.

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

C. Virtualization Formalizm

In parallel with the pragmatic work on the networking, there is 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. This lack of networking virtualization formalism motivates our high-level definition of network virtualization in the next section.

III. Virtual Device Representation

In our approach, the VDR aims to address the mobile cloud computing virtualization paradigm. We have identified three types of VDRs, and each type has a specific role within the mobile cloud.

1. Sensor VDR (SVDR): it represents a physical sensor or actuator within to the mobile cloud.
2. Device VDR (DVDR): it represents a physical mobile device within to the mobile cloud.
3. Composite VDR (CVDR): it represents a composition of SVDR, DVDR, and mobile cloud resources.

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) [15], 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 and the 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), devices, and sensors. It is a software composite component that 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 have a network connection.

A VDR can be associated to a physical device nor sensor in this case, a 1:1 association is control their interactions (ex: SVDR and DVDR). We call this category “Emulated VDR” A VDR can be free of any hardware association, in this case, it is a composed VDR (CVDR) that aggregates it components hardware associations and have then a 0:n association to the hardware devices. 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) nor a service call. We define an event vector representing all interfaces of a VDR. This event vector, illustrated in (1) contains channels that are used to exchange messages

$$\overrightarrow{ev} \overset{\text{def}}{=} \left[camera_m, micro_n,\ nfc_o, keyboard_p, \ldots\right] \qquad (1)$$

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 that represents a web service based exchange.

$$VDR(\overrightarrow{ws}) \overset{\text{def}}{=}$$

$$(\nu\ \overrightarrow{ev}) \begin{pmatrix} (\lambda\ \overrightarrow{ev}\ ws_i)SVDR \\ + \\ (\lambda\ \overrightarrow{ev}\ ws_j)DVDR \\ + \\ (\lambda\ \overrightarrow{ev})CVDR \\ + \\ \emptyset \end{pmatrix} \qquad (2)$$

We define the generalization called VDR as a nondeterministic choice between the three types of VDRs as illustrated in (2)

The term $VDR(\overrightarrow{ws})$ have a vector $\overrightarrow{ws}$ of web services channels as parameter, these channels are shared with the mobile cloud system and are transmitted to the specific VDRs to allow the communication with the physical devices. The term VDR creates a new $\overrightarrow{ev}$ vector containing the channels that are used to interface the specific VDRs. We benefit in the VDRs definition of the use abstractions where $(\lambda\ \overrightarrow{ev}\ ws_i)SVDR$ is a natural way to write $SVDR(\overrightarrow{ev}, ws)$, and so on for the two other VDR types. 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 SVDR is activated behind 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 (5) that give back the identification data if requested through the right event channel.

$$SVDR(\overrightarrow{ev}, ws) \overset{\text{def}}{=}$$

$$ws(id).\tau.\begin{pmatrix} DevId(ev_i, id) \\ |VirtualSensor(\overrightarrow{ev}, ws) \end{pmatrix} \qquad (3)$$

As illustrated in (3), the term $SVDR$ uses the term $VirtualSensor$ defined in (4) 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 (4): if a Stop command (15) is received, the process will end, else, the dispatching action is executed. The parallel composition of the term $SVDR$ allows the administrator to retrieve the VDR identifier using the environment channel ev_i and don't impact the execution of the virtual sensor.

$$VirtualSensor(\overrightarrow{ev}, ws) \stackrel{\text{def}}{=}$$

$$ws(sens).\left(\begin{array}{c}[sens = Stop]\,Stop \\ + \\ \tau.\overline{ev_i}\langle sens\rangle.VirtualSensor(\overrightarrow{ev}, ws)\end{array}\right) \quad (4)$$

$$DevId(req, id) \stackrel{\text{def}}{=} req(cb).\overline{cb}\langle id\rangle.DevId(req, id) \quad (5)$$

Messages sent through the ws channel are initiated by the mobile device or the sensor. However, these messages are forwarded to the target VDR by the networking infrastructure defined in (19) and (22).

The DVDR specification respects the same fundamentals as the SVDR. As illustrated in (6), it uses the term $DevId$ to persist and give back the device identifier and use term called $VirtualDevice$ to manage the virtual device behavior. However, the DVDR can run applications instead of SVDR that only proxy the sensor events.

$$DVDR(\overrightarrow{ev}, ws) \stackrel{\text{def}}{=}$$

$$ws(id).\tau.\left(\begin{array}{c}DevId(ev_i, id) \\ |VirtualDevice(\overrightarrow{ev}, ws)\end{array}\right) \quad (6)$$

We need to dissociate between sensing events sent from the device embedded sensors and the application offloading requests. To do that, we define a type called App (7) that encapsulate the offloaded application.

$$App\big(BackEndProc(ws)\big) \stackrel{\text{def}}{=} BackEndProc(ws) \quad (7)$$

We define in (8) the term $VirtualDevice$ that execute the offloaded application if need, else, it proxies the sensing data.

$$VirtualDevice(\overrightarrow{ev}, ws) \stackrel{\text{def}}{=}$$

$$ws(msg).\left(\begin{array}{c}[msg = Stop]\,Stop \\ + \\ \left(\begin{array}{l}case\ msg\ of \\ : App\big(P(x)\big) \Rightarrow P(x) \\ : msg \Rightarrow \tau.\overline{ev_i}\langle msg\rangle\end{array}\right) \\ .VirtualDevice(\overrightarrow{ev}, ws)\end{array}\right) \quad (8)$$

We used to this definition the syntactic sugar introduced by R. Milner in [16] by using the "case of" instruction to distinguish between the offloading action represented by $App\big(P(x)\big)$ and sensing actions. Where we run the higher-order parameter $P(x)$ within the DVDR on the offloading action, elsewhere, we proxy the message to the corresponding event channel as we do for SVDR. 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 (9) have no direct association with a physical device, its interactions pass through a SVDR nor a DVDR. The term $CVDR$ is defined as an aggregation of SVDR and DVDR that are sharing the same events vector.

$$CVDR(\overrightarrow{ev}) \stackrel{\text{def}}{=} (\nu\ id)$$

$$DevId(ev_i, id)|CompositeDevice(\overrightarrow{ev}) \quad (9)$$

A identifier is created the term $CVDR$ and returned trough the right event channel ev_i.using the term $DevId$.

$$CompositeDevice(\overrightarrow{ev}) \stackrel{\text{def}}{=}$$

$$ev_i(\vec{e}).CompositeDevice(\overrightarrow{ev}\,^\wedge\vec{e}) \quad (10)$$

The term $CompositeDevice$ defined in (10) is used to aggregate the events channels $\overrightarrow{ev}$ (the event channel associated with the actual $CompositeDevice$) and $\vec{e}$ (the event channel associated with the VDR to add to this composition) using the concatenation operator ^.

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 must 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, directing 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 (11), we define these three components as common orchestration tasks: 1) Configuration where the cloud orchestrator manages the storage, compute, 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 next sub section. 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 describe the details of this aspect also on a separate paper where we describe our implementation and detailed algorithms.

$$Orchestrator(\overrightarrow{api}) \stackrel{\text{def}}{=}$$

$$Configuration(\overrightarrow{api})|\text{Provisioning}(\overrightarrow{api}) \quad (11)$$

$$\big|(\nu\ data)Monitoring\big(\overrightarrow{api}, \overrightarrow{data}\big)$$

In the term $Configuration$ in (12), 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 contravariant indicates the target module (ex: api^c where $^{\boldsymbol{c}}$ stand for $\boldsymbol{C}onfiguration$). The covariant indicates the service called within the module (ex: api_a where $_{\boldsymbol{a}}$ stand for $\boldsymbol{a}llocate$). The system administrator will use the vector $\overrightarrow{api}$

$$Configuration(\overrightarrow{api}) \stackrel{\text{def}}{=}$$

$$\begin{pmatrix} api_a^c(allocate).\tau.(\nu\, res)\overline{allocate}\langle res\rangle \\ |api_f^c(free).\tau \\ |api_s^c(suspend).\tau \end{pmatrix} .Configuration(\overrightarrow{api}) \tag{12}$$

The term Provisioning in (13) uses also an api to ask the configuration module for resource allocation. Once allocated, it delegates the creation of the VDR to the term Run defined in (14). We use the abstraction of the resources information 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.

$$\text{Provisioning}(\overrightarrow{api}) \overset{\text{def}}{=}$$

$$\begin{pmatrix} \begin{pmatrix} api_r^p(Run).(\nu\, allocate)\overline{api_a^c}\langle allocate\rangle \\ |allocate(res).(\lambda\, res)Run(\ \ ) \end{pmatrix} \\ |api_s^p(suspend).\overline{api_s^c}\langle suspend\rangle \\ \left|\begin{pmatrix} api_t^p(terminate).terminate(ws) \\ .\overline{ws}\langle Stop\rangle.\overline{api_f^c}\langle terminate\rangle \end{pmatrix}\right. \end{pmatrix} .\text{Provisioning}(\overrightarrow{api}) \tag{13}$$

The suspension is delegated to the term $Configuration$ where it is represented as an invisible 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 (14) composes a vector depending on the type of the VDR that the initiator wants to create. After the creation of the VDR, it creates and sends a new identifier using the ws channel to start the new created VDR.

$$Run(ws, \overrightarrow{type}) \overset{\text{def}}{=}$$

$$\tau.\begin{pmatrix} [type_v = type_s]\ VDR(ws\hat{}\emptyset\hat{}\emptyset) \\ |[type_v = type_d]\ VDR(\emptyset\hat{}ws\hat{}\emptyset) \\ |[type_v = type_c]\ VDR(\emptyset\hat{}\emptyset\hat{}\emptyset) \end{pmatrix} \Bigg| (\nu\, id)\overline{ws}\langle id\rangle \tag{14}$$

$$Stop(\ \ ) \overset{\text{def}}{=} \emptyset \tag{15}$$

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 $\overrightarrow{ev}$ channels, an information must be sent to the monitoring module using the api_{put}^m channel. This information is stored in data vector $\overrightarrow{data}$ on the recursive call in (16) to the term $Monitoring$.

$$Monitoring(\overrightarrow{api}, \overrightarrow{data}) \overset{\text{def}}{=}$$

$$\begin{pmatrix} api_{put}^m(datum).\tau.(\nu\, id)api_{ret}^m(id) \\ |api_{get}^m(id).api_{res}^m(data_{id}) \end{pmatrix} .Monitoring(\overrightarrow{api}, \overrightarrow{data}\hat{}datum) \tag{16}$$

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's located. The VDR should be movable between the hosts based on the need. We commit our networking definition to allow VDRs across 2 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 (13) to manage the virtual network component like a VDR and hide the complexity from the user. The model allows also to bypass the scale perspective 4096 VLAN limit as proposed on VXLAN by the Internet Engineering Task Force (IETF) RFC 7348 [17]. Our model definition is composed from two terms: $vSwitch$ defined in (19) and $vRouter$ defined in (22).

For our network modelling, we define the structure of the packet transiting on the networking infrastructure. The vector $\overrightarrow{ethernet}$ in (17) 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 the message as $ip_{payload}$. The names that composes the vectors in (17) and (18) are abbreviations of header fields of the packets as described in the IETF RFC 791.

$$\overrightarrow{ethernet} \overset{\text{def}}{=} [dst, src, tag, type, \overrightarrow{ip}, check] \tag{17}$$

$$\overrightarrow{ip} \overset{\text{def}}{=} \begin{bmatrix} version, ihl, tos, len, id, flag, frag, ttl \\ , proto, check, src, dst, opt, payload \end{bmatrix} \tag{18}$$

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.

$$vSwitch\,(cntl, \overrightarrow{adr}) \overset{\text{def}}{=}$$

$$Control\,(vSwitch\,(cntl, \overrightarrow{adr}), cntl, \overrightarrow{adr}) \\ \Big| adr_i(\overrightarrow{ethernet}).\tau.\overline{ethernet_{dst}}\langle ethernet_{ip_{payload}}\rangle \\ .vSwitch\,(cntl, \overrightarrow{adr}) \tag{19}$$

$$Control\,(Target, cntl, \overrightarrow{adr}) \overset{\text{def}}{=}$$

$$cntl_{connect}(link).Target \\ \Big| cntl_{disconnect}(link).(\nu\, \vec{p}) \\ \Big(Disconnect\big(Target, \overrightarrow{adr}, (\nu\, \vec{p}), link, O\big)\Big) \tag{20}$$

$$Disconnect(Target, \overrightarrow{old}, \overrightarrow{adr}, port, i) \overset{\text{def}}{=}$$

$$[i = \|\overrightarrow{old}\|](\lambda\ \overrightarrow{adr})Target \\ \Big|[old_i = port] \\ Disconnect(c, \overrightarrow{old}, \overrightarrow{adr}, port, i+1) \\ |Disconnect(c, \overrightarrow{old}, \overrightarrow{new}\hat{}port, port, i+1) \tag{21}$$

The term $vSwitch$ defined in (19) represents the virtualization of the L2 switch. It is modelled as a congruency between a control (20) that manage the VDRs connections and a L2 network bridge.

The term $Control$ has three parameters, the first one is higher-order 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 one the vector containing connected VDRs channels. The $Target$ parameter is passed also to the term $Disconnect$ defined in (21), 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.

$$
\begin{aligned}
&vRouter\left(ipAdr, cntl, \overrightarrow{adr}\right) \stackrel{\text{def}}{=} \\
&\quad Control\left(\begin{matrix} vRouter\left(ipAdr, cntl, \overrightarrow{adr}\right) \\ \qquad , cntl, \overrightarrow{adr} \end{matrix}\right) \\
&\left| adr_i\left(\overrightarrow{ethernet}\right).\tau.\left(\begin{matrix} \left[ipAdr = ethernet_{ip_{dest}}\right] \\ \overline{ethernet_{dst}}\langle\overrightarrow{ethernet}\rangle \\ + \\ \overline{ethernet_{ip_{dest}}}\langle\overrightarrow{ethernet}\rangle \end{matrix}\right)\right. \\
&\qquad\qquad .\,vRouter\left(cntl, \overrightarrow{adr}\hat{\ }link\right)
\end{aligned}
\tag{22}
$$

The term $vRouter$ defined in (22) represents the virtualization of the L3 routing. It is modelled as a congruency between a control (20) that manage the virtual switches nor VDRs connections and a L3 network bridge. In this model, we don't illustrate some features like IP forwarding to keep our definition clear.

The management of the networking infrastructure in exposed as a part of the provisioning API. To do so, we illustrate in (23) an extension of the term Provisioning defined initially in (13).

$$
\begin{aligned}
&\text{Provisioning}(\overrightarrow{api}) \stackrel{\text{def}}{=} \\
&\left(\begin{matrix}
\cdots \\
\cdots \\
\cdots \\
\left| api^{p}_{vsCreate}(ret).(\nu\ cntl) \right. \\
\quad \left(\begin{matrix} \tau.(\nu\ \overrightarrow{adr})vSwitch\left(cntl, \overrightarrow{adr}\right) \\ \mid \overline{ret}\langle cntl\rangle \end{matrix}\right) \\
\left| api^{p}_{vsConnect}(cntl, adr).\tau.cntl_{connect}(adr) \right. \\
\left| api^{p}_{vsDisconnect}(cntl, adr).\tau.cntl_{disconnect}(adr) \right. \\
\cdots
\end{matrix}\right) \\
&\qquad\qquad .\,Provisioning(\overrightarrow{api})
\end{aligned}
\tag{23}
$$

In (12), 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 like the Switch one, the two differences is that the api^{p}_{vs*} channels are defined as api^{p}_{vr*} and the $api^{p}_{vrCreate}$ is used to create a virtual router, to keep our definition clear, we omit this part of the definition.

The previous terms are formally defined in the objective to model a new architecture of cloudlet. The definitions are useful not only for this current work but also for all software researcher in cloud computing domain.

IV. ARCHITECTURE

The definition presented in the previous section is made

$$
\begin{aligned}
&Control\left(Target, cntl, \overrightarrow{adr}\right) \stackrel{\text{def}}{=} \\
&\quad cntl_{connect}(link).Target \\
&\left| cntl_{disconnect}(link).(\nu\ \vec{p}) \right. \\
&\quad \left(Disconnect\left(Target, \overrightarrow{adr}, (\nu\ \vec{p}), link, 0\right)\right) \\
&Disconnect\left(Target, \overrightarrow{old}, \overrightarrow{adr}, port, i\right) \stackrel{\text{def}}{=} \\
&\left[i = \left\|\overrightarrow{old}\right\|\right](\lambda\ \overrightarrow{adr})Target \\
&\left| [old_i = port] \right. \\
&\quad Disconnect\left(c, \overrightarrow{old}, \overrightarrow{adr}, port, i+1\right) \\
&\left| Disconnect\left(c, \overrightarrow{old}, \overrightarrow{new}\hat{\ }port, port, i+1\right) \right.
\end{aligned}
$$

based on the state-of-art regarding the MCC research [18] [19] that converge into the Cloudlet-based MCC. The cloudlets are defined as trusted and resource-rich network computers that offer bridging capabilities to the Internet and is available for use by nearby mobile devices through a direct and well-connection. In this section, we describe our Cloudlet-based architecture by illustrating some of the technical aspects that was abstracted in the formal definition. We also link the technical implementation with their correspondent formal model. Moreover, 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 determine how data and processes are transmitted between the different components of the MCC. The Fig. 1 illustrate our vision of the MCC that is composed by three layers: the first is the Device Layer (DL) composed by physical sensor and mobile devices. The second is the Cloudlet Layer (CL) that is composed from the network of Cloudlets, each Cloudlet may contain the VDRs, Virtual Service Representation (VSR), and local services. The third layer is the Internet Layer (IL) composed by the central Cloud that contains Cloud services and needed registries in addition to Internet services like the media sensors.

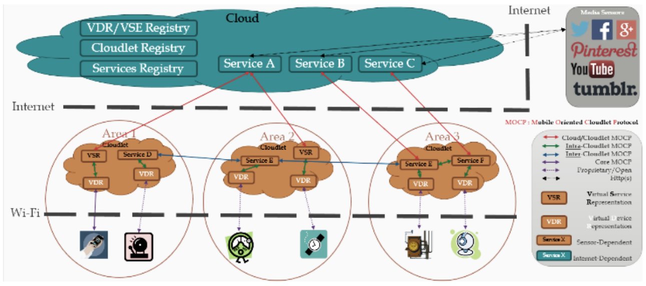

Fig. 1. Global architecture

In the CL, we define a networking infrastructure based on the NFV. As illustrated in Fig. 2, the device is connected to the VDR through a vRouter defined in (22) and a vSwitch defined in (19). The networking infrastructure is managed using the cloud orchestrator API, in our implementation model, we use OpenStack [20] that contains a powerful networking module called Neutron. Is module is based on Open vSwitch [21]. This implementation and provide a ReST [22] API for the creation and the managing of the provided virtual networking infrastructure.

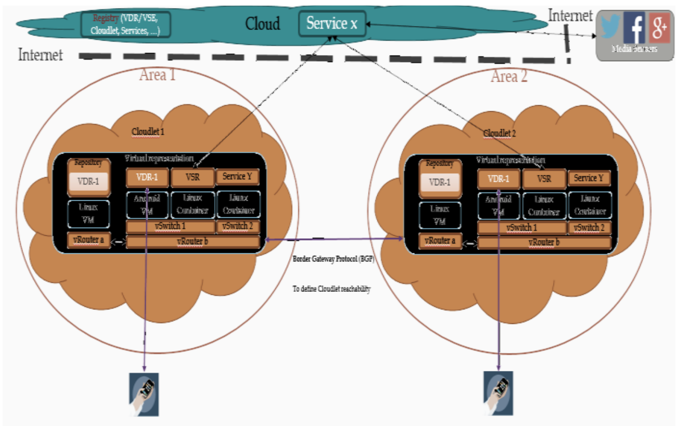

Fig. 2. Cloudlet structure and networking

B. *Mobile application offloading*

As a part of our contribution with the Mobile Oriented Cloudlet Protocol (MOCP), the formal definition of this paper focus on the communications especially used by the Core MOCP for the migration of the Apps from the physical device to the VDRs. Our implementation model, as illustrated in Fig. 3, extends the formal definition in (6) by adding technical details to the abstract definition. The two components of the VDR are the *Device Descriptor* that is modelled by the $DevId$ in (5) and the *Virtual device* is modelled in (8). The *Backend app* is modelled as the higher-order parameter $BackEndProc$ in (7). The *OSGi* [23] *container* operations are considered as non-observable operations.

Our offloading approach differs from the actual overlays oriented [24] approaches. We consider the Backend application as an ACID service that can migrate from one host to another one. Our definition of the DVDR in (8) allows the ACID properties by isolating the *Backend app* in an atomic process, which runs that makes 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 service execution. The Apache Felix [25] OSGi implementation in used in our architecture due to an Android porting effort that Apache has been supporting since the version 1.3. This mechanism works with stateless Backend services that provides a response after for the Frontend Cloudlet Android Application Package (CAPK) request, and then requires no further attention. Regarding the stateful Backend service where subsequent Frontend CAPK requests depend on the results of the first request, they are more difficulties to manage because a single action typically involves more than one request. We thus need another isolation level in top of the OSGi.

To address the issue of the state management, we use a *chroot* 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.

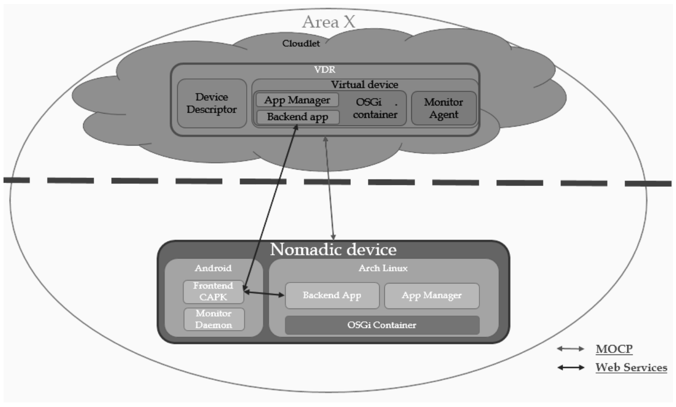

Fig. 3. DVDR implementation model

V. Case Studies

Our case of study aims to show the structural congruence between a Backend app offloaded in a VDR and the same backend app running in the device. Our objective is to illustrate that a Backend App (7) that runs in a VDR are identical up to structure 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 first define 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 (26) and (27).

The term $FrontEnd$, defined in (24), is a model of a "web view" which sends messages to the Backend using the channel ws, once the response received from the Backend, the Frontend execute another iteration as a recursion. This term has also the $touch$ channel as parameter to communicate with the user defined in (29).

$$\begin{aligned} FrontEnd(touch, ws) \overset{\text{def}}{=} \ & (\nu\, cb) \\ & touch(event).\tau.\overline{ws}\langle event, cb\rangle \\ & |cb(res).FrontEnd(ws) \end{aligned} \tag{24}$$

The term $BackEnd$, defined in (25), 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. Else, the Backend send a message containing a copy of itself to the corresponding VDR and terminate the local execution. The execution continues into the VDR after the offloading.

$$\begin{aligned} & BackEnd(ws) \overset{\text{def}}{=} (\lambda\, intra) \\ & ws(event, cb).\tau \\ & \quad .\begin{pmatrix} [intra = ws].\overline{intra}\langle\ \ \rangle \\ +\ \overline{ws}\langle App\big((\lambda\, ws)BackEnd(ws)\big)\rangle.\emptyset \end{pmatrix} \\ & \qquad |intra(\ \ ).\tau.(\nu\, res)\overline{cb}\langle res\rangle \end{aligned} \tag{25}$$

We define two parallel composition as models for the mobile devices. The first mobile device is defined in (26) as the parallel execution of a Frontend and a locally executed Backend. The second mobile device is defined in (27) as the parallel execution of a Frontend and a Backend which is configured to be offloaded to the VDR.

$$Devicelocal(ws, touch) \stackrel{\text{def}}{=} FrontEnd(touch, ws)|(\lambda\, ws)BackEnd(ws) \quad (26)$$

$$DeviceRemote(ws, touch) \stackrel{\text{def}}{=} (\nu\ local) (FrontEnd(touch, local)|(\lambda\, local)BackEnd(ws)) \quad (27)$$

To keep the clarity of our specification, we omit the details of the definition of the term $Admin$, we define just the signature in (28). It is important to note that this term send all needed messages using the vector $\overline{\overrightarrow{api}}$. It starts the networking infrastructure and the VDRs.

$$Admin(ws, \overline{\overrightarrow{api}}) \stackrel{\text{def}}{=} \cdots \quad (28)$$

The term $user$ defined in (29) 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 to have a system which can be reduced manually by a human is a reasonable time slot.

$$user(touch) \stackrel{\text{def}}{=} (\nu\ event)\overline{touch}\langle event\rangle \quad (29)$$

B. Systems

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 (30) represents the system that will give raise to a Backend offloading after some reductions.

$$SystemMig \stackrel{\text{def}}{=} (\nu\ ws) \left(\begin{array}{c} (\nu\ touch)\left(\begin{array}{c} user(touch) \\ |DeviceRemote(ws, touch)\end{array}\right) \\ \left|(\nu\ \overrightarrow{api})\left(\begin{array}{c} Admin(ws, \overline{\overrightarrow{api}}) \\ |Orchestrator(\overrightarrow{api})\end{array}\right)\right. \end{array} \right) \quad (30)$$

The term $SystemLocal$ defined in (31) represents the system that initiate a Backend after some reductions.

$$SystemLocal \stackrel{\text{def}}{=} (\nu\ ws) \left(\begin{array}{c} (\nu\ touch)\left(\begin{array}{c} user(touch) \\ |Devicelocal(ws, touch)\end{array}\right) \\ \left|(\nu\ \overrightarrow{api})\left(\begin{array}{c} Admin(ws, \overline{\overrightarrow{api}}) \\ |Orchestrator(\overrightarrow{api})\end{array}\right)\right. \end{array} \right) \quad (31)$$

C. Structural congruence

We have performed some computations steps to fully to reach a stable system starting from $SystemMig$. We call this stable state reached 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 $SystemLocal \xrightarrow{\overline{touch}\langle event\rangle,\ldots} SystemLocal'$.

Only some bound names and non-observables actions composes the difference between the two reduced systems. We have thus find that $SystemMig' \equiv SystemLocal'$.

The structural congruence is commutative and associative. We can then write:

$$\begin{array}{lrl} \text{given that} & SystemMig & \equiv SystemMig' \\ \text{and} & SystemLocal & \equiv SystemLocal' \\ \text{and} & SystemMig' & \equiv SystemLocal' \\ \text{then} & SystemMig & \equiv SystemLocal \end{array} \quad (32)$$

VI. Conclision and future works

In this paper, we present our formal definition of the MCC. This specification focus on the communications interactions on the MCC. Moreover, architectural aspects dedicated to the realization of a MCC solution are described. The case studies proof the structural congruence between offloading and local execution of a mobile application and shows the transparency of the offloading in our MCC system. On our future work, we will focus on two aspects of the MCC. 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.

Disclaimer

References

[1] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions.," Future Generation Computer Systems, vol. 79, pp. 849-861, 2014.

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

[3] A. Hosseinian-Far, M. Ramachandran and C. L. Slack, "Emerging Trends in Cloud Computing, Big Data, Fog Computing, IoT and Smart Living," in Technology for Smart Futures, vol. 53, Springer, 2018, pp. 29-40.

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

[11] 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.

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

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

[24] 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.

[25] Felix, Apache, "Apache Felix-welcome," Apache Software Fundation, 2 03 2018. [Online]. Available: http://felix.apache.org. [Accessed 2 03 2018].