TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF ABREVIATIONS

3D	Three-Dimensional space
3GPP	Third Generation Partnership Project
3GPP2	Third Generation Partnership Project 2
A	Anchor Point unit
ACK	Acknowledgement message
ARM	Advanced RISC Machine
AS	Application Server
BGCF	Breakout Gateway Control Function
C	Call Session unit
CLCK	Clock
CMW	Communication Middle Ware unit
CPU	Central Processing Unit
CS	Circuit Switched
CSCF	Call Session Control Function
CU	Computing Unit
DB	Database
DDS	Deployment Database Service
DHCP	Dynamic Host Configuration Protocol
Diah	Diameter Handler unit
DSL	Domain Specific Language
DTMF	Dual Tone Multiple Frequency

EBS	Ericsson Blade Server
EAI	European Alliance for Innovation
EM	Element Manager
ETS	École de Technologie Supérieure
ETSI	European Telecommunication Standard Institute
FM	Fault Management
G.711	ITU-T standard for audio companding
GND	Ground
GPIO	General Purpose Input and Output
GPU	Graphical Processing Unit
H	HSS Front-end unit
H.248	Media Gateway Control Protocol
HSS	Home Subscriber Server
HTTP	Hyper Text Transfer Protocol
IaaS	Infrastructure as a Service
IBCF	Interconnection Border Control Function
I-CSCF	Interrogating Call Session Control Function
IDS	Information Distribution Service
IEEE	Institute of Electrical and Electronics Engineers
IM MGW	IP Multimedia Media Gateway
IP	Internet Protocol
IPC	Inter Process Communication
IMS	IP Multimedia Subsystem

I/O	Input and Output
ISCC	IEEE Symposium on Computers and Communications
IT	Information Technology
IWSC	International Workshop on Software Clones
LDS	Load Distribution Service
LED	Light Emitting Diode
LGS	Log Gathering Service
M	Media Processor unit
ManO	Management and Orchestration
MD5	Message Digest algorithm 5
MDD	Model Driven Design
MGCF	Media Gateway Control Function
MGW	Media Gateway
MMO	Meta Manager and Orchestrator
MMTEL	Multimedia Telephony
MOSI	Master Output Slave Input
MRB	Media Resource Broker
MRF	Media Resource Function
MRFC	Media Resource Function Controller
MRFP	Media Resource Function Processor
MTBF	Mean Time Between Failure
MTTR	Mean Time To Recovery
NAS	Network Attached Storage

NFV	Network Function Virtualisation
NIST	National Institute of Standard and Technology
NPN	Semiconductor device in which a P-type region is sandwiched between two N-type regions
NSS	Node Selection Service
O	Orchestrator unit
OA&M	Operation, Administration and Maintenance
OS	Operating System
PaaS	Platform as a Service
P-CSCF	Proxy Call Session Control Function
PM	Performance Management
PNF	Physical Network Function
PNP	Semiconductor device in which a N-type region is sandwiched between two P-type regions
PoC	Proof of Concept
PS	Persistence Service
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
RFC	Request For Comments
RGB	Red Green Blue
RISC	Reduced Instruction Set Computer
RPi	Raspberry Pi
RTP	Real-time Transport Protocol

S2CT	Smart Sustainable City Technologies
SBG	Session Border Gateway
S-CSCF	Serving Call Session Control Function
SD	Secure Digital
SDL	Specification and Description Language
SDP	Session Description Protocol
SDS	State Database Slave unit
SIP	Session Initiation Protocol
SIPh	SIP Protocol Handler unit
SIPp	SIP performance testing tool
SLA	Service Level Agreement
SLF	Subscriber Location Function
SPI	Serial Peripheral Interface
SOA	Service Oriented Architecture
T	Telephony Server unit
TCP	Transmission Control Protocol
TrGW	Translation Gateway
TTCN-3	Testing and Test Control Notation version 3
UA	User Agent
UDP	User Datagram Protocol
UE	User Equipment
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

USB	Universal Serial Bus
VM	Virtual Machine
VNF	Virtual Network Function
VNFD	Virtual Network Function Descriptor
Wi-Fi	Wireless Fidelity wireless internet
XaaS	Anything as a Service

LIST OF SYMBOLS AND UNITS OF MEASUREMENTS

A	Ampere
GB	Giga Byte
GHz	Giga Hertz
kΩ	kilo-Ohm (noted as “K” in diagrams)
kbit/s	kilo-bits per second
MB	Mega Byte
ms	milli-second
TB	Terra Byte
V	Volt

INTRODUCTION

In recent years telecommunication operators have been more and more demanding for a cloud-native deployment of the IP Multimedia System (IMS) Core (3GPP, 2015) and the telecommunication platform in general. Operators have embraced the cloud for their Information Technology (IT) operations and want to deploy the telephony system on a comparable platform (AT&T, 2012).

Most current telecommunication vendors' software architecture is based on the concept of function defined, monolithic telecommunication nodes built for a given capacity based on a chosen hardware configuration (Fried et Sword, 2006). This approach is at the antipode of a cloud-based environment, where capacity is only limited by the available computing resources. Moreover, the scalability of the telecommunication networks, while possible, requires human intervention in order to deploy new specific hardware, wire them in a specific way and provision the nodes, a long and tedious process. Cloud-native deployments on the other hand are expected to provide automated scalability and elasticity on the available hardware with no or little human intervention.

Some telecommunication vendors are planning the migration of their hardware gears to a cloud platform through virtualization (4G Americas, 2014). This is the first step towards Network Function Virtualisation (NFV) (ETSI, 2014). In its first iteration, this replicates the monolithic telecommunication node architecture on an Infrastructure as a Service (IaaS) cloud platform. This approach limits the elasticity potential and bounds the scaling to thousands (or more) subscribers. It allows for decoupling from the underlying hardware but limits the deployment to specific Virtual Machines (VM). The next step requires defining a cloud-based architecture for telecommunication applications that guarantee the quality of service to which operators and subscribers are used to.

Cloud-native software architecture for telecommunication applications should provide us with a nimble system, able to adapt automatically to different service scenarios with little to

no human intervention. Such architecture should provide and an elastic system able to adapt its required resources automatically based on the current usage of the system in order to: maximize the computing resource utilization thus reducing the operating cost and, maximize the Quality of Service (QoS) thus providing better user satisfaction and loyalty.

This thesis is structured as follows. Chapter 1 presents the context and motivation for our research. It shows the lack of cloud-native solutions for the telecommunication sector and presents our objectives. Chapter 2 presents relevant state-of-the-art research and a selected set of cloud architectural patterns we intend to use with some adaptation for the telecommunications sector in the following chapters. Chapter 3 describes the software architecture framework we propose and the software architecture of the simplified IMS core application we built on top of the proposed architecture framework. It explains how we used the cloud architectural patterns and applied them to the telecommunication sector. Chapter 4 describes our experimental protocol and how our architecture and our sample application were instantiated and deployed on various hardware. Chapter 5 provides our experimental results and shows how the architecture we proposed is validated through experimentation. Finally, in our conclusion, we summarize the research performed for this thesis, our contribution and prospects for the future.

CHAPTER 1

CONTEXT AND MOTIVATION

In this chapter, we first establish the context of our research. When the stage is properly set, we present the problem statement and the issues we see on the road toward the establishment of telecommunication cloud software architecture. From there, we establish our objectives for the current research.

1.1 Context

In this section, we first take a look at the traditional telecommunication networks in order to identify their limitations to see why a cloud-based approach might be required today. We examine next the current ongoing evolution of those networks toward a virtualized deployment through the Network Function Virtualisation (NFV) standardization initiative (ETSI, 2014). This brings us to discuss the cloud, the telecommunication cloud and the cloud programming paradigm as an evolutionary step for telecommunication networks.

The requirements for a telecommunication cloud come to us from two main sources: the Ericsson software model (Ericsson, 2014), due to the fact that Ericsson is sponsoring this research and their vision of future telecommunication software should be taken into account, and the expected characteristics of cloud-native applications.

1.1.1 Traditional telecommunication networks

As the main path toward next generation telecommunication networks, the IMS (3GPP, 2015) is a standardized solution to address the need of operators to provide advanced services on top of both mobile and fixed networks. The IMS provides end-to-end services via IP-based mechanisms and is built, amongst others, on the Session Initiation Protocol (SIP) (Rosenberg et al., 2002) to establish and manage sessions and the Real-time Transfer

Protocol (RTP) (Schulzrinne et al., 2003) for the data planes. Figure 1.1 presents a simplified view of the IMS, where we have circled the main functions we consider in our research scope.

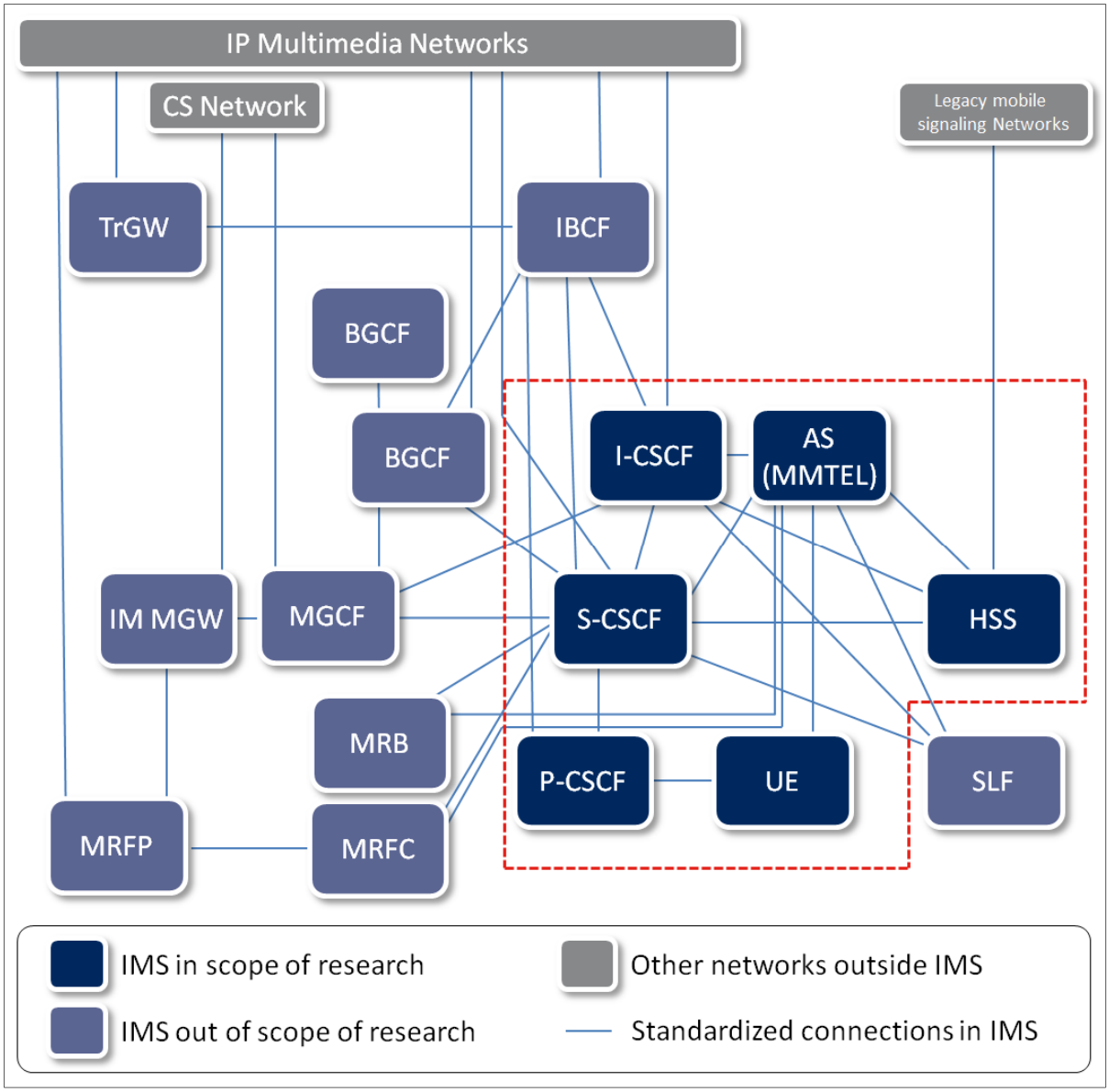

Figure 1.1 The IP Multimedia Subsystem (IMS) and circled the simplified view we consider for our research scope
Adapted from (Potvin et al., 2015a)

The main functions of the IMS are:

- Call Session Control Functions (CSCF),
- Home Subscriber Server (HSS),
- Multimedia Telephony (MMTEL),
- Media Resource Functions (MRF).

For current deployments of IMS, each of those functions is customarily deployed on dedicated physical nodes on vendor specific hardware. As an example, Ericsson has a family of hardware platforms (Ahlforn et Ornulf, 2001) for IMS deployment purposes. Figure 1.2 illustrates a possible current deployment of the core IMS functionality on server racks. Telecommunication operators deploy telecommunication networks built of telecommunication nodes made from hardware and software provided by multiple vendors in order to provide services to individual end-users, the subscribers.

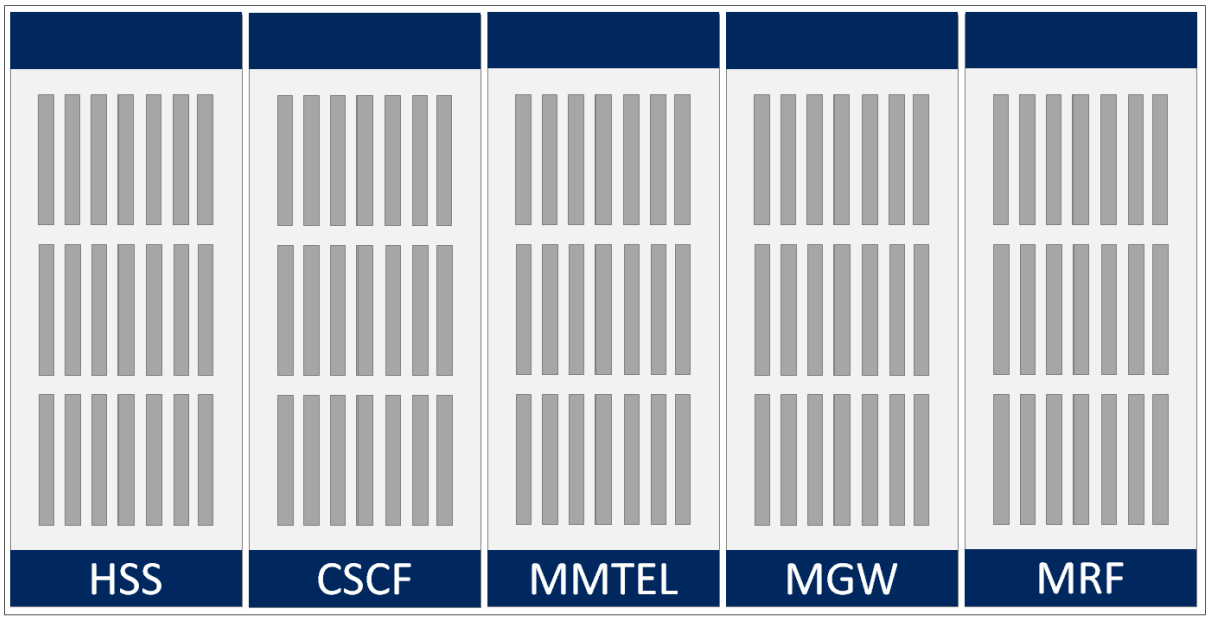

Figure 1.2 A possible current IMS deployment on server racks
Taken from (Potvin et al., 2015a)

The CSCF is in charge of exchanging the signalling between the User Equipment (UE) and the other IMS functions. Furthermore, the CSCF handles establishment and termination of the SIP sessions and provides authentication, security and monitoring. The specific tasks of the CSCF are split amongst three main specialisations:

- Proxy-CSCF (P-CSCF) that is the entry point to the IMS network from the point of view of the UE;
- Interrogating-CSCF (I-CSCF) that is responsible for identifying the network serving a user and its associated Serving-CSCF (S-CSCF);
- Serving-CSCF (S-CSCF) that is responsible for a specific user's services.

The HSS is the main database in the IMS. It keeps the profile information of all subscribers including the relevant triggers required to provide the services they are registered to.

The MMTEL provides the functionality necessary to offer telephony services such as voice calls, conference calls, redirection of calls, abbreviated dialing, etc. It supports different media content including audio, video, messaging and file sharing.

Lastly, the MRF, which is normally split in a control function (MRFC) and a media processing function (MRFP), provides the media related functions such as voice and video mixing, transcoding, etc.

The current IMS deployments suffer the consequence of being manual and are thus expensive in human capital. They also lack elasticity and as such fail to maximize computer resource usage which leads to non-optimized capital and operational expenses. With the increased demand for telecommunication services, providers have to invest further in their infrastructure. Furthermore, in order to remain competitive, the providers have to upgrade their network to provide the latest multimedia services to their subscribers. The increased market pressure to provide lower cost solutions to the subscribers adds to the operator's cost of expanding and upgrading their telecommunication network, and leads operators to look at

ways to reduce capital expenditure as well as operational expenses in order to remain competitive in the market.

1.1.2 Network Function Virtualisation

With the advent of virtualization techniques, the sharing of computing, storage and network resources has been made possible. This is probably one of the factors that made possible the creation and growth of cloud computing (Buyya, Vecchiola et Selvi, 2013). By abstracting the hardware and software, Infrastructure as a Service (IaaS) provides a pool of computing and storage resources which isolates the software from the complexity of individual hardware devices. Virtualisation also allows easy sharing of hardware devices amongst different software.

The virtualization techniques lead to the currently privileged path to address the main drawbacks of the current IMS deployments via the NFV (ETSI, 2014) standardization effort. The NFV aims at introducing virtualization platforms for telephony functions and IMS. The NFV standard defines an evolution of the currently predominantly vendor-based hardware deployment called Physical Network Functions (PNF) to a vendor-agnostic hardware platform running on virtualized hardware called Virtual Network Functions (VNF). NFV introduces the concept of elasticity for telephony application deployment, but allows a wide range of potential implementation of the elasticity concept from none to fully automated. Until very recently, the deployment of the VNF is still proposed to be executed on a per node basis and can thus provide coarse scalability and limited elasticity.

The problems associated with such coarse scalability are well covered in (Glitho, 2014) and the general problem of scaling the IMS (Hammer et Franx, 2006) is considered in (Agrawal et al., 2008) and (Bellavista, Corradi et Foschini, 2013). Namely, it leads to overprovisioning as a solution for scalability and therefore poor resource utilization derived from the node-based scaling.

1.1.3 Cloud technology

Although cloud technology is still being defined, it is easy to find ample literature about it. As a proof of initial maturity, Gartner, in their July 2015 Hype Cycle for Emerging Technologies (Anderson, 2015), have rated cloud computing as being "Trough of Disillusionment", meaning the next steps are the "Slope of Enlightenment" followed by the "Plateau of Productivity". The same is being said about hybrid cloud computing.

In the past years, two main definitions of the cloud have been widely accepted. The first of those definitions was introduced in 2008 (Vaquero et al., 2008) followed by a definition published in 2010 by the National Institute of Standards and Technology (NIST) in the United States (Mell et Grance, 2010).

From those definitions, we derive our own definition of the cloud for the purpose of the telecommunication domain: the cloud consists of a pool of shared computing resources communicating over a network, providing elasticity and an on-demand measured service with a QoS guarantee.

1.1.4 The Ericsson Software Model

In September 2014, Ericsson launched the Ericsson Software Model (Ericsson, 2014). That model emerged from noticing five trends requested by operators in the telecommunications market:

1) technology shift toward virtualization,
2) simplicity,
3) predictability of software investments,
4) network performance,
5) easy software upgrades.

The Ericsson Software Model addresses these trends by introducing three concepts:

1) value packages (selling base and value packages which correspond to a tangible benefit for the operator),

2) different software subscriptions (pay as you grow, term-based licensing and capacity buyout),

3) parameter and control (reducing manual work).

These trends and concepts should be taken into account while we develop the proposed software architecture.

1.2 Problem statement

Now that we better understand the need for the telecommunication operators to migrate toward cloud technologies we present in this section the issues and concerns associated with such a transition. Those problems are associated to two main themes: the definition of the telecommunication cloud and the application of the cloud programming paradigm to the telecommunication domain. The following sub-sections present individual problems we need to address.

1.2.1 Telecommunication cloud (telco cloud)

We said earlier that the cloud computing, in the general IT sense, is reaching the beginning of its maturity. It is not quite the same story when we look at telecommunication specific concerns in the cloud. Cloud technology in that field is really in its infancy.

The telecommunication domain brings its own set of necessities that are not usually found in the general IT cloud. IT cloud is mainly targeting world wide web (WWW) type applications (e-commerce, database backed portals for photo, music, messaging, …) or large scale analytics (search engine, machine learning, …). Among other considerations, the QoS goal for a telecommunication cloud is to provide a minimal response time for the control and setup of the different services (control plane latency), and maximum quality for the service

while ongoing (e.g. minimal jitter and latency on the media plane for a voice or video call) on an individual end user basis.

1.2.2 Heterogeneous deployments

Cloud providers usually build their cloud on homogeneous commodity hardware to reduce acquisition and operating cost. Telecommunication vendors might have to deploy their software on an operator's cloud which may very well be different from one operator to the next. Some of the telecommunication vendor software functionality could be better suited for a certain type of hardware characteristics. As such, the ability to deploy the same software in a solution-defined heterogeneous pool of computing resources is desirable.

A heterogeneous cloud still has an unclear definition. Some associate it to the cloud software stack being built from many vendors (Wellington, 2012), e.g. a management tool from one vendor driving a hypervisor from another. Others associate it to the use of hardware clusters that contain heterogeneous equipment (Xu, Wang et Li, 2011), (Crago et al., 2011), e.g., general purpose computing platforms sitting next to specialized accelerators or, mixed characteristics general computing platforms where some equipment is faster at processing, better at I/O or provides different memory/storage capacities.

1.2.3 Cloud programming paradigm

We defined earlier what cloud computing is and which characteristics we decided to focus on for this research. As we've seen, cloud computing is still a relatively new paradigm. We still have not described what the cloud programming paradigm is. As one might expect, cloud computing being new, the cloud programming paradigm is also in its infancy as a definition (Shawish et Salama, 2014). Cloud computing is a technology and can also be considered a platform. The cloud programming paradigm is the programmatic answer to how an application optimally built for the cloud computing environment should be programmed to respond to the characteristics of the cloud we mentioned earlier.

The interest here lies in identifying the programmatic constructs, techniques or approaches which enable or facilitate building an application that best answers to cloud computing characteristics. More specifically, for the scope of our research, we look at it from the angle of a telephony application point of view, using the IMS core as our base application for this research.

1.3 Research questions

As explained in the problem statement, the issues we face are those associated with two main themes: the definition of the telecommunication cloud and the application of the cloud programming paradigm to the telecommunication domain. The research questions presented in this section address those problems.

From the characteristics we mentioned while discussing the cloud and the cloud programming paradigm, we can infer secondary characteristics or contributors to the technology as mentioned in the literature (Buyya, Vecchiola et Selvi, 2013). The way those characteristics and contributors can be taken into consideration for building telecommunication cloud provides us with a number of research questions in itself. Our first four questions derive directly from that concern.

Next the need for heterogeneous deployments provides us another interesting research question. The applicability of the cloud programming paradigm to the telecommunication domain closes our list of research questions.

1.3.1 Elasticity issue

In our context elasticity should not be confused with scalability. Scalability deals with the potential to grow the number of subscribers to huge numbers. Elasticity deals with the ability to grow and shrink usage of resources on a need basis. Current telecommunication equipment deployments provide scalability. We can scale telecommunication systems to handle millions

of subscribers but those systems are not elastic. The scalability is provided via overprovisioning and when the limit is reached we need to add some number of nodes (telecommunication equipment and software fulfilling a specific function, e.g. a S-CSCF) to cope with the predicted future traffic in the network. Elasticity implies that as traffic lowers overnight for example, the excess computing resource capacity could be used for another purpose. It could be machine learning based network optimization based on the previous day acquired data, data analytics, overnight post processing etc.

In order to provide elasticity, we need to be able to allocate and use resources across the cloud on-demand as the need arises and the level of resources used should be proportional to the current demand, leaving the balance of computing resources free to be used for another "service", thus enabling resource pooling. The absence of service should not consume any resources and when a service need arises, the resource required should be swiftly instantiated anywhere on the available computing resources of the cloud platform being used.

A cloud-based application should be scalable and the scalability should only be limited by the quantity of available hardware. Software instances should be able to spread linearly on that hardware. In a telecommunication system, the instantiation of these resources should be as transparent and natural as possible to the subscriber and the operator. This should allow for operational expense reductions as unused computational resources can be assigned to a different task or even left turned off.

Q1. How can elasticity be provided to cloud-based telecommunication applications?

1.3.2 Quality of Service provisioning issue

A secondary challenge consists in studying the relationship between the cloud-provided metrics, e.g., processor load, memory load, storage usage, number of running instances, among others, and telecommunication related metrics, e.g. Quality of Service (QoS) metrics.

A telecommunication cloud application should ensure QoS based on the type of service offered on a per subscriber basis, e.g. registration and call establishment do not require the same QoS qualities, and respecting QoS on "average" cannot be accepted by the subscribers who happen to have a bad response time.

Q2. How can Quality of Service (QoS) be provided to cloud-based telecommunication applications?

1.3.3 Statefulness issue

One of the main challenges of using cloud technologies, which were by design initially developed for operations in a stateless web environment, resides in their adaptation to strictly stateful applications, such as those found in telecommunications. Specifically, the question is how a mechanism developed for the cloud architecture can be adopted by telecommunication applications dependant on a lot of state information, e.g., current SIP session or, transaction state.

Cloud-based applications should also make use of a distributed storage system, where data is spread on the cloud and where processing of the data can be performed close to where the data is located, to avoid long fetching delays. This is especially important to telecommunication applications, where latency should generally be minimized.

Q3. How can state information be maintained and provided to cloud-based telecommunication applications without adversely affecting latency?

1.3.4 Communication issue

The resources required to provide a service should be available through the cloud network. The service should be available to the subscriber no matter where he is located, as long as he has a connection to the operator cloud network. As a corollary to this, the services in the

operator cloud network should have access to each other through the operator's cloud network.

In Service Oriented Architecture (SOA) systems, part of the software is providing service functionality to other parts of the application. This type of architecture is well suited when a number of computing resources are connected over a network to provide functionality. In SOA, the units of functionality should be self-contained and allow the developers to combine pre-existing functionality to form a new application built mainly from existing services. Without forcing the architecture at this stage to be strictly SOA, the principles of SOA could be beneficial.

Q4. How can communication between services be insured across telecommunication cloud platforms?

1.3.5 Heterogeneous deployment issue

The telecommunication vendors will have to deploy their software on different cloud infrastructures, either because of telecommunication operators cloud choices or because of QoS requirements of a specific functionality. For those considerations among others, telecommunication software may have to be deployed on hybrid clouds consisting of different cloud technologies.

Q5. Can we propose a software architecture which enables writing software once and transparently deploying it on different cloud infrastructure, ultimately enabling hybrid deployments?

1.3.6 Cloud architectural pattern selection issue

The cloud programming paradigm is the programmatic answer to how to build an application optimally for the cloud computing environment. A number of architectural patterns for cloud

computing exist, but they were conceived primarily for the IT stateless cloud. Applying them to the stateful nature of the telecommunication domain may require non-trivial adaptations.

For example, in the IT cloud, load balancing is mostly done in a round robin fashion on the received HTTP requests (each request is sent to a single computing unit and as new requests come in they are sent to the next computing unit in a pool). This is possible since each request is stateless and no session needs to be maintained from one request to the next. This makes scaling of the application relatively easy, as we can instantiate more of it on the cloud and we can simply add those new instances to the pool served by the round-robin load balancer.

In the telecommunication domain, the SIP over UDP requests received for a specific session needs to be handled by the same process for the duration of the session, since state information about the session needs to be maintained. Moreover, information in the SIP request itself identifies the path that has been followed by the session, and that path should be preserved throughout the session. In such a case, a simple round-robin load balancer would not work, making scaling more difficult.

Latency is also an issue. In the IT cloud, low latency is preferred but most applications can still provide their functionality with variation in the latency. A longer page load is inconvenient but does not prevent the user from reading it. In the mobile telecommunication domain, longer latency could mean that a handoff between cell sites fails, leading potentially to call failure. This is not an acceptable situation.

Q6. Which cloud architectural patterns are applicable to the telecommunication cloud to increase its efficiency? How should they be adapted for the telecommunication cloud?

1.4 Objectives

Telecommunication vendors have not yet fully embraced cloud technologies. Some recent developments aligned with the NFV standardization effort are adapting to the cloud, but we have not yet seen a cloud-native deployment of a telecommunication system.

Our main objective is to design a cloud-native software architecture for telecommunication systems, and implement this architecture in order to evaluate its merits. In order to do so, we have to determine if we can re-architecture IMS in order to provide its functionality in an on-demand, per subscriber and per service basis through the usage of the actor model (which can be seen as a specialization of the microservice pattern).

Our specific objectives to accomplish this objective consist of the following:

O1. Propose a mechanism to efficiently allocate computing resources through multiple cloud platforms in order to prevent overloading of computing resources and the resulting adverse impact on QoS. This mechanism implements an elastic and automatic scalability scheme. To the best of our knowledge, no research has addressed this question in the telecommunication sector.

O2. Propose a mechanism to allow for application state information to be distributed on compute instances, providing resiliency while minimizing impact on latency.

O3. Propose an architecture providing portability between multiple cloud environments for telecommunication applications, enabling a solution-oriented, heterogeneous cloud deployment. The deployment could be on pools of bare-metal servers, IaaS, Platform as a Service (PaaS) or a mix of these. The architecture should enable communication between services or application components even if distributed on different platforms.

O4. Implement the architecture and measure its characteristics and to validate its performance and compare it to a traditional telecommunication node-based deployment.

CHAPTER 2

STATE OF THE ART

In this chapter we give an overview of research related to ours. In order to get acquainted with the telecommunication cloud state of the art, we take a look at the issues of scalability of telecommunication systems and how the cloud could address those issues. We follow with a look at the telecommunication cloud research and what is lacking in that area. We then look at the heterogeneous cloud research and identify what is missing in order to address the problems we identified. Finally, in order to identify the relevant cloud architectural patterns for our research, we describe the actor model and complete the chapter with an overview of some relevant software architectural patterns for the cloud.

2.1 Telecommunication systems scalability

As we discussed in our problem statement, telecommunication specific concerns in the cloud have not yet reached maturity. The telecommunication domain brings its own set of necessities that are not usually found in the general IT cloud. In order to better understand those specificities, we first look at the issue of scalability for telecommunication systems in the cloud and attempts at cloudifying telecommunication systems.

The Third Generation Partnership Project (3GPP) standardization effort has attempted to design the IMS in such a way that its core functionality is to some extent scalable as (Glitho, 2014) reminds us, although scalability does not translate into elasticity. Some work has addressed the scalability of parts of the IMS.

In (Lu et al., 2013), the authors propose a resource allocation scheme which satisfies the time requirements of a telecommunication network. They achieve those results by using static and dynamic groups for assignment of virtual computation units to physical computation units. The approach allows for coarse grained elasticity through the dynamic allocation of VMs

providing node-based functionality of the IMS core. It is an interesting foray into cloud territory but relies mainly on virtualization technology and is not a cloud-native deployment.

In (Yang et al., 2011), the authors focus on the scalability aspect of an individual functional unit of the IMS. In this case the scalability of the Home Subscriber Server (HSS) is addressed through the concepts of distributed databases, but it does not address any other nodes.

Dynamic distribution or concentration of the IMS functionality has been proposed (Bellavista, Corradi et Foschini, 2013) but still maintains node-based coarse scaling. It helps on the resource utilization front but fails to solve the overprovisioning issue. The same could be said of the "Merge-IMS" (Carella et al., 2014) approach which proposes a pool of IMS VMs holding the CSCF and HSS functionality where an instance is assigned to a subscriber at registration.

2.2 Cloud and QoS

A lot of research has been done on the IT cloud and some of it addresses the QoS issues for large scale systems. Once again in order to understand the problematic of the telecommunication cloud, we look at some of the work done for IT cloud and how QoS can be insured in the IT cloud.

Work has been done on assuring the QoS on a per user basis for IT clouds (Turner, 2013). However, the approach is applied to web technologies and specifically to e-commerce, and would require adaptations for the stateful telecommunication cloud.

In (Verma et al., 2015), the authors describe a large-scale cluster management system at Google named Borg. It explains how jobs are distributed on a large-scale cluster using master managers (the BorgMaster) and multiple instances of slave resource supervisors (the Borglets). Jobs are scheduled and deployed on the cluster by these entities based on configuration files, command line or web browser inputs. The large-scale management

concepts are interesting and we can certainly learn from their implementation. However, the batch deployment is not particularly suitable for the telecommunication domain as requests must be handled as they come and not in bundled batch. In general the whole approach would require a redesign for the telecommunication domain.

In a telecommunication cloud, different services are offered which are not differentiated at the TCP or UDP layer. In fact, differentiating those services at the SIP layer could even prove to be tricky. Moreover, the QoS guarantee can vary based on the service itself, e.g. registration does not require the same stringent QoS as a voice call establishment. The specificities of the telecommunication cloud are not widely mentioned in the literature.

2.3 Heterogeneous cloud research

One of the problems we identified is the necessity to deploy a single software base on many different cloud platforms. In this section, we take a look at research made on heterogeneous cloud in order to determine what is missing for the telecommunication domain.

Some work has been done on the heterogeneous cloud. In (Xu, Wang et Li, 2011), the authors propose a solution to schedule tasks based on their computing requirements, memory usage requirements, storage requirements, etc. to the hardware that best fit these. In (Crago et al., 2011), the authors propose a cloud built of a mix of Central Processing Unit (CPU) based computing resources and Graphical Processing Unit (GPU) based computing resources through virtualization. Another CPU/GPU study is described in (Lee, Chun et Katz, 2011) and looks at how proper allocation and scheduling on such heterogeneous clouds can benefit the Hadoop workload. All these works propose interesting ideas but are not specifically adapted to the telecommunication application space where, the definition of heterogeneous cloud is different, and would require adaptation for that domain.

2.4 Actor model

One of the problems we identified is to determine which cloud architectural patterns are good matches for telecommunication cloud software architecture. We found that the actor model presents some nice attributes that are well suited for the nature of telecommunications systems, and provides the potential to be distributed on a cloud. As a proof of suitability of the actor model, we can mention that Erlang (Armstrong, 1997), a programming language developed at Ericsson for the telecommunication domain, has been built on the principles of the actor model (Vermeersch, 2009). In this section we describe the actor model.

Hewitt (Steiger, 1973) defines the formalism of the actor model, where actors communicate through message-sending primitives. His work was geared toward artificial intelligence; however, the actor model mechanism fit well the separation of functionality found in telecommunication networks when we look at how telecommunication standards are defined.

Kheyrollahi (Kheyrollahi, 2014) abridges the definition of actors. An actor can have the following response to a received message:

- Send a finite number of messages to the address of other actors (the actors are decoupled and only know of each other via a way to address each other);
- Create a finite number of actors;
- Set the behavior for the next messages it will receive (state memory).

Kheyrollahi also explains the distinction between an imperative actor system and a reactive one. Imperative actor systems exchange messages between themselves based on a predefined knowledge of their existence. In a reactive actor system, the actors go through a message broker to forward messages to the handling destination. The reactive actor model appears more like an evolution of the actor model towards the multi-agent model.

As we can see in (3GPP, 2015) and (ETSI, 2014), telecommunication networks are based on the passage of messages between nodes. Through those messages, computing resources in the

nodes are seized or released in order to provide the telecommunication functionalities. As such, we can make a parallel between telecommunication networks and the actor model as defined previously.

The actor model is defined as a group of processes that communicate by means of messages. It provides a coherent method to organize and control many processors in parallel. Thus, it has been developed at its heart to be deployed on a platform consisting of many available computing resources and is well adapted to the cloud.

2.5 Cloud architectural patterns

As mentioned in section 1.3.6, one of the problems we identified is to determine which cloud architectural patterns are good matches for telecommunication cloud software architecture. That work has not been done previously to our knowledge. However, some architecture patterns are well adapted to cloud architectures and could be applied with adaptation for telecommunication cloud software architecture.

Wilder (Wilder, 2012) presents some cloud architectural patterns. One point Wilder makes is: "*using PaaS does not imply that the application is cloud-native, and using IaaS does not imply that it isn't. The architecture of your application and how it uses the platform is the decisive factor in whether or not it is cloud-native*" (Wilder, 2012).

In the next sub-sections we take a look at the major architectural patterns for the cloud. These patterns were extracted primarily from Wilder's book and we mention other sources where they served as inspiration. These patterns are of special interest, since we needed to use and adapt a number of them for the design of the proposed architecture, described in the following chapter.

2.5.1 Scaling axes

Three axes can be recognized in scaling an application. The axes were introduced in (Abbott et Fisher, 2009) through the concept of the scale cube. Figure 2.1 presents the three scaling axes.

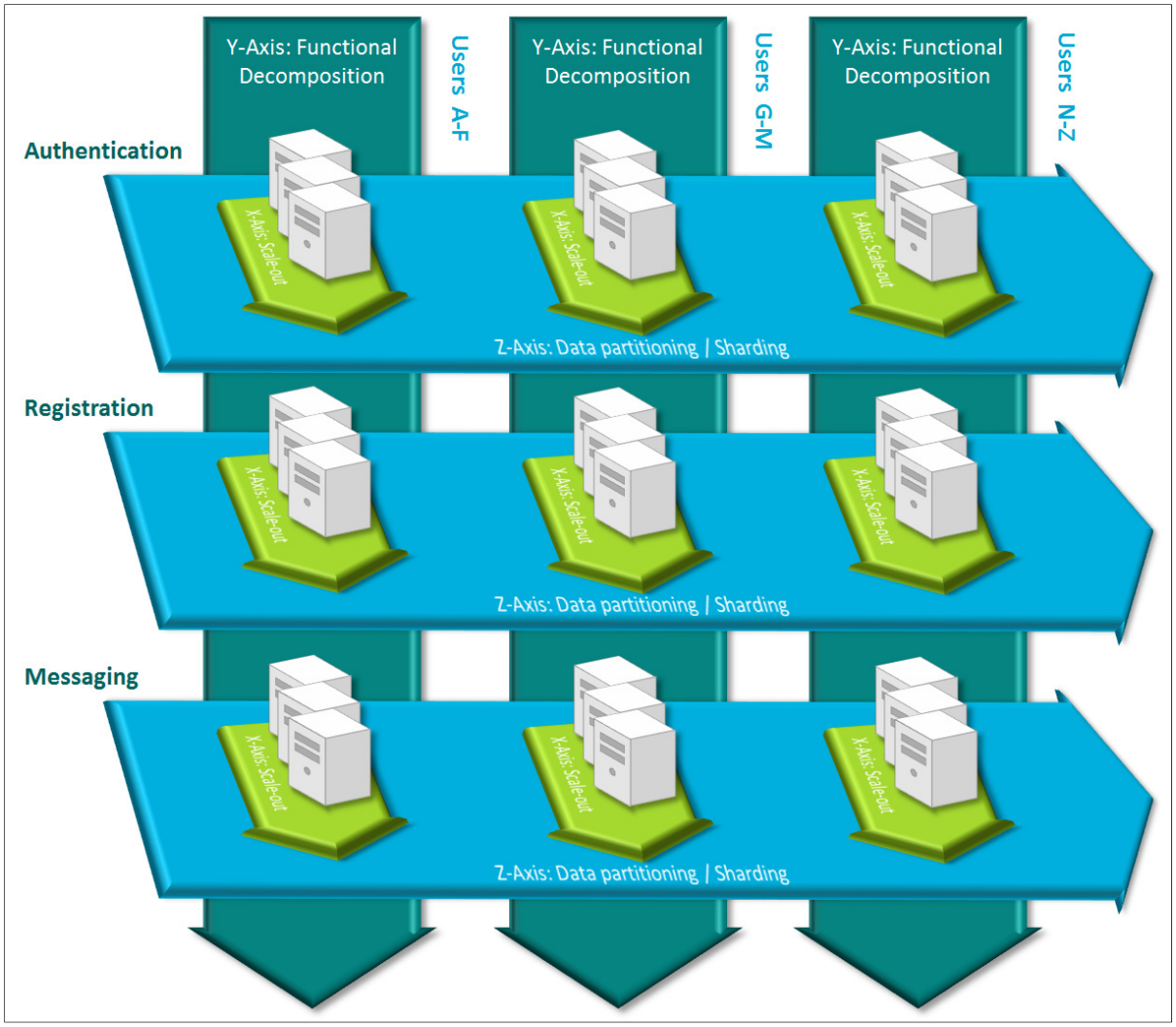

Figure 2.1 The Three Scaling Axes

The first axis, the x-axis, is known as horizontal scaling (or scale out). It consists of running multiple instances of an application behind a load balancer. The y-axis is the functional decomposition axis and scaling along this axis consists of splitting the application into

multiple different services responsible for one function. Lastly, the z-axis makes use of data partitioning or sharding. The y-axis is first discussed in the next section, and then we concentrate on the x and z axes in the following sections.

2.5.2 Microservices and actor model (y-axis)

The actor model is meant to enable concurrent computation where actors are the central primitives. An actor can create more actors, and send messages based on local decisions in response to a message that it receives. Messages are exchanged between actors via the use of addresses. As such, microservices can be seen as a subset of the actor model from the communication point of view, where each microservice can send messages based on a local decision in response to a message that it receives, but normally will not create another microservice. Messages are exchanged between microservices via the use of an address which is known or discovered via another service. Microservices are usually responsible for their own elasticity and scaling and are thus independently responsible for creating their own instances if needed. Besides this difference, microservices are also a means to enable concurrent computation. One could say that the actor model is a computation model while the microservice model is an implementation model. Microservices are well covered in (Newman, 2015).

Decomposition of an application into actors or microservices enables distributing the computational load of the application among different hardware instances or even different geographical locations. This in turns allows maximum flexibility for the management of computational resources that can deploy an actor where it best fits.

Decomposition of an application into microservices also facilitates parallel and continuous development of the individual parts. Development of one microservice can be decoupled from the other as long as strict interface definitions are maintained between the service user and the service provider. By decomposing an application into smaller services we:

- Facilitate parallel development for many developers;
- Facilitate continuous deployment of the application;
- Facilitate re-use of the microservices for other applications (this is not a silver bullet to re-use, but if combined with a good library and search engine and combined with well-defined interfaces it makes re-use simpler);
- Maximize deployment flexibility (as we get finer granularity in the deployment options).

Each microservice implements a narrow set of functionality that can be deployed anywhere and can communicate with other microservices via synchronous or asynchronous protocols, hence the benefit of having a common communication framework.

One of the main advantages with respect to the architecture we propose is the potential to deploy anywhere, thus not being bound to a specific node and allowing seamless scalability. However, microservice architecture may increase the system complexity.

2.5.3 Horizontal scaling (x-axis)

One of the first architectural patterns of the cloud is to build systems that can be horizontally scaled (in or out). The application is scaled by adding more computing resources or clones and load is shared through the usage of a load balancer. When applied to a stateless protocol such as HTTP, it is straightforward to implement such a load balancing scheme as such an architectural pattern has reached maturity for web services. Some thinking is required to apply it to the stateful telecommunication domain. This is discussed in section 3.2.1.

2.5.4 Data sharding (z-axis)

Data sharding is the approach usually proposed when scalability is required for the access of data. In a nutshell, database sharding is the concept of dividing up the data across many databases (shards). Each of the shards uses the same schema and the sum of the shard data would consist of all of the content in the original database. For instance, each row of data is located in exactly one shard. In this case, we need a shard key that identifies which shard holds the relevant data.

This concept is being used in IMS via the functionality of the Subscriber Location Function (SLF). The SLF provides information to the I-CSCF and the S-CSCF about which HSS to reach for a specific user profile. Figure 2.2 shows a simplified example on how the SLF could determine which HSS to reach based on a user URI, but in general the SLF relies on a database mapping users to HSS.

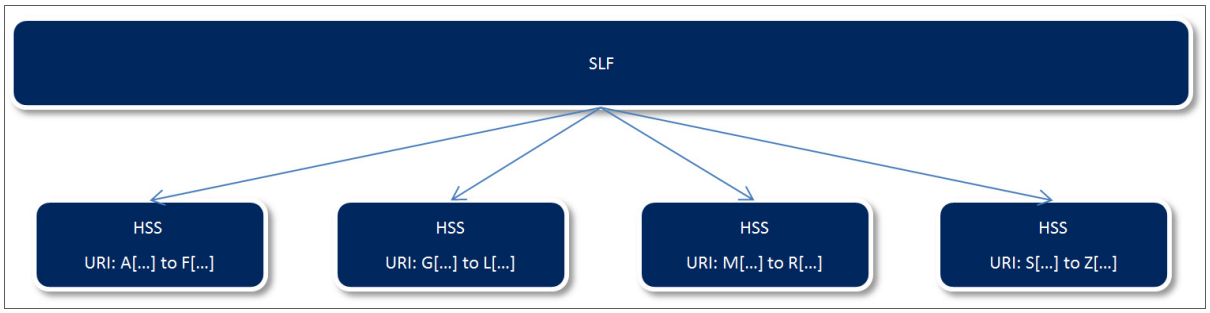

Figure 2.2 Database sharding in the IMS

This leads us to explore different approaches for a load balancer to assign the load to a computing instance. The case of a shard-aware load balancer is of special interest for us, as we discuss in the following chapter. In the next few sections, we explore shard-aware load balancer approaches.

In a stateless system, information relative to a subscriber is stored on a central database. Accessing a central database adds time to the processing of requests as information needs to

be retrieved over the network. Using a shard-aware load balancer ensures to reach the same computing instance for each request of a specific subscriber. This allows us to store locally in the memory of that computing unit (the cache) the information relative to that subscriber, hence, removing the need to retrieve that information from a centralized database. Shard-aware load balancer should always direct request to the proper computing resource. This is defined as a cache hit. Other types of load balancers may not always reach the intended computing unit. This is defined as a cache miss. In the case of a cache miss, we need to retrieve the information from a centralized database in order to populate the cache with the subscriber information. Reaching the database is performed at the expense of the time required to perform that action.

2.5.5 Round robin scheduling

Described in (Arpaci-Dusseau et Arpaci-Dusseau, 2012), the usage of a round robin or spreader is undeniably simple. However, if the goal is to reach a cache and maximize cache hits, a spreader is not a viable option, as it achieves a cache hit only *1/n* of the time (*n* being the number of processing instances), which means a cache miss occurs *(n-1)/n* of the time which, for a large value of *n*, quickly tends to one. In such a case a cache, is not an effective mechanism as most of the time, requests will result in a cache miss. As such, this is not a viable option for a shard-aware load balancer stage.

2.5.6 Modulo hashing

Hash tables are well explained in (Cormen, 2009). The concept behind modulo hashing consists in hashing a shard key and then calculating the modulo of that value by the number of computing instances available. The obtained value is the computing instance which should be chosen. For a telecommunication application we could use the subscriber URI as a shard key, a unique identifier for a subscriber. The user URI hash is mapped to a number, e.g. using MD5 we obtain a 128-bit number. Applying the modulo of the number of computing instances to that hash (Hash % *n*) allows us to determine which instance should handle the

processing for a specific user (where instances are numbered from 0 to *n*). As long as the number of instances stays constant, there is a cache hit. However, in a cloud deployment, where elasticity should be allowed through horizontal scaling, this is not the case for an extended period of time. Moreover, failures might remove instances. Obviously, as soon as *n* changes, the modulo operation will lead to a new value and each instance that is represented by a value from 0 to *n* will have to be re-mapped, leading to cache misses. Hence scaling operations will mostly lead to cache misses for most of the following requests. Since cloud systems are built on the principle of quick elasticity, this is not a viable option for shard-aware load balancer functionality.

2.5.7 Consistent hashing

Consistent hashing (Karger et al., 1997) is based on the principle of assigning a number of random values in a range from 0 to 1 to each computing instance (e.g. eight per computing instance). These are named partitions. The shard key is hashed and the resulting value is normalized in the range from 0 to 1. The value associated to a partition, thus the associated computing instance is the one associated to the closest partition of the normalized shard key. Figure 2.3 illustrates this concept with eight partitions per computing instances, or nodes, and four of such nodes, thus in total 32 (8 x 4) partitions.

As long as the compute instances do not change, consistent hashing ensures a cache hit. In case of scale-in, “nodes” in this picture are removed and the ones around them take up the space, thus only instances that were on the removed “node” will need relocation. In the case of adding new compute instances (called scale-out), *1/n’th* of the elements will be relocated to a new “node”. Assuming a uniformly distributed hashing function, this ensures that traffic amongst the compute instances is uniformly distributed.

When multiple shard-aware load balancers are present, the issue of having to propagate the “random” selection information between them still exists. The best case might use an algorithmic random function and only need to share the seed, but still a minimum set of

information needs to be shared. This could be a good way to ensure a cache hit while minimizing the synchronized information that needs to be shared between the load balancers.

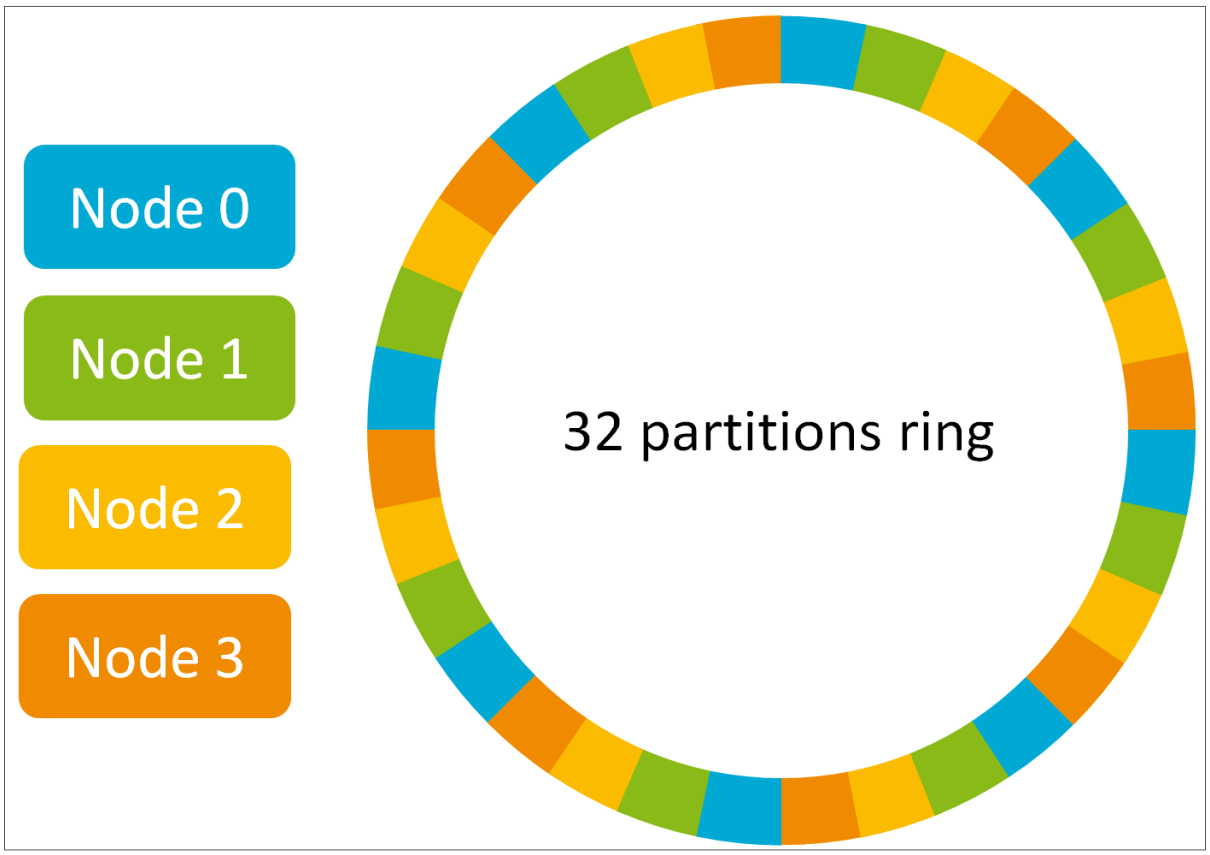

Figure 2.3 Consistent hashing illustration

2.5.8 Rendezvous hashing

Rendezvous hashing works by using the highest random weight mapping technique introduced in (Thaler et Ravishankar, 1998) and can be summarized as follows:

1) A shard key (key-m) is calculated. For example, for an originating side SIP-based service chain we could use the from-URI as an instance key;

2) The computing instance keys (key-c) are calculated to identify the computing instances uniquely. For example, it could be the IP address and port number of the computing instance;

3) For each computing instance, the hash of the concatenated key-m with key-c is calculated;

4) The computing instance with the highest hash value is selected as the instance to process the received message.

Specifically:

$$selected\ node = \underset{node_i}{\arg\max}\ \mathcal{H}(< URI > + < host_i >) \quad (2.1)$$

Where $\mathcal{H}(.)$, $< URI >$ and $< host_i >$ represent the hash function, the originating side URI (from-URI) and host identification (e.g., host name or, IP address and port) of the computing node, respectively. Moreover, + denotes the concatenation operation for two strings.

By using always the subscriber URI, and if no computing resource is added or removed, messages from a specific SIP-URI will always hit the same computing resource. If a computing instance is added or removed (assuming a uniformly distributed hashing function) *1/n'th* (*n* being the number of compute instances) of the SIP-URIs will be relocated to a new home.

With rendezvous hashing, no information needs to be shared between the load balancers as the computing unit selected to service the request is programmatically computed. Such an approach may be the best choice of a shard-aware load balancing algorithm in many practical cases.

2.5.9 Auto-scaling and busy signal pattern

In order to obtain a system which does not rely on human intervention to scale and is ready to accept variation of the traffic inflow, there is a need to automate the scaling process. In the previous section, we described how a sharding technique could be used to scale the

computing capacity. The goal of auto-scaling is to ensure scaling of the computing capacity based on decision factors that can be automated.

In the IT cloud, QoS is usually provided in a best-effort fashion and individual QoS may vary based on the overall load of the system. As such, metrics on the CPU usage, the number of messages being processed or the available memory are usually sufficient indicators to determine if scaling is required. While a scaling operation is performed, the busy signal pattern dictates that overload traffic can be discarded with an appropriate response code.

Obviously for the telecommunication domain, we have to adapt the concept, as dropping services or calls is unacceptable.

2.5.10 MapReduce principles

The MapReduce pattern is a cloud architectural pattern derived from functional programming. In this pattern, we distribute the processing of a large set of data on multiple computing resources. The first step is done by a map function that is distributed and takes a portion of the large set of data and "extracts" some information from it or applies some algorithm to it. Then the mapped data is sent to a reduce function which takes all of the mapper's individual results and collates them together in final form.

There are a number of variants to the MapReduce pattern:

- Map only: we distribute the map function and the output is directly the desired outcome, without reducing required;
- Classic MapReduce: as explained previously;
- Iterative MapReduce: the output of the reducer function generates data for another cycle of MapReduce. This can go on until the criterion to exit the MapReduce iterative cycle is reached.

In our solution, we explore how the MapReduce principles could be used for some aspect of a telecommunication cloud.

2.5.11 Node failure

Traditionally, telecommunication equipment vendors have worked hard to ensure a high Mean Time Between Failures (MTBF) figure by using high-end, reliable components or through the deployment of redundant hardware. This is how the “five nines” performance figure was guaranteed and it implied that when a component failed, a technician should replace it as early as possible to ensure continued redundancy and thus availability in the future. This was made a necessity because most of the data paths were a single chain of hardware and software; if the chain broke, it needed an available alternate path.

With the advent of the cloud, where commodity hardware is used, we need a shift toward a Mean Time To Recovery (MTTR) focus. In such a mindset, you need to ensure your processing is as distributed as possible and can run anywhere, such that if a hardware computing unit fails or some software processes fail, further processing can continue on another computing unit or in another software process. This way, the failing hardware could be replaced as time permits or not at all, and no effect would be noticed on the overall system.

2.5.12 Collocate pattern

Collocating functionality on the same computing resource or geographically close computing resources helps reduce network latency. The “Merge-IMS” (Carella et al., 2014) approach discussed earlier is based on this pattern. Although this approach has the limitations of providing coarse scaling, the collocation principle could be retained for our proposed architecture.

2.6 Discussion

Although the cloud is still being defined, a lot of work has been done covering the IT cloud. We have seen in the previous section that most of the research related to the IT cloud cannot be directly applied to the telecommunication cloud and would require adaptations.

Some research has been done on what could be called the telecommunication cloud. However, in those cases only the virtualization path has been explored where existing telecommunication functionality is deployed on an IaaS cloud. There is no telecommunication cloud native research to our knowledge at this point in time.

The heterogeneous cloud approach could be beneficial to the telecommunication domain as telecommunication vendors might very well have to deploy their telecommunication functions on various types of infrastructures or even a hybrid deployment. However the heterogeneous cloud research is concentrated on a more traditional, batch processing type of request or more common IT cloud applications. Thus the telecommunication cloud concept has not yet been explored to our knowledge.

Finally, the actor model seems a good fit to the general landscape of telecommunication applications. At the same time, a number of cloud architectural patterns have good potential to be used with some adaptation in order to produce cloud-native telecommunication applications. Hence, we can see good prospects in defining and evaluating an architecture and providing a platform for the development of cloud-native applications in the telecommunication domain. This could cater to the need for a heterogeneous deployment.

CHAPTER 3

PROPOSED ARCHITECTURE

In order to propose a suitable architecture we first look at the requirements we can derive for that architecture and make an analysis of those. We then explore the patterns we presented earlier and see how they can be applied or adapted to the telecommunication cloud domain, and we show how we implemented them in our proof of concept. We then proceed to introduce the pouch concept we came up with in order to address resource allocation issues on heterogeneous platforms and explain how it is applied in our proof of concept. Afterwards we dive into the software framework, the communication principles and our answer to the delays introduced by starting actors. At this point we are ready to present a high level view of the proposed architecture and describe all its elements. We then present the elements of the telecommunication application we built on top of the proposed architecture. We conclude with a discussion about the proposed architecture.

3.1 Requirements for the proposed architecture

In order to address the trends and concepts proposed by the Ericsson Software Model we discussed earlier, and in order to get the most benefit from the cloud programming paradigm, the architecture we propose should address a number of requirements.

R1. Decoupling software from hardware: usage of virtualization technologies and other cloud technologies should be incorporated in the architecture. It is especially important for the telecommunication vendor to avoid a situation of platform lock-in, as they migrate from a hardware platform native solution to a cloud solution. To that effect, we propose a heterogeneous architecture where an application can be built once and then deployed on any cloud technology whether it is IaaS, PaaS, bare metal server pools, public cloud or private cloud.

R2. Simplicity: in order to do so we should first define what is simplicity. From the telecommunication vendor's point of view, simplicity is a software product line which can be easily configured to address the needs of many different operators (customers). From the operator's point of view, enabling specific features should be as simple as purchasing the license for it and enabling it in the network. The configuration should be minimal and kept simple for the operator. Hence the architecture we propose enables the production of different solutions for specific markets or different packages through parameterization. Our architecture makes the development of a service simpler, as service chains can be developed, deployed and maintained independently. Our proposed architecture makes capacity increase simple by defining new pools of resources available for the computing platform.

R3. Predictability of the operator's investment on the deployed hardware and software: through its cloud principles, our architecture offers predictability on investments, as capacity is determined by available hardware and the software cost can be adapted to any of the Software Subscriptions mentioned in the Ericsson Software Model. Since our proposed architecture is based on the actor model and has a coarse granularity, we could sell software on a pay-per-use model base. Furthermore, our architecture hardware usage should vary linearly with the number of subscribers served, in order to easily predict the required investment.

R4. Elasticity: so that as the service usage changes, resources do not stay locked providing an unsolicited service.

R5. Observability: in order to assure problems in the network or bugs in the software can be quickly and properly identified. Our architecture provides a fine granularity, e.g. each service chain being independent enables observability of the individual service provided to a subscriber or any aggregation level of the information thus providing maximal observability.

R6. Enhancing network performance: making it easy to upgrade the system. Our architecture allows for the upgrade of a single service chain. This makes it possible to

upgrade a single subscriber for test purposes, then once the new version is proven, to upgrade a ratio of random subscribers, e.g. 5% of subscribers. As the performance indicators validate the new version, this ratio can be increased to full deployment.

R7. Automatic upgrade: so the system can determine the valid upgrade paths, which versions of software are valid for which service chains, have data be transformed for new versions on the fly, etc. At the same time, the solution, being cloud-native, must adhere to the way resiliency is achieved in the cloud. Traditionally, resiliency in telecommunication networks was achieved through maximizing the MTBF. In the cloud, resiliency is normally achieved by minimizing the MTTR.

In summary, a number of characteristics point us toward the selection of an architecture based on the actor model as described earlier:

- Being able to distribute processing independently of the hardware;
- Having the processing instantiated with per-service granularity as needed, guaranteeing QoS on a per service level;
- Having a high level of autonomy for each service instance;
- Having the possibility to split the functionality in parts that can be re-used for different services in a service oriented architecture.

In order to address the aforementioned requirements, we settled on developing a heterogeneous and scalable software architecture. A heterogeneous software architecture allows software to be defined once and deployed on varying pools of hardware, varying pools of virtualization platforms (IaaS) and varying pools of cloud development and development platforms (PaaS). The deployment on these pools can be performed concurrently on a mix of pools (hybrid cloud) or only on one pool. Thus the architecture we propose can be deployed on a set of different hardware infrastructures, using a mix of management tools and a mix of deployment technologies. In other words, part of the deployment may be on Virtual Machines (VMs), on containers (e.g., Docker (Docker, 2016) or, Apcera Cloud Platform (Apcera, 2016)) and on bare metal servers, to take advantages of

the various platforms and their availability. Our goal is to build a system and software architecture which allows a single software base to be deployed on heterogeneous hardware and cloud platforms. Specific requirements are met through deployment configuration rather than a design for a specific platform set. A Scalable software architecture allows software processes to scale in or out (elasticity) with a linear resource utilisation, where the only limitation is the available hardware, thus allowing scaling of the solution by simply adding additional hardware in existing or new pools.

As the proposed software architecture is primarily meant for telecommunication-based applications we split our work in two main concerns:

1) Developing and building a heterogeneous and scalable framework to support telecommunication applications in the cloud;

2) Developing a telecommunication application on top of the framework to demonstrate the architecture.

Building a simplified IMS system on a microservices-based architecture (Newman, 2015) allows us to study the characteristics of the heterogeneous and scalable cloud software architecture we are proposing. This gives us the flexibility to distribute the IMS functions on a combination of platforms, through a descriptor file which defines the available pools of platform resources and the deployment model of the microservices. The Meta Manager and Orchestrator we built can deploy the functionality on heterogeneous platforms. This approach allows defining hybrid deployments since the defined platforms could as well be provided by a public cloud.

The list of microservices developed for the framework and the IMS functionality implemented is detailed in later sections. We first focus on the software architecture and infrastructure enabling a heterogeneous and scalable cloud deployment. We first revisit some of the cloud architectural patterns we reviewed earlier and explain how we use them in the proposed architecture and the proof of concept we built.

3.2 Application of architectural patterns

As detailed in our research question Q6, need to identify which cloud architectural patterns are applicable to the telecommunication cloud to increase its efficiency, and establish how they should be adapted for the telecommunication cloud. This section looks at the selected architectural patterns, their adaptation and their application in our proof of concept.

3.2.1 Application of horizontal scaling and sharding patterns

The application of the horizontal scaling and sharding patterns enables us to address objective O2 by proposing a mechanism to allow for application state information to be distributed on compute instances, providing resiliency while minimizing impact on latency. This answer as well research question Q3 and provide a mechanism such that state information can be maintained and provided to cloud-based telecommunication applications without adversely affecting latency.

As described before, horizontal scaling and sharding are well established in the stateless environment of the web. However, the stateful nature of telecommunication applications, where usage of session-based stateful messaging is common, requires exploration of new ideas.

We applied data sharding principle to an IMS system where each computing node of the system, which is a synchronized cache, contains a subset of the HSS database. For instance, as in Figure 3.1, consider that the Core IMS is distributed based on the first letter of the SIP URI (sharding) and the load balancer is aware of the shard's location.

Each of the shards contains an in-memory cache (synchronized with the HSS) of the user profile information for a number of subscribers. The load balancing function is aware of the sharding algorithm and can determine the key to hit the proper shard. This way, the

information contained in the HSS is spread to in-memory cache locations on individual computing units which can provide the functionality via requests to that cache.

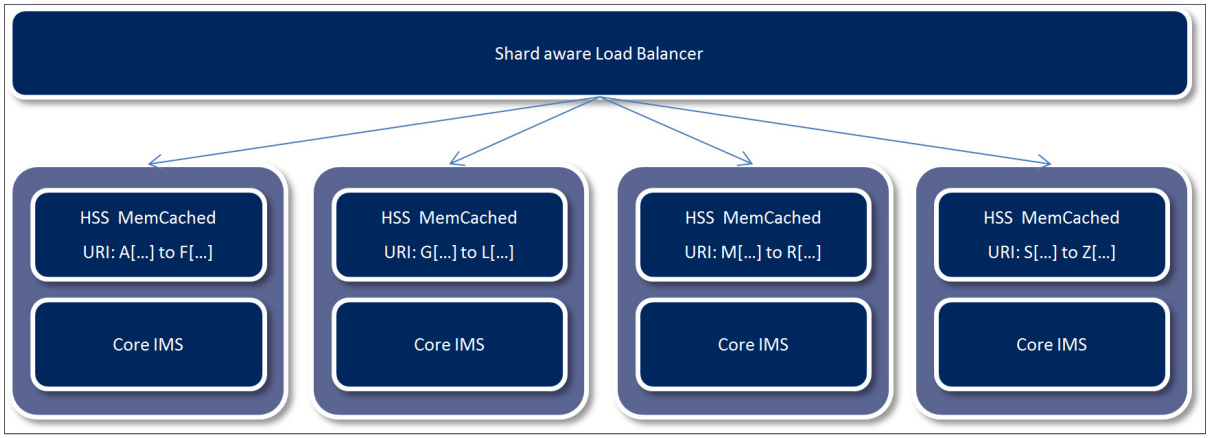

Figure 3.1 Cache sharding

Using an efficient and proper sharding algorithm, it becomes possible to perform horizontal scaling by adding new shards. In case of failure of one of the computing resources, the cache can be re-spawned on a new computing node incurring a one-time cost of rebuilding that cache at a new location. The same principle applies in the case of congestion of a computing node, where users can be migrated to another node for that same cost.

The underlying principle is the distribution of data and its in-place processing. This must be done while maintaining the ability to fully distribute the data and processing on the whole cloud.

The same principles can be applied for application state information which is not necessarily stored in the HSS. For example, assuming you are building on a microservice-based architecture, each of your service streams should have a way to horizontally scale. Moreover you probably want to be resilient to faults which could bring down computing instances.

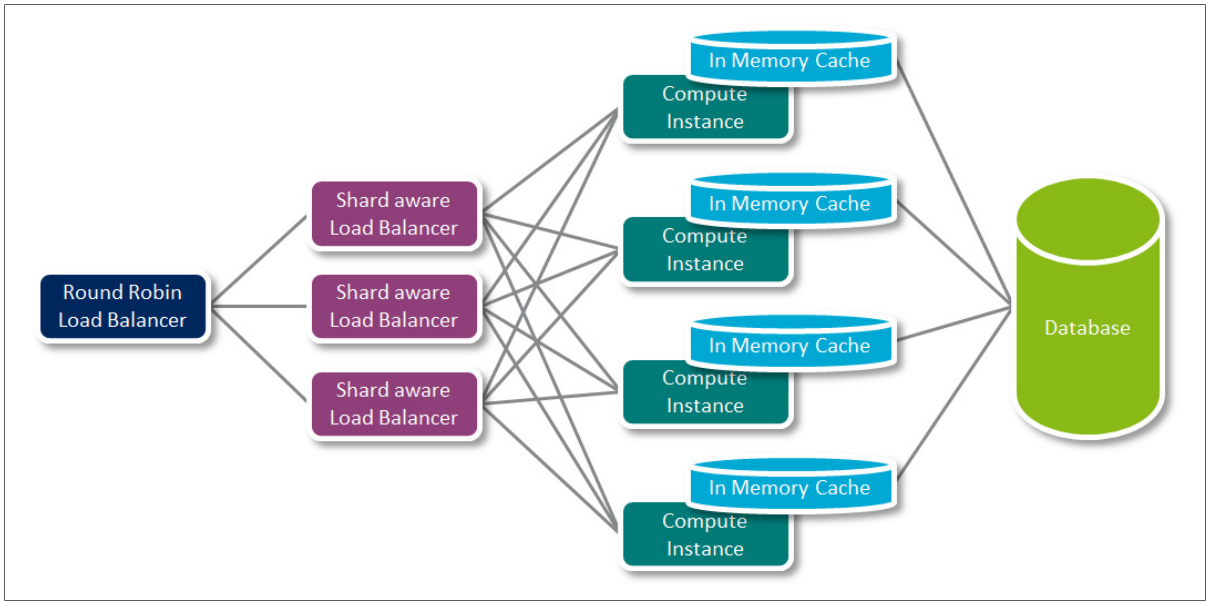

Figure 3.2 Cache sharding generalized

Figure 3.2 depicts a good architectural pattern for that task. The process follows the next steps:

1) A simple round robin load balancer, which requires minimal processing on the messages, dispatches the messages to a shard-aware load balancer;

2) The shard-aware load balancer dispatches the messages to the appropriate computing instance, maximizing cache hits. Different requests for the same user go to the same computing instance;

3) On a per-message basis, the in-memory cache content is fetched and the message is processed. Changes to the session are stored in the in-memory cache;

4) When updated, the in-memory cache pushes its data to a database which could be distributed and replicated for resiliency (or it could be as simple as a number of replicated caches on other compute instances). That database should ideally be optimized for write operations, as we mostly use it to write. We only need to read in case of re-allocation of computing to another compute instance. This can occur if one compute instance fails and we need to redistribute its traffic.

The main element is to enable the processing on a per-message basis. As messages are processed, the changes they impart on the session state are stored in the in-memory cache. We use this in-memory cache to avoid the latency of accessing a remote cache or database. However, using a collocated in-memory cache does not protect us from failure of the

compute instance. Thus the need to push that cache data to a resilient database from which state information will be fetched in case of such a failure.

State information is maintained through in-memory cache and resiliency is insured by the database (or information replication/distribution service). Horizontal scaling is made easy by adding computing instances with their associated in-memory cache. Usage of stateless shard-aware load balancers allows us to add as many as required to cope with the load. From our survey of significant hashing algorithms, we can conclude that a Rendezvous hashing based shard-aware load balancer can provide such characteristics.

The pattern we describe here is implemented through a number of services in our proof of concept. The round robin load balancer functionality is implemented by the Load Distribution Service (LDS). The shard-aware load balancer functionality is implemented by the Node Selection Service (NSS). The computing instances are deployed as pouches, a concept we introduce shortly. The in-memory cache is implemented by the State Database Slave (SDS), and finally the Database itself is implemented by the Persistence Service (PS).

We also use this pattern to store in in-memory cache the information retrieved from the HSS. In that case the computing instance selection is performed as previously described. The in-memory cache is implemented by the HSS Front-end (H) unit described lower. The database is the HSS itself, accessed via the Diameter Protocol Handler (Diah) or, for testing purpose, the Database (DB) unit.

The conventional approach in IMS for handling multiple instances of the HSS deployed for different user profiles requires the interrogation of a Subscriber Location Function (SLF) to direct the calls to their respective HSS (Poikselkä et Mayer, 2013). Compared to the SLF mechanism, the proposed approach minimizes the number of over-the-network queries to the HSS, thus minimizing the latency of the overall system.

3.2.2 Application of auto-scaling and busy signal patterns

The application of auto-scaling and busy signal patterns allow us to address objective O1, to propose a mechanism to trigger efficiently the allocation of computing resources through multiple cloud platforms, in order to prevent overloading of computing resources and the resulting adverse impact on QoS.

In order to obtain a system which does not rely on human intervention to scale and is ready to accept variation of the traffic inflow, there is a need to automate the scaling process. In the previous section, we described how a sharding technique could be used to scale the computing capacity. In this section, we discuss when a new shard should be added.

One of the differentiating factors of a web-based service and a telecommunication grade service is the QoS assurance. In the IT cloud, QoS is usually made in a best-effort fashion and individual QoS may vary based on the overall load of the system. However, on average the QoS requirement should be met. In a telecommunication grade system, the guarantee to meet the QoS requirements should be stricter. They should guarantee QoS on a per service basis, e.g. each individual call should meet QoS requirements.

One way to ensure such strict QoS requirements are met is to perform QoS monitoring on an individual service basis, on a limited number of metrics per computing resource. For example, we could think of control plane signaling latency or data plane jitter measurement, etc. Not meeting the QoS requirements for a small number of services for a short period of time can then be used as a signal that a scale out is required for the traffic on that computing resource. No new traffic would be admitted and could instead be temporarily redirected to an overflow-handling resource or simply discarded while the scale-out operation takes place.

We performed some experiments where the decision was purely based on CPU usage, but we concluded that better results can be achieved by having a set of metrics that includes, without being limited to, CPU usage. In order to maintain a generic load balancing scheme, a simple

load indicator could be provided by all computing nodes (possibly via the IaaS or PaaS infrastructure) and individual computing resources could compute a more sophisticated metric. Upon the reception of a new request, if the computing units violate the acceptable load range, they would reply to the load balancer with a busy signal. This busy signal would trigger the load balancer to forward the request to an overflow-handling computing node. This way the load balancer logic stays generic and the specifics of the QoS requirements can be handled by the individual computing nodes and the applications that run on it.

The principle here is to allow dynamic scaling while ensuring QoS for the subscribers. The main remaining question is how the shard-aware load balancing function selects a computing node. How could the busy signal be handled with Rendezvous hashing? We made the choice for our proof of concept to trigger a scale out operation, but this may take time, hence the need for a predictive algorithm to perform scaling before the need arises. Our current implementation simply looks at the load metrics (CPU and memory usage, network processing delays and number of unit instantiated) in order to make a decision to scale-out or scale-in, based on configurable rules and thresholds put in the descriptor file read by the MMO at system start.

The following QoS metrics have been considered in our architecture:

- round trip communication delays,
- processing load on the distributed computing nodes (CPU usage and memory usage),
- history of QoS experienced by users for their subscription policies.

In order to implement the auto-scaling and busy signal patterns, we distributed the functionality as follow in our proof of concept. The platform framework deployed with each pouch reports its load figures through the Information Distribution Service (IDS) Client to the IDS, which implements publish-subscribe functionality. The Meta Management and Orchestration (MMO) is subscribed to the load figure and receives that information. The MMO can then assess the need for scale-in or scale-out based on rules and thresholds configured in the descriptor file it reads at system start.

3.2.3 Application of the MapReduce pattern

The application of the MapReduce pattern supports the fulfillment of objective O4 to implement a software architecture for the telecommunication cloud, by providing a way to efficiently distribute certain processing tasks on the cloud.

As stated earlier, the MapReduce pattern is well adapted at processing large sets of data. As such in our architecture implementation it could be well suited for usage in:

- Log aggregation: log files can be locally generated, but should not be aggregated all the time in order to limit bandwidth and storage usage. Computing resources can compile local logs based on classical log levels. When they receive log commands to provide some of those logs, they can map the command to their local log file, i.e. filter the local log file for some properties. The result can then be sent to the log aggregator, which reduces all the received logs for a certain map (or filter), order them and present them to the requester;
- Performance Management (PM) and Fault Management (FM): in the same way we apply MapReduce for logging, it could be applied to PM and FM monitoring and collection. Instead of being on-demand, the mapping could be asked for at regular intervals and reduced for presentation on an Operation Administration and Maintenance (OA&M) interface;
- Analytics: since the subscriber data would already be spread over a number of computing resources, and since usage data could be kept as well in-memory cache on those computing nodes, analytics requests could be mapped to the cloud and then reduced for consumption.

These are a few of the potential uses of the MapReduce pattern. Further study is required to see how off-the-shelf MapReduce packages (e.g. Hadoop) could be used efficiently for that purpose.

We have implemented the MapReduce pattern in our proof of concept for the Log Gathering Service functionality. Via the console, it is possible to instruct the Log Gathering Service of the level of logs desired (INFO, WARNING, ERROR, etc.) and some filters, e.g., a specific keyword in the log or, a specific service chain. This information is then mapped to all pouches, which only report back the required information to the Log Gathering Service which acts as a reducer as well, ordering the information based on time stamp and presenting it to the user on the console or in log files.

The other potential usage of the MapReduce pattern (Performance Management, Fault Management and Analytics) could be considered in future work.

3.2.4 Application of the collocate pattern

The application of the Collocate pattern supports the fulfillment of objective O4, to implement a software architecture for the telecommunication cloud by reducing latency in the service chain, thus allowing for telecommunication grade expected response time.

Collocating functionality on the same computing resource or geographically close computing resources helps reduce the network latency. Some initial trials of our solution showed us that the full distribution of the functionality over a cloud cluster had worse latency response than when it was functionality collocated on a single physical host. We also quickly looked at the possibility of using alternative communication procedures in the communication middleware (thus transparent to the application) when communication occurs within a physical host (usage of memory buffers or Inter Process Communication (IPC)). These approaches should be further investigated as in our current implementation we rely on network communication.

We implemented the Collocate pattern by allowing application defined units to specify their preference of collocation. Hence in our proof of concept the Call Session (C) unit, Orchestrator (O) unit, the Anchor Point Controller (A) unit and the Telephony Server (T) unit

express to the framework their preference to be collocated and are hence normally started together.

3.2.5 Application of the actor model

The application of the actor model is the first corner stone that allows us to fulfill objective O3, to propose an architecture for telecommunication applications providing portability between multiple cloud environments, enabling a solution-oriented, heterogeneous cloud deployment.

In order to address the needs mentioned earlier and specifically to address the need to allow building software for which capacity is only limited by the available hardware in the cloud, we settled on an actor-based architecture. The telecommunication application (in this case a simplified Core IMS we describe in the following sections) functionality is split amongst a number of communicating actors linked together to form a service chain (or call chain) that is formed for the sole purpose of serving one subscriber for the current service.

An actor can be seen as a small program which handles only one aspect of the service chain. The smaller and more well-defined the actors, the easier it is to recombine them for a new purpose or a new behavior in a different service chain. This behavior shows the link with the microservices architecture we mentioned earlier. A unit is a piecemeal sized portion of a complete service offering. You can think of a unit as a microservice. Being actor-based also gives us per-service and per-subscriber granularity and provides us with the fine granularity we are seeking.

Since our architecture is implemented as a development framework for heterogeneous cloud-native applications, we are calling individual actors “units”. Units in our architecture have various lifespans and we categorize them according to their lifespan.

The first class of units is the ephemeral units, since once the service they were invoked to perform is completed, the units are released and destroyed, releasing all resources held for that service. For example, in the current implementation our orchestrator, the O units are ephemeral units and as soon as they have provided the information they were required to, they get released.

The second class of units is the dialog units. The dialog units live for the duration of a subscriber service, e.g. a voice call. They are normally created by permanent units (described next) in the system and form what we have called earlier the service chain. Once that service is completed, those units are released and destroyed. Examples of those units are the C units (more or less equivalent to CSCF functionality), the T units (handling the MMTEL functionality) and the M units (performing the MRF functionality).

The third and last class of units is the permanent units. Permanent units have a longer lived duration, usually serving the system as a whole, for example, LDS (Load Distribution Service) units or, DB (Database) units are always executing. Such units will be shut down only in case we need to upgrade them.

In general, ephemeral units provide a service to a dialog but do not need to outlive that service. Dialog units serve a single subscriber for the duration of a specific service only. Permanent units serve functionality required to keep the system up and running and are of use for a short duration by most of the subscribers in the system.

Figure 3.3 shows the units involved in a two-way call between subscribers in order to illustrate how units are linked in a service chain. It shows a typical two-party call using the proposed architecture. Units in the call chain serve a single subscriber service and they can exchange messages following the actor model.

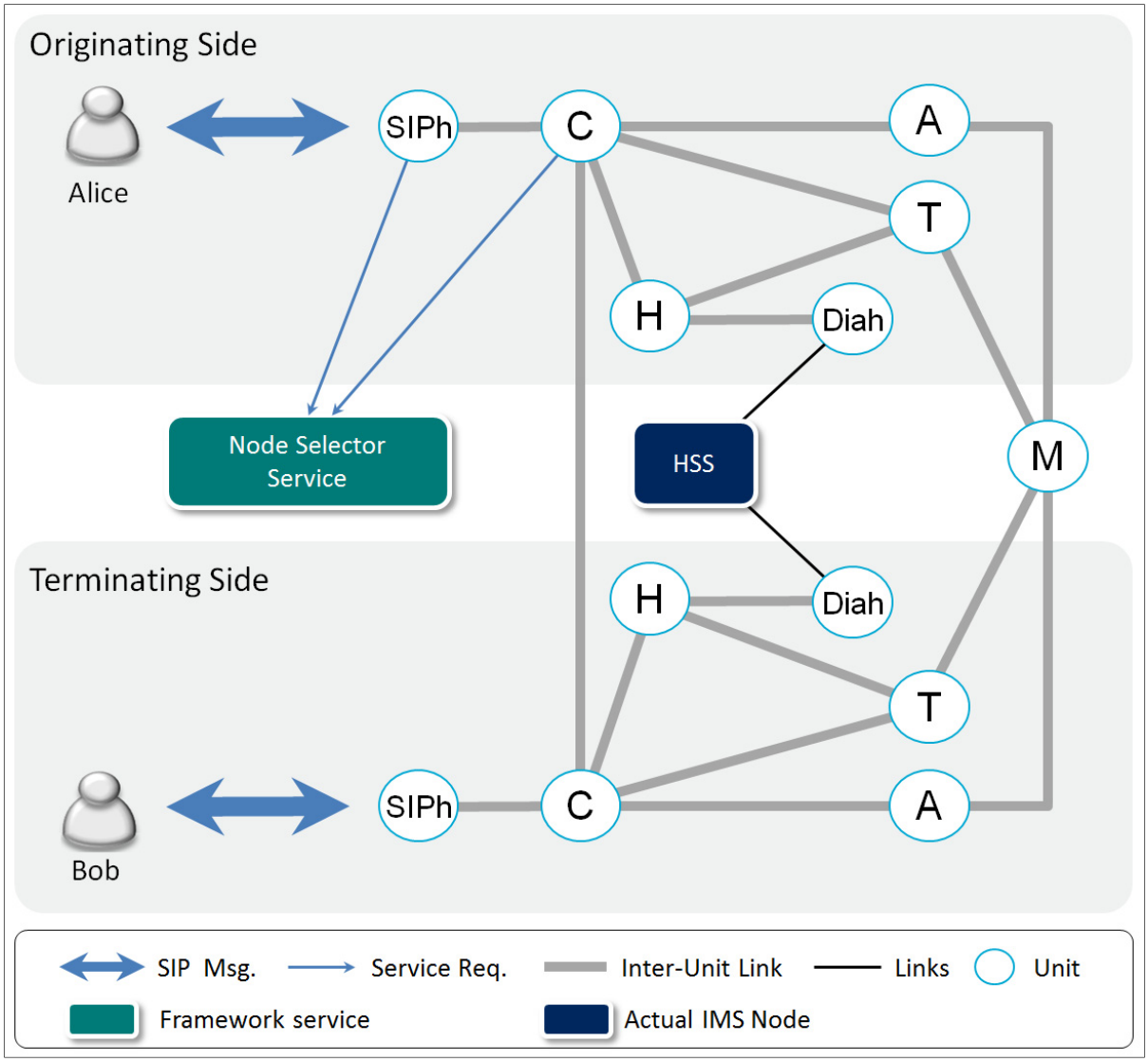

Figure 3.3 Two-party call service chain

3.3 The pouch concept for resource allocation

The definition of the pouch concept is the second corner stone that allows us to fulfill the objective O3, to propose an architecture for telecommunication applications providing portability between multiple cloud environments, enabling a solution-oriented, heterogeneous cloud deployment. By doing so, we answer the research question Q5 and develop a software architecture which enables writing software once and transparently deploying it on different cloud infrastructure and even provide hybrid deployments. Incidentally, by doing so, it also

provides a partial answer to the research question Q1, as the pouch concept provides elasticity to cloud-based telecommunication applications. Research question Q2 is also partially answered here as Quality of Service (QoS) is provided to cloud-based telecommunication applications via the scale-out of pouches.

In the proposed architecture, the cloud platform is responsible for allocating computing, network and storage resources to provide the required telecommunication functionality on a per-user or per-service basis. In order to cater to the heterogeneity of the platforms (PaaS, IaaS, Bare Metal, etc.), we introduce an abstraction layer which represents an instance of a computing resource on a platform. We define a pouch (Figure 3.4 and Figure 3.5) as a computing resource combined with a lightweight platform framework.

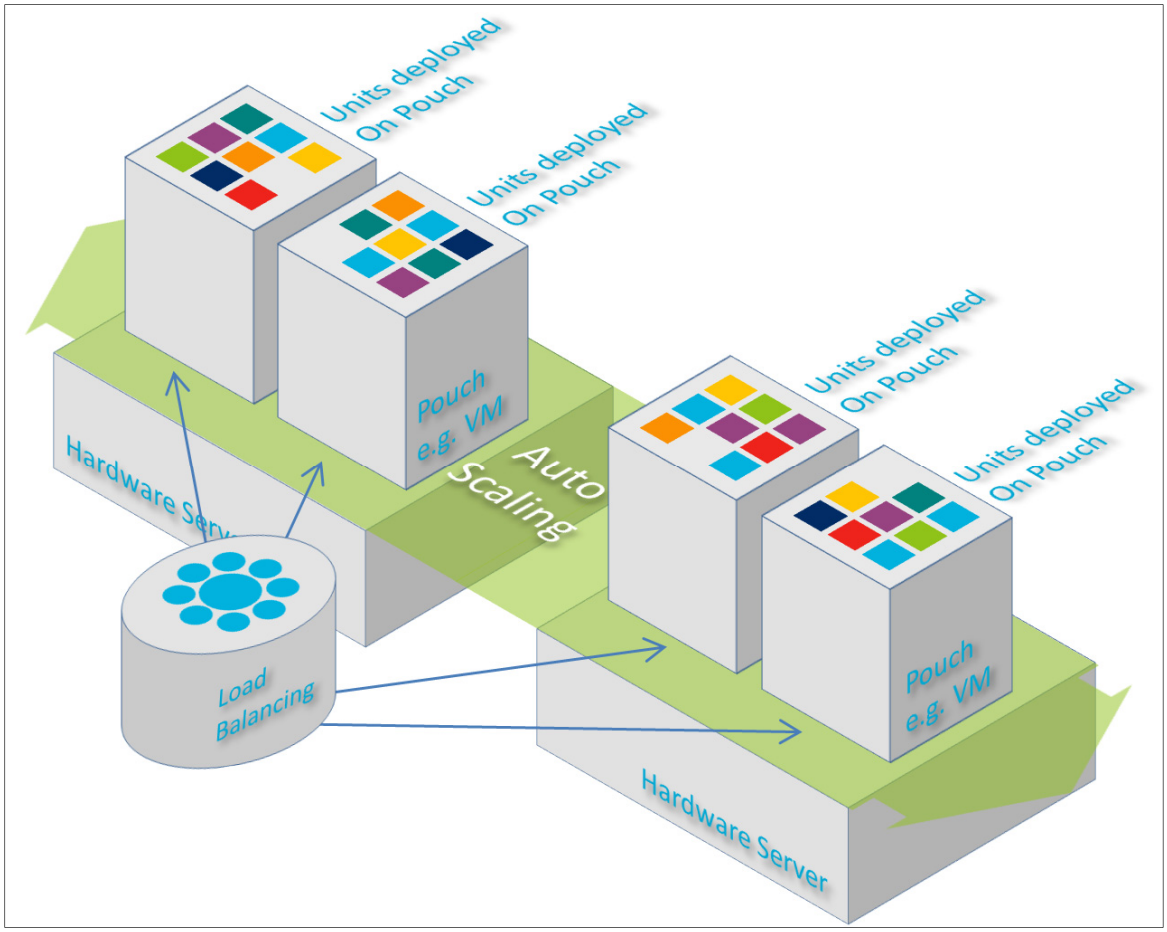

Figure 3.4 Distributed deployment of units on pouch scaling as needed
Taken from (Potvin et al., 2015a)

The framework supports functions that are offered as a library to the application code rather than an over the network as a service. The pouch can be seen as a set of libraries and daemons running on a computing resource to support the microservices and facilitate access to other services. In practice, a pouch can be a bare metal server, a virtual machine on IaaS, a container/job on PaaS, a microservice on a Unikernel, etc. The number of microservices and instances held by a pouch can vary from one to thousands depending on the characteristics of the host where the pouch is deployed.

One can scale-out any number of pouches on a platform. In these pouches, we instantiate units which can be assimilated as mentioned earlier to an actor in the actor model. A unit is able to transparently communicate with other units within any pouch through the Communication Middleware. Each unit is independent and can proceed to the creation of other units through the help of the platform framework.

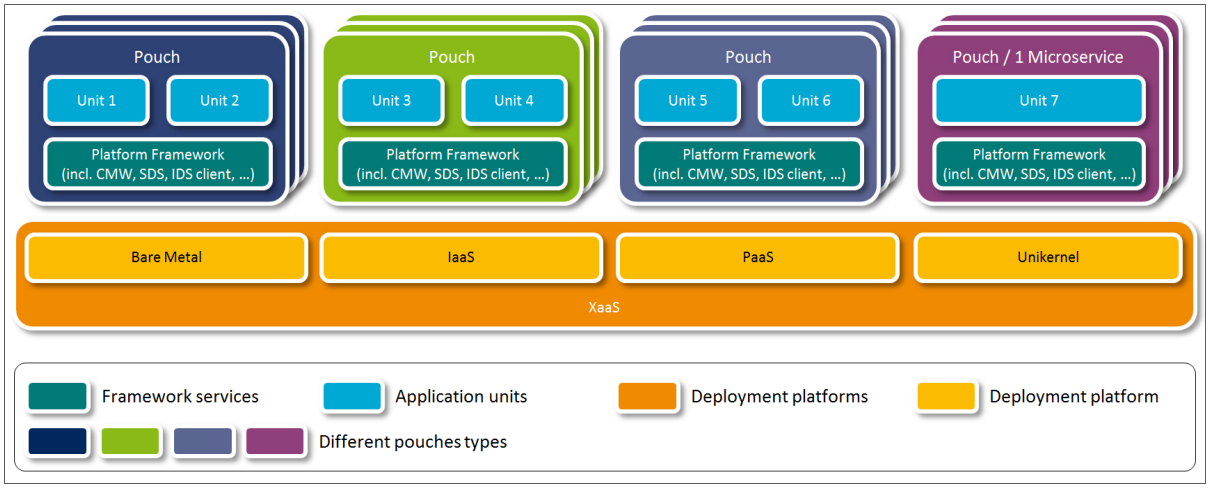

Figure 3.5 The concept of pouch deployed on XaaS

Our proof of concept was successfully deployed on a number of different platforms and even a hybrid deployment was made. In order to do so, we need to define a number of pouches to be used for those deployments. Figure 3.6 depicts such a target deployment on a mixed platform consisting of OpenStack VMs and Apcera Cloud Platform containers. Details about the different elements in that figure and the next are explained in later sections.

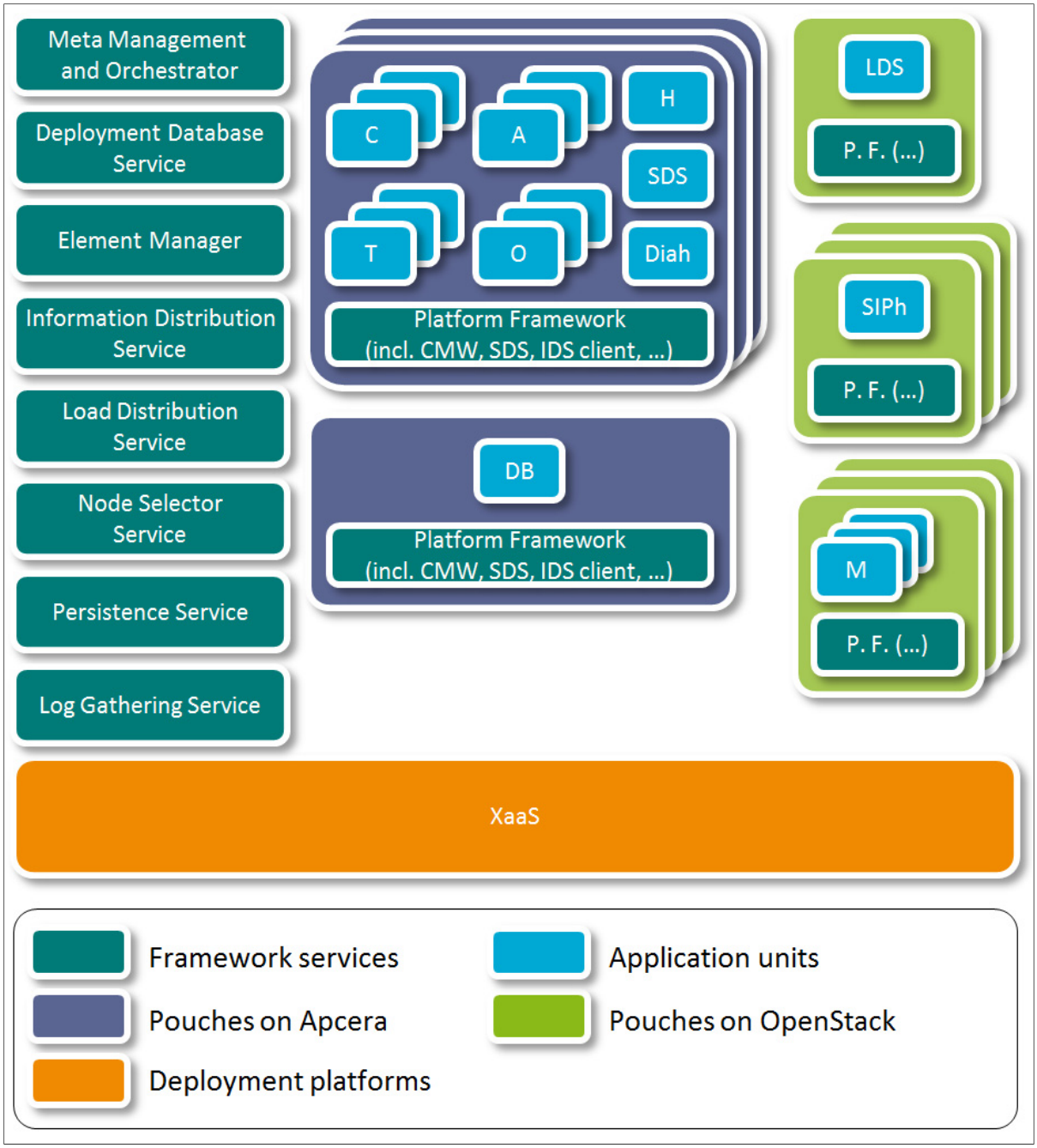

Figure 3.6 A potential deployment in a datacenter with OpenStack and Apcera Cloud Platform

In Figure 3.6, the pouches containing the main control functions of the Core IMS (units C, A, T, O, H, SDS, Diah and DB) as well as most of the services are deployed on Apcera Cloud Platform containers (jobs). Pouches on the Apcera Cloud Platform are depicted in dark blue.

The pouches containing the user plane functions of the Core IMS (units M) as well as the Load Distribution Service and SIP protocol handlers (units LDS and SIPh) are deployed on OpenStack VMs. The pouches on OpenStack are depicted in green. The main reason to deploy functionality on OpenStack VMs rather than on Apcera Cloud Platform containers has to do with the external protocol usage. Apcera Cloud Platform containers are best suited for HTTP traffic, which is one of the modes of operation for our Communication Middleware (CMW). On the other hand the LDS and SIPh units are facing the User Equipment which normally provides control plane traffic on SIP over UDP. The M units are also facing the User Equipment which provides user plane traffic on RTP over UDP. UDP traffic is not supported by the Apcera Cloud Platform, so that traffic needs to be handled in some other way. Hence the split between the platforms, handling the control plane over HTTP on containers and handling User Equipment facing UDP traffic over VMs on OpenStack.

In order to demonstrate hardware independency and the architecture heterogeneity we deployed our solution on a bare metal cluster of Raspberry Pi, low cost, credit-card sized computers. In Figure 3.7, we depict deployment on eight Raspberry Pi boards. The only difference between the deployment in Figure 3.6 and the deployment in Figure 3.7 is the descriptor file which specifies different target platforms, interfaces and pouch content. The application software source code is the same.

Lastly, it should be noted that the content of the pouches (units and services) and where they are to be deployed (Containers, VM, Bare Metal …) is completely configurable and could be different from one solution to the next. The deployment is controlled via a descriptor file handled by the Meta Manager and Orchestrator. The following subsections describe the pouches we defined via the descriptor file for the configuration we tested.

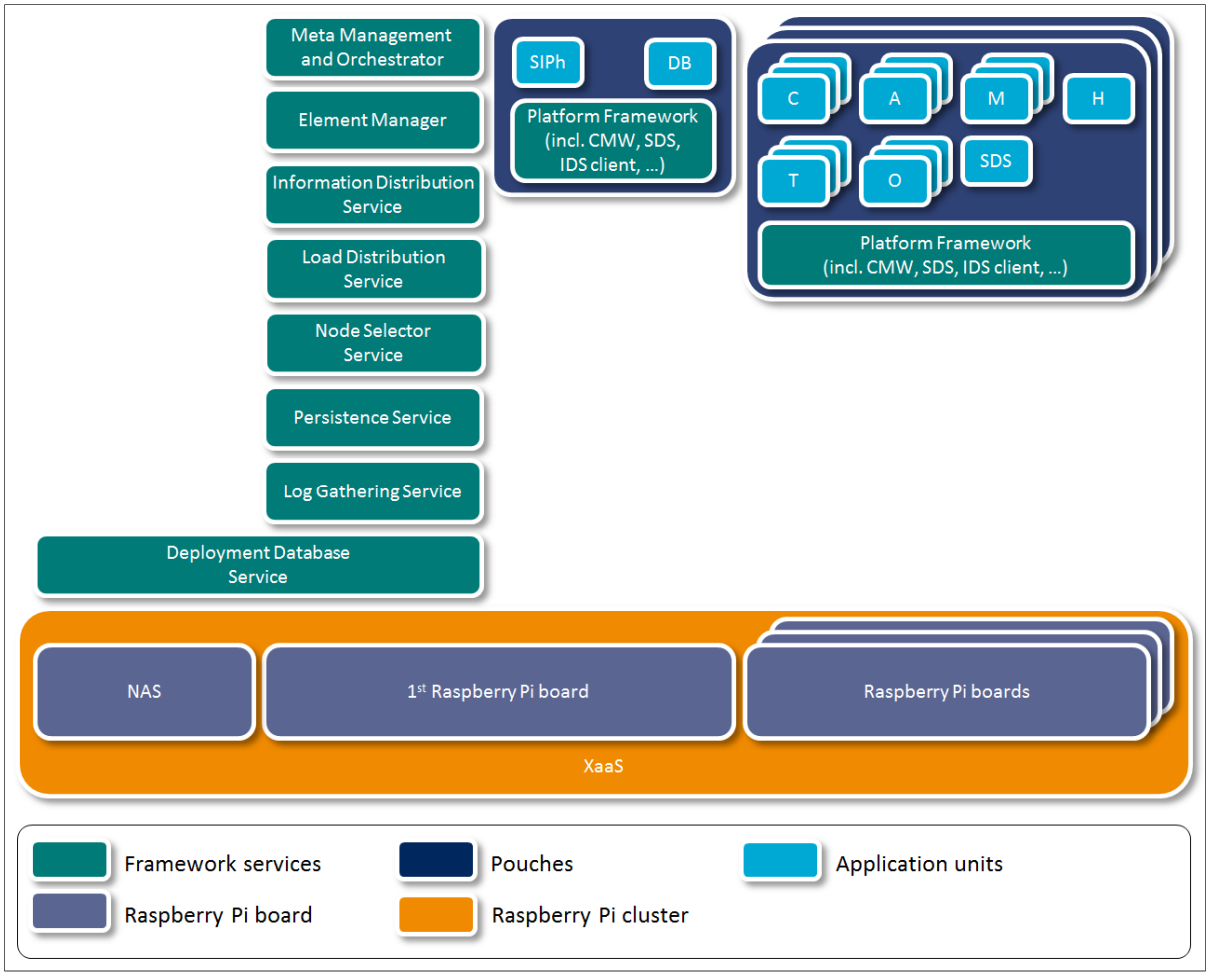

Figure 3.7 A potential deployment on bare metal Raspberry Pi cloud

3.3.1 Platform service pouches

All common functions are deployed in individual pouches. These include the Information Distribution Service, the Log Gathering Service, Node Selection Service, etc. Their deployment model is specified in a descriptor file used by the Meta Management & Orchestrator function. Where those functions are deployed and how they scale is transparent to the business logic application that makes use of these services through the platform.

3.3.2 Application pouches

Application pouches are used to deploy any type of application. The content of the pouches is solution dependent and configuration is driven based on the content of the descriptor file. In the case of our IMS prototype deployed on OpenStack and Apcera Cloud Platform, we define five types of application pouches: Control Plane pouches, Database pouch, Protocol Interface pouches, Load Balancing pouch, and User Plane pouches. For the Raspberry Pi cloud deployment, we define two types of application pouches: the Protocol Interface / Database pouch and the Core IMS pouches.

We should emphasize that we propose an approach where the application software is built once and can be deployed in multiple different configurations where the hardware and software platform is abstracted. For example, in chapter 5, we discuss a deployment where the pouches contained a single actor type, either a C unit or a T unit, in order to mimic a node-based deployment. This type of deployment can easily be done by modifying the descriptor file where the pouches are defined.

Control plane pouches

These pouches can scale horizontally on a need basis and run all the actors required to handle a SIP dialog (register or invite) in Apcera Cloud Platform containers. Basically they are defined to handle the control logic of the system.

Database pouch

This pouch contains the database used to represent the HSS instead of using an actual deployed HSS in order to facilitate testing.

Protocol interface pouch

As it was previously mentioned, this pouch hosts the protocol handling related functionality (SIPh).

Load balancing pouch

This pouch contains an LDS unit, providing round robin load balancing for UDP traffic. It should be noted that we cannot simply re-use OpenStack or Apcera Cloud Platform load balancers as they support only TCP.

User plane pouches

These pouches host the Media plane related functionality (M units) and are horizontally scalable.

Raspberry Pi deployment – protocol interface / database pouch

In the case of a Raspberry Pi cloud Deployment, this pouch is defined to handle the load distribution and the database. Since we do not have access to an ingress load balancer for this configuration, we define a single pouch of this type and directly use the SIPh as an entry point, instead of going through an LDS. It is however possible to allow multiple instances via the usage of an ingress load balancer.

Raspberry Pi deployment – core IMS pouches

In the case of a Raspberry Pi cloud Deployment, these pouches are horizontally scalable and contain all of the functionality of our Core IMS application distributed on the units.

3.4 Software architecture high level view

For the rest of this chapter, we assume that a pouch runs on XaaS and many instances of pouches can be scaled out on different platforms. It could be argued whether the platform framework should be an integral part of those platforms: should the Communication Middleware be integrated in the PaaS? To a certain extent this is not necessary for the architecture. If a part of the framework is part of the platform then the framework for that specific platform is simply lighter. If it is not part of a specific platform then it is available in the framework. Therefore the set of functionality which cannot be provided as an "over the network" service should be provided on all platforms we could wish to support. We do not have to develop for all platforms upfront, but simply ensure such development is possible if support for another platform is required at a later time. We are proposing a cloud architecture as platform-agnostic as possible. Then the choice of platform, Operating System (OS) and programming language can become merely a developer and customer preference decision (assuming availability of the framework for that platform).

The heterogeneous cloud architecture we propose provides the benefits of the cloud, while allowing the potential to get the benefits using purpose-built hardware where beneficial (pouch on specific target hardware with better characteristics for a task). Another benefit is the ability to adapt to the available operator's data center and to deploy a solution-oriented system/network based on the configuration of building blocks.

Figure 3.8 is an illustration of a basic model for this architecture. In this case, a number of pouches are instantiated on any platform (XaaS), and those pouches contain a number of units which are parts of a service for a subscriber. A complete service for a subscriber requires the linking of a number of units, which could all be collocated in the same pouch or be distributed between any numbers of pouches.

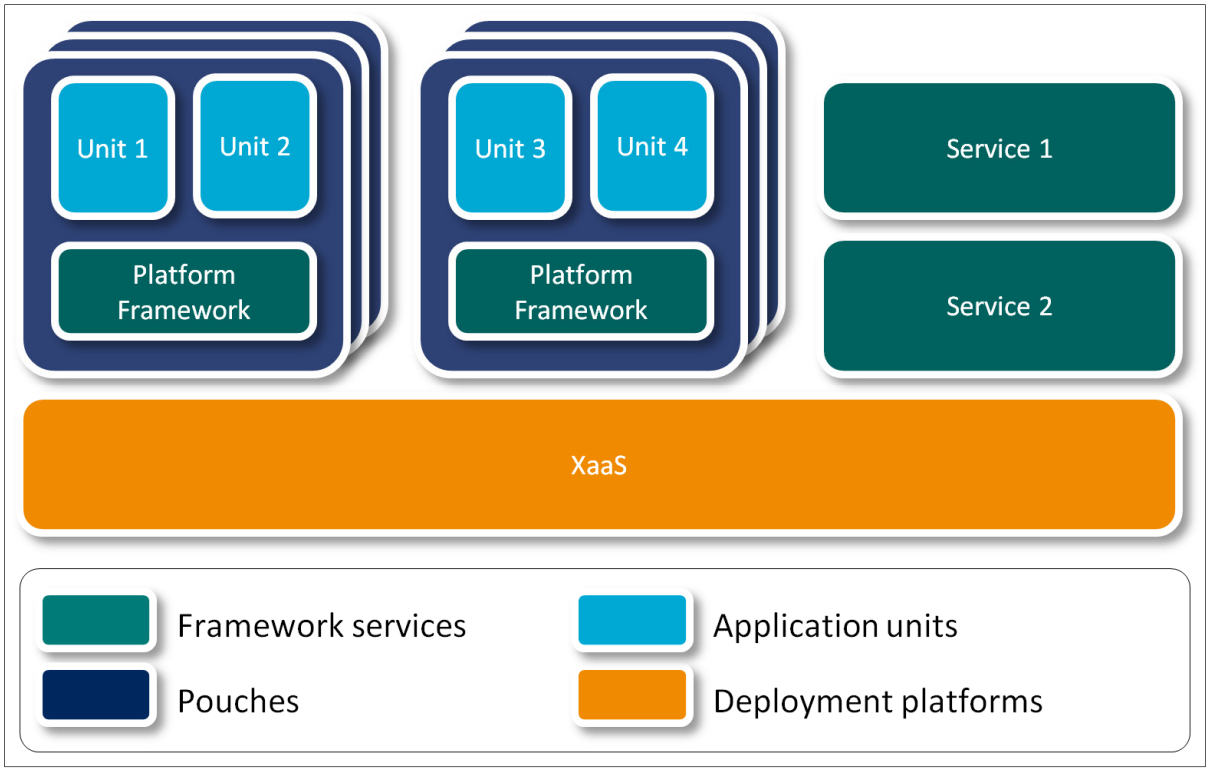

Figure 3.8 Distributed deployment through our proposed architecture and Microservices

Services are presented as separate entities from the pouches. This is because we might want to use services offered on different platforms (XaaS) through regular network communications such as REST. One such service is the Meta Manager and Orchestrator that we mentioned in the beginning of this chapter. Since we aim at deploying and scaling pouches on different platforms, there is a need for a Meta Manager and Orchestrator which knows where and when to deploy pouches based on a given configuration. The best way to visualize this Meta Manager and Orchestrator is to think of it as the NFV Management and Orchestration (ManO), where a VNF Descriptor (VNFD) describes the deployment configuration. This way, the configuration describes how to deploy the VNF (in our case a collection of pouches) and coordinates with the Element Manager to provide proper configuration to the units which run on a pouch.

Since it is difficult to assess the precise resource requirements needed by an application, the scaling should be based on some QoS measurements. This also decouples the scaling from

the platform which could vary from a deployment to the next. This however introduces the need for an interface between the individual pouch and the Meta Manager and Orchestrator to trigger scaling when necessary.

3.5 The platform framework

What enables units to be created and to communicate is provided by the platform framework. The platform framework is deployed together with all deployed pouches. The platform framework provides the essential services for such a system to function. This includes:

- unit start-up,
- unit configuration,
- unit shutdown,
- unit hibernation,
- unit state Storage,
- unit wakeup,
- maintaining pool of units,
- communication,
- logging,
- service resolving.

3.5.1 Unit start-up

The platform framework is responsible for figuring out where a unit should be started. Once a specific pouch has been selected, the framework performs the actual task of starting the unit.

3.5.2 Unit configuration

The platform framework is responsible for forwarding the configuration information to a newly started unit. The framework provides the communication mechanism to do so.

3.5.3 Unit shutdown

The platform framework provides the facility to enable a unit to properly shutdown and to propagate the information to its children and parents if needs be.

3.5.4 Unit hibernation

A currently running unit which determines it is not actively processing can request hibernate in order to save computing and memory resources at the expense of additional latency when the units needs to later be woken up. The hibernation process consists of:

1) The unit saves its current state information with the help of the framework which stores that state information;
2) The framework on behalf of the unit informs the connected peers of the hibernation;
3) The unit disconnects and quits (the unit running process is terminated).

3.5.5 Unit state storage

At hibernation or at any interval deemed appropriate by a unit, a unit can store its state information so that it can be recovered in case the unit is stopped. A unit can be stopped as part of the hibernation process, but a unit could also be stopped because of a failure of the software or hardware supporting the pouch holding the unit. As such, periodic state storage is a good strategy for resiliency.

State information is stored by a unit with help from the framework. The unit sends its state information to the framework, which in a first stage stores it locally in-memory. Then the in-memory information is replicated toward a database (ideally that database is replicated and distributed on the cloud for resiliency).

3.5.6 Unit wakeup

When a message is addressed to a hibernated unit, the framework starts the unit wakeup process. The wakeup process consists of:

1) The framework buffers the message addressed to the hibernated unit;
2) The framework starts the hibernated unit (start a new process);
3) The framework provides the unit with its stored state information and the unit restores itself to the saved state;
4) The framework forwards the buffered message the newly awoken unit.

3.5.7 Maintaining pool of units

Early on in the measurement stage it was discovered that starting an actor has its cost (in the 100ms range depending on the platform). In the actor model, we rely on using a number of actors linked together. Any delay associated with the creation of the actors can have a big effect on the performance of the system.

We found an elegant and generic solution at the platform framework level to this challenge. We pre-instantiate a small pool of every kind of actor, such that when an actor is needed, it is simply grabbed from that pool. The pool is replenished in the background to keep its level constant. The size of the pool just needs to be big enough to cover the number of instantiations of a specific actor versus the time it takes to instantiate such an actor, which on a production system would be a considerably small value. Moreover, assuming QoS implements a traffic shaping function, that value would be quite stable, i.e. the rate of requests would be stable with not much variation. Therefore, the optimal amount for the pool size can be precisely adjusted.

3.5.8 Communication

The communication facilities are provided by the platform framework. The platform framework supervises socket creation and termination, and determines the Actor Reference (ActorRef), a unique reference to a unit in the cloud. It also stores the Service to ActorRef information in a local in-memory database in order to perform Service Resolving functionality when required. The communication facilities of the platform framework are meant to address research question Q4 and insure communication between services across the telecommunication cloud platforms.

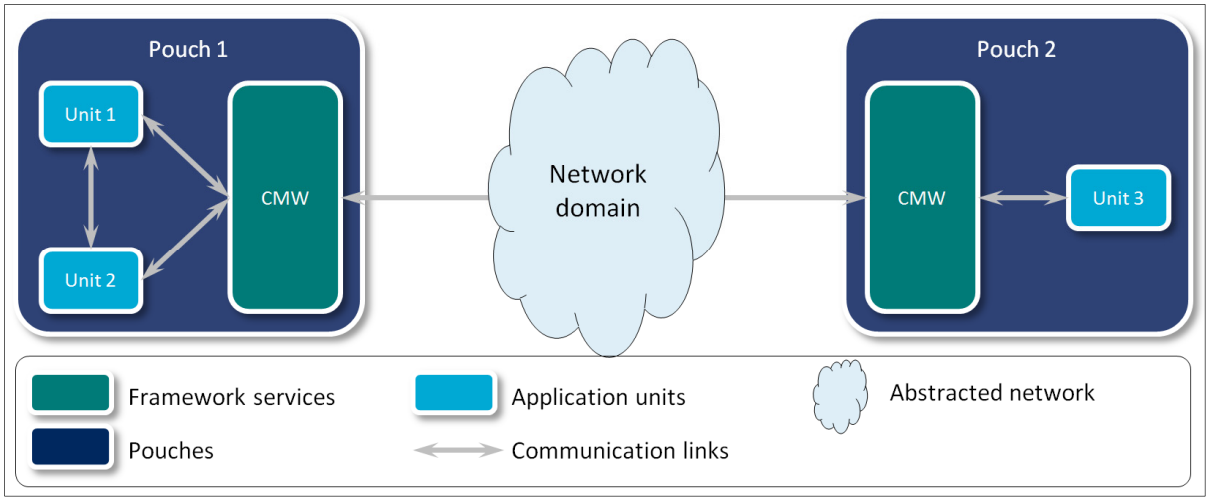

Figure 3.9 Communication scheme

Communication between units is mediated through a Communication Middleware (CMW) we developed and which is part of the platform framework. Figure 3.9 shows the communications scheme using tunneling. Co-located units on a same pouch can directly communicate using a TCP link. The establishment of that link and the communication is enabled via the CMW framework, which is provided as a library to the unit developers. When messages have to cross the pouch boundary, the link used to communicate with the CMW is used as a tunnel to route messages toward units located in other pouches. In order to cater for different platforms, the tunnelling can be performed over TCP or HTTP. The CMW uses the host name contained as part of the ActorRef to figure out which pouch to send the

message to. The receiving CMW looks at the "to_actorRef" parameter and ensures the destination actor is present. If it is, the message is simply routed using the link already established when the actor was launched.

If the peer unit is not present, the CMW responds with an error code indicating the unit has not been found. In turn, the originating CMW, upon receiving the error, informs the originating actor of the condition by sending a routing error message. This scheme emulates the TCP connection/delivery handshake and allows the detection of errors of a communication-oriented scheme in a connectionless scheme. From the application's point of view, there is no difference between an actor located on the same pouch from an actor located in another pouch and whether an HTTP or a TCP link is used.

For platforms where direct TCP communication is allowed and possible between the pouches, we also implemented the option for a unit to communicate directly to the peer unit without having to tunnel the message via the CMW.

When a pouch is started, the configuration file passed to the CMW by the Meta-Manager and Orchestrator (MMO) contains all the required information to operate. Depending where the pouch is running (on an OpenStack (OpenStack, 2016) VM or a Apcera Cloud Platform job, etc.), the MMO provides different configuration values that take into consideration the limitations and specificities of the platform the pouch runs on, and set the communication accordingly.

3.5.9 Logging

The framework provides facilities for logging and also provides the primitives necessary to forward the logging information to a central logging area, the Log Gathering Service (LGS), detailed in a later section.

3.5.10 Service resolving

The framework provides name resolving. It takes a service name and returns an ActorRef handling the requested service.

3.6 Other software architectural elements

Figure 3.10 depicts the target deployment when including all elements developed in this architecture. The proof of concept (PoC) telecommunication application architecture is presented in later sections. The architecture implementation can be deployed on a mixed platform consisting of OpenStack VMs, Apcera Cloud Platform containers or bare metal servers. In the current implementation of our proof of concept only the Linux OS is supported. Also only the C/C++ programming languages have a full support. We have partial support for the Java programming language. In order to support other platforms, OS or programming languages, one would have to port the CMW to those other platforms and may need to extend the MMO to support those platforms.

The proposed architecture defines a minimum set of functionality allowing a microservices-based application to be deployed on heterogeneous cloud platforms. The microservices or units each perform a specific task and cover a single scaling domain. For example, an application microservice or unit may handle a limited number of related telephony services, such as the HSS interrogating functionality of an application. The minimum set of functionality is itself built as microservices which are deployed as “units” as follows in the next sub-sections.

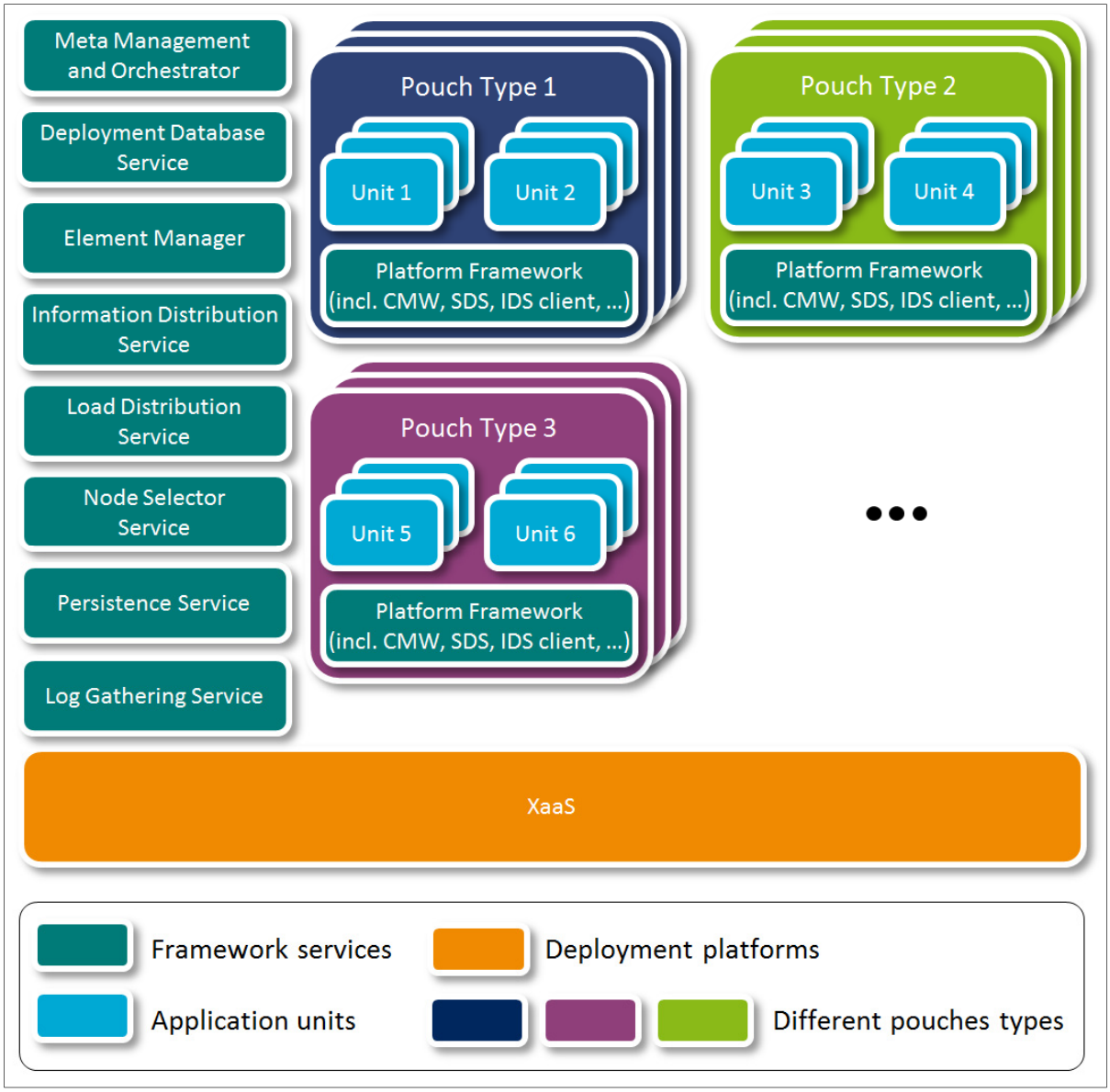

Figure 3.10 Framework proposed architecture

3.6.1 Meta Management and Orchestrator (MMO)

The Meta Management and Orchestrator (MMO) is responsible to read the descriptor file and deploy the appropriate pouches on the available platform pools. It also monitors usage information from the CMWs via the Information Distribution Service (IDS) and instantiates new pouches to provide elasticity to the cloud. The MMO is the final part of the answer to

research question Q1, as it provides elasticity to cloud-based telecommunication applications by scaling the pouches; and research question Q2 as Quality of Service (QoS) is provided to cloud-based telecommunication applications via the scale-out of pouches.

The main tasks of the MMO are to:

- Allow the user to configure and monitor the deployment through the Operation, Administration and Maintenance (OA&M) interfaces;
- Parse the descriptor file and use it to deploy the appropriate pouches (VM, containers or bare metal servers) through the proper interfaces;
- Monitor the pouch status (in terms of resource usage, alarms) and react accordingly by either alerting the end user or scale the pouches in or out;
- Monitor the computing pool resource allocation and report error conditions to the operator.

There are two main entry points to the Meta Management and Orchestrator: the operator OA&M and the Element Manager (EM). Through the OA&M access point, the operator can configure the various aspects of the deployment as well as monitor its operation for fault and performance evaluation. The EM provides the configurations of the business logic in collaboration with the MMO.

The MMO should preserve its current status through the use of permanent storage in order to be able to resume its operation in case of a crash.

Elasticity

As mentioned earlier, the MMO is responsible to perform the scale in/out of pouches thus implementing one level of elasticity in the architecture. In order to guide the scale in/out of the pouches, the MMO reads the descriptor file that defines:

- The content of the various pouches (which units it can run);
- What platforms it can be deployed and run on (VM, container, bare metal server, etc.);

- The minimum and maximum number of pouches allowed;
- The conditions under which a scale in/out should be performed;
- …

The conditions to perform scale in/out are based on average CPU usage, memory usage, network usage and number of instantiated actors. For each of those conditions we can specify a weight which is used to determine if a pouch is sufficiently idle to perform a scale in or too busy and requires a scale out. In order to evaluate these conditions, each pouch provides its load figures through the Information Distribution Service (IDS) as detailed in Figure 3.11.

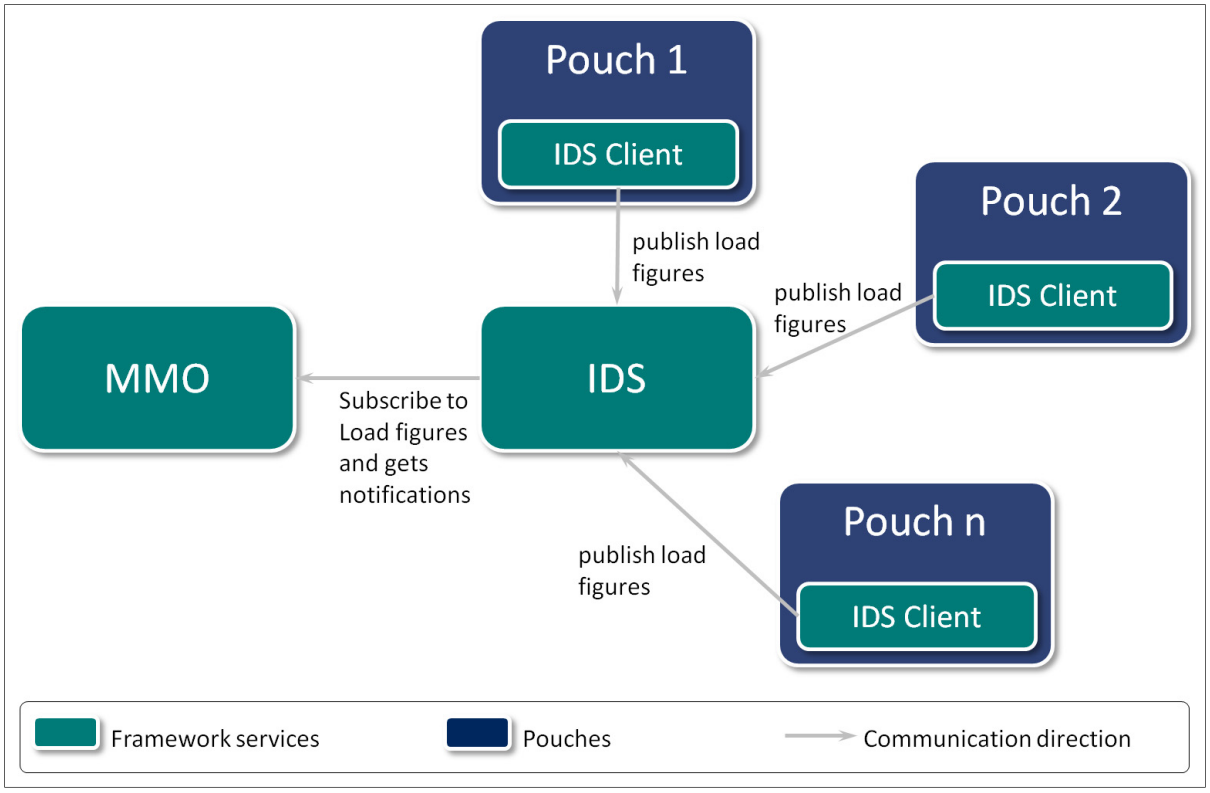

Figure 3.11 Load figures publication and subscription

3.6.2 Element Manager (EM)

The Element Manager (EM) is responsible for configuring the various applications running on various pouches. It also performs monitoring of the pouches in co-operation with the MMO.

3.6.3 Communication Middleware (CMW)

The Communication Middleware (CMW) unit is the basis of the proposed architecture. Each pouch is required to run a single instance of a CMW. It manages the most basic functionality that is required for cloud operation such as inter-unit communication, unit spawning, pouch monitoring and unit/service address resolving. It implements the communication principles detailed in the platform framework section above.
Communication between telecommunication nodes is mostly if not exclusively defined in terms of standardized communication protocols. However, communication between the processes constituting a telecommunication node or between proprietary systems is usually not bound by a requirement to follow a standardized protocol. This leads to a multitude of in-house communication principles being used and developed, which makes software re-use more painful as designers must learn new techniques for all functionality. It also makes distribution over a large network of computing resources more difficult, as each of those in-house communication principles needs to be adapted for distribution over such network.

Selection of a proper communication middleware can alleviate if not solve those issues by relinquishing the burden of establishing communication in the system to a universal mechanism. It also limits adaptations required for message distribution to the communication middleware functionality.

Discovery service

With the usage of a Communication Middleware comes the necessity of a discovery service or a name resolver service. This enables the application to find the address of another service/agent/actor/unit by its name or some other reference. This functionality is also implemented in the Communication Middleware.

3.6.4 Load Distribution Service (LDS)

The Load Distribution Service (LDS) is the functionality involved in a service setup scenario. It is basically a sticky load balancer which selects in a round robin fashion the proper message handler to use, i.e. it selects which SIP Handler (SIPh) unit to send the SIP messages too.

The LDS unit is one of the few permanent units in our architecture implementation. In our current implementation, it is the SIP proxy (thus linked to the control plane) for all the subscribers. This first stage load balancer distributes the load evenly to a semi-stateless load distributor stage (shard-aware load balancer) in order to spread the load and ensure QoS.

The LDS unit is started when the system is provisioned and is intended as a service to all subscribers' service instances. Since the LDS is stateless, any number of LDS units could be started in the system as long as the User Agent (UA) configurations (pointing to the LDS unit as a proxy) are properly balanced to distribute the load across them. LDS units do not have to be aware of each other's existence.

As the UA only knows about one proxy, the LDS unit is always the first unit involved in a service setup scenario.

3.6.5 Node Selection Service (NSS)

The Node Selection Service (NSS) implements the logic of spreading the subscribers' service instances on the available pouches. It ensures that most of the service requests for a subscriber are made to the same pouch in order to maximize local memory cache hit.

It is important for a telephony system to limit the query time to a database. It is consequently a desirable feature to use an in-memory cache which synchronizes itself with a centralized or distributed database. The NSS unit ensures that most of the service requests for a subscriber are made to the same computing resource, which has stored the in-memory cache of that subscriber's profile thus maximizing the cache hits.

On originating calls, the SIPh uses the services of the NSS to locate where the Originating C unit should be started. Later in the call chain, the Originating C unit uses the service of the NSS to locate where the Terminating C unit should be started.

In order to ensure that most of the requests from a specific subscriber are made to the same computing resource, we use Rendezvous hashing as described earlier, where the served subscriber URI is used as sharding key.

The current implementation of NSS is such that you can change its node selection algorithm to two different modes. In the first mode, it picks the nodes based on the Rendezvous hashing load balancing. In the second mode, it picks the least loaded node to spawn the target unit at. Specifically, when the NSS receives a spawn request, it looks at the spawning tag field of the request. If it is empty (meaning that no tag is provided), it uses least loaded algorithm for selection. If a spawning tag is provided with the request, the NSS uses Rendezvous hashing algorithm to find a node for it.

- Rendezvous hashing based: the NSS sifts the pouches which are not allowed to spawn the target actor, according to the pool configuration messages that it has received from them. When the NSS receives a spawn request with a spawning tag, it tries to find the hosting

computation node according to a deterministic algorithm. Assume that each pouch has a unique name, say NODE_i. Representing the received spawning tag as TAG, the Rendezvous hashing algorithm selects the node with the lowest hash. As a result, if the set of pouches stays the same, the spawn requests with the same TAG is directed to the same pouch;

- Least loaded: in this mode, after sifting out the pouches which are not allowed to spawn the unit, the pouch with the smallest load figure (calculated from the load figures distributed through the IDS) is selected as the node to handle the request.

3.6.6 Information Distribution Service (IDS)

The Information Distribution Service (IDS) allows information exchange based on a publish/subscribe system. Some of the information disseminated through it includes: resource utilization, service/unit resolving updates, system status updates, log levels and log entries, global configuration, etc.

In order to efficiently distribute information between different entities within the cloud, an efficient and sufficiently generic publish-subscribe notification service is required. Such a service is founded on an event-based or notification-based interaction pattern for inter-object communication. This type of notification pattern is being increasingly used in the context of IT cloud and is necessary in a telecommunication cloud.

The IDS is a key service in the proposed architecture framework and it is the first pouch to be deployed for without it there can be no exchange of information amongst the pouches.

3.6.7 Deployment Database Service (DDS)

The Deployment Database Service (DDS) maintains copies of VM images, service and microservices binaries, etc. which are necessary to deploy software on the various pouches of the system.

3.6.8 Log Gathering Service (LGS)

The Log Gathering Service (LGS) receives the logging information from various pouches, services and microservices, and sorts and consolidates it for consumption. It can be assimilated to both the roles of a mapper and a reducer in the MapReduce architectural pattern. As a mapper, it indicates to the pouches enabled log levels and filters based on a number of keywords. As a reducer, it receives the logs from the pouches and orders them based on the time stamp before presenting them to its output which could be a file, a console, etc.

3.6.9 Persistence Service (PS)

The purpose of the Persistence Service (PS) is to store a unit's state information from a local in-memory cache. This cache is backed up by the PS on a distributed/replicated storage solution that permits its recovery when its assigned compute resource fails. In case of failure, a unit may be recovered from the last saved stable state and be restarted on a new compute unit. In conjunction with the Communication Middleware and the State Database Slave (SDS), it allows units to "hibernate" within the context of a session.

An in-memory cache system is a necessity to limit access to Telecommunication databases. For example, an HSS cache can keep local copies of information while processing services and potentially in-between successive service accesses. In general, it is desirable to process data in-place as it can be bigger than the control plane signaling. As such, maintaining the information solely in a central database can become a bottleneck. The usage of an in-memory cache service combined with replication to a distributed database as described before is our mean to address the need to fetch the information once from the central database and then maintain it on a specific computing node, thus minimizing networking traffic for that purpose.

In the current incarnation of the system we have a centralized database implementation for the PS. However, the concept supports the more general case where the centralized database is replaced by a distributed database on the cloud.

3.6.10 State Database Slave (SDS)

The State Database Slave (SDS) unit is a permanent unit created at a pouch start. It is responsible for managing the memory cached database tables in a pouch. This results in a two stage database system: in-memory cache handled by the SDS unit, plus a replication database handled by the Persistence Service.

3.7 Telecommunication application software architectural elements

The microservices (also called units) developed for the Cloud IMS Telephony application are listed below, with notes as to which of the IMS functions they provide. Although the application we developed is a simplified Core IMS system, any application could be developed under this framework, whether it is telecom-based or web-based. Anecdotally, we built a chess game as a development example on our proposed architecture framework, which plays concurrent tournaments between virtual players and which can display individual games as well as overall results. This demonstrates that, as a distributed actor model based framework, our proposed framework can be of use for different type of applications, even non-telecommunication type applications.

3.7.1 SIP Handler (SIPh)

The SIP Handler (SIPh) is the SIP signaling entry point. The SIPh is stateless and any number of instances can be started in the system, as long as the User Agent (UA) configurations (pointing to the LDS unit as a proxy) are properly balanced to distribute the load across them. Alternatively, a regular load balancer (or SBG) or a single SIPh could be used as an entry point.

The SIPh Implements the SIP processing functions of the P-CSCF and the I-CSCF. It is the first unit involved in a service setup scenario. The SIPh uses the services of the NSS in order to figure out where the service setup should be instantiated in the cloud. The SIPh instantiates the C unit as it is the first service unit in the service chain and forwards it the received SIP message.

Due to the statefulness of the SIP stack, any given pair of originating and terminating CSCF actors must connect to the same SIPh. When receiving an external request, the SIPh extracts the "from" and "callid" headers and combine them with the CSCF actor name to generate a unique identifier "unit/shard/instance". This allows the SIPh to properly connect to any CSCF that is associated to it.

3.7.2 Call Session (C)

The Call Session (C) unit deals with the requests coming from the subscriber. It performs the functionality of an S-CSCF. It handles the request coming from the UA, fetches the subscriber profile based on service triggers and builds the appropriate service chain to provide the requested service. The C unit makes use of the Node Selection Service in order to figure out where the terminating Call Session unit should be instantiated.

The C unit is a dialog actor instantiated on request to handle the subscriber’s current service and it is terminated when the service has completed, e.g. for a call, it stays in the picture from the Initial Invite until after a timeout supervision period after the SIP Bye has been forwarded.

One subtlety with respect to an S-CSCF (to which the C unit is the closest to in functionality) lies in the fact that if a subscriber registration is about to expire during a call, the subscriber’s UA automatically triggers a re-register process. This re-register transaction is done using a separate dialog that is handled by a separate C unit instance. The new C unit updates the database accordingly. When the registration period expires, the C unit requests the

registration expiry time from the database. If it is updated appropriately by the other C unit, nothing needs to be done. If for some reason, the re-registration process fails, the C unit terminates the current call.

3.7.3 Orchestrator (O)

The Orchestrator (O) unit is an ephemeral actor which tells if a specific service/feature is allowed to run on the cloud platform based on provisioning or license activation. In its current incarnation, this actor acknowledges all requests as no consideration for provisioning or licensing has been included yet.

3.7.4 HSS Front-End (H)

The HSS Front-End (H) unit is a permanent unit created at a pouch start. The H is used to fetch a user profile. It is responsible to verify if the information is present in the in-memory cache. When it is absent, the H unit must query the database or HSS in order to get that information and to store the newly received data in the in-memory cache. Once the information is stored in the in-memory cache it is responsible to remain synchronized with the database or HSS for that set of data. The cached data is removed if the registration timer is allowed to expire.

Two configurations are possible. In the deployment configuration, the H unit accesses the HSS via the Diah, which it contacts via the Diameter protocol. In the test configuration, the H unit accesses the subscriber information stored in the system Database (DB), therefore bypassing the Diah unit.

3.7.5 Database (DB)

The system Database (DB) unit is a permanent unit created at system start. It is responsible for managing the various subscriber database tables in the system. The DB provides a central

repository for the user profile information in a test deployment, where we do not want to use an external HSS. The DB basically ensures replication for the in-memory cache handled by the H units.

In the current incarnation of the system, we have a centralized database. However, the concept supports the more general case where the centralized database is replaced by a distributed database on the cloud.

3.7.6 Diameter protocol Handler (Diah)

The Diameter protocol Handler (Diah) unit is used to interface with the HSS via the Diameter protocol. It serves as a protocol handler service for the H unit.

3.7.7 Media Processor (M)

The Media Processor (M) unit is a dialog actor which handles the media plane of the call through RTP as an MRFP would do. It provides point-to-point connectivity for basic two-way calls and provides voice mixing in the case of conference calls. In its current simple incarnation, it supports only G711 8 kbit/s μ-law encoding.

When involved in a conference call, the M unit performs basic source-exclusive audio mixing. This means that each participant receives a mix of all the terminations except his own. For example, in a three participants call, the first subscriber receives a mix of the second and the third subscribers, the second subscriber a mix of the first and third subscribers and the third subscriber a mix of the first and second subscribers. The same principle applies for more participants.

The M unit is, for our small scale prototype, the most resource intensive. It uses in the range of 5% of the processor capacity of a single Raspberry Pi computing resource (for the case of a Raspberry Pi deployment as described later on). For that reason, in the current model, it is

shared between all the legs of a call instead of following the traditional model where each call leg would have its own MRF. In the long run however, we could associate independent M units to each call leg.

The current implementation uses a private protocol which looks like a lightweight version of H.248.

3.7.8 Anchor Point Controller (A)

The Anchor Point Controller (A) unit covers more or less the MRFC functionality controlling the M (Media Processor) unit. It is in charge of creating the M unit as needed for the service scenario and informs the interested units in the call chain of the availability of the functionality in the service chain.

The A unit's main function is to negotiate the media codec and to modify the SDP codec list according to the responses from the M unit so that a subscriber's UA can properly exchange media with the M unit.

3.7.9 Telephony Server (T)

The Telephony Server (T) unit provides the various telephony related features to the subscriber as an IMS MMTEL would do. For example, it can listen to DTMF subscriber activities to trigger supplementary services, e.g. ad-hoc conferencing by adding another call leg to the current call.

The T unit is created by the C unit on both the originating and terminating sides based on the user profile triggers. The subscriber information that is required for the T unit to perform its tasks is fetched from the subscriber's profile via the H unit.

The T unit connects to the M unit in order to receive the various media plane telephony events the subscriber might generate and to control the connectivity at the media plane level.

The T unit provides the different services provisioned for the subscriber. Currently supported services in our lite implementation are:

- Two-way call establishment: this service covers the basic establishment of a two-way call as detailed earlier;
- Audio conference: a subscriber is able to start an ad-hoc conference when he is in a call with another subscriber by pressing the sequence '#'<third subscriber number>'#'. The process establishes a new call leg toward a third subscriber (or fourth, etc.) complete with its own C, A and T units. And the service invoking T to operate the M unit to connect and mix that new call leg within the current call.

3.8 Discussion

As exposed in the first chapter, our main objective is to develop a cloud-native software architecture for telecommunication systems and perform and realize an implementation of the architecture in order to evaluate its merits. Through this chapter, we developed such an architecture and showed how it is implemented. Still, as part of the main objective, we wanted to determine if we could re-architect an IMS in order to provide the functionality in an on-demand, per subscriber and per service basis through the use of the actor model, which can be considered as a specialization of the microservices pattern. Again, as seen in this chapter, re-architecting the IMS is indeed possible and through the use of the actor model and the concept of units we introduced, we can provide the functionality in an on-demand, per subscriber and per service basis.

We fulfill specific objective O1 with the pouch concept and the MMO by providing a mechanism to efficiently allocate computing resources through multiple cloud platforms in order to prevent overloading of computing resources and the resulting adverse impact on QoS. This MMO implements an elastic and automatic scalability scheme. The MMO unit

implements this by monitoring the conditions on the different pouches and allowing new pouch instances to be added or removed based on need.

We answer specific objective O2 with the application of horizontal scaling and sharding patterns though the LDS, NSS, SDS and PS in the framework and via the use of the LDS, NSS, H and DB or Diah and a real HSS in the telephony application. Together those units allow for application state information to be distributed on compute instances, providing resiliency while minimizing impact on latency.

We also have specific objective O3 to propose an architecture providing portability between multiple cloud environments to enable a solution-oriented, heterogeneous cloud deployment where the deployment can be on bare metal pools of servers, IaaS, Platform as a Service (PaaS) or a mix of those. Through this chapter we developed a service-based architecture where the main services are accessed via the network through a unified communication mechanism. Moreover, we proposed the concept of a pouch, which allows deploying instances of the architecture on any deployment platforms. In the next chapters, we see how implementation of the architecture have been deployed on a small cloud made of Raspberry Pi computing platforms (Raspberry Pi, 2016), Open Stack VMs and Apcera Cloud Platform containers. Support for other platforms can be integrated in the architecture in a plug-and-play fashion. We also show that support for other OS or programming languages can be added to the architecture in that same plug-and-play fashion.

Lastly, specific objective O4 is answered partly as we detailed the implementation of the architecture. In a following chapter, we proceed to measure the architecture characteristics and to validate its performance and compare it to a traditional telecommunication node-based deployment.

Some other additional benefits of the architecture are listed in the following sections.

3.8.1 Service orientation and re-usability

The actor model relies heavily on Service Orientation in order to provide the functionality of its applications. As such, it is a natural enabler for re-usability, where services are defined and built once and re-used by many different actors. This opens the door to a system where Solution Orientation is promoted, where research and development can develop the building blocks that can be configured in different ways in order to answer the needs of different markets. As such, our proposed architecture is an enabler for providing a software product line. As mentioned earlier, the software product line model is a benefit for telecommunication vendors as it allows them to cater to the needs of the many telecommunication operators they serve as customers.

3.8.2 Scalability from one computing unit to infinity

As computing for a subscriber service can be started on any computing unit, the system can scale from a single computing unit to any number required to handle the traffic. Seen from another standpoint, this means that all computing units but one could fail and the system would still be able to perform its function for any of the subscribers or their services albeit at a lower capacity.

3.8.3 Model driven friendliness

One of the units we built in the implementation of our architecture, the T unit, was done using Bridgepoint (xtUML, 2016), a modelling tool allowing for automatic software generation. The actor model upon which our proposed architecture is based has a really good fit with Model Driven Design (MDD). The fact that we only need to concern ourselves with one subscriber service at a time simplifies the problem enough to make MDD manageable as a development tool.

3.8.4 Testing

One of the obvious advantages of the actor-based, one subscriber approach that our proposed architecture offers is simplified testing. Testing of the solution needs only be performed for one subscriber as the solution to elastically scale is part of the framework. Hence, once the framework is proven to support elastic scaling, there is no need to re-test that functionality for each application.

3.8.5 Software problems lessened

Through the use of our proposed architecture implementation, we can see that some software problems are lessened. For example, since most units have finite life duration, i.e., duration of the service or, duration of the call, memory leak problems becomes less severe as the leaking process will be terminated after a finite duration.

3.8.6 Upgradability

One of the requirements we initially considered was upgradability. The architecture we propose offers a highly flexible upgradability path. Since the units supporting a service or a call are deployed on a need basis, upgradability can be performed in the same way. Upgrade principles used for web services (e.g. Facebook) could be applied for a system based on our proposed architecture. For example, we could upgrade a random sample of 1% of the new calls and look at performance indicators to ensure the quality of the upgrade. If the upgrade goes well, we could grow to percentage of upgraded calls to 5%, 10%, and so on until we reach 100%. Therefore upgradability becomes highly flexible and can be performed on per call or per service basis.

3.8.7 QoS

One of the main advantages of the proposed architecture is that it allows building a system where only the available computing resources dictate the scalability of the deployment. The software is no longer built with a specific target in mind and thus no specific scaling value is set. We can delegate the scaling aspect to the infrastructure of the cloud. The cloud infrastructure can also take care of the QoS aspects, thus relieving the individual application from that consideration and transforming it into a service of the platform.
Even without redundancy on a per call basis, losing a computing unit would only lead to dropping the calls served on that computing unit, but it would not prevent or delay the re-establishment of those calls. The capacity of the system to handle calls would still be available. This is quite an advantage over a node-based architecture, where you need to have redundancy at the node level, otherwise the loss of a node would prevent all the served subscribers to have access to their services. In our proposed architecture, the subscribers are simply served on another computing unit.

3.8.8 Integration with legacy systems

We've shown that our architecture allows integration with legacy systems by enabling the use of a real HSS from the sample telecommunication application we built. We could add boundaries in multiple places and allow systems of mixed paradigms to interoperate, e.g. a system consisting of some applications built on our proposed architecture and some built on legacy node-based architectures. Boundary units can be added to offer the transition toward the legacy systems if needed. These boundary units would not be used otherwise, saving resources in the system.

CHAPTER 4

EXPERIMENTAL PROTOCOL

As we mentioned earlier, we built the architecture such that applications developed on it can be deployed on any platform, which is commonly called “Anything as a Service (XaaS)”. For the purpose of demonstrating our proposed architecture, we deployed it on a number of platform mixes:

- On a cloud of low capacity Linux-based servers, consisting of eight Raspberry Pi computing platforms. This was our main development and testing platform for a long time;
- On a single OpenStack VM on an individual developer computer, in order to facilitate development and testing (no measure are reported on this development setup);
- On various OpenStack managed server clusters: one consisting of Ericsson Blade Servers (EBS) at ETS, and another one of Dell Servers at Ericsson premises;
- We deployed the system on a hybrid set of platforms consisting of an OpenStack managed server cluster, running some pouches on VMs and an Apcera Cloud Platform running some pouches on containers all cooperating as a whole.

4.1 Raspberry Pi deployment

As we discussed earlier, we deployed our microservices IMS telephony application on two main distinct deployments. The first deployment platform (Figure 4.8) is based on eight Raspberry Pi (Raspberry Pi, 2016) boards. It provides twofold benefits. First, it is a cost effective way to have a 24/7 cloud we can experiment on. Secondly Raspberry Pi computing platforms being simple single core computers, this limits the number of variables required to consider while studying the system.

Figure 4.1 Eight Raspberry Pi boards cluster cabinet designed for 3D printing and the actual cabinet

Raspberry Pi[1] is a small (credit card size) and low cost (approximately 50$) board containing an Advanced RISC Machine (ARM) processor with 512MB of RAM, a 100 Base-T Ethernet connection and two USB 2.0 ports. It boots from an SD card and can run a multitude of operating systems. It benefits from a huge community and is used for all kind of tasks. For example, it was used to build arcade game cabinet, mobile phone, home automation controller, tablet computer, Tor Router, media player, picture frame, etc.

Our Raspberry Pi (RPi) deployment is built of:

- 8 RPi Model B boards, stacked together in a custom made cabinet (more details on the cabinet later on), where each RPi is set in a sliding tray and can be inserted or removed from the cabinet enclosure;

[1] https://www.raspberrypi.org

- 8 custom-made RPi Daughter Boards enabling the display of information via two RGB LEDs and allowing input via a button;
- 1 Gigabit Ethernet switch providing the backbone network for the system;
- 1 Wi-Fi router providing access to UEs (hosting the UA) and providing the NAS functionality on a USB Storage Device;
- 1 Power Supply for the RPi's Cabinet.

Figure 4.2 shows the interconnection of these elements in the RPi cluster we built.

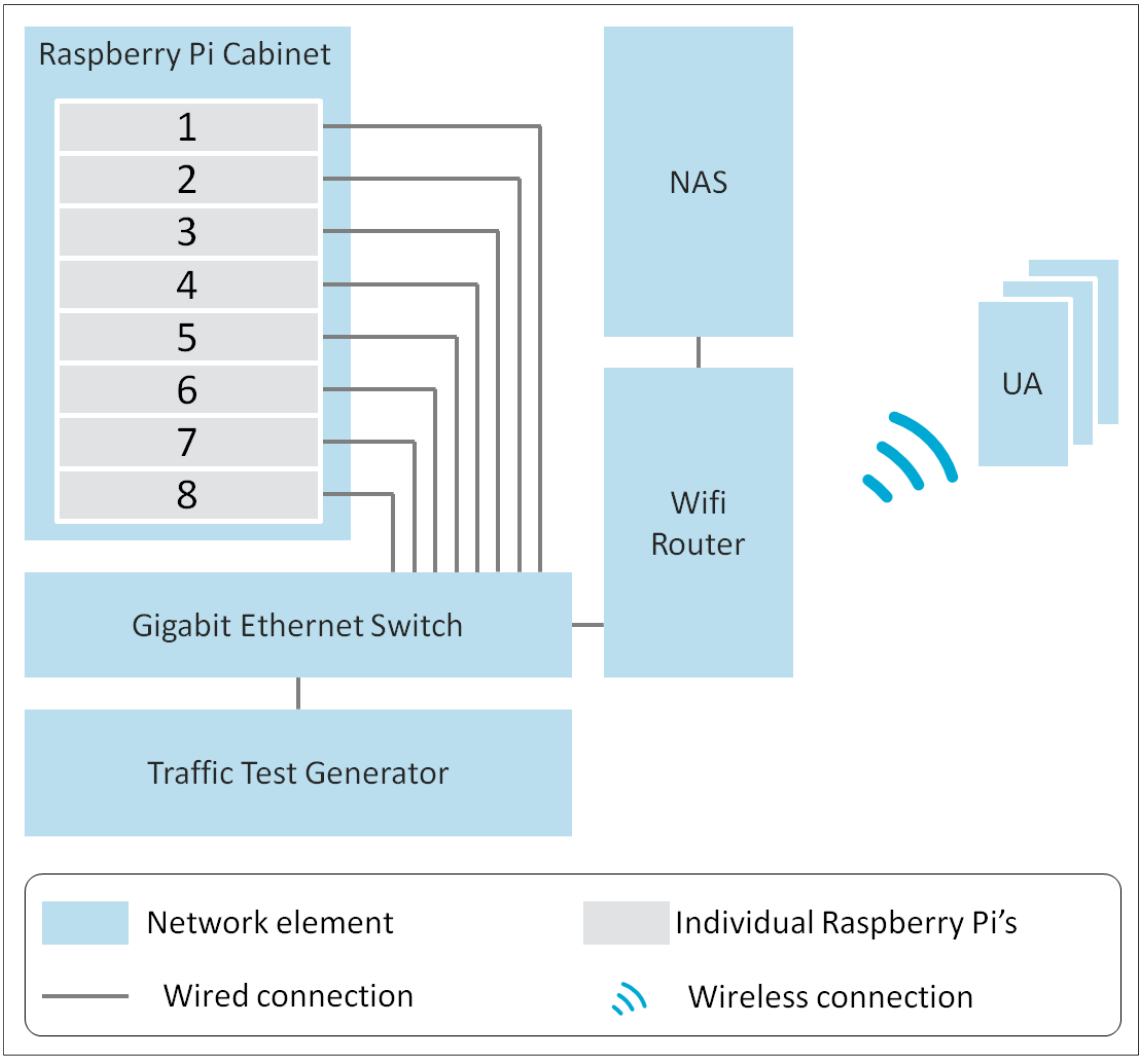

Figure 4.2 Hardware experimentation setup for Raspberry Pi cluster

For the purpose or holding securely the Raspberry Pi's we designed and 3D printed eight drawers which can be inserted in the main cabinet. Figure 4.3 shows one of the drawers holding a RPi, and one can see the location openings on the front where the LEDs and the button is located on the drawer. The LEDs are used as an indication of the CPU and network bandwidth usage and the button to perform a restart or shut down for each of the RPi boards.

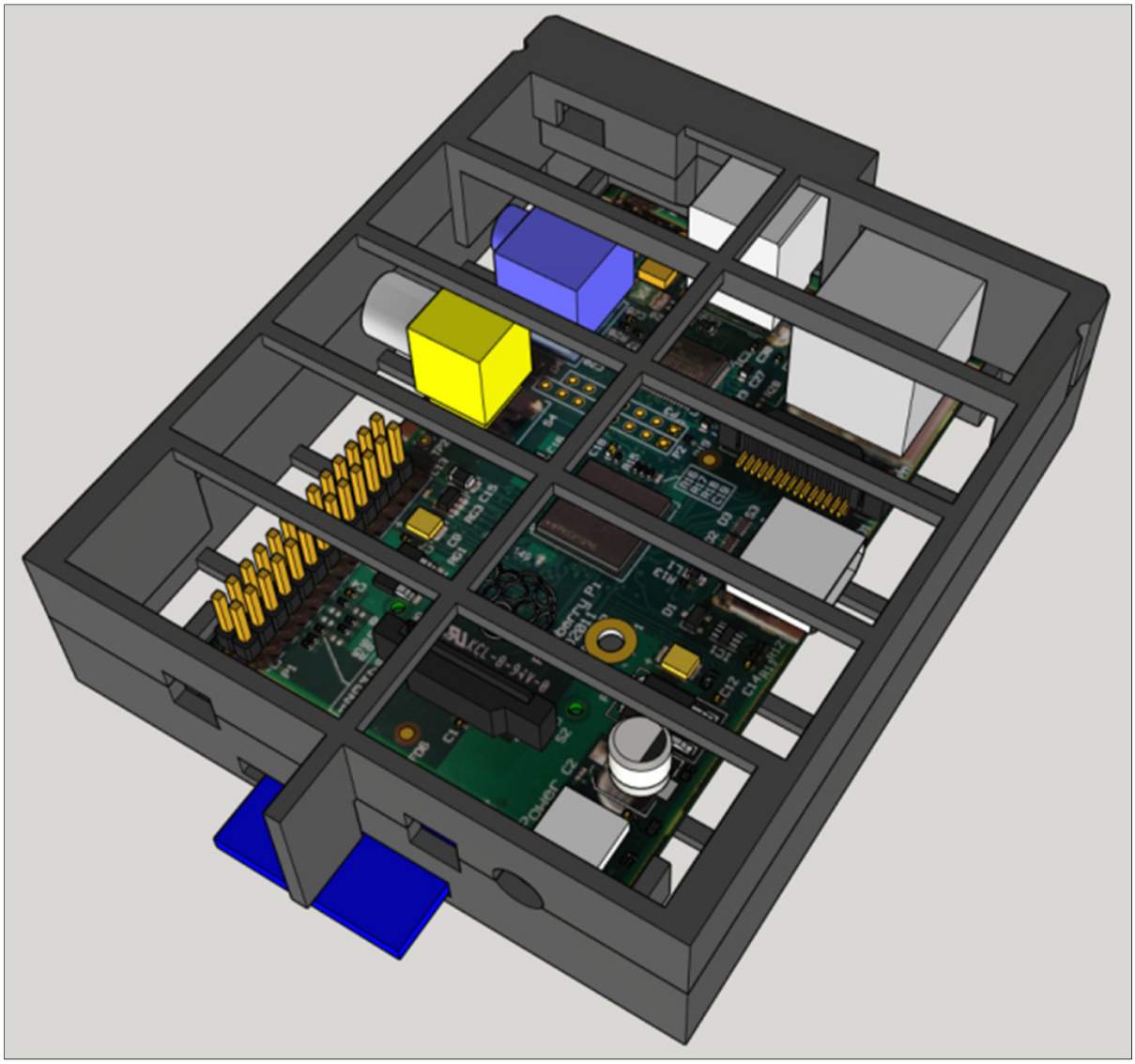

Figure 4.3 One Raspberry Pi drawer designed to slide in the cabinet

Figure 4.1 shows the design of the 3D printed cabinet used to hold the eight drawers containing the eight RPi boards and their daughter boards with the RGB LEDs and buttons.

Finally, Figure 4.4 shows the electrical diagram of the Daughter Board, enabling connection of the RPi to two RGB LEDs controlled via SPI protocol and terminations on the RPi and the button which is mapped to a specific General Purpose Input and Output pin (GPIO 23) on the RPi.

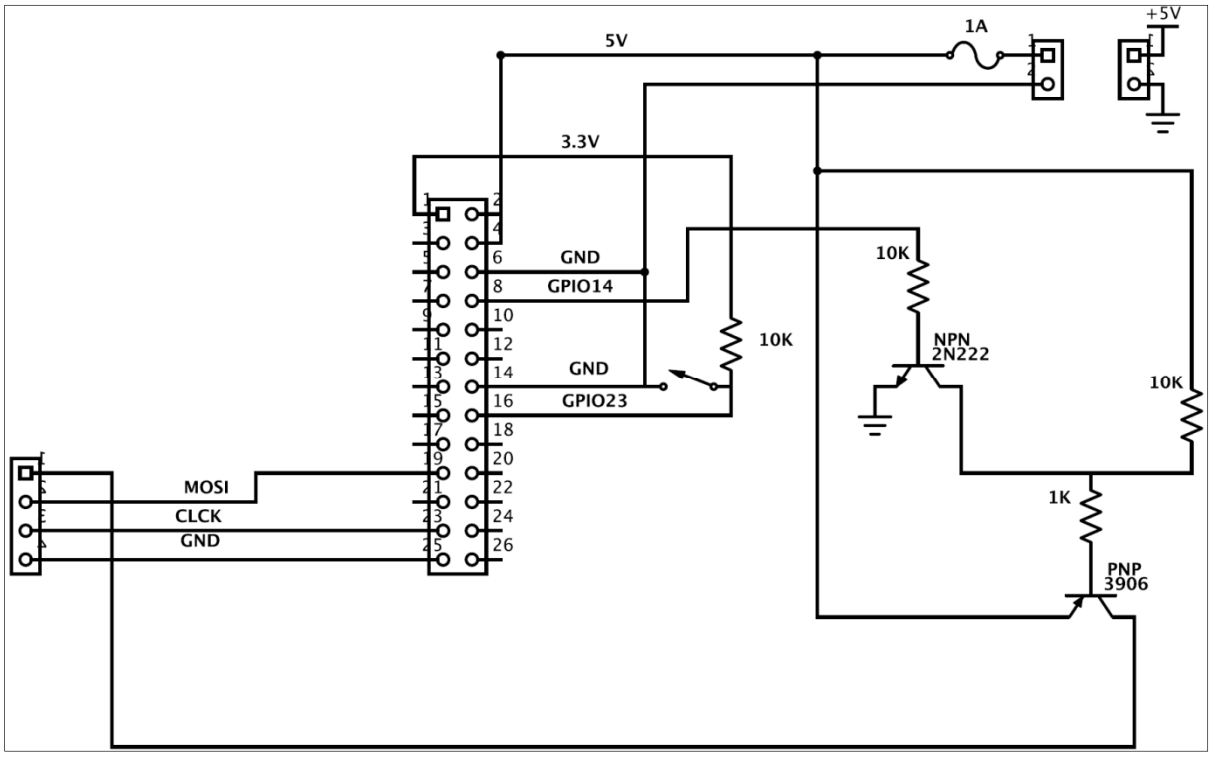

Figure 4.4 Custom RPi daughter board driving two RGB LEDs and one switch

4.1.1 Pouch allocation on Raspberry Pi

On the Raspberry Pi cluster we built, we deploy the *Raspberry Pi Deployment – Protocol Interface / Database pouch* and the *Raspberry Pi Deployment – Core IMS pouches* as defined in the previous chapter. This deployment is illustrated in Figure 4.5.

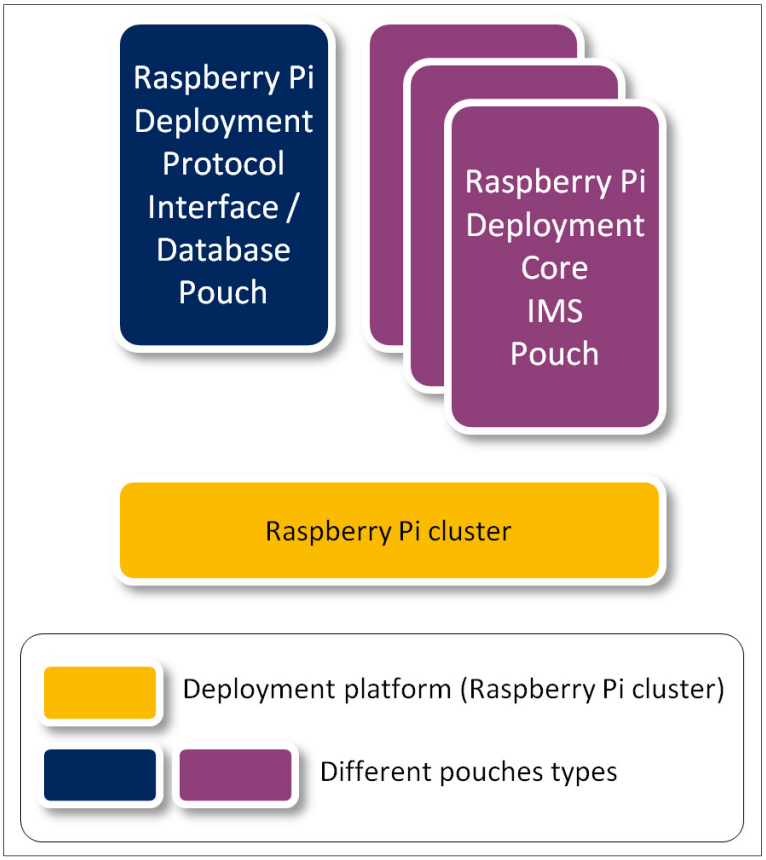

Figure 4.5 Pouch allocation on Raspberry Pi

4.2 OpenStack managed cluster deployment

The second deployment was made on top of OpenStack (an open source IaaS platform providing virtualization), deployed on a cluster of five Dell PowerEdge R620 Servers, which are interconnected through a 1 Gigabit Ethernet switch from Extreme Networks. The first server is used as the OpenStack controller and the four remaining host OpenStack compute instances.

Each of the servers has the following characteristics:

- 2 Intel Xeon CPU E5-2640 @ 2.50GHz,
- 96 GB of RAM,

- 2 Seagate 1 TB Hard Drive (model ST91000640SS) in a RAID configuration,
- 4 Gigabit Ethernet ports (one being dedicated to the messages exchanged via the CMW).

4.2.1 Pouch allocation on OpenStack managed cluster

For the OpenStack managed cluster, we deploy each on their own VM one of each of the *Database pouch*, the *Protocol Interface pouch* and the *Load Balancing pouch* detailed in the previous chapter. We also instantiate on individual VMs at least one of the *Control Plane pouches* and the *User Plane pouches*, and allow them to scale-in or scale-out on a need basis, starting and stopping VMs running those pouches as required. This deployment is illustrated in Figure 4.6.

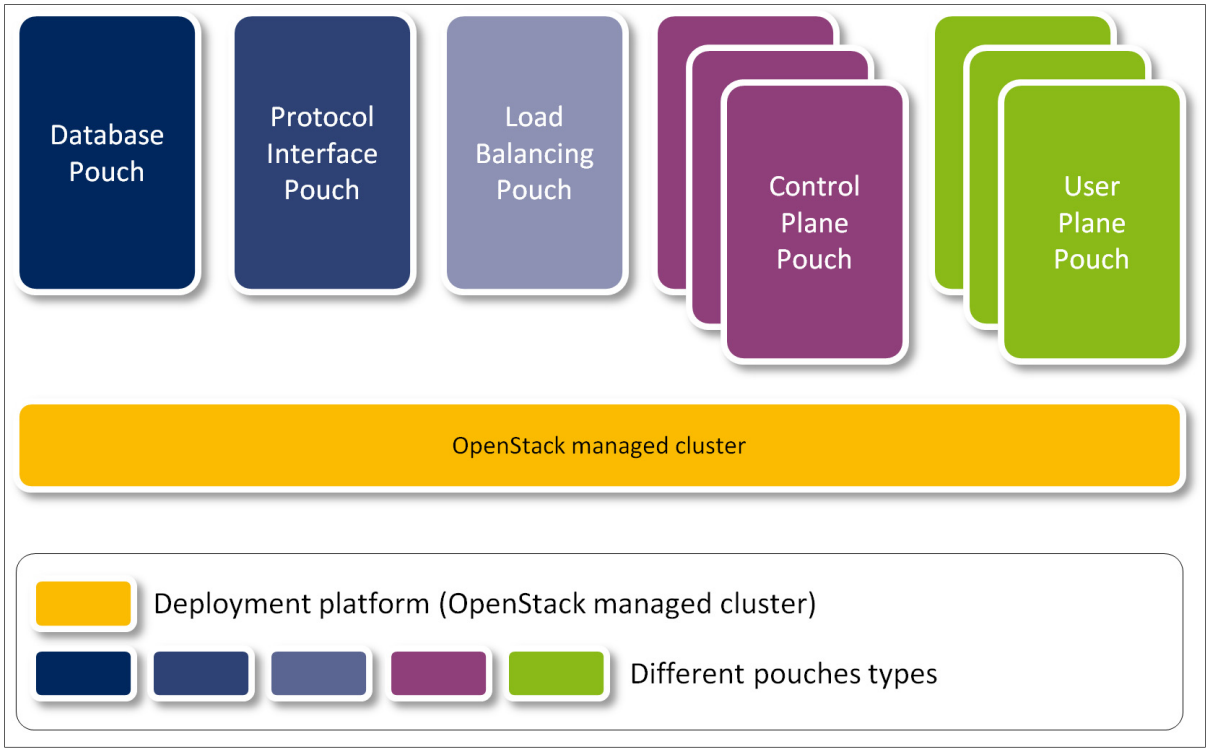

Figure 4.6 Pouch allocation on OpenStack managed cluster

4.3 Hybrid deployment on OpenStack and Apcera Cloud Platform

For the case of the hybrid deployment on OpenStack and Apcera Cloud Platform (a PaaS platform providing containers), the Apcera Cloud Platform was deployed on top of two of the OpenStack managed servers we detailed in section 4.2. Two other servers were used for the OpenStack VM deployments. The last server is used as the OpenStack controller.

4.3.1 Pouch allocation for the hybrid deployment

For the hybrid deployment on OpenStack and Apcera Cloud Platform, we deploy on OpenStack each on their own VM one of each, the *Protocol Interface pouch* and the *Load Balancing pouch* detailed in the previous chapter. We also instantiate on individual VM at least one *User Plane pouch* and allow it to scale-in or scale-out on a need basis, starting and stopping VM running those pouches as required.

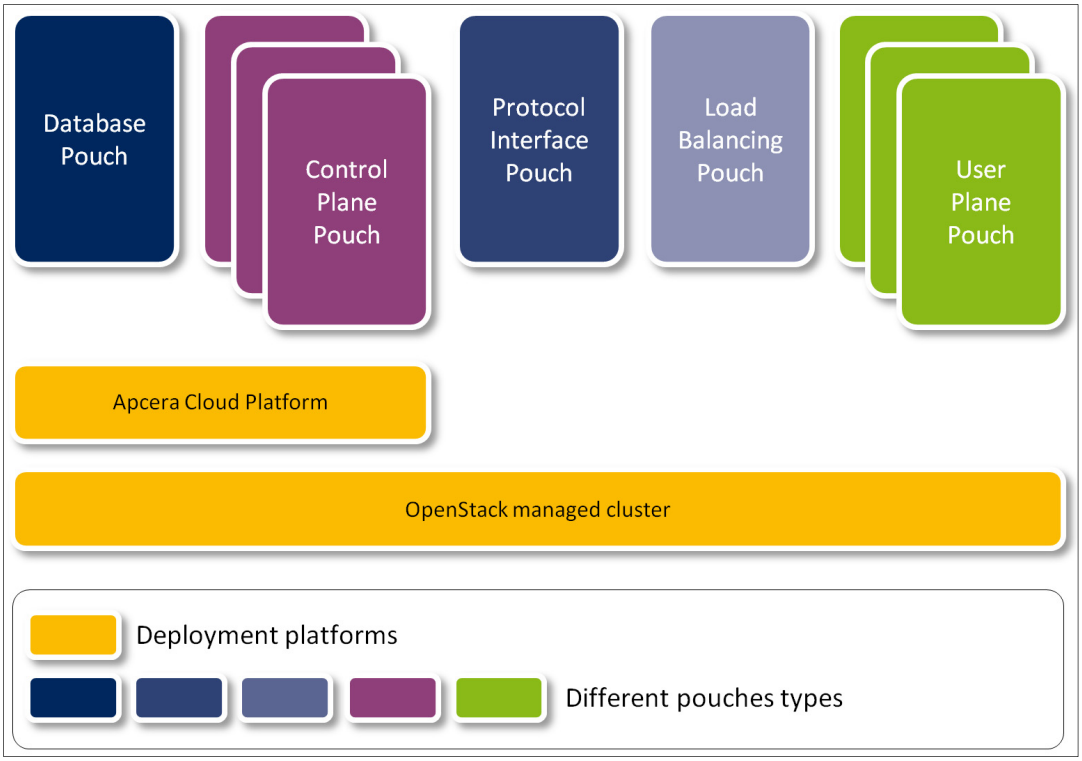

Figure 4.7 Pouch allocation for the hybrid deployment

On Apcera Cloud Platform, we deploy in its own container the *Database pouch*. We also instantiate on an individual container at least one *Control Plane pouch*. We allow it to scale-in or scale-out on a need basis, starting and stopping containers running those pouches as required.

This deployment is illustrated in Figure 4.7.

4.4 Activity display

In order to visualize the activity on the cloud architecture implementation, we developed a rudimentary activity display which shows the instantiated units (colored circles), how they are grouped together to serve a call (vertical slices) and how they are spread across the available computing resources.

On this display depicted in Figure 4.8, we see the different computing resources on the cloud named by the IP addresses they were allocated via DHCP. Then the first column (marked with a “1”) represents the permanent units on the System (as detailed in the Architecture section) of the framework and also of the application. The next columns (marked with a “2”) are each a separate instance of a call between two subscribers with the Originating side (with the C, A and T units and while in call establishment we can also see the O unit), the Terminating side (again C, A and T units) and the Media Processor (M unit) shared by both sides. In this specific example, we have seven calls established. As shown, units processing calls are spread all over the cloud and exist only for the purpose of serving those calls, disappearing once the calls terminate (technically 30 seconds later for most actors for timeout supervision control purpose to respect the SIP RFC). This represents an instantaneous snapshot of the system state, i.e. which units are running where at the screen capture time.

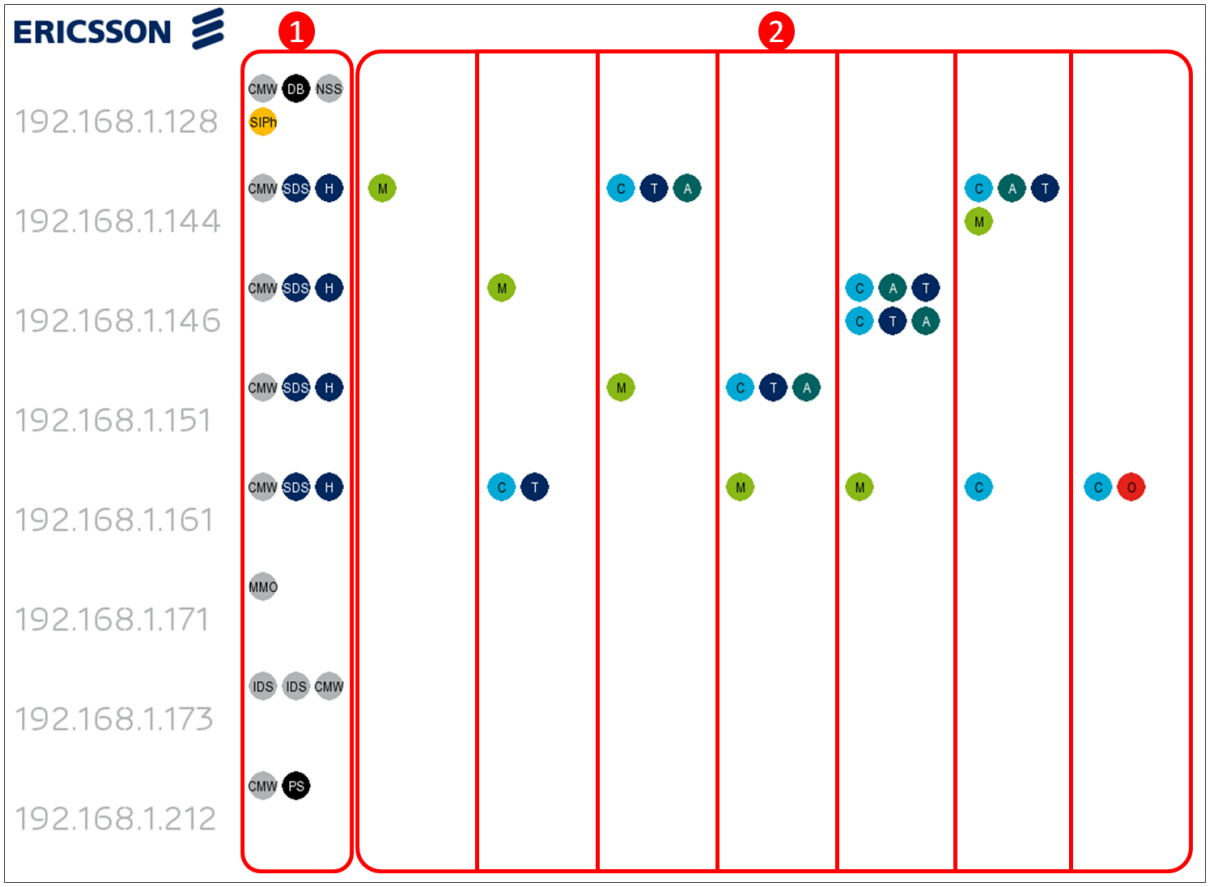

Figure 4.8 Activity display main screen

A second view is available on the activity display where we can see an historical view of the number of calls established at a specific time, and the number of pouches instantiated at that time. In Figure 4.9, we see in blue the traffic we generated toward the system following a sinusoid curve, between no calls at all and 48 calls at maximum traffic (in blue, scale on the left). At the same time, we see the number of pouches that were instantiated at that time, which varies between five and eight pouches (in green, scale on the right).

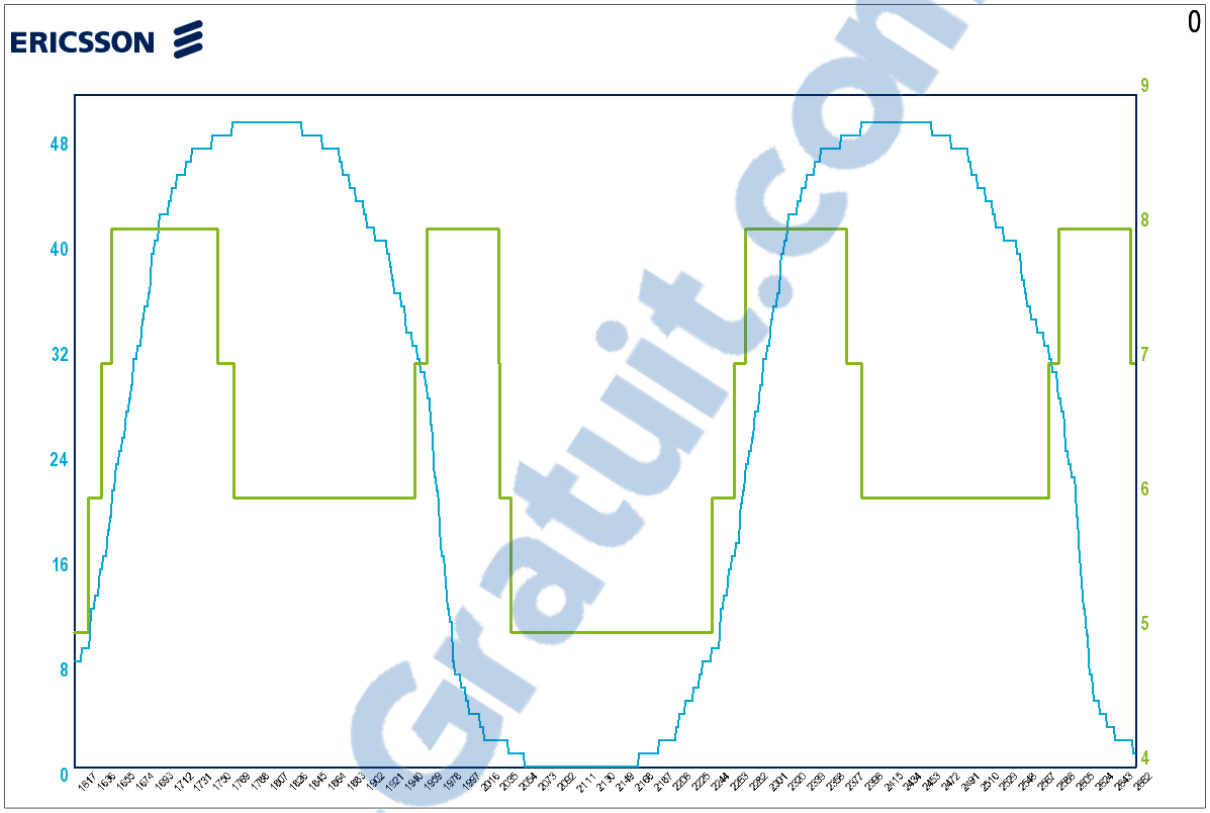

Figure 4.9 Activity display historic data view

4.5 Measurement scenario

Our experiment consists in making a number of two-way SIP voice calls between two subscribers via our simplified IMS system. For each individual experiment, we mention the experiment parameters:

- Number of calls established per unit of time;
- Hold time of individual calls;
- Maximum number of concurrent calls reached before we stop establishing new calls;
- Number of iterations of the experiment performed, i.e. once the maximum number of concurrent calls is reached, we wait until all ongoing calls terminate and then we start a new iteration.

Our measurement approach consists of a collection of time stamped in-process logs which are collected in a file on our system. These logs are then post processed to generate the various graphs we show in the next sections. In order to generate the traffic required to take those measurements, we used a free and lightweight open-source tool, SIPp (SIPp, 2014).

Most of our latency curves (unless otherwise stated) measure the delay from the INVITE reception by our system until it is sent to the terminating UA. This way, we keep the measurement to the portion which is directly dependent of our system processing and keep the measure independent from UA delays.

4.6 SIPp

SIPp is a test tool enabling traffic generation for the SIP protocol and enables registration of UAs via the REGISTER message, and establishment and release of multiple calls via the INVITE and BYE messages.

4.7 Discussion

In chapter 3, we developed a cloud-native software architecture for telecommunication systems and realized an implementation of the architecture through a simplified re-architected IMS. In this chapter, we described how we deployed this telecommunication application on various heterogeneous and hybrid platforms. SIPp is used to generate the traffic towards the telecommunication application and an Activity Display can be used to visualize the internal working of the application. In the next chapter, we will present the results of executing the telecommunication application on those platforms with generated traffic.

CHAPTER 5

VALIDATION AND RESULTS

In this chapter, we present some of our experimental results. Most of the units developed in the simplified IMS system following our proposed architecture are involved in the establishment of a two-way call session. As such, we use the two-way call establishment procedure as a reliable indicator of the overall performance of the system, especially on the control plane side. Once the call is established, the voice call is handled by the media processor units and does not require much interaction with other units. Therefore, the media plane unit performance is a good indicator of the overall performance of the system on the user plane side.

5.1 Measurements on Raspberry Pi

The first experiment is carried out to measure the characteristics of our architecture. We specifically look at the latency of call establishment, the time between the processing of each voice frame, the CPU usage and the memory usage as we load the system with more calls. For this experiment, we did not allow elasticity by setting the number of Core IMS Pouches on the system to a fixed value.

We used the following parameters:

- Call rate: 25 two-way calls established / minute,
- Call duration: 4:10 minutes,
- Maximum number of concurrent calls: 60,
- Number of rounds: 20,
- Subscribers: 200 registered users.

5.1.1 Control plane measurements

In this section, we specifically look at the latency of call establishment as we load the system with more calls.

We initially noticed unusual 40 ms delays introduced in an apparent random fashion when sending TCP messages. It appears we were suffering from Nagle's algorithm (Cheshire, 2005) which accumulates outgoing data until the maximum segment size is reached or an Acknowledgement message (ACK) is received. For small packets, this has the effect of basically capping the latency to the round-trip time for the message. By disabling that algorithm (NOWAIT flag in the TCP socket parameters), we were able to reclaim a number of 40 ms induced latency, leading to a slightly better latency curve for the call establishment pictured in Figure 5.1.

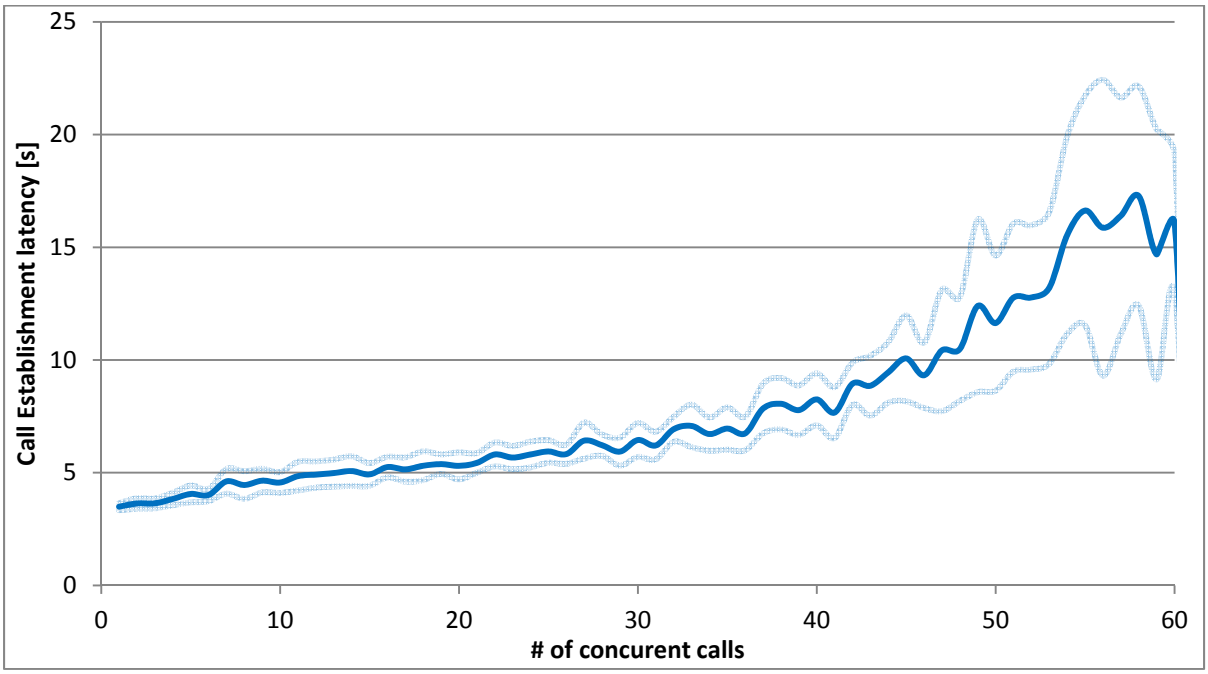

Figure 5.1 Average and standard deviation of the call establishment latency on Raspberry Pi cluster

We can see that the latency of call establishment follows pretty much a linear increase with the standard deviation (pictured in light blue), relatively constant until a breaking point at around 40 calls, where the standard deviation starts to increase sharply and the latency picks up at a higher rate.

Figure 5.2 also depicts the call establishment latency, but this time versus the average CPU load on the pouches. Two curves are provided, as we look at the average CPU load on the originating and terminating side. Both curves are similar. We can see the same behavior as before, where latency increases linearly until a break point at around 65% average CPU load, where the rate of increase of latency increases faster. As in this experiment we were not allowing for elasticity, there was no way for the system to cope with the degrading QoS figures. In a system where we would allow elasticity, the system would regulate itself to a maximum latency value until all resources are exhausted.

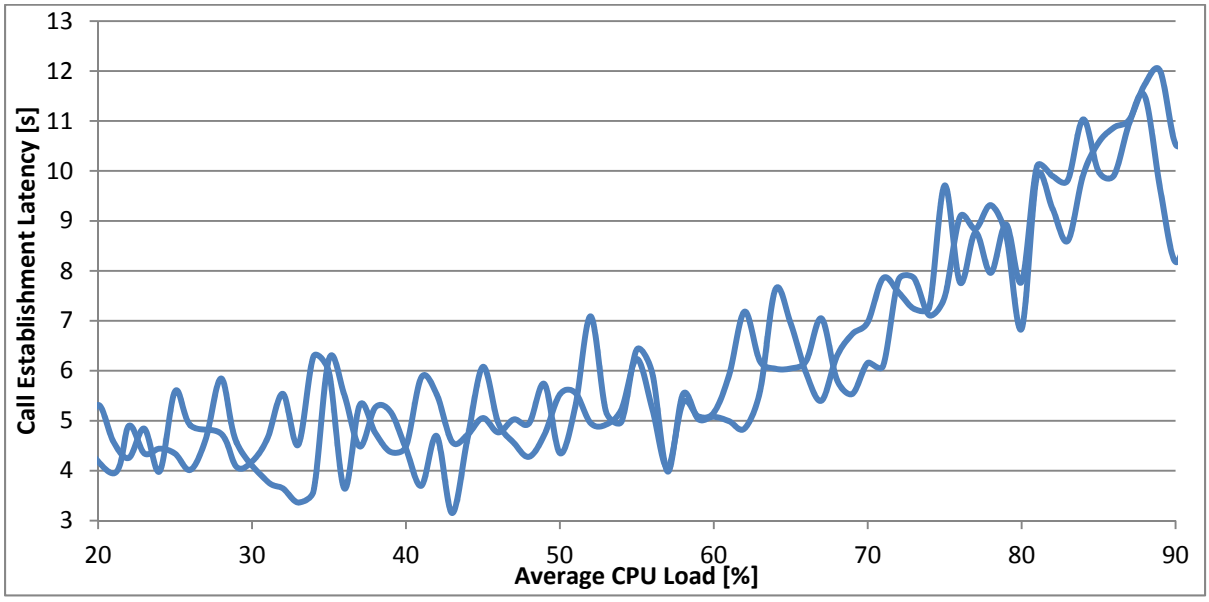

Figure 5.2 Call establishment latency vs. the average CPU load on Raspberry Pi cluster

This shows that above the QoS control we have through fast scaling operations (elasticity), we also need an admission control system which would potentially reject calls if no more

resources are available for a scale-out operation, in order to avoid affecting the QoS of all currently established calls.

Still we can see for loads below 65%, we experience a linear increase on the call establishment latency with the number of calls increasing, thus a predictable behavior.

5.1.2 Data plane measurements

In this section, we specifically look at the time between the processing of each voice frame as we load the system with more calls.

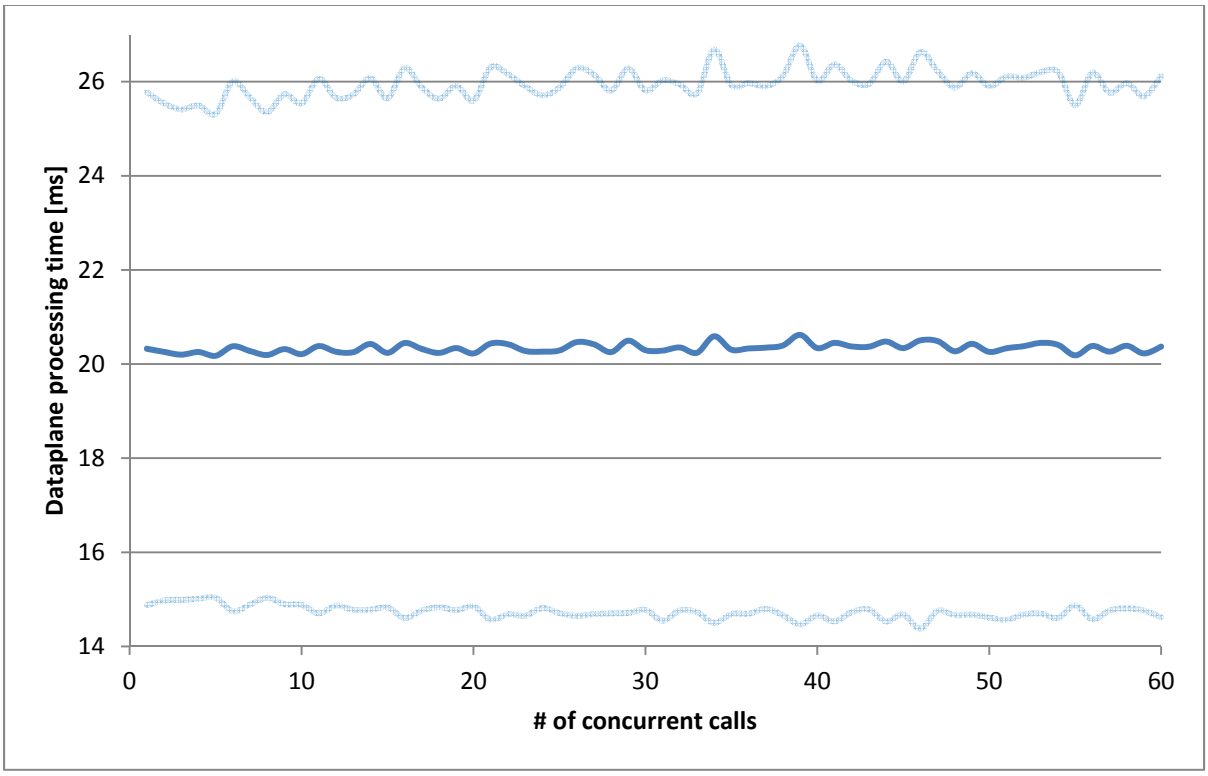

Figure 5.3 Data plane processing time vs. the number of concurrent calls on Raspberry Pi cluster

Ideally we should process frames every 20 ms. However, due to the load on the computing units, that delay may vary. Figure 5.3 shows the average time between the processing of each

frame (20 ms) and the standard deviation (lighter shade of blue) from that ideal processing time, based on the number of concurrent calls being processed.

We can see that the system stayed in its normal operation zone, as the standard deviation did not vary much. This is a desirable characteristic from the point of view of the QoS. Even though the call establishment rises, the actual call processing occurs on time, insuring good voice quality.

5.1.3 CPU usage vs number of calls

In this section, we specifically look at the average CPU load on the system as we load the system with more calls.

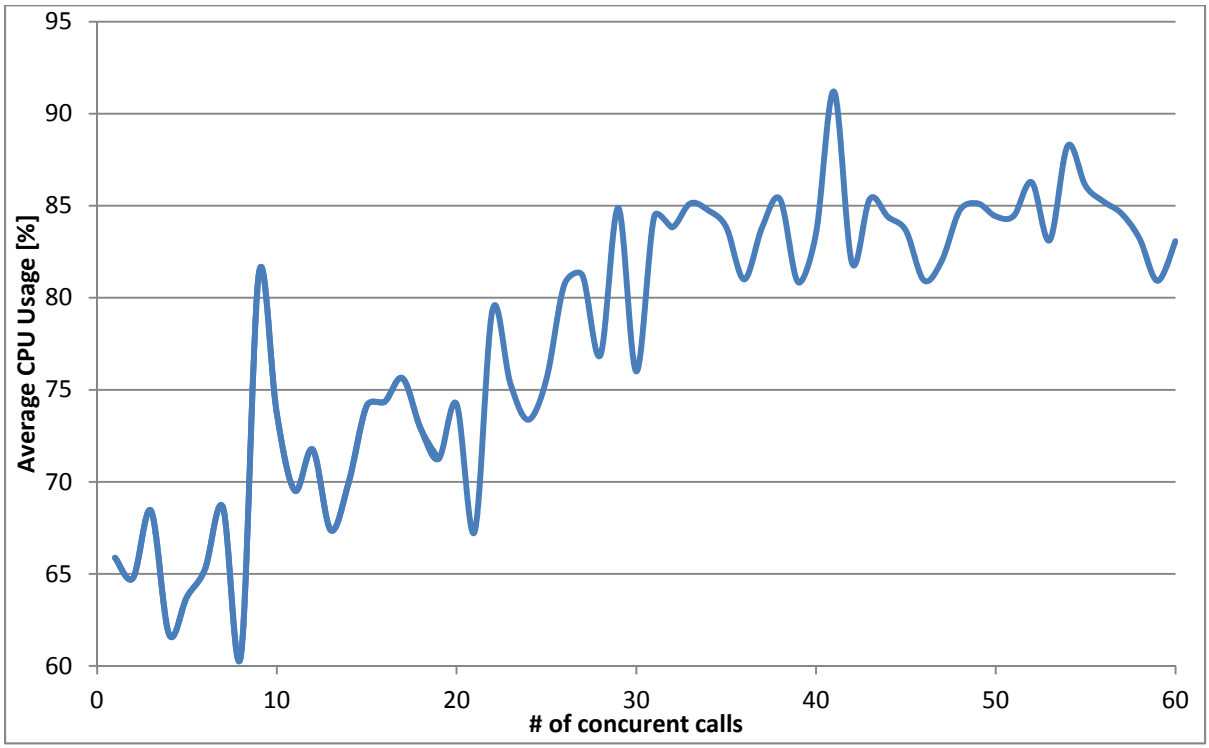

Figure 5.4 CPU usage vs. the number of concurrent calls on Raspberry Pi cluster

Ideally the CPU usage in a cloud-based system should be a linear function of the number of service instances. We see this desired linearity in Figure 5.4, where until the point of 40 concurrent calls, the CPU usage increases pretty much linearly with the number of calls.

If we remember the control plane measurements in section 5.1.1, the 40 calls point is where the call establishment latency starts to rise sharply, thus everything is consistent. When the system reaches an average CPU load of 85%, a lot of time is spent actually at 100% CPU, and only some of the time is spent below this level. When the system reaches 100% usage, it can no longer maintain the expected linear increase in the call establishment latency.

This experiment confirms that for the current system, the actual limit is probably around a maximum of 40 concurrent calls. It is important to note that this experiment is aimed at conceptually validating the proposed architecture, and uses very limited computing hardware. Our goal here was not to reproduce the capacities of large scale equivalent systems. Nonetheless, reaching 40 concurrent calls is better than we could have expected of such a limited setup.

5.1.4 Memory usage vs number of calls

In this section, we specifically look at the average memory (RAM) usage on the system as we load the system with more calls.

As for the CPU usage figure, we would expect the memory usage to display the same linear increase with the number of calls. We can again see that linearity in Figure 5.5. Thus the usage of memory is predictable and is a function of the number of concurrent calls.

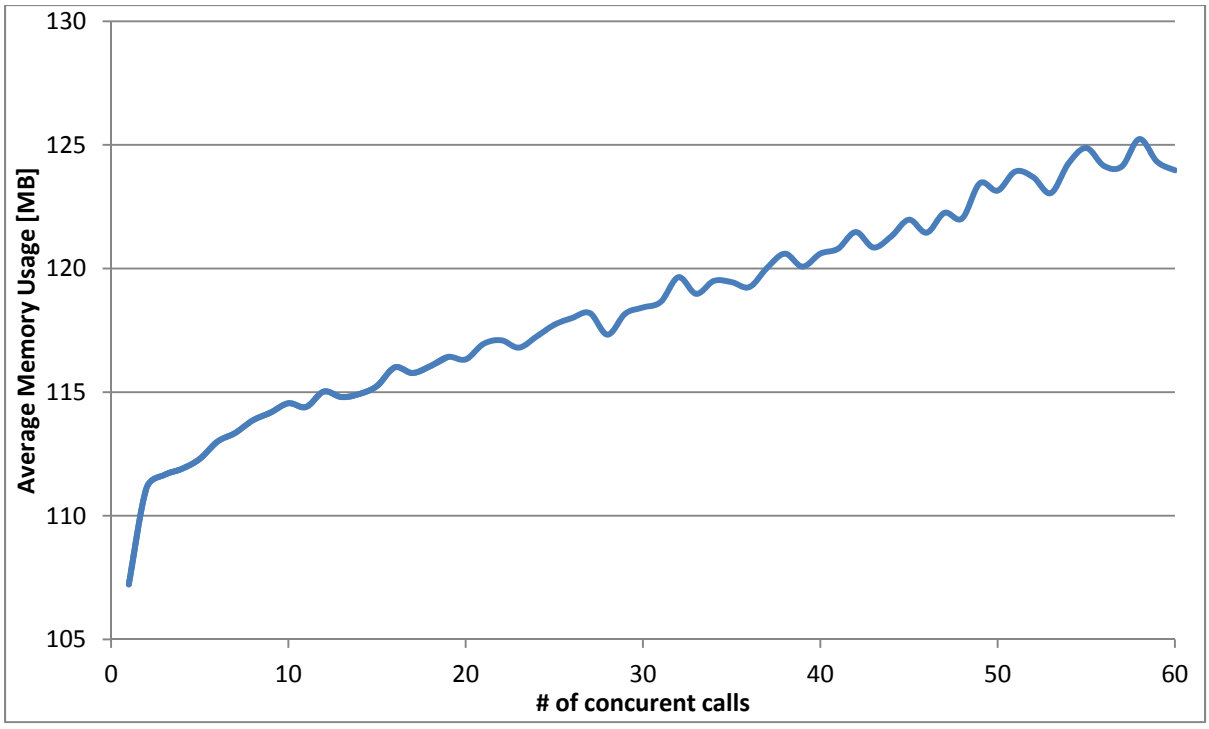

Figure 5.5 Average memory usage vs. the number of concurrent calls on Raspberry Pi cluster

5.2 OpenStack measurements

The second experiment is carried out again to measure the characteristics of our architecture. This time, the experiment is performed on an OpenStack deployment, but consists of the same type of traffic and measurements as those performed in the first experiment. We specifically look at the latency of call establishment, the time between the processing of each voice frame, the CPU usage and the memory usage as we load the system with more calls. For this experiment, we did not allow elasticity, by setting the number of pouches on the system to a fixed value. One of each of the Control Plane pouch, Database pouch, Protocol Interface pouch, Load Balancing pouch and User Plane pouch we deployed. Moreover, we limited the OpenStack cluster server usage to a single server in order to measure the performance of that single server.

We used the following parameters:

- Call rate: 60 two-way calls established / minute,
- Call duration: 9 minutes,
- Maximum number of concurrent calls: 500,
- Number of rounds: 20,
- Subscribers: 200 registered users.

5.2.1 Control plane measurements

In this section, we specifically look at the latency of call establishment as we load the system with more calls.

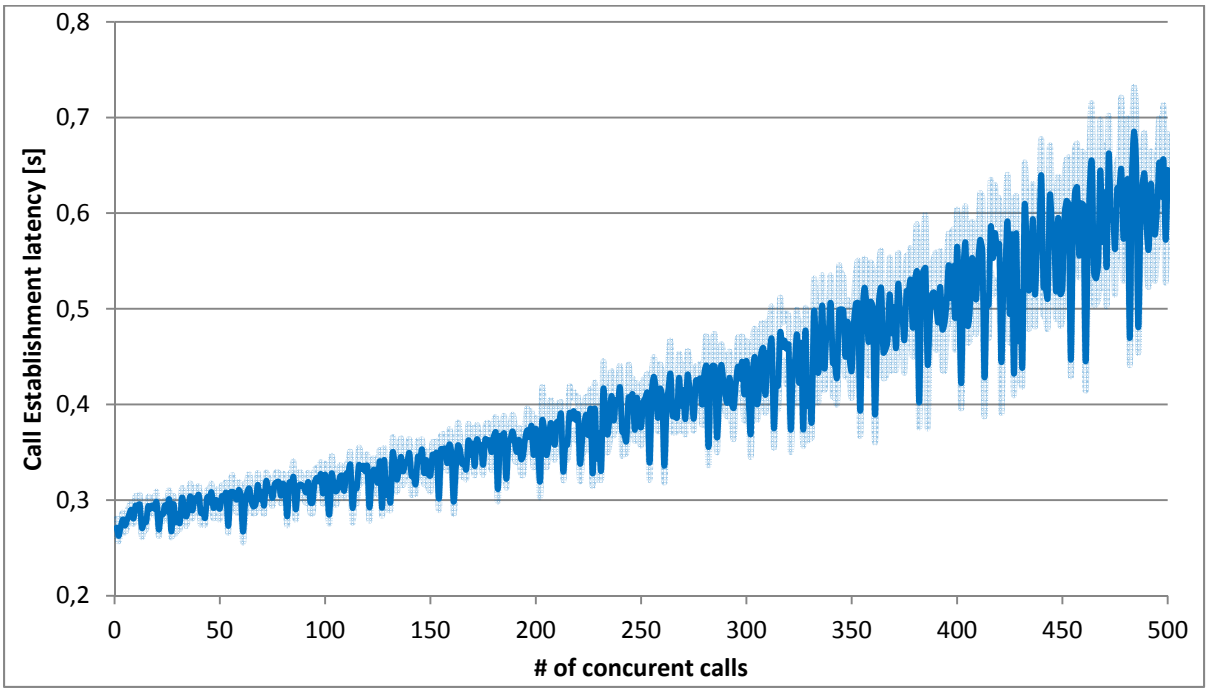

Figure 5.6 Call establishment latency vs. the number of concurrent calls on OpenStack cluster

Figure 5.6 shows a smoother increase on the call establishment latency than what we had on the Raspberry Pi platform. The system could probably take more load before experiencing the same behavior as what we saw on the Raspberry Pi. However, as before, QoS control and

admission control would need to be taken into account in order to maintain the call establishment time at acceptable values.

Figure 5.7 depicts the call establishment latency but this time versus the average CPU load on the pouches. We see the latency increasing relatively linearly with the increased CPU load.

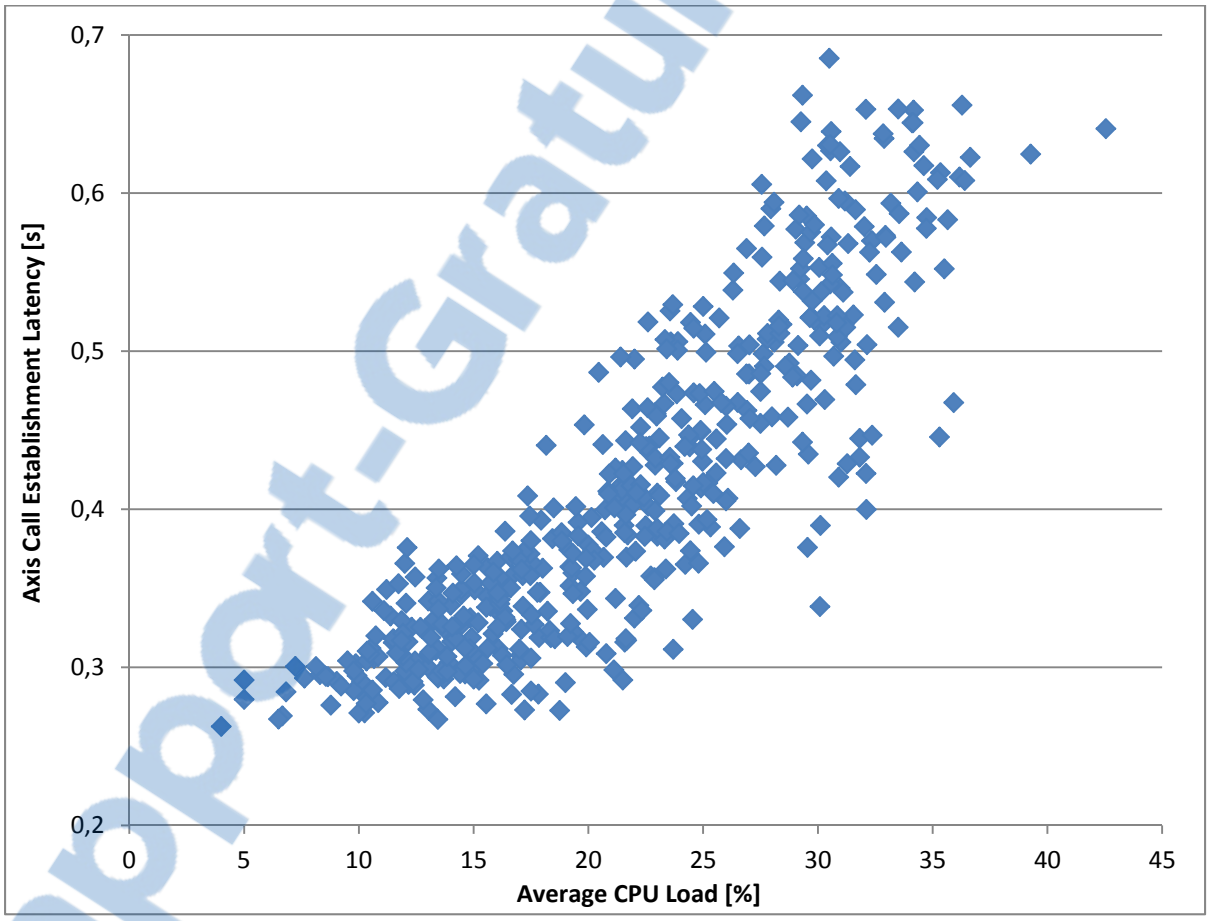

Figure 5.7 Call establishment latency vs. the average CPU load on OpenStack cluster

For the OpenStack deployment, an average CPU load of 40% is still in the low range. It is therefore expected that we see a linear relation. If we were to stress the system at a higher load, we could expect to see a behavior equivalent to what we saw on the Raspberry Pi

deployment, where at around 65% average CPU load, the rate of increase of latency is greater.

However, for similar ranges, we can observe similar behavior between the Raspberry Pi platform and the OpenStack platform. The call capacity is different, obviously, as the OpenStack platform can handle a lot more load at a higher rate, but the behavior stays the same.

5.2.2 Data plane measurements

In this section we specifically look at the time between the processing of each voice frame as we load the system with more calls.

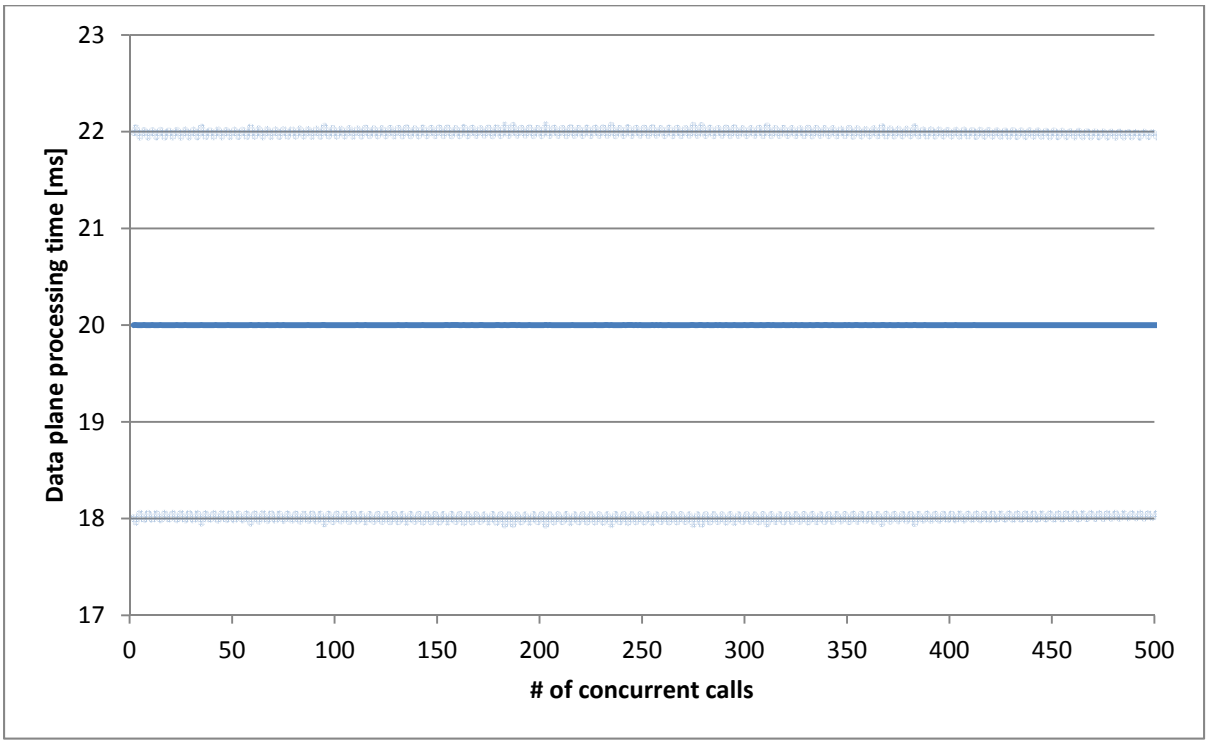

Figure 5.8 Data plane processing time vs. the number of concurrent calls on OpenStack cluster

Ideally, as we mentioned earlier, we should process frames every 20 ms. However, due to the load on the computing units, that delay may vary. Figure 5.8 shows the average time between the processing of each frame (20 ms) and the standard deviation (lighter shade of blue) from that ideal processing time, based on the number of concurrent calls being processed.

As for the case of the Raspberry Pi platform, we can see that the system stayed in its normal operation zone, as the standard deviation did not vary much. In fact, having a more powerful platform helped lower the standard deviation noticeably. This is again desirable from the point of view of QoS, as even though the call establishment increases, the actual call processing occurs on time, thus insuring good voice quality.

5.2.3 CPU usage vs number of calls

In this section we specifically look at the average CPU load on the system as we load the system with more calls.

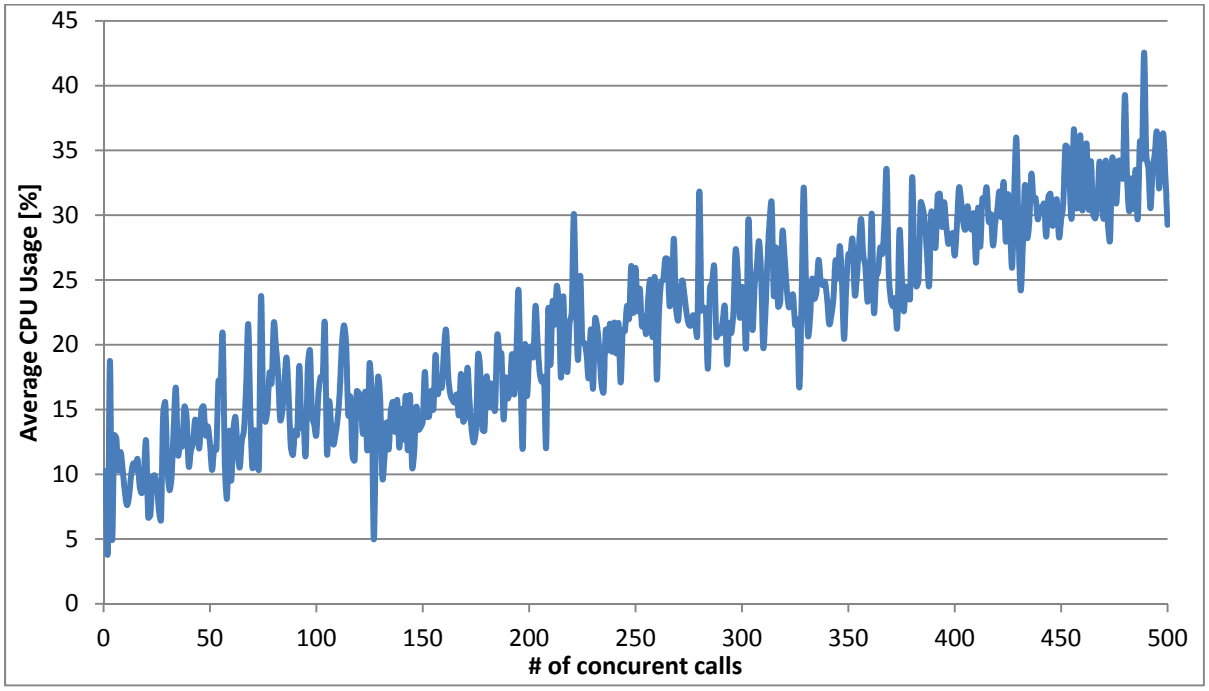

Figure 5.9 Average CPU usage vs. the number of concurrent calls on OpenStack cluster

We once again see the desirable linearity in Figure 5.9, where the CPU usage increases linearly with the number of calls. We do not reach the point where the CPU is the bottleneck on the OpenStack deployment.

5.2.4 Memory usage vs number of calls

In this section we specifically look at the average memory (RAM) usage on the system as we load the system with more calls.

We once again note the expected linearity in Figure 5.10. Thus the usage of memory is predictable and is a function of the number of concurrent calls.

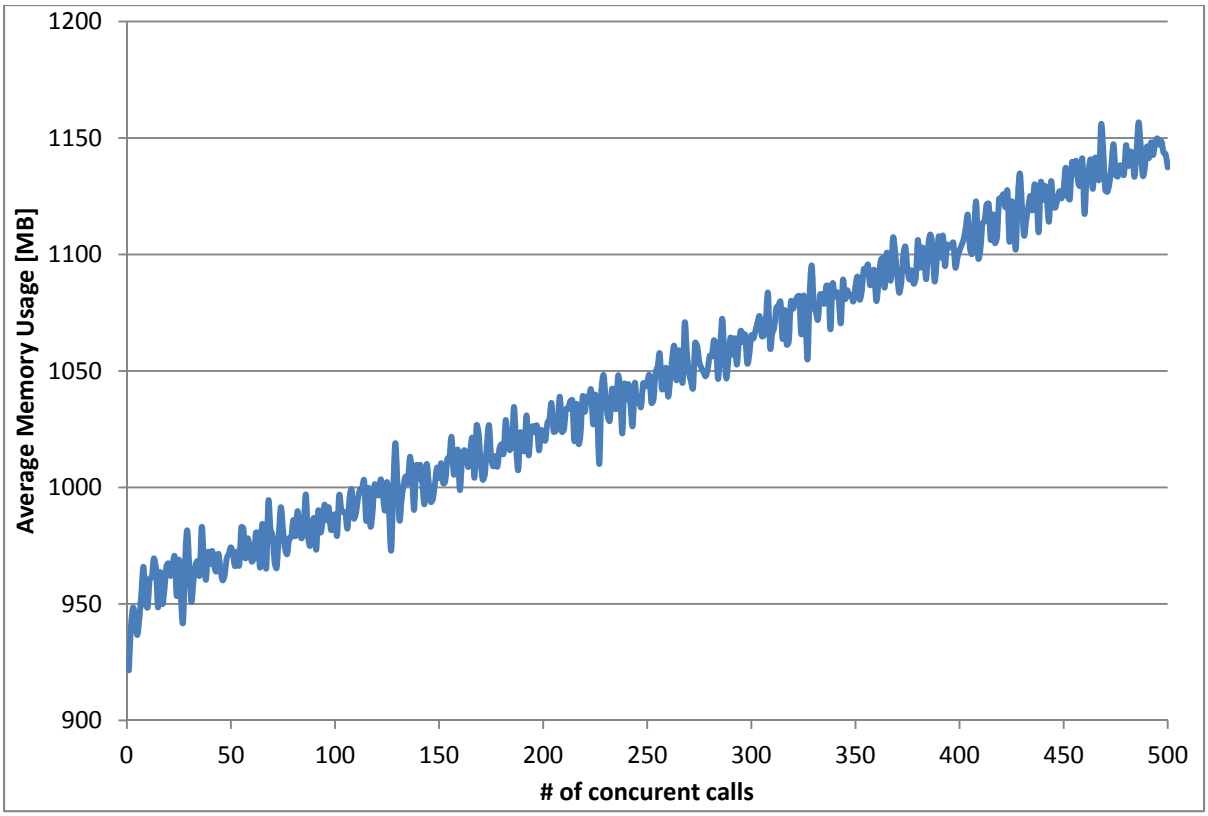

Figure 5.10 Average memory usage vs. number of concurrent calls on OpenStack cluster

5.3 Comparison of direct and tunnelled communication

The next experiment is carried out to measure the effect of tunnelling communication through the CMW, when actors are on different pouches, versus accessing them directly. When using TCP, we have the possibility to send messages directly. Thus we evaluate the effect using the TCP transport mechanism on the Raspberry Pi deployment. For this experiment, we did not allow elasticity, by setting the number of Core IMS pouches on the system to a fixed value.

We used the following parameters:

- Call rate: 10 two-way calls established / minute,
- Call duration: 5:30 minutes,
- Maximum number of concurrent calls: 50,
- Number of rounds: 20,
- Subscribers: 200 registered users.

Figure 5.11 shows the effect of tunnelling communication through the CMW when we have to access units in other pouches. Since most of the traffic is collocated on a same pouch, and since communication is only a part of the processing, adding two hops in order to tunnel the communication has the overall impact of increasing the latency by about 20% for the whole call establishment process. This shows the importance of making sure the tunnelling communication is optimized.

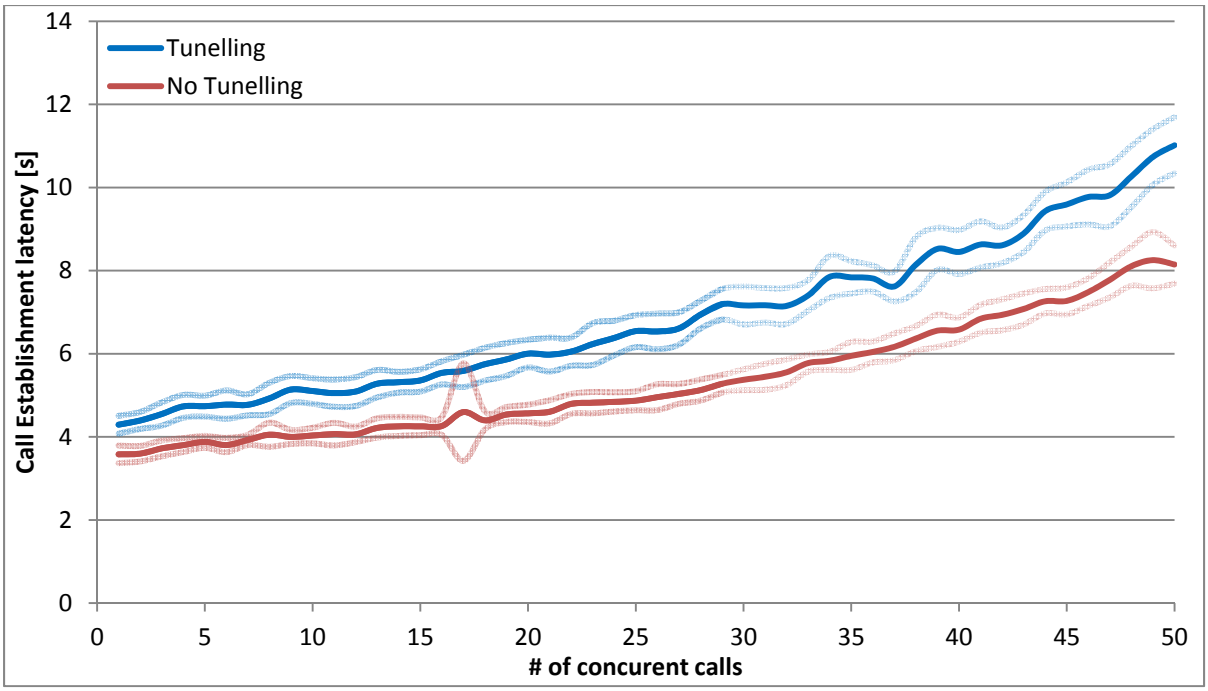

Figure 5.11 Call establishment latency with and without tunnelling vs. the number of concurrent calls

5.4 Comparison of HTTP and TCP tunnelling

The next experiment is carried out to measure the effect of using the TCP or HTTP transport mechanism, when using tunnelled communication through the CMW. When using the HTTP transport, we do not have a choice to use tunnelling communication, as the number of HTTP ports on a target computing resource is limited. Still, this only affects messages that have to go outside a pouch. We evaluate the effect of using the TCP or HTTP transport mechanism on the OpenStack deployment. For this experiment we did not allow elasticity by setting the number of pouches on the system to a fixed value. One of each of the Control Plane pouches, Database pouch, Protocol Interface pouch, Load Balancing pouch and User Plane pouch we deployed. Moreover, we limited the OpenStack cluster server usage to a single server in order to measure the performance of that single server.

We used the following parameters:

- Call rate: 30 two-way calls established / minute,
- Call duration: 4 minutes,
- Maximum number of concurrent calls: 100,
- Number of rounds: 20,
- Subscribers: 200 registered users.

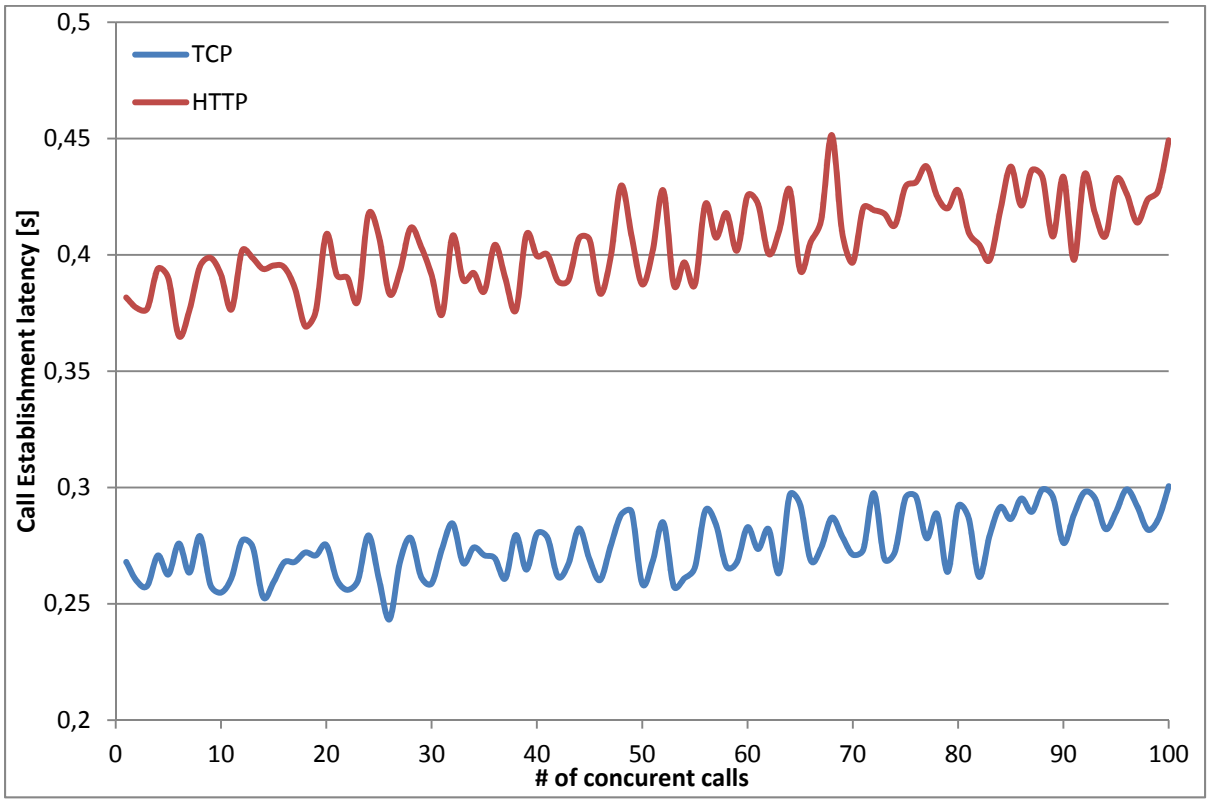

Figure 5.12 Call establishment latency with TCP and HTTP tunnels vs. the number of concurrent calls

Figure 5.12 shows the effect of using HTTP transport instead of TCP transport. We see an increase on the latency in the range of 40% with HTTP. Our analysis of the situation indicates that we would need to multi-thread the wait for the acknowledgement response when using HTTP. In our current implementation, we have a single threaded process. Therefore, waiting for an acknowledgment becomes a blocking operation for the reception of

other messages and no other processing can be done. This is not the case with the TCP transport mechanism, as we do not have to wait for an acknowledgement.

Moreover, we had to limit the rate and the number of concurrent calls in order not to reach an unstable state. Hopefully, usage of multi-threading would solve the issue with HTTP transport, but this would needs to be confirmed, especially since a 40% increase in latency is worrisome.

5.5 Comparison of distributed and node-based architecture

In order to compare the proposed microservice-based architecture to the currently prominent node-based architecture adopted by telecommunication equipment vendors, we conducted a set of experiments where the functions developed for our proposed architecture were statically bound to a specific computing unit, thus replicating the node-based architecture. We performed this experiment on the Raspberry Pi deployment, where we did not allow elasticity, by setting the number of pouches on the system to a fixed value. This experiment was performed with an earlier build of the system, where a number of cloud management functions were not yet developed. This explains why the capacity of the system was higher. However, we could expect comparable results with the most current release although with a lower capacity.

In a node-based architecture, the provisioning of the nodes needs to be optimally engineered. However, since we only had a limited number of computing units available (eight Raspberry Pi) and did not have proper methods to manually engineer the provisioning, we evaluated a number of configurations. These configurations and their functionality distribution are shown for the eight available computing units (CU) in Table 5.1.

Table 5.1 Experimental configurations for node-based measurements: X (cloud management services), S (SIPh), D (DB), all other units as previously described

Config.	CU1	CU2	CU3	CU4	CU5	CU6	CU7	CU8
NO1	XS	D	C	C	A	T	M	M
NO2	XSD	C	C	A	T	M	M	M
NO3	XS	D	C	A	T	M	M	M
NO4	XS	D	C	C	T	MA	MA	MA
NO5	XSD	C	C	T	MA	MA	MA	MA

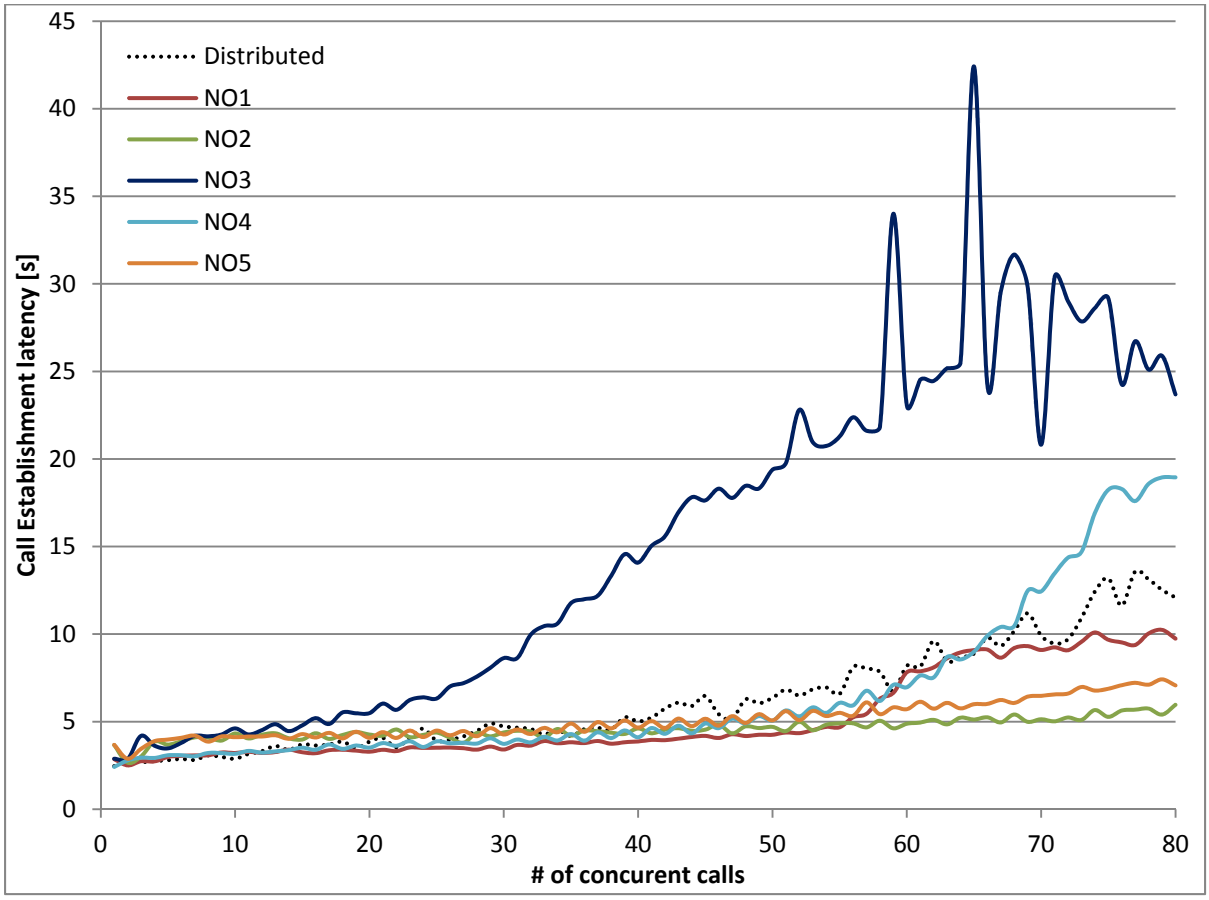

Figure 5.13 Call establishment latency vs. the number of calls for the different experimentation configurations

Figure 5.13 shows that the distributed cloud-based approach gives similar average control plane QoS characteristics, compared to node-based approach. However, poor provisioning engineering (NO3) in a node-based approach can have a dramatic effect on the call establishment latency. In this case, we suffer a starvation of call session control function (CSCF) capacity, which degrades the overall figure. This is avoided in the cloud-based approach, which ensures an average result.

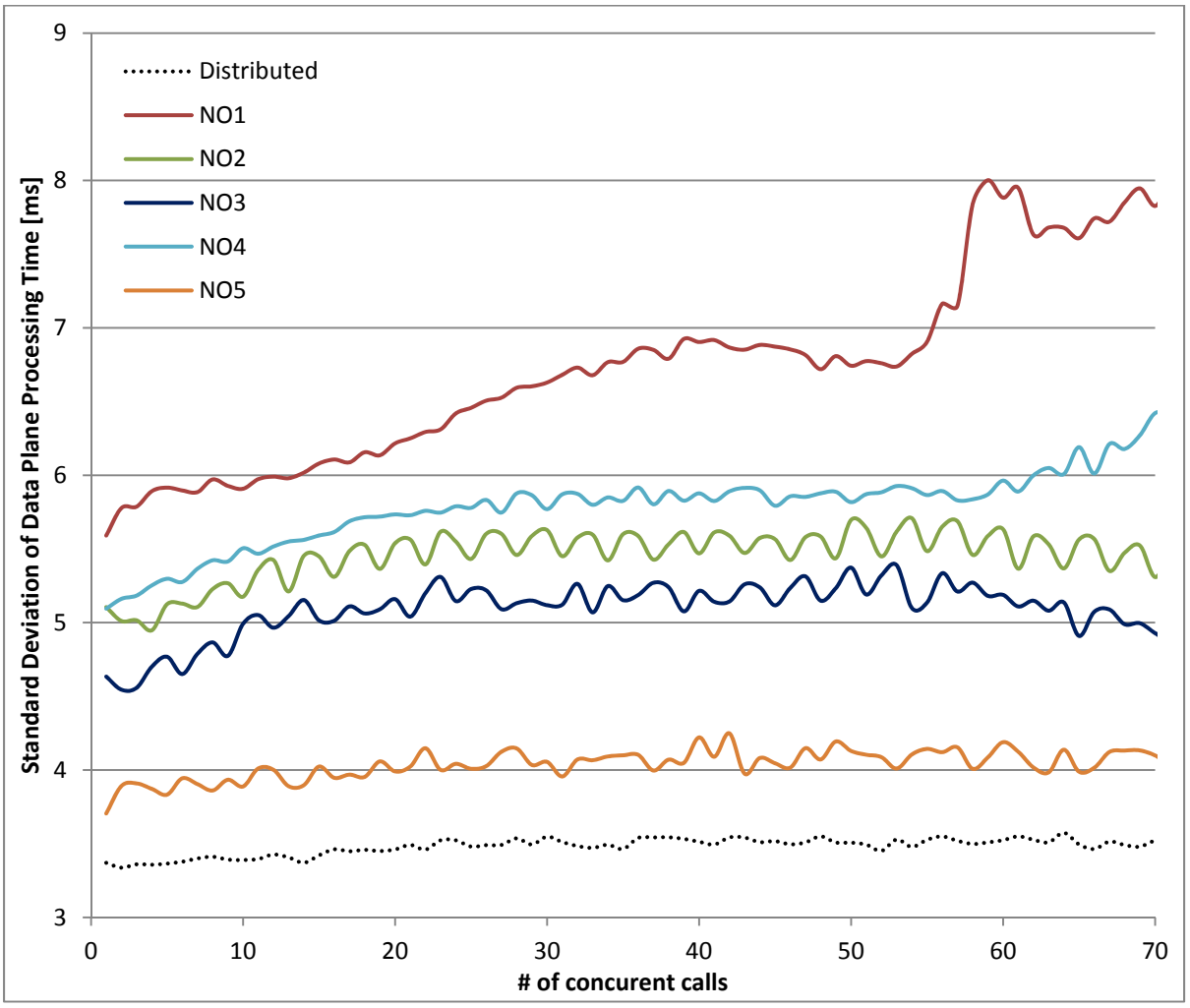

Figure 5.14 Data plane standard deviation to ideal processing time (20 ms) vs. the number of calls for the different experimentation configurations

It is worth noting that the cloud-based approach exhibits the best data plane QoS characteristics (Figure 5.14), compared to all node-based deployment configurations. With

the distributed cloud-based approach, the same amount of resources is given to both media and control plane and to each unit, because distribution is based on CPU usage. As such, we notice a better media plane performance and an average control plane performance.

In summary, the distributed cloud-based approach provides a platform resource allocation which cannot be easily achieved by the node-based architecture, even with precise allocation engineering (assuming the same hardware resources are provided to both approaches). Failure to properly weigh the node resource allocation engineering in the node-based architecture may yield major impacts on performance. Finally, node-based deployment reduces the reliability of the overall system, since if a node is deployed only on one CU and it fails, then the whole system fails and the service remains unavailable until that function is restored. This is the major reason why high MTBF are required on traditional node-based telecommunication networks. In the distributed cloud-based model, such a failure terminates the services hosted on a single CU, but the system remains available to provide new service instances spread on the other available CUs, although with a lower capacity. In order to promptly resume the original capacity, one must optimise the MTTR of such a cloud-based system.

5.6 Hibernation demonstration

As mentioned earlier, the implemented framework allows for individual unit hibernation. The capacity to hibernate units comes from the mechanism to distribute state information through the computing units. It provide resiliency and allows elastic scalability as units can be hibernated then revived on another computing unit. To demonstrate this capability, we performed an experiment on the Raspberry Pi cluster where we performed a number of two-way calls and screen captured the output of the activity display main screen.

Figure 5.15 shows the screen capture of the activity display main screen. Circled and marked with “1”, we see a two-way call where the control plane has fully hibernated and only the Media Processor (M) unit is active. Circled and marked with “2”, we see a two-way call in

the process of hibernating the control plane. Finally circled and marked with “3”, we see a fully established and non-hibernated two-way call with the complete control plane and the media plane in place.

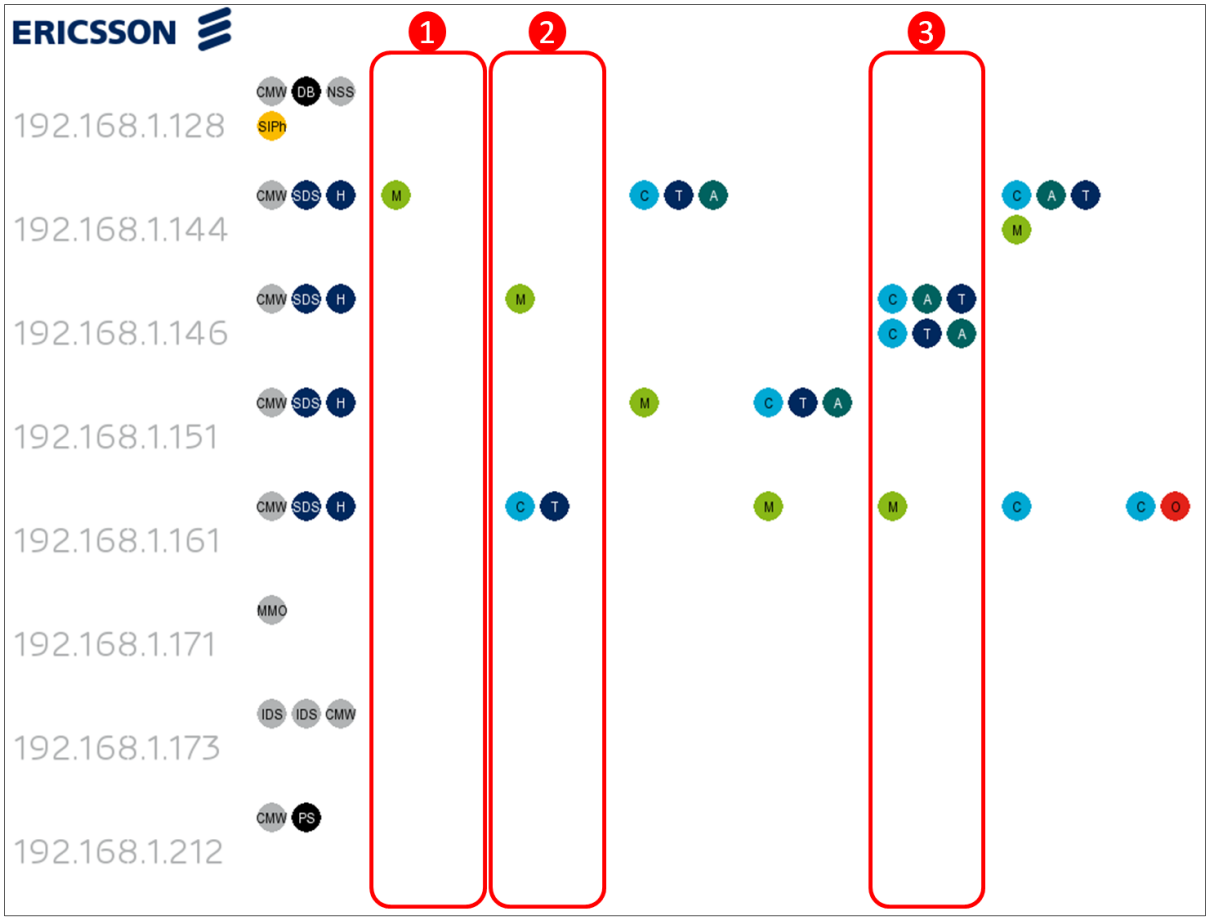

Figure 5.15 Hibernation demonstration through the activity display

5.7 Elasticity demonstration

There are two levels of elasticity provided by our proposed architecture. In order to demonstrate the elasticity, we performed an experiment on the Raspberry Pi cluster, where we generated traffic toward the cluster in a sinusoidal fashion. We start with no calls, increase toward a maximum, then reduce progressively back to no calls and are ready for a new cycle.

The first level of elasticity is the elasticity provided by the fact that units are instantiated, hibernated or released on-demand as the need arise. This can be observed from the previous experiment in Figure 5.15, where besides the cloud management units, the only running unit instances are for currently ongoing calls. As detailed in section 3.5, units not required to perform work are hibernated and when their associated service has completed, those units are released.

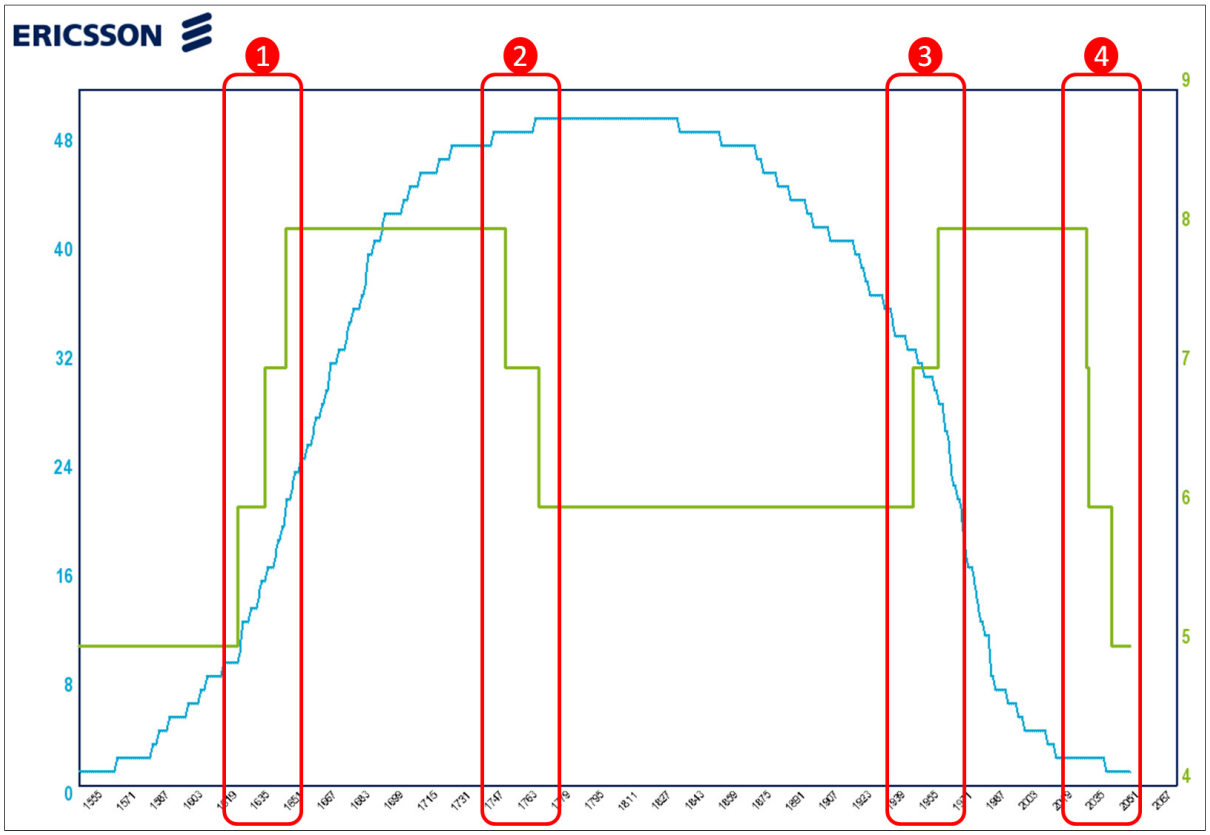

Figure 5.16 Elasticity demonstration through the activity display

The second level of elasticity is the elasticity provided by performing scale-out or scale-in operations. Figure 5.16 shows scale in and scale out operations of the computing units (in green) as the traffic on the system varies (blue line). Traffic varies from none to a maximum following a sinusoidal pattern. In the first stage (marked “1”), we can see successive scale out operations as the traffic picks up. In the second stage (marked “2”), the calls are fully established. Hence, only the media plane is busy and the control plane performs no

processing. The system is thus allowed to perform scale-in on the control plane pouches. The third stage (marked “3”) occurs as the calls are terminating. The control plane pouches are busy processing the call termination requests and require a scale out in order to maintain proper characteristics. Finally in the fourth stage (marked “4”), no calls are processed anymore and the system can scale-in both the control plane and the media plane.

5.8 Hybrid deployment

We discuss at last an experiment we performed of a hybrid deployment on OpenStack VMs and Apcera Cloud Platform containers. Through deployment of an appropriate descriptor file, we were able to deploy the control plane logic on Apcera Cloud Platform containers, and the media plane logic plus the protocol handling processing on OpenStack VMs. However, two issues limited our potential to take measurements.

The first issue is the limitations discussed earlier related to the usage of HTTP tunnelled communication (as it was made necessary by the usage of containers). Our Communication Middleware being single-threaded, the performance of HTTP is relatively poor since the communication becomes blocking while waiting for acknowledgements to a message.

Secondly, at the time of our experiment, the Apcera Cloud Platform system was still in early development. We used an early build, plagued with a number of problems, making the start of new containers an unreliable process. Nonetheless, we were able on a few occasions to properly start a static setup and perform a few calls on that system. We were however never able to submit this deployment to automated traffic, and as such we do not have the same kind of statistical measurements we performed in the first two experiments. We however produced a video of a limited number of subscribers being registered in the system, followed by two two-way call establishments. Figure 5.17 shows the activity display while performing that experiment. We see the M units deployed on OpenStack (OpenStack VMs based pouches are identifying themselves to the activity display with their IP addresses). We see the control logic, C, A and T units deployed on Apcera Cloud System containers (Apcera

Cloud System container based pouches are identifying themselves to the activity display with their Uniform Resource Locator (URL) name). This shows communications through hybrid deployment, as the M units, in an OpenStack based pouch, are created by the C units, in an Apcera Cloud System container based pouch.

Hybrid deployment is indeed possible, but our choice of an early development container technology and the lack of thread support in our HTTP tunnel implementation prevented us from having the same kind of statistical measurements we performed in the first two experiments.

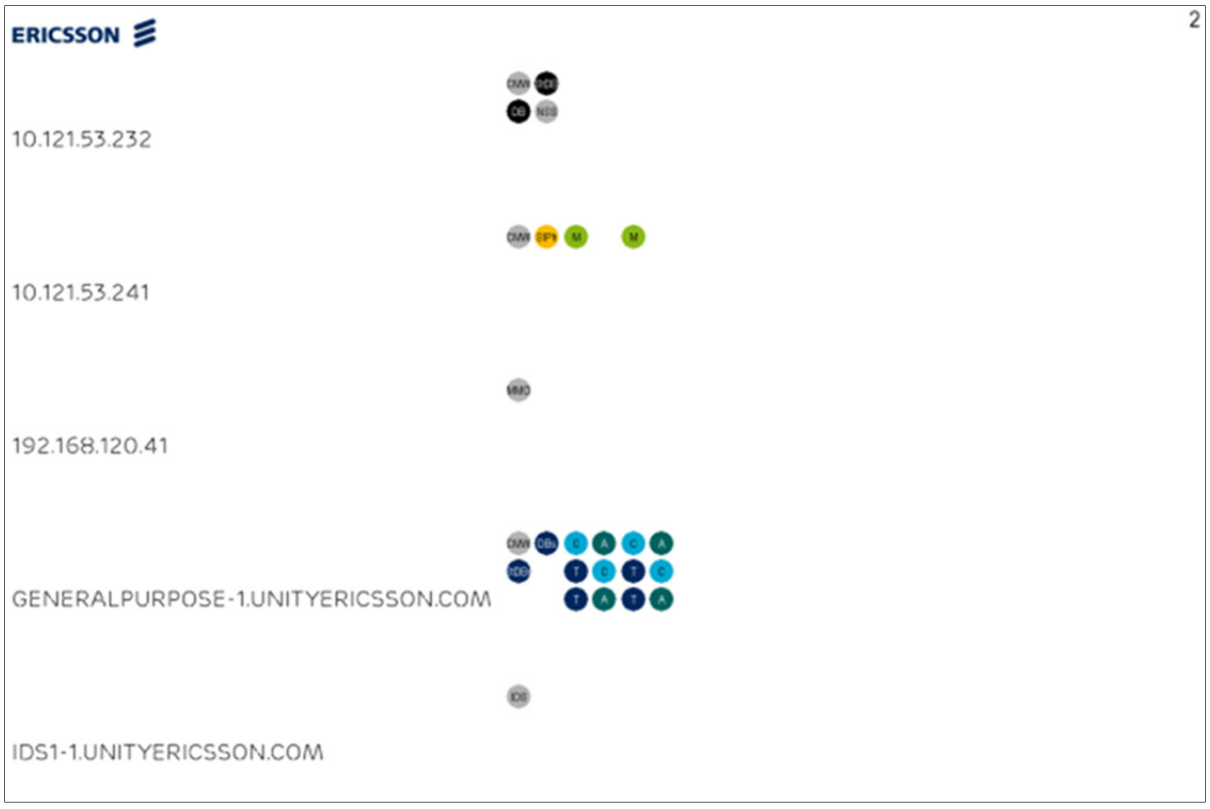

Figure 5.17 Hybrid deployment demonstration through the activity display

5.9 Discussion

As discussed in the first chapter, our main objective was to develop a cloud-native software architecture for telecommunication systems and code an implementation of that architecture

in order to evaluate its merits. Through this chapter and the previous ones, we demonstrated the implementation of the architecture through a telecommunication application we developed. The deployment of that application was performed on three platforms: a Raspberry Pi cluster, an OpenStack deployment and a hybrid, although limited, deployment on a mix of OpenStack VMs and Apcera Cloud Platform containers. As part of the main objective, we wanted to determine if we can re-architect IMS in order to provide its functionality in an on-demand, per subscriber and per service basis through the usage of the actor model (which can be seen as a specialization of the microservices pattern). We demonstrated in earlier chapters that such a re-architecting of the IMS is indeed possible and we demonstrated in this chapter that we can indeed provide the functionality in an on-demand, per subscriber and per service basis through our experimental results.

We also had the specific objective O1 for which we propose a mechanism to efficiently allocate computing resources through multiple cloud platforms in order to prevent overloading of computing resources and the resulting adverse impact on QoS. This mechanism had to implement an elastic and automatic scalability scheme. We demonstrated in this chapter how a deployment of the architecture performs on two levels of elasticity.

We also demonstrated through the hibernation and elasticity experiments how we answered specific objective O2, and proposed a mechanism to allow for application state information to be distributed on compute instances, providing resiliency while minimizing impact on latency.

We have shown how specific objective O3, and proposed an architecture providing portability between multiple cloud environments. This architecture enables a solution oriented, heterogeneous cloud deployment where the deployment could be on bare metal pools of servers, IaaS, Platform as a Service (PaaS). The architecture also show promises for a deployment on a mix of platforms. Through this chapter we have demonstrated deployment on two distinct platforms: a small cloud made of Raspberry Pi's computing platforms and

OpenStack VMs. We also have hope for a hybrid deployment on OpenStack VMs and Apcera Cloud Platform containers in the near future.

Finally we completed our demonstration of specific objective O4 where the implemented architecture characteristics were measured to validate its performance and we compared it to a traditional telecommunication node-based deployment.

Characteristic measurements show comparable results, albeit different in scale, on different platforms. A larger number of calls can be handled on the OpenStack deployment but the trend of QoS characteristics stays the same on all platforms we experimented with. This suggests that we achieved our goal of defining a cloud-based software architecture that can be easily deployed on heterogeneous hardware clusters using a single application code base.

Lastly, through the deployment of our architecture we wanted to measure its characteristics and to some extent compare it to a traditional telecommunication node-based deployment. We achieved this objective in this chapter by measuring the characteristics on a number of deployments of the architecture and comparing it to a traditional node-based deployment.

We observed that accurately allocating resources is required for each node in a node-based system, and this must be done statically due to the static configuration of the node-based system. For example, dramatic performance degradation may be experienced where the lack of resources allocated to some functionality might degrade the characteristics in a very noticeable fashion.

Our deployment of a microservices-based IMS telephony solution on different cloud platforms (RPi cluster, OpenStack platform and to a certain extent Apcera Cloud Platform containers) shows that this architecture enables one-time development of the business logic across multiple deployments and on various platforms through modification of deployment configuration only. Changing the descriptor file to deploy to a different supported platform is a trivial activity of a few minutes. Adding support for new cloud platforms is an effort of

about three man-weeks per platform as measured lately while implementing support for the Amazon Web Services (AWS) platform. The developed architecture demonstrates the possibility of defining an architecture supporting cloud features, especially automatic scaling-out of the business logic for the telecommunication sector.

There is however a cost to this architecture, the application needs to be re-written with that architecture in mind. We cannot simply re-use existing applications on this new framework. For some applications that cost will be prohibitive. In that case, virtualization might be the best that can be achieved on the road toward the cloud.

CONCLUSION

In this thesis we developed a cloud-native software architecture for telecommunication systems and we implemented that architecture in order to evaluate its merits. We re-architect the IMS as a simplified IMS core application which provides functionality in an on-demand, per subscriber and per service basis through the usage of the actor model. To our knowledge no other approach proposes the same thing at this point in time.

We proposed a mechanism to efficiently trigger the allocation of computing resources through the available cloud to accommodate overloaded nodes, using the unit and pouch concept, thus, successfully implementing an elastic and automatic scalability scheme.

We also proposed a mechanism to allow for application state information to be distributed on the computing instances, providing resiliency while minimizing the impact on latency through the usage of two stages of load balancing, local in-memory cache and a database. To that effect we proposed the usage of a local in-memory cache as an effective approach to reduce the number of queries to the HSS as well as reducing the latency associated with such information retrieval. Moreover, we proposed the usage of local in-memory cache for microservices state information. Replication of that cache on a distributed and replicated database enabled resiliency and supported elasticity by periodically storing the state information. Coincidentally this approach enables us to hibernate microservices if deemed appropriate.

Still as a mechanism to allow for application state information to be distributed on the computing instances, we discussed the usage of Rendezvous hashing load balancing in order to obtain a uniform load distribution amongst the computing resources while still maximizing the local in-memory cache hits, thus limiting the communication overhead.

Furthermore we proposed an architecture providing portability between multiple cloud environments enabling a solution oriented, heterogeneous cloud deployment. The

deployment could be on bare metal pools of servers, IaaS, Platform as a Service (PaaS) or a mix of these.

Finally we implemented the architecture and a sample telecommunication application and measured the characteristics of our architecture and compared it to a traditional telecommunication node based deployment. Through those measurements we observed that a cloud-based, distributed approach has benefits in terms of automated scalability, minimizing human intervention need and potential errors in dimensioning and in terms of elasticity over the traditional node-based architecture. Furthermore we saw that the cloud-based distributed approach presents a linear pattern in the usage of resources and thus can be predictable.

To sum up, a distributed cloud-based approach can provide automatic platform resource allocation which cannot be easily achieved by a node-based architecture. Failure to properly engineer node resource allocation in a node-based architecture can lead to major impacts on performance. Node-based deployment also reduces the reliability of the overall system since if a node is deployed on only one CU and it fails, then the whole system fails and the service remains unavailable until that function is restored. In the distributed cloud-based model, such a failure terminates the services hosted on a single CU but the system remains available to provide new service instances spread across other available CUs.

The heterogeneity aspect of the proposed architecture enables us to deploy an application that is designed once across different cloud platforms, easing the job of telecommunication equipment and software vendors to deploy network functions on various operator-owned clouds. The heterogeneity aspect also allows tailoring of the deployment to take advantage of the benefits of specific platforms for a given application. For example, an accelerator-based cloud might be beneficial for media resource processing functions while a general-purpose cloud might be more appropriate for control information.

Contributions of our thesis

The main contributions made so far in this thesis can be summarized as follow:

1. We proposed an adaptation of the actor model for the telecommunication domain as part of an answer to scalability and elasticity;
2. We introduced the pouch concept in order to abstract the platform the application software is running upon, hence addressing the need of heterogeneous deployment. Also, the pouch concept through scale-out operations completes answering the scalability and elasticity issue in the telecommunication cloud. As a side note, it is noteworthy to mention that the pouch concept is inspired from the game of Scrabble where a pouch contains the letters to play. The letters do not care in which pouch or what is the size or the color of the pouch. The letters have a meaning in the way they are linked together to form words. In the same way, units or actors have a meaning in the way they are linked together and do not have to care on what platform they are running upon since the pouch abstract it for them;
3. Based on the pouch concept, we proposed a Heterogeneous Architecture which can be deployed on hybrid clouds and provide QoS to telecommunication applications built upon it;
4. We proposed a mechanism using round robin load balancer, rendezvous aware load balancers for data sharding, and a mix of in memory databases and distributed database to provide application state storage while minimizing latency in a telecommunication cloud;
5. We built a proof of concept showing how the IMS could be re-architected along the principles outlined in this thesis. We deployed and measured the proof of concept of that implementation on a number of platforms in order to demonstrate it;
6. We have published two papers (Potvin et al., 2015a) and (Potvin et al., 2015b) which briefly describe our research. These papers were presented at the 2015 EAI International Conference on Smart Sustainable City Technologies in Toronto, and are available at this time in the soft proceeding of the conference on the S2CT.org website.

7. We also published a paper at the 16th International SDL Forum on Model-Driven Dependability Engineering (Potvin et al., 2013). It describes some work done on Domain Specific Languages of which some aspects were integrated in our current research as part of the message passing interface implementation. The details are beyond the scope of this document.

Future research direction

Lots of stones were left unturned and new ones were discovered along the way. Below is a list of some of the research questions that could be explored in the future:

- The actor model proves to be a good fit for the cloud programming paradigm as we have seen in this research. Could the multi-agent architecture be a good fit as well? A well implemented multi-agent architecture might improve the modularity of the system;
- We implemented a rudimentary QoS regulator where new pouches are instantiated when some characteristics measures are reached. Finer grained QoS regulation by the application is desirable and would need some research;
- Could Machine Learning help insuring QoS while still maintaining the highest utilization of the computing resources? It could enable taking into consideration the power usage, the source of the electricity (renewable or not) or other environmental and economic factors.

LIST OF REFERENCES

3GPP. 2015. *3rd Generation Patrnership Project; Technical SPecification Group Services and System Aspects; IP Multimedia Subsystem (IMS); Stage 2 (3GPP TS 23.228 V13.2.0 (2015-03) (Release 13))*.

4G Americas. 2014. *Bringing Network Function Virtualization to LTE*. < http://www.4gamericas.org/files/1014/1653/1309/4G_Americas_-_NFV_to_LTE_-_November_2014_-_FINAL.pdf >.

Abbott, Martin L, et Michael T Fisher. 2009. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Pearson Education.

Agrawal, Prathima, Jui-Hung Yeh, Jyh-Cheng Chen et Tao Zhang. 2008. « IP multimedia subsystems in 3GPP and 3GPP2: overview and scalability issues ». *Communications Magazine, IEEE,* vol. 46, n° 1, p. 138-145.

Ahlforn, Goran, et Erik Ornulf. 2001. « Ericsson's family of carrier-class technologies ». *ERICSSON REV(ENGL ED),* vol. 78, n° 4, p. 190-195.

Anderson, Ed; Smith, David Mitchell. 2015. « Hype cycle for cloud computing, 2015 ». *Gartner Inc., Stamford*.

Apcera. 2016. « Apcera ». < https://www.apcera.com/ >. Consulté le 2016-02-02.

Armstrong, Joe. 1997. « The development of Erlang ». In *ACM SIGPLAN Notices*. Vol. 32, p. 196-203. ACM.

Arpaci-Dusseau, Remzi H, et Andrea C Arpaci-Dusseau. 2012. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books.

AT&T. 2012. *Network Functions Virtualization; An Introduction, Benefits, Enablers, Challenges & Call for Action*. SDN and OpenFlow World Congress, Darmstadt-Germany. < https://portal.etsi.org/NFV/NFV_White_Paper.pdf >.

Bellavista, Paolo, Antonio Corradi et Luca Foschini. 2013. « Enhancing Intradomain Scalability of IMS-Based Services ». *Parallel and Distributed Systems, IEEE Transactions on,* vol. 24, n° 12, p. 2386-2395.

Buyya, Rajkumar, Christian Vecchiola et S Thamarai Selvi. 2013. *Mastering cloud computing: foundations and applications programming*. Newnes.

Carella, Giuseppe, Marius Corici, Paolo Crosta, Paolo Comi, Thomas M Bohnert, Andreea Ancuta Corici, Dragos Vingarzan et Thomas Magedanz. 2014. « Cloudified IP Multimedia Subsystem (IMS) for Network Function Virtualization (NFV)-based architectures ». In *Computers and Communication (ISCC), 2014 IEEE Symposium on*. p. 1-6. IEEE.

Cheshire, Stuart. 2005. « TCP Performance problems caused by interaction between Nagle's Algorithm and Delayed ACK ». *Self-published online: http://www.stuartcheshire.org/papers/NagleDelayedAck*.

Cormen, Thomas H. 2009. *Introduction to algorithms*. MIT press.

Crago, Steve, Kyle Dunn, Patrick Eads, Lorin Hochstein, Dong-In Kang, Mikyung Kang, Devendra Modium, Karandeep Singh, Jinwoo Suh et John Paul Walters. 2011. « Heterogeneous cloud computing ». In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*. p. 378-385. IEEE.

Docker, Inc. 2016. « Docker ». < https://www.docker.com/ >. Consulté le 2016-05-12.

Ericsson. 2014. « Ericsson Networks Software 15A ». < http://www.ericsson.com/ourportfolio/telecom-operators/networks-software-15a >.

ETSI. 2014. *Network Functions Virtualisation (NFV); Virtual Network Functions Architecture*.

Fried, Jeff, et Duane Sword. 2006. « Making IMS work: current realities, challenges and successes ». *Business Communications Review,* vol. 36, n° 5, p. 43.

Glitho, Roch. 2014. « Cloudifying the 3GPP IP Multimedia Subsystem: Why and How? ». In *New Technologies, Mobility and Security (NTMS), 2014 6th International Conference on*. p. 1-5. IEEE.

Hammer, Manfred, et Wouter Franx. 2006. « Redundancy and Scalability in IMS ». In *Telecommunications Network Strategy and Planning Symposium, 2006. NETWORKS 2006. 12th International*. p. 1-6. IEEE.

Karger, David, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine et Daniel Lewin. 1997. « Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web ». In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. p. 654-663. ACM.

Kheyrollahi, Ali. 2014. « Reactive Cloud Actors: An Evolvable Web of Events ».

Lee, Gunho, Byung-Gon Chun et Randy H Katz. 2011. « Heterogeneity-aware resource allocation and scheduling in the cloud ». *Proceedings of HotCloud*, p. 1-5.

Lu, Feng, Hao Pan, Xiao Lei, Xiaofei Liao et Hai Jin. 2013. « A virtualization-based cloud infrastructure for ims core network ». In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*. Vol. 1, p. 25-32. IEEE.

Mell, P., et T. Grance. 2010. « The NIST Definition of Cloud Computing ». *Communications of the Acm,* vol. 53, n° 6, p. 50-50.

Newman, Sam. 2015. *Building Microservices*. " O'Reilly Media, Inc.".

OpenStack, Project. 2016. « OpenStack ». < https://www.openstack.org/ >. Consulté le 2016-02-02.

Poikselkä, Miikka, et Georg Mayer. 2013. *The IMS: IP multimedia concepts and services*. John Wiley & Sons.

Potvin, Pascal, Mario Bonja, Gordon Bailey et Pierre Busnel. 2013. « An IMS DSL Developed at Ericsson ». In *SDL 2013: Model-Driven Dependability Engineering*. p. 144-162. Springer.

Potvin, Pascal, Hanen Garcia Gamardo, Kim-Khoa Nguyen et Mohamed Cheriet. 2015a. « Hyper Heterogeneous Cloud-based IMS Software Architecture: A Proof-of-Concept and Empirical Analysis ». In *EIA International Conference on Smart Sustainable City Technologies*. (Toronto, Canada). < http://proceedings.dev.icstweb.eu/2015/USB-s2ct2015.zip >.

Potvin, Pascal, Mahdy Nabaee, Fabrice Labeau, Kim-Khoa Nguyen et Mohamed Cheriet. 2015b. « Micro Service Cloud Computing Pattern for Next Generation Networks ». In *EIA International Conference on Smart Sustainable City Technologies*. (Toronto, Canada). < http://proceedings.dev.icstweb.eu/2015/USB-s2ct2015.zip >.

Raspberry Pi, Foundation. 2016. « Raspberry Pi ». < https://www.raspberrypi.org/ >. Consulté le 2016-01-27.

Rosenberg, Jonathan, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley et Eve Schooler. 2002. *SIP: session initiation protocol*.

Schulzrinne, Henning, Stephen Casner, Ron Frederick et Van Jacobson. 2003. *RTP: A transport protocol for real-time applications*.

Shawish, A., et M. Salama. 2014. « Cloud computing: Paradigms and technologies ». In *Studies in Computational Intelligence*, sous la dir. de Xhafa, F., et N. Bessis. Vol. 495, p. 39-67. In *Scopus*.

SIPp, Project. 2014. « SIPp ». < http://sipp.sourceforge.net/ >. Consulté le 2016-01-27.

Steiger, Carl Hewitt; Peter Bishop; Richard. 1973. « Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence ». In *IJCAI 73*. (Stanford University, Stanford, California, USA).

Thaler, David G, et Chinya V Ravishankar. 1998. « Using name-based mappings to increase hit rates ». *IEEE/ACM Transactions on Networking (TON),* vol. 6, n° 1, p. 1-14.

Turner, Andrew J. 2013. « Input Shaping to Achieve Service Level Objectives in Cloud Computing Environments ». Ph.D. Ann Arbor, Carnegie Mellon University, 187 p. In ProQuest Dissertations & Theses Full Text.

Vaquero, Luis M., Luis Rodero-Merino, Juan Caceres et Maik Lindner. 2008. « A break in the clouds: towards a cloud definition ». *SIGCOMM Comput. Commun. Rev.,* vol. 39, n° 1, p. 50-55.

Verma, Abhishek, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune et John Wilkes. 2015. « Large-scale cluster management at Google with Borg ». In *Proceedings of the Tenth European Conference on Computer Systems*. p. 18. ACM.

Vermeersch, Ruben. 2009. « Concurrency in Erlang & Scala: The Actor Model ». < https://rocketeer.be/articles/concurrency-in-erlang-scala/ >. Consulté le 2016-02-27.

Wellington, Dominic. 2012. « Homogeneous vs. heterogeneous clouds: pros, cons, and unsolicited opinions ». < http://www.bmc.com/blogs/what-price-homogeneity/ >. Consulté le 2016-01-25.

Wilder, Bill (182). 2012. *Cloud Architecture Patterns : Develop cloud-native applications*. O'Reilly Media.

xtUML, Project. 2016. « eXecutable Translatable UML with BridgePoint ». < https://xtuml.org/ >. Consulté le 2016-02-04.

Xu, Baomin, Ning Wang et Chunyan Li. 2011. « A cloud computing infrastructure on heterogeneous computing resources ». *Journal of computers,* vol. 6, n° 8, p. 1789-1796.

Yang, Tianpu, Xiangming Wen, Yong Sun, Zhenmin Zhao et Yuedui Wang. 2011. « A new architecture of HSS based on cloud computing ». In *Communication Technology (ICCT), 2011 IEEE 13th International Conference on*. p. 526-530. IEEE.