Pós-Graduação em Ciência da Computação

Thiago Felipe da Silva Pinheiro

Performance Prediction for Supporting Mobile Applications' Offloading

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

Thiago Felipe da Silva Pinheiro

Performance Prediction for Supporting Mobile Applications' Offloading

A Master Dissertation presented to the Informatics Center of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master in Computer Science.

Concentration Area: Performance and Dependability Evaluation
Advisor: Paulo Romero Martins Maciel

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

P654p Pinheiro, Thiago Felipe da Silva

Performance prediction for supporting mobile applications offloading / Thiago Felipe da Silva Pinheiro. – 2019.

194 f.: il., fig., tab.

Orientador: Paulo Romero Martins Maciel.

Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.

Inclui referências e apêndices.

1. Ciência da computação. 2. Avaliação de desempenho. I. Maciel, Paulo Romero Martins (orientador). II. Título.

004 CDD (23. ed.) UFPE- MEI 2019-044

THIAGO FELIPE DA SILVA PINHEIRO

Performance Prediction for Supporting Mobile Applications' Offloading

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 18/02/2019.

BANCA EXAMINADORA

I would like to dedicate this dissertation to God and my mother who helped me to overcome great obstacles during these last years.

Acknowledgements

I am immensely grateful to Professor Paulo Maciel for accepting me in his research group. He gave me all the support I needed to achieve my goals. I remember well when I asked if he would accept me as his student. He gave me a vote of confidence without even knowing me! I am immensely grateful for all the support and friendship, Paulo. All your students had been very lucky to have an advisor like you. I would like to mention my co-authors — Airton and Iure — for our mutual support and our continuous collaboration. A special thank to Airton for the valuable tips and for always being available. I want to thank my colleagues — Príscila, Danilo, Carlos Melo, Érico, Jamilson, and Ronierison — for helping me at some point during the previous two years. I also would like to thank FACEPE for the financial support, without which I would not have been able to finish this work.

Foremost, I would like to thank my mom for supporting me throughout this period. She motivated me to stand firm when things got tough. Thank you for always being by my side, mom. I love you all very much! God bless you!

I am grateful to God for giving me health, strength, and patience to complete this journey. God had given me the strength to overcome the various obstacles that have appeared.

I am immeasurably grateful for everything that happened!

“You are your own worst enemy. You waste precious time dreaming of the future instead of engaging in the present. Since nothing seems urgent to you, you are only half involved in what you do. The only way to change is through action and outside pressure. Put yourself in situations where you have too much at stake to waste time or resources –if you cannot afford to lose, you won’t. Cut your ties to the past; enter unknown territory where you must depend on your wits and energy to see you through. Place yourself on "death ground," where your back is against the wall and you have to fight like hell to get out alive."

(The 33 Strategies of War - Robert Greene)

Resumo

A Computação na Nuvem Móvel (Mobile Cloud Computing - MCC) é a integração da computação móvel e a computação em nuvem que pode aumentar o desempenho de aplicativos móveis e reduzir o seu consumo de energia por meio do *offloading* de códigos e dados. Os desenvolvedores podem implantar sistemas MCC em uma nuvem pública. A nuvem pública pode oferecer economias de escala, mas há algumas considerações a serem levadas em conta. Provedores de nuvem cobram seus clientes pelo tráfego de dados e uso de *virtual machines* (VMs), e decisões erradas de *offloading* podem levar a prejuízos financeiros. Essa dissertação propõe uma abordagem para estimar o desempenho de aplicações, uso de VMs e tráfego de dados gerado pelo *offloading* de tarefas e os seus custos em uma nuvem pública. Este trabalho propõe duas estratégias de modelagem formal baseadas em Redes de Petri Estocásticas (Stochastic Petri Nets - SPNs) para representar aplicações MCC e um modelo de custo para estimar o volume de tráfego de dados e o uso de VMs. A primeira estratégia representa os aplicativos MCC executando em dispositivos de usuários. A segunda representa uma infraestrutura remota implantada em uma nuvem pública para suportar o *offloading* de usuários móveis. Combinando diferentes tipos de VMs, trabalhos simultâneos por VM e *thresholds* para dimensionamento do sistema, é possível oferecer diferentes tempos de resposta para cada cenário de *offloading*. Além disso, usando ambas as estratégias de modelagem, é possível representar o processo de comunicação entre o aplicativo em execução no dispositivo do usuário e uma infraestrutura remota. Tornando possível estimar o desempenho do aplicativo MCC. Nossa abordagem permite que os projetistas planejem e ajustem arquiteturas de MCC com base em quatro métricas de desempenho: *tempo médio de execução* (Mean Time to Execute - MTTE), *tempo médio de resposta* (Mean Response Time - MRT), *função de distribuição acumulada* (Cumulative Distribution Function - CDF) e *vazão*. O MTTE está relacionado ao desempenho no dispositivo móvel. Por outro lado, o MRT corresponde ao desempenho da infraestrutura remota implementada em uma nuvem pública. Nossa estratégia de modelagem permite representar o uso e o compartilhamento da largura de banda disponível para operações de *offloading*, bem como o efeito da sua variação nas métricas avaliadas. Ela possibilita uma avaliação mais precisa sobre o desempenho de aplicativos, levando em consideração requisitos específicos de rede, usuários e cenários para *offloading*. Quatro estudos de caso foram executados. Nossa abordagem provou ser viável e destaca os cenários mais adequados. Ela suporta os desenvolvedores em tempo de projeto, fornecendo informações estatísticas sobre o comportamento dos aplicativos e estimativas de custos. Além disso, nossa estratégia pode ser adaptada para oferecer suporte a aplicativos MCC em tempo real, fornecendo estimativas probabilísticas imediatas de desempenho (*on-the-fly*).

Palavras-chaves: Nuvem móvel. Nuvem pública. Avaliação de desempenho. Avaliação de tráfego de dados. Elasticidade. Modelagem estocástica.

Abstract

Mobile Cloud Computing (MCC) is the integration of mobile computing and cloud computing, and it can increase the performance of mobile apps and reducing their energy consumption through code and data offloading. Developers may build MCC systems on a public cloud. The public cloud may offer economies of scale, but there are some considerations to take into account. Cloud providers charge their customers by data traffic and use of virtual machines (VMs), and wrong offloading decisions may lead to financial losses. This dissertation proposes an approach for estimating applications' performance, use of VM instances and data traffic generated by tasks offloading and its related costs on a public cloud. This work proposes two Stochastic Petri Net (SPN)-based formal modeling strategies to represent MCC applications and a cost model to predict data traffic volume and use of VM instances. The first SPN-based modeling strategy represents MCC applications running on user devices. The second one represents a remote infrastructure deployed in a public cloud for supporting offloading making by mobile users. By combining different instance types, simultaneous jobs per VM instance and thresholds for scaling the system, it is possible to offer different response times for each offloading scenario. In addition, using both strategies it is possible to represent the communication process between the app running on the user's device and a remote infrastructure. Thus, making possible to estimate the performance of the MCC application. Our approach enables designers to plan and tune MCC architectures based on four performance metrics: *Mean Time to Execute* (MTTE), *Mean Response Time* (MRT), *Cumulative Distribution Function* (CDF) and *Throughput*. MTTE is related to the performance on the mobile device. On the other hand, MRT corresponds to the performance of the remote infrastructure deployed in a public cloud for supporting offloading. Our modeling strategy allows for the representation of the use and sharing of available bandwidth for offloading operations, as well as the effect of bandwidth variation on the metrics evaluated. It allows a more accurate evaluation by developers about the performance of their applications taking into account specific network requirements, users, and offloading scenarios. Four case studies were performed to evaluate our approach. Our approach has proven to be feasible and it highlights the most appropriate scenarios. Supporting developers at design time by providing statistical information about applications' behavior and costs estimations. In addition, our approach may be adapted to support MCC applications in real time providing on-the-fly probabilistic performance predictions.

Keywords: Mobile cloud. Public cloud. Performance evaluation. Data traffic evaluation. Elasticity. Stochastic modeling.

List of Figures

List of Tables

List of Acronyms

AMI	Amazon Machine Image
API	Application Program Interface
AR	Arrival Rate
AWS	Amazon Web Services
BW	Bandwidth
CDF	Cumulative Distribution Function
CI	Confidence Interval
CPN	Colored Petri Net
CPU	Central Processing Unit
CT	Communication Time
CTMC	Continuous Time Markov Chains
CV	Coefficient of Variation
DoE	Design of Experiments
DTMC	Discrete Time Markov Chain
EC2	Amazon Elastic Compute Cloud
FCFS	First Come, First Served
GPU	Graphic Processing Unit
GSPN	Generalized Stochastic Petri Nets
IaaS	Infrastructure-as-a-Service
ISS	Infinite Server Semantic
IT	Information Technology
KSS	K-Server Semantic
MCC	Mobile Cloud Computing
MRT	Mean Response Time
MSS	Mean System Size

MTTA	Mean Time to Absorption
MTTE	Mean Time to Execute
NMon	Nigel's Monitor
ODI	On-demand Instance
OpenCV	Open Source Computer Vision Library
PaaS	Platform-as-a-Service
PN	Petri Net
RI	Reserved Instance
RMI	Remote Method Invocation
RTT	Round-Trip Time
SaaS	Software-as-a-Service
SI	Service Instance
SIT	Scaling In Threshold
SLA	Service Level Agreement
SOT	Scaling Out Threshold
SPE	Software Performance Engineering
SPI	Spot Instance
SPN	Stochastic Petri Net
SSS	Single Server Semantic
TP	Throughput
TTB	Total Transferred Bytes
UML	Unified Modeling Language
VM	Virtual Machine

Contents

Chapter 1
Introduction

“ *Everything is possible to him who has faith.* ”

The Holy Bible, *Mark 9:23*

The first call from a handheld mobile phone occurred in 1973 (MOTOROLA, 1973). Since then, the technology available in the user’s hand has evolved from a simple phone to a smartphone that offers a myriad of functions and capabilities. Today, each smartphone model has its own hardware specification. Hardware specifications are determined by the combination of components such as processor, volatile memory and memory for storage, wireless technologies, screen width and resolution. The evolution process of all technologies involved in theses devices naturally gave rise to a large ecosystem of devices. This ecosystem consists of more than 600 brands (BUSINESS STANDARD, 2018).

Cisco forecasts that, by 2021, the mobile connection speed will overpass 20 Mbps and the total number of smartphones will be over 50 percent of global devices and connections (CISCO, 2019). The report forecasts that there will be 11.6 billion mobile devices connected to the internet. More specifically, there will be 8.3 billion handheld or personal mobile devices. According to the same report, between 2016 and 2021 mobile data traffic will be increased sevenfold and 86 % of global mobile traffic will come from smartphones and phablets. A large part of the increase in mobile data traffic will come from cloud computing applications.

This huge number of users, myriad of mobile device brands and increased data traffic stimulates researches aiming to meet ever more demanding performance requirements and the increasing demand. Today, many applications have real-time constraints and the processing power available on mobile devices does not satisfy the performance requirements. Complex applications — with complex methods — usually only reach users permanence satisfaction through expensive devices. These complex methods are the critical points which require the software engineer attention. This issue prompted researchers to find ways to increase the processing power available to mobile applications running on limited devices. Sending mobile processing to powerful servers deployed on remote infrastructures can improve the performance of mobile systems (CHUN et al., 2011), (KEMP et al., 2012), (SATYANARAYANAN et al., 2009), (KOSTA et al., 2012), (CUERVO et al., 2010), (GORDON et al., 2012), (SHI et al., 2014), (SHI et al., 2012), (SILVA et al., 2016).

Considering this, a new field of study known as Mobile Cloud Computing (MCC) has emerged. MCC is the integration of mobile computing and cloud computing, and it can increase the performance of mobile applications and reduces their energy consumption through the offloading technique (KUMAR; LU, 2010), (KOCJAN; SAEED, 2012). The offloading process sends tasks to be processed on remote servers in the cloud. The first step in performing task offloading is to split an application into different *parts*. Next, there is a decision of which ones are most convenient for remote processing. Tasks may be partially or completely offloaded, depending on the application requirements. The convergence of mobile devices and cloud computing has been studied by a large number of works and MCC still has open issues that motivate further researches (ARAUJO et al., 2014; MATOS et al., 2015; COSTA et al., 2015; ABOLFAZLI; GANI; CHEN, 2015; SILVA et al., 2015; SILVA; MACIEL; MATOS, 2015; SILVA et al., 2018; SILVA et al., 2017; ZHANG et al., 2018; AKHERFI; GERNDT; HARROUD, 2018; ARAUJO et al., 2016).

Today, many mobile apps that have performance constraints benefit from using cloud resources. Many companies have deployed most of their Information Technology (IT) resources in the cloud. The cloud may be composed of physical and virtual heterogeneous components that offer diverse computing power. This can help remove many of the complexities and limitations of computing processing such as physical space, energy, time, and cost. However, a physical cloud infrastructure may have limitations regarding the computing power available for supporting the expected workload, taking into account the increasing demand generated by mobile users. While powerful infrastructures become accessible for even small businesses, a large number of physical resources can be difficult to manage. In addition to issues regarding the processing power, these infrastructure may tackle communication issues. Cloud resources may be distributed geographically and issues such as network latency must be considered by MCC systems.

MCC service providers are interested in tuning their underlying resources to face variations in user demands. Dynamic provisioning also known as elasticity is one of the most important features of cloud computing (HAN et al., 2014). The elasticity characteristic inherent in cloud computing makes it possible to a system to dynamically provision and deprovision resources for it according to its demand. As demand increases, the system inserts processing power in the infrastructure in order to comply with performance requirements defined in Service Level Agreements (SLAs). Considering a mobile app that may have users distributed around the world, a private infrastructure may have limited processing capacity for supporting increasing demand. On the other hand, public clouds offer unlimited processing capacity.

Today, there are some Infrastructure-as-a-Service (IaaS) public clouds providers available on the market. As an example, we may cite Amazon Web Services (AWS) [1], Google Cloud Platform [2], and Microsoft Azure [3]. These cloud providers offer a large number of infrastructure resource as services. Public clouds offer resources that makes it possible to deploy systems for supporting a myriad of types of workloads. When using these IaaS resources, customers may have full control from the kernel to the entire software stack. AWS is the most mature cloud computing platform available today. AWS supports the most well-known mobile services available to the general public such as Airbnb [4], Netflix [5], Reddit [6], Pinterest [7], and Spotify [8] (VOXMEDIA, 2018; CNBC, 2017; AWS, 2018c). One of the major issues regarding mobile applications that will be made available to the general public is related to the user base that may grow rapidly. For example, Pokémon Go (stylized as Pokémon GO) is a location-based augmented reality game developed by Niantic (POKEMONGO, 2018). Niantic released the game in selected countries in July 2016 and the Pokémon Go had its downloads peak within first six days after its launching. The mobile game had 21 million daily active users only a few days after its launching (FORBES, 2016). Considering only the United States, more than 34 million people had downloaded the game within a period of one month after its release (BUSINESS OF APPS, 2017). Of course, public clouds can support applications like Pokémon Go, with millions of users in their user bases and increasing demand. Public clouds offer unlimited processing power and can be geographically distributed for supporting requests sent by users around the world.

However, offloading to public clouds does not come for free. IaaS cloud service providers charge their clients for resource usage. A wrong offloading decision may lead a company to financial losses. The higher the resource usage, the higher the amount to be paid to the IaaS provider. Method-call offloading is a partitioning strategy that enables to split a code into multiple parts. Deciding which method to partition is not an easy task. It is necessary to analyze many possible scenarios. There may be a large number of offloading scenarios depending on the number of parts the developers splits their MCC applications. Each offloading scenario may have a large number of remote configurations for supporting offloading making by mobile users. These configurations correspond to the number of virtual machines (VMs) —also known as instances —, available to process external requests, and scaling policies. Public clouds charge for resources consumption

[1] AWS: https://aws.amazon.com
[2] Google Cloud Platform: https://cloud.google.com
[3] Azure: https://azure.microsoft.com
[4] Airbnb: https://www.airbnb.com
[5] Netflix: https://www.netflix.com
[6] Reddit: https://www.reddit.com
[7] Pinterest: https://www.pinterest.com
[8] Spotify: https://www.spotify.com

and it is necessary to use them appropriately.

This dissertation proposes an approach for predicting performance, use of resources and its resulting costs of MCC applications deployed in public clouds. This work proposes two Stochastic Petri Net (SPN)-based modeling strategies to represent the workload of MCC applications being processed both locally and remotely. The first SPN-based modeling strategy represents MCC applications installed on mobile devices. It may represent the structure of the application's source code. Representing the source code enables the software engineer to access a more accurate result. In addition, this modeling strategy represents the use and sharing of the network bandwidth (BW) available for supporting offloading operations as well as the effect of BW variation on the evaluated metrics. In this way, making possible to represent the communication time to transfer data and code. The second modeling strategy represents elastic MCC systems deployed in public clouds for supporting offloading generated by mobile users. We have considered for modeling purposes VM instances and data traffic as resources of the public cloud. These are the two basic resources for supporting a remote MCC system. As the number of users grows, the greater may be the data traffic and the consumption of VM instances for supporting the demand. The public cloud provides an elastic capability capable of supporting varying user demand patterns. Thus, making it possible to reduce costs and comply with performance requirements defined in SLAs. On the other hand, it is a difficult task to identify a configuration to deploy the system to meet performance and cost requirements. Our SPN-based cloud modeling strategy makes it possible to represent remote MCC systems deployed in a heterogeneous infrastructure with variable processing and buffers capacities, variable demands, scaling policies, and a large number of VM instances running in the same infrastructure. By combining different instance types, simultaneous jobs per VM instance, stepsizes and scaling thresholds, it is possible to offer different response times for each offloading scenario. Thus, impacting the total time to execute the mobile app on user devices. However, each of these variables affects resource consumption as well as the cost that an MCC service provider pays to an IaaS provider. An application may use an unlimited amount of resources and this affects the prices that the service provider needs to pay at the end of a period. For that aim, we propose a mobile cloud cost model to predict data traffic volume, use of VM instances and its costs applying SPNs. The cost of using VM instances corresponds to the cost of using reserved instances (RIs) and on-demand ones (ODIs). Next, the proposed models are used to estimate four performance metrics of the application: *Mean Time to Execute* (MTTE), *Mean Response Time* (MRT), *Cumulative Distribution Function* (CDF) and *Throughput* (TP). MTTE corresponds to the execution time of the mobile app as a whole. MRT corresponds to the time a remote MCC infrastructure takes to process a request sent by a mobile device. CDF indicates the maximum probability that the processing will finish in a specific time interval. Throughput corresponds to the number of requests per unit of time made by each user as well as processed by a remote infrastructure.

On the remote side, we have evaluated the deployment of a mobile cloud face recognition system using the proposed approach. Many works have evaluated facial recognition algorithms in the context of MCC (CIDON et al., 2011; CUERVO et al., 2010; KEMP et al., 2012; KOVACHEV; YU; KLAMMA, 2012; KWON; TILEVICH, 2012; SOYATA et al., 2012; HUANG; WANG; NIYATO, 2012; WANG; DONYANAVARD; CHENG, 2012; ZHANG et al., 2012). We have validated our cloud modeling strategy considering this type of workload. However, it is important to highlight that the strategies proposed in this dissertation can support the planning of MCC systems deployed in a remote infrastructure for supporting a myriad of types of workloads. In some situations, there may be some refinement in our modeling strategies in order to adapt them to more accurately represent the architecture and system under consideration.

Monetary cost evaluation based on stochastic models may support companies in planning their applications on the public cloud. A very few papers have proposed strategies based on stochastic models in order to support cost evaluations of resource consumption on public clouds (FE et al., 2017; RIBAS et al., 2015b; RIBAS et al., 2015a; GUERFEL; SBAÏ; AYED, 2018; SILVA et al., 2015). These works only addressed the monetary cost of using VMs.

Our approach may assist software engineers to evaluate their applications at design time and deciding where to process the application's workload considering local and remote processing resources and networking requirements. Although many studies focus on the optimization of the offloading process, in the recent literature, few studies have addressed stochastic performance modeling in the context of MCC infrastructure planning. In addition, the tradeoff between performance, data traffic and use of VM instances — and their related costs — for offloading on public clouds have been ignored by these studies. Our work is the first one, to the best of our knowledge, that provides an integrated modeling approach that represents MCC workloads running locally and remotely and the communication process between both sides supporting performance and costs evaluations.

1.1 Problem Statement

Figure 1 illustrates a scenario where a mobile application offloads tasks to an elastic infrastructure on a public cloud. Offloading these tasks may improve the application's performance. However, the infrastructure's configuration as well as the workload being processed by it impacts the total performance and resource consumption. In addition, networking conditions may impact the time necessary to transfer data and code.

In this context, one question arises:

- How to represent MCC applications running on mobile devices and in an elastic MCC infrastructure deployed on a public cloud, and the communication process between both sides, in order to support performance and remote resource consumption evaluations?

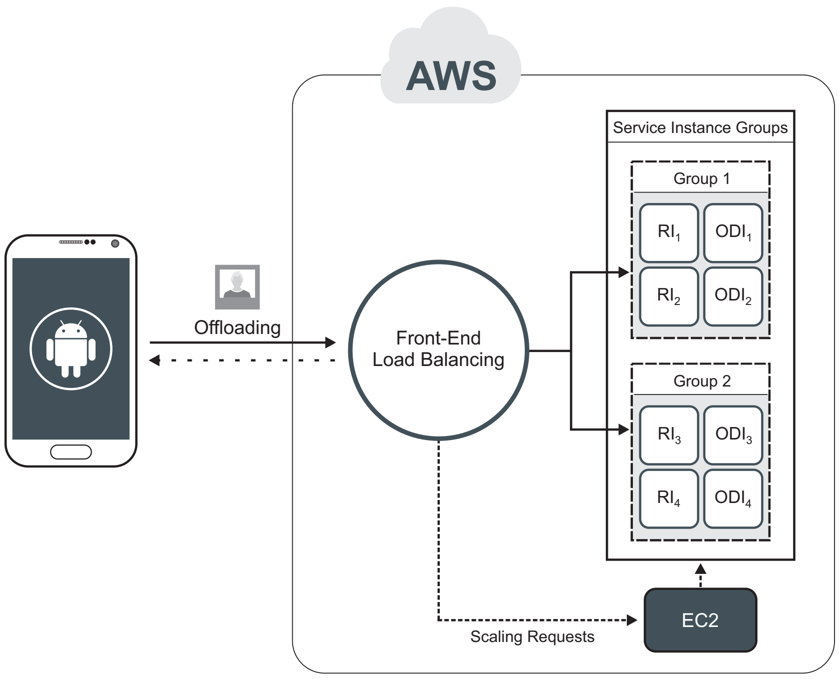


Figure 1 – Scenario Depicting a Mobile Application Offloading a Face Recognition Task to an Elastic MCC Infrastructure on AWS

1.2 Research Scope

There are three main perspectives that service providers need to take into consideration when designing MCC applications. Each perspective is related to an aspect of the MCC system that may impact the performance and the way the system performs offloading. MCC systems usually explore one or more of the following three perspectives.

- **What to Offload?** The first step when using the offloading technique is to define how the application's workload should be represented and splitted. The workload is defined in the application's source code. Thus, service providers need to decide at which level of granularity their applications must be represented. Granularity here defines how the source code is represented for supporting the partitioning process. For example, among other types of representations, source code may be partitioned per class, method, and components. The most advanced offloading frameworks adopt method-call as partitioning granularity (CUERVO et al., 2010; KOSTA et al., 2012; CHUN et al., 2011; KEMP et al., 2012). In this work, we have adopted the method-calls as granularity for supporting offloading scenarios generation.

- **When to Offload?** The mobile application should decide whether it is worthy to perform the offloading of its workload to a remote infrastructure. The perspective "*when*" corresponds to the process to decide the most appropriate moment to perform offloading. For this, it is necessary to consider data traffic and networking condition. Networking condition takes an important role in the decision-making process. For example, when executing under worst networking conditions, the time required to an application to send and receive data may degrade its performance. An MCC application must have the context-awareness capability, so that it could adapt itself on-the-fly to the changing of the environment conditions.

- **Where to Offload?** This perspective corresponds to the location at which the application's workload should be processed. It means that an application may be partitioned in different parts and each of them may be processed at different locations. These parts are defined considering the granularity choosed in the "*What to Offload?*" perspective. After the partitioning, there may be a decision related to where each part must be processed. That is, based on the application's requirements, the user's device may process the workload or it may send the workload or part of it to be processed in a more powerful remote infrastructure. On the remote side, there may be a large number of servers with different processing capabilities available to handle users' requests. The capacity of the remote infrastructure as well as the workload processed by it takes an important role in order to define the application's performance. The characteristics of such an infrastructure should be considered to construct an offloading solution.

Figure 2 depicts the scope of this research. As we can see, this master research has focused mainly on the "*where*" perspective due to its challenging and important features.

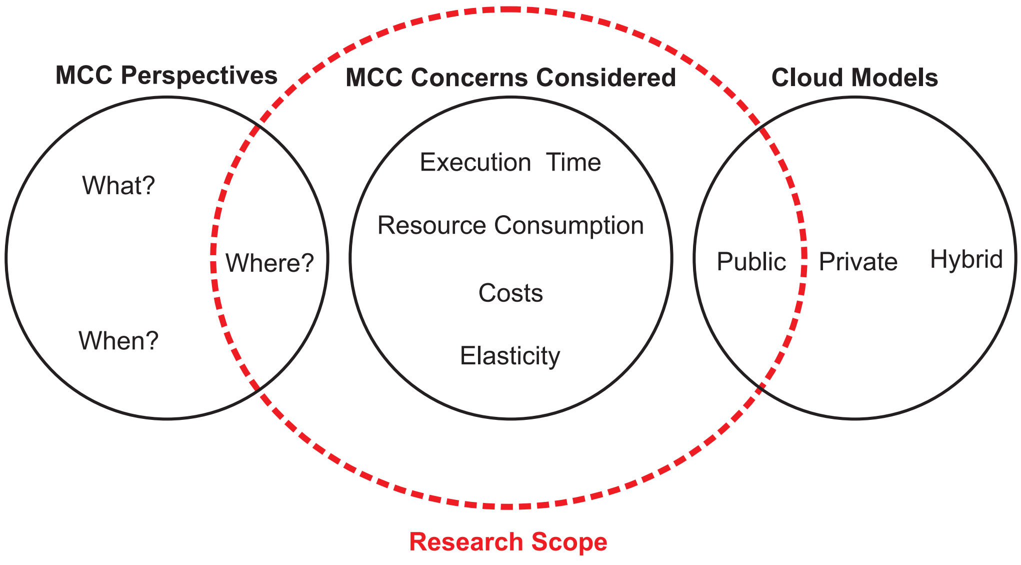


Figure 2 – Master Research Scope

1.3 Objectives

The main objective of this research is to proposes an approach for predicting performance, use of resources and its resulting costs of MCC applications deployed in public clouds.

Among the specific goals of the research, we can list:

- Develop an SPN-based modeling strategy to evaluate the performance of MCC applications deployed on mobile devices that consider the use of the actual bandwidth for supporting offloading operations.
- Develop an SPN-based modeling strategy to evaluate the performance of elastic MCC systems running on a public cloud that considers the use of virtual machines of different types on the same infrastructure.
- Develop cost models considering the use of virtual machines and data traffic for supporting MCC systems in public clouds.
- Evaluate the proposed strategies in case studies.

1.4 Publications

Following, we present a list with the published papers related to this research.

As main author:

- Thiago Pinheiro, Francisco Airton Silva, Iure Fe, Sokol Costa and Paulo Maciel. Performance Prediction for Supporting Mobile Applications' Offloading. The Journal of Supercomputing. 2018. ISSN: 0920-8542 (Print) 1573-0484 (Online).
- Thiago Pinheiro, Francisco Silva, Iure Fe, Sokol Kosta and Paulo Maciel. Performance and Data Traffic Analysis of Mobile Cloud Environments. In: 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC). October 7-10, 2018 – Myiazaki, Japan.

1.5 Organization of the Document

The remainder of the dissertation is organized as follows. Chapter 2 provides an overview of the main concepts about cloud computing, mobile cloud computing, performance evaluation of systems, and face recognition technique. Chapter 3 discusses and compares noteworthy works found in literature that have some topics in common to those addressed in this dissertation. Chapter 4 presents detailed information about the methodologies proposed to support the decision-making process related to the definition of offloading scenarios and remote configurations to deploy the MCC system in public clouds. Chapter 5 describes the MCC architeture considered in this work. Chapter 6 and 7 describe the core contribution of this dissertation. Chapter 6 describes our SPN-based modeling strategy that represents mobile apps installed on user devices. Chapter 7 describes our SPN-based modeling strategy that represents elastic MCC systems deployed in public clouds. Chapter 8 details case studies considering the proposed approaches; and finally, Chapter 9 traces conclusions and presents our ideas about future directions.

Chapter 2 Background

“ *Never argue. In society nothing must be discussed; give only results.* ”

Benjamin Disraeli, *1804-1881*

This chapter discusses the basic concepts needed to understand our work. More specifically, the concepts presented here should provide the necessary knowledge for a clear understanding of the chapters below, including aspects involving the proposed methodology and subsequent case studies. The remainder of the chapter is organized as follows. Section 2.1 highlights the main concepts about cloud computing; Section 2.2 shows the main concepts about Mobile Cloud Computing (MCC) and method-call offloading; Section 2.3 presents concepts related to performance evaluation, such as Design of Experiments (DoE), Continuous Time Markov Chains (CTMC), Petri Nets (PNs), Stochastic Petri Nets (SPNs), and phase-type approximation technique; and finally, Section 2.4 provides detailed information on how the face recognition approach considered in this dissertation works.

2.1 Cloud Computing

Cloud computing is a model that allows the use and delivery of Internet-based services, which typically involves the provision of dynamically scalable and often virtualized resources over the Internet. Acordingly to the National Institute of Standards and Technology (NIST), cloud computing is “a model for enabling ubiquitous, convenient on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction” (MELL; GRANCE, 2011). Cloud computing is a fairly widespread business model today and it offers cost savings, more mobility and operational flexibility. It allows users to access application services from anywhere using a variety of terminals. A computational infrastructure is considered a cloud when five essential characteristics are presented.

- **On-demand self-service.** Users can provision computing resources, such as virtual machines, storage, and network capacity as needed, automatically, and without requiring human interaction with human resources responsible for managing the service provider.
- **Broad network access.** Resources are available on network and may be accessed by heterogeneous computing platforms through standard communication interfaces.
- **Resource pooling.** Computing resources are grouped to serve multiple users using a multi-tenant model. Resources are provisioned and de-provisioned dynamically according to the demand received by applications deployed on the cloud. Generally, cloud users are not aware of the exact geographic location of the provisioned resources.
- **Rapid elasticity.** Rapid elasticity is the ability to deliver, increase, or reduce services and features automatically at any time without additional costs. Resources may be automatically allocated and removed from the system according to demand variation. In some service providers, the number of resources available for provisioning and use may seem unlimited and may be made available in any amount.
- **Measured service.** Cloud systems automatically control and optimize the available resources and provide tools for consumption measurement appropriate to the type of service and contracted resource.

Cloud computing stack comprises nine layers and three types of service models, as we can see in Figure 3. Figure 4 depicts the three types of cloud computing service models. As we can see, Software-as-a-Service (SaaS) offers the highest level of abstraction to the cloud user, and Infrastructure-as-a-Service (IaaS) provides greater control over all infrastructure and resources. The infrastructure layer is at the bottom, the platform layer is in the middle, and at last the software layer is at the top.

- **Infrastructure-as-a-Service (IaaS):** IaaS is the first layer of the cloud computing model. IaaS is a model that provides computational and networking infrastructure through virtualization technology as a service for users over the Internet. More specifically, IaaS service providers provide resources such as hardware, network, connectivity, virtual machines, operating systems, among others. Instead of buying servers, software, special network equipment, users may receive these features in the form of outsourcing. IaaS normally employs a pay-as-you-go model with vendors typically charging by a specific period, offering an economic way to hire resources.

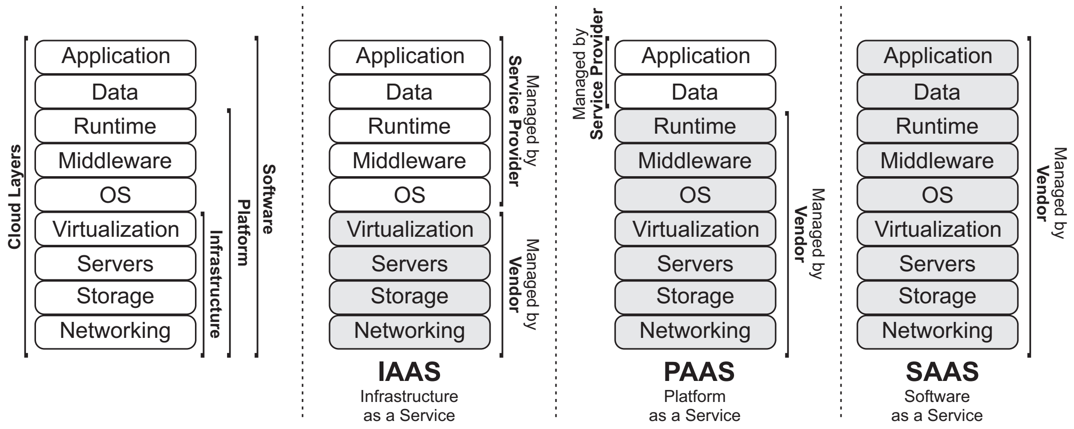


Figure 3 – Cloud Layers.

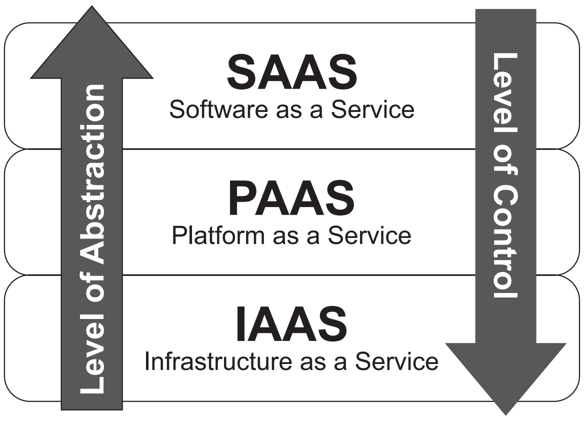


Figure 4 – Cloud Service Models.

Computational resources are virtualized and made available in a way that simplifies the provisioning, and configuration of an infrastructure. Basically, the provision of infrastructure is performed in an environment that provides a variety of computing and networking resources where users can hire and manage them through a web interface or through an Application Program Interface (API). Generally, features provided by the IaaS model are not preconfigured, and the responsibility for managing them is unique of the cloud users. Depending on application demands, the infrastructure provisioned for it may be dynamically extended or reduced. Some of the largest IaaS service providers include Amazon, Microsoft, VMWare, and Red Hat.

- **Platform-as-a-Service (PaaS):** PaaS is the second layer of the cloud computing model, sometimes called middleware. PaaS provides a platform for cloud applications that make able to businesses or individuals to develop software for themselves or to other users, and they only pay for the resources usage. PaaS make easy the development and deployment of applications, while allows developers to save money on hardware and software. PaaS providers offer a variety of solutions for developing and distributing applications such as virtual servers, operating systems, web application management, application design, hosting, storage, security, and collaborative development tools. Generally, a PaaS platform supports the whole process of software development that includes design, deployment, testing, installation, and hosting of applications. Some of the most well-known PaaS providers are Google App Engine, Microsoft Azure, Force.com, Heroku and Engine Yard.

- **Software-as-a-Service (SaaS):** SaaS is the top layer of the cloud model. SaaS is a model that enables the delivery of ready-to-use applications to end users as an on-demand service, such as enterprise resource management over the Internet. Users access such applications over the network, and they are usually made available as a web application. Generally, SaaS is the only cloud layer that end users have direct access to. Instead of buying software, users rent software from service providers and pay only for the contracted period and the chosen features. There are several well-known services that may be classified as SaaS such as Netflix, Dropbox and iCloud, for example.

There are four deployment models for cloud computing (MELL; GRANCE, 2011; ERL, 2013).

- **Private Cloud:** Private cloud is an infrastructure built and used by an organization in a context that only authorized people have access to. Usually private clouds are built to provide resources for corporate users. The data center architecture and servers maintenance is planned and executed by the infrastructure owner or an outsourced team. The company owns the infrastructure and controls how applications are deployed and updated on it, as well as the network security rules. Private cloud is a model that provides higher control over service quality, private data, and infrastructure security.

- **Community Cloud:** Community Cloud is a cloud model that provides features and services to a limited number of users and organizations. The cloud may be managed and secured commonly by everyone involved, a group of them, or an outsourced team. This is a hybrid model derived from the private cloud model, where users with similar requirements and goals share the same infrastructure. Sometimes this is the ideal cloud computing model for companies or groups of users who want to work together on researches or shared projects.

- **Public Cloud:** Cloud computing provided in public infrastructures is the cloud model most adopted by businesses and individual users. Public clouds may be accessed over the Internet and cloud features is not restricted to a single user or company. The main idea behind the public cloud is that resources made available for users and contracted by them are shared among other cloud's users. Thus, the public cloud is more accessible to various sizes of business, being cheaper and elastic. Contracted resources may be scaled easily, and users only pay for resources consumption. Features may be scaled automatically or users only need to make a few clicks to scale up/down the capacity of the contracted services. As a result, companies do not have to buy new servers or storage devices to ensure maximum performance for their systems. Public cloud is the ideal model for adoption by startups that need fast scalability or may have high seasonality. However, this model is not recommended for those who need a platform with high control over the features, for security reasons. In general, IaaS and SaaS services are among the most popular products licensed in this mode.

- **Hybrid Cloud:** Hybrid cloud is defined as a combination of a public cloud and a private one. For security reasons, companies are more willing to store data in private clouds, but at the same time expect to get public cloud computing resources. This solution has proven to be a viable, economic and secure option.

Among a large number of benefits associated with the implementation and use of cloud services, we can highlight the centralized resources management and the reduction of energy consumption and maintenance costs (MILLER, 2008).

2.1.1 IaaS Public Clouds

Today, there exists some Infrastructure-as-a-Service (IaaS) cloud providers. Amazon Web Services (AWS), Azure, and Cloud Oracle are among the most well-known IaaS cloud providers. In this work, we evaluate resource consumption of applications running on AWS (AWS, 2018b). Considering this, in this section, we briefly describe the features provided by AWS and that were considered in this work.

AWS is a cloud services platform that offers to everyone computational power, storage, content delivery among other powerful resources that make possible to organizations and startups to scale and grow effortlessly. Infrastructure services are provided by AWS as a commodity, which means that features are delivered on demand, made available in seconds, and customers pay only for them in a pay-as-you-go model. AWS consists of more than 50 availability zones in 18 geographic regions around the world, as well as a large number of services. Availability zones allow developers to provide virtual machines close to their end users, as well as providing a geographic distribution of data. Table 1 shows some of the most popular services provided by AWS. A large number of customers are developing advanced applications in a flexible, scalable, and reliable environment.

Customers may scale resources provisioned for their applications to target the defined requirements. EC2 offers a large number of instance types optimized for different processing workloads. Instance types comprise combinating of CPU, memory, networking

Table 1 – AWS's Featured Services (AWS, 2018b)

Service	**Description**
Amazon EC2	Virtual servers
Amazon Simple Storage Service (S3)	Scalable storage
Amazon Aurora	High performance managed relational database
Amazon DynamoDB	NoSQL database
Amazon RDS	MySQL, PostgreSQL, Oracle, and SQL Server
AWS Lambda	Run code without thinking about servers
Amazon VPC	Isolated cloud resources

capacity, and storage. EC2 provides flexibility to consumers by allowing them the freedom to select the appropriate set of feature to handle the demands for their applications. Developers may customize an Amazon Machine Image (AMI) considering the operating system and their application in order to the VM instances in the infrastructure use the same system image with the same configuration.

Developers need to evaluate the application requirements and select the appropriate instance family for supporting them. Amazon EC2 provides a large number of options across ten different instance types. Each instance with one or more size options, organized into instance families optimized for different workloads. AWS provides a wide set of services, such as computing power, storage, networking, and databases. All of them delivered as a commodity, that is delivered on-demand, and customers are charged considering a pay-as-you-go pricing model (AWS, 2017).

Reserved Instances (RIs) Users may contract RIs for long periods and pay fixed amounts for them. Customers can purchase RIs for a 1-year or 3-year term. RIs are specific to handle the normal demands of systems. AWS offers to customers a discount up to 75 % when using RIs compared with on-demand ones (ODIs). Users may contract RIs adopting three upfront payment options such as follows: No Upfront, Partial Upfront, and All Upfront. The discount rate applied to the contract depends on the chosen option. It means contracts' total value may be low when using RIs compared when using ODIs. However, there is a downside when using RIs. Contracted RIs are available for using them during the contract period. In other words, users pay for RIs regardless of their actual use. However, customers have the option to sell their unusable RIs to other AWS users through the Amazon EC2 Reserved Instance Marketplace [1]. By only using RIs, users may be required to maintain idle resources to process transient increases in users demand. As we will see, ODIs are valuable in dealing with transient increases in demand. It is important for developers to find out the normal usage pattern of their applications. Making

[1] https://aws.amazon.com/ec2/purchasing-options/reserved-instances/marketplace/

possible for them to avoid the RIs over-provisioning and pay much for deploying their applications on the public cloud.

On-demand Instances (ODIs). ODIs are a cost-effectively option to scale applications on the cloud. The main idea behind them is that customers only pay for the resources that are really needed to support the needs of their applications in order to maximize performance and minimize cost. In other words, applications have the option to provision and use them only when they need them. ODIs are an appropriate option to handle with a transient increase in application demand, as customers pay a fixed price per hour for using them. More generally, they are used to compose an autoscaling instances group. Developers need to evaluate their applications to find out their transient needs, and to define a cost-effective autoscaling strategy. Applications may completely deprovision instances when they are no longer needed, and stopping the billing related to them. However, it is important to highlight that the price paid for using ODIs for long periods may be higher than when using RIs. Developers need to carefully define autoscaling thresholds for their applications.

Spot Instances (SPIs). SPIs offer a contracting model that may be even more advantageous for AWS customers. SPIs are an appropriate option to avoid over-provisioning resources and, as a consequence, wastage of financial resources. Spot prices may be up to 90 % lower; related to on-demand ones. SPIs are equal to ODIs in terms of computational attributes. The difference between them falls in the way provisioning occurs. Spot prices are automatically adjusted according to the supply and demand of AWS customers. Each SPIs type has a transient value that is updated at periodic intervals. The provisioning process consists in customers setting bids to hiring SPIs. If a customer's bid is greater than or equal to the current price for the selected SPI, then it is provisioned. Despite offering great savings, this modality has some drawbacks. SPIs are the extra computational capacity available in AWS cloud. It means that when AWS needs extra computational capacity, some provisioned SPIs will be requested to supply the current cloud needs. For that aim, AWS interrupts some SPIs that are currently in use after two minutes it has sent a notification. This SPIs characteristic may be a limiting factor for applications that require high availability or reliability. SPIs are a good option to fault-tolerant and flexible applications. We have not evaluated them in this work.

AWS offers some options for developers to provision VM instances. Developers may use the web management console, the CLI, or the EC2 API, which makes it possible to provision instances programmatically. To scale multiple services offered by AWS, developers may use the AWS Auto Scaling service. Likewise, to scale only EC2 instances, developers may use the EC2 Auto Scaling service.

2.1.2 Autoscaling

Scaling is a technique that may offer performance and costs optimization (AL-DHURAIBI et al., 2018; CHEN; BAHSOON; YAO, 2018). According to NIST, on-demand self-service and rapid elasticity are key features related to cloud computing (MELL; GRANCE, 2011). Rapid elasticity is related to the capacity to add and remove resources to support demands variation. On-demand self-service and rapid elasticity offer the possibility to allocate computing resources without interacting with human resources, and only using the resources that are really necessary. The main idea behind autoscaling is making possible for an application to support transient increases in demands while maintaining the performance levels defined in SLAs.

An auto-scaling system can configure services and applications in a cloud taking into account a set of requirements and configurations. The cloud provider inserts or removes resources based on predefined thresholds set by the service provider. Adding and removing resources may be performed automatically through settings defined by developers. The system deployed on the cloud may perform resource provisioning through API calls provided by the cloud provider. Considering the AWS provider, another option is to use a high-level tool, such as AWS Cloud Watch (AWS, 2018a). Cloud Watch monitors AWS features such as Amazon EC2 and applications running on AWS. Some auto-scaling techniques have been proposed to provide an automatic solution for allocation and deallocation of resources.

Configuring auto-scaling strategies usually depends on the mapping of resources needed to meet performance requirements defined in SLAs. Therefore, it is necessary to identify the quantity of each resource that must be provisioned and released according to demand variations (LORIDO-BOTRAN; MIGUEL-ALONSO; LOZANO, 2014). A large number of parameters may be used to identify the needs of an application in relation to computational resources insertion or removal in the adjacent infrastructure, such as CPU utilization, disk usage, memory usage, queue size, number of requests in the system, and response time (GHANBARI et al., 2011).

In addition to the identification of computational resources, it is necessary to identify the time at which resources must be provisioned and destroyed. Generally, the techniques available to identify the instant of time to add or remove resources in the system are divided into reactive and predictive techniques (AL-DHURAIBI et al., 2018; CHEN; BAHSOON; YAO, 2018; CARDELLINI et al., 2018). Reactive techniques consider predefined rules and they respond to changes in the system state when any predefined threshold is found (LORIDO-BOTRAN; MIGUEL-ALONSO; LOZANO, 2014). This technique is widely used in commercial systems (GALANTE; BONA, 2012). Predictive techniques try to predict the resource needs for the system considering a future instant of time. Usually, they use techniques such as time series to support this analysis (HAMILTON, 1994). Resources are provisioned or released before the system needs them based on the result obtained. This is to ensure that the necessary resources are always available whereas trying to avoid holding resources when they are not needed. This technique can also be used in combination with a reactive approach (CARDELLINI et al., 2018).

There are two main approaches to scaling a system deployed on a cloud (AL-DHURAIBI et al., 2018).

- **Vertical Scaling:** Vertical scaling, also called scale up and scale down, means changing the capacity of a resource in the system infrastructure. For example, the amount of RAM in a virtual machine may be increased or decreased according to changes in workload. However, as the resources in the virtual machines have changed, it is necessary to reboot the virtual machine so that the system can recognize changes in hardware capacity. Therefore, it is not uncommon to automate vertical scaling.
- **Horizontal Scaling:** Horizontal scaling, also known as scale-out and scale-in, means adding or removing VM instances in the infrastructure for supporting a change in workload. The system deployed on a cloud continues running without interruption while new instances are provisioned. Once the provisioning process is complete, instances are deployed in the system infrastructure. As demand diminishes, the system may terminate and deallocate additional resources previously inserted.

2.2 Mobile Cloud Computing

Mobile Cloud Computing (MCC) is a paradigm that increases the performance of mobile applications and reduces their energy consumption through offloading technique (KUMAR; LU, 2010), (KOCJAN; SAEED, 2012). The offloading process sends tasks to be processed on remote servers in the cloud. The first step in performing task offloading is to split an application into different *parts*. Next, there is a decision of which ones are most convenient for remote processing. Tasks may be partially or completely offloaded, depending on the application requirements. Method-call offloading is a partitioning strategy that enables to split a code into multiple parts. Deciding which method to partition is not an easy task. It is necessary to analyze many possible scenarios.

2.2.1 Method-Call Offloading

An application may offload method calls without restrictions to the cloud. The above-mentioned scenario is called Full Method-Call Offloading (LI; WANG; XU, 2001), (BALAN, 2006), (KUMAR; LU, 2010). An alternative to the Full Method-Call Offloading is the Partial Method-Call Offloading. In a method-call partitioning strategy, the source code is partitioned at method level (KOSTA et al., 2012), (KEMP et al., 2012), (CUERVO et al., 2010), (PATHAK et al., 2011), (SILVA et al., 2015), (RIM et al., 2006), (SAARINEN et al., 2012), (LI; WANG; XU, 2001), (BALAN, 2006), (KUMAR; LU, 2010). Instead of all methods, only a subset of them is offloaded. Data dependencies must be observed to perform a correct partition. If some parts of an application can be executed in parallel, then, these tasks may be offloaded to more than one server concurrently. For example, considering the methods *m1()* and *m2()*, two decisions should be made (as illustrated in Figure 5): **Where to execute *m1()* and *m2()*?**

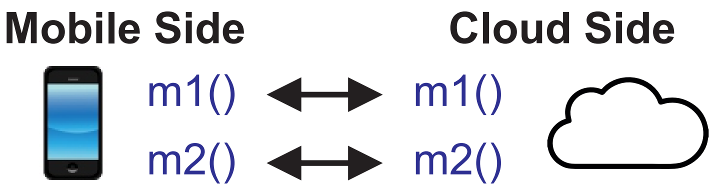


Figure 5 – Partial Method-Call Offloading

Combining these possibilities (mobile device or the cloud), four scenarios may be exploited (see Table 2). The number of possibilities increases proportionally to the application complexity. In a real-world scenario, the number of combinations of method-calls to offloading may be very large. Such variety makes harder the software engineer's work when deciding the most appropriate offloading distribution. The developer would expect that partitioning and distributing a set of method-calls on multiple servers would reduce its total execution time. However, one method-call may present a low level of processing, not justifying the offloading process. Besides, other aspects, such as the mobile device current CPU and energy consumption level, may influence the decision.

Table 2 – Possible Offloading Scenarios.

Possibility	***m1()***	***m2()***
Scenario #1	mobile	mobile
Scenario #2	mobile	cloud
Scenario #3	cloud	mobile
Scenario #4	cloud	cloud

2.3 Performance Evaluations of Systems

System administrators need to provide the highest performance at the lowest cost. A performance evaluation is necessary when a system administrator wants to compare a number of alternative configuration scenarios to find the best one. It is also used to compare two similar systems and decide which one is better for a given task. Performance evaluation can also help to determine how well a system is performing certain tasks, and if some improvements are necessary. Generally, evaluating the performance of a system means to verify its behavior according to a defined set of metrics. The researcher must select appropriate evaluation techniques (e.g.: analytical modeling, simulation or measurement), perform a statistical analysis to identify possible bottlenecks and propose improvement solutions. This work has applied a performance analysis from the analytical modeling with SPN and CTMC models, and measurements based on the Design of Experiment (DoE) technique.

2.3.1 Measurement

Design of Experiments (DoE) technique allows to obtaining a maximum of information about a system, regarding many factors, with a reasonable number of experiments and effort (JAIN, 1990; MONTGOMERY, 2017). A set of experiment executions planned through DoE can be analyzed to determine if the factors have significant effects, or if the differences in the observed effects are due to variations caused by measurement errors and not controlled parameters (JAIN, 1990; MONTGOMERY, 2017).

This study adopts the General Full Factorial Design, which uses all possible combinations of levels for all factors, i.e., there are no limits to the number of factors and the number of levels. This type of DoE allows every configuration to be examined, so we can find the effects of all factors and their interactions, which is an advantage; the disadvantage is that the cost of analysis can be very high if the number of factors and levels is too high, and also considering that each of these experiments may have to be repeated several times. It is possible to reduce the number experiments by reducing the number of factors, and/or the number of levels for each factor, or using Fractional Factorial Design instead (JAIN, 1990).

2.3.2 Continuous Time Markov Chain

Markov chains are used for evaluation of performance and dependability of computer and communication systems. Markov chains make it possible to model the interaction between different components that are part of an infrastructure. Markov chains may be described as a state space diagram associated with a Markov process (BOLCH et al., 2006). Markov chains are a stochastic process with memoryless property (CHUNG, 1954). Lack of memory is a property of some probability distributions, and it is formally known as the Markov property. Markov chains and stochastic processes form the basis for model-based system evaluations.

A stochastic process is characterized by a set of random variables X, indexed by an ordered set T, representing the evolution of a system over time (CASSANDRAS, 2008). A random variable is a quantitative variable that may assume different numerical values influenced by random factors. A random variable may represent the time of an activity or event that is in a sample space Ω, and it is related to some probability distribution. A stochastic process can be regarded as a set of random variables X defined as $X_t : t \in T$, of which each instance X_t is characterised by a probability distribution function. The index set T is mostly associated with the passing of time. $Pr\{X_t\}$ defines the probability that a given event X_t occurs. A stochastic process is the opposite of a deterministic process, where times are not influenced by random factors. The set of all possible values of X_t ($\forall t \in T$) is called the state space of the stochastic process (BOLCH et al., 2006; CASSANDRAS, 2008), named here as set S.

A Markov chain is a discrete-valued Markov process. Discrete-valued means that the state space of possible values of the Markov chain is finite or countable. A Markov process is a stochastic process in which $Pr\{X_{t_{n+1}}\}$ depends only on the previous value of X_{t_n}, $\forall t \in T$ and $\forall s_i \in S$. A Markov process is commonly referred to as a memoryless process, in which the future state of the system depends only on its current state (BOLCH et al., 2006; HAVERKORT, 2001). In other words, it means that the path followed by a process from its initial state to its current state is irrelevant for evaluation purposes. Furthermore, in a memoryless process, the transition to the next state is not influenced neither by the time passed in the current state, nor on states visited previously (BOLCH et al., 2006; CASSANDRAS, 2008). Due to the Markov property, the time associated with an activity must follow a memoryless distribution. In the case of Markov chains, the probability distribution related to the times involved follows an exponential distribution.

Some abstractions for the Markov chains have been proposed to represent the way transitions between states occurs over time. Depending on the type of random variables related to the process's activities, times can be characterized as discrete time or continuous time. Understanding the main difference between discrete and continuous Markov chains is a key factor in choosing the most appropriate abstraction for representing processes. Discrete Time Markov Chain (DTMC) represents systems that evolve through discrete time steps. It means the system only changes at one of discrete time values in the set T, which $t \in T$ represents only non-negative integer values. On the other hand, Continuous Time Markov Chain (CTMC) represents systems which the state changes may happen at any time along a continuous interval (it means that $T \in R_+$). Basically, CTMC is similar to DTMC with the difference that transitions in CTMC may occur any instant of time (BOLCH et al., 2006). The way metrics are calculated is directly related to the Markov chain abstraction chosen.

Graphically, CTMCs are represented by a directed graph. Vertices represent states while directed arcs indicate how transition between states occurs. In Markov chains, states represent different conditions a system may follow. Each arc has a transition rate or probability assigned to itself, indicating the way at which transitions occur from one state to another. Transitions between states indicate events occurrence (SILVA et al., 2013).

Figure 6 shows an example of a CTMC.

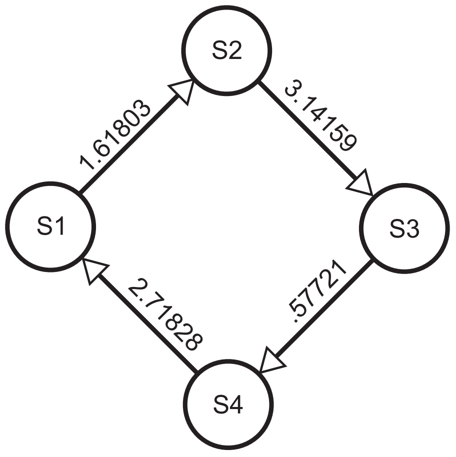


Figure 6 – A CTMC

Mathematically, CTMC is defined by a probability vector for states π and a transition rate matrix Q, also known as infinitesimal generator matrix. Matrix Q is a square matrix that has a dimension equal to the number of states in the state space S. Matrix Q's elements represent the transition rates between the states of the chain. The elements q_{ij} (for $i \neq j$) represent the rate departing from i and arriving in state j. The main diagonal elements q_{ii} are always defined by $-q_{ii} = \sum_{j,j\neq i} q_{ij}$. The off-diagonal elements are always nonnegative, whereas the main diagonal elements are always negative. The sum of each matrix row equals zero. Once the model structure has been defined so that the infinitesimal generator matrix Q is known. On the other hand, the probability vector for states π is a one row vector containing the transition probability from the current state for all states, including itself. The initial probability vector is defined as $\pi(0)$. Below we can see the initial probability vector $\pi(0)$ and the matrix Q related to the CTMC shown in Figure 6 defined by the state space $S = \{\text{S1, S2, S3, S4}\} = \{0, 1, 2, 3\}$. As we can see, the probability vector $\pi(0)$ defines that the probability of starting in state S1 is equal to 1. That is, the process always starts from the state S1. Through the Q matrix and the probability vector π it is possible to compute stationary and transient metrics.

$$\pi(0) = \begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

$$Q = \begin{pmatrix} q_{00} & q_{01} & q_{02} & q_{03} \\ q_{10} & q_{11} & q_{12} & q_{13} \\ q_{20} & q_{21} & q_{22} & q_{23} \\ q_{30} & q_{31} & q_{32} & q_{33} \end{pmatrix}$$

$$Q = \begin{pmatrix} -1.61803 & 1.61803 & 0.0 & 0.0 \\ 0.0 & -3.14159 & 3.14159 & 0.0 \\ 0.0 & 0.0 & -0.57721 & 0.57721 \\ 2.71828 & 0.0 & 0.0 & -2.71828 \end{pmatrix}$$

Non-time dependent metrics are obtained through stationary analysis (BOLCH et al., 2006). Equation 2.1 allows the computation of the state probability vector for steady-state analysis. The resolution of the system of differential equations obtains a probability distribution that converges to a vector of stationary probabilities, that is independent of the initial distribution $\pi(0)$. Based on the limiting probability distribution obtained it is possible to find out what happens in the long run.

$$\pi \times Q = 0, \sum_{i \in S} \pi_i = 1 \tag{2.1}$$

Time-dependent metrics are obtained through transient evaluation (BOLCH et al., 2006). Transient evaluation makes possible to find out what happens on the system taking into account a specific period of time. Transient evaluation allows evaluating how a system behaves before it reaches its stationary state. The row vector $\pi(t) = [\pi_1(t), \pi_2(t), \ldots, \pi_n(t)]$ represents the transient state probability vector of the CTMC in time t. The behavior of the CTMC can be described by the Kolmogorov Equation 2.2 given the initial probability vector $\pi(0)$. Equation 2.3 gives the expected total time the CTMC spends in state i during the interval [0, t). The performance metrics can be derived for each evaluated system from its state probability vector π and its Q matrix.

$$\frac{d\pi(t)}{dt} = \pi(t) \times Q \tag{2.2}$$

$$L_i(t) = \int_0^t \pi_i(x)dx \tag{2.3}$$

Sometimes, it is necessary to evaluate the amount of time a process will take to complete a task. The measure of interest is the Mean Time to Absorption (MTTA). To calculate MTTA, the model must have at least one absorbing state. For a better understanding of MTTA computation, it is necessary to define the difference between transient state and absorbing one. Transient states are defined as temporary states. In other words, when the system leaves a transient state, there is a likelihood of never coming back to it again. On the other hand, an absorbing state is a state that when the system reaches it, there is no way out. An absorbing state may represent the end of a task. For MTTA evaluation, the state space S is partitioned into two sets, the set of absorbing states A and the set of non-absorbing states N. The next step is to create the initial probability vector for transient states $\pi_N(0)$. From the matrix Q a new matrix Q_N can be constructed by restricting Q to the transient states. Matrix Q_N is a square matrix with dimension defined by the cardinality of the set of non-absorbing states N. The time spent before absorption can be calculated restricted to the states of the set of non-absorbing states (N) by $lim_{t \to \infty} L_N(t)$. Thus, $L(t)$ satisfies the Equation 2.4 where $\pi_N(0)$ is the vector $\pi(0)$ restricted to the states in the set N. Q_N is the infinitesimal generator matrix restricted to the non-absorbing states. Finally, MTTA may be described as Equation 2.5 (BOLCH et al., 2006).

$$L_N(\infty)Q_N = -\pi_N(0) \tag{2.4}$$

$$MTTA = \sum_{i \in N} L_i(\infty) \tag{2.5}$$

2.3.3 Petri Nets

Carl Adams Petri formalized the initial conceptions of Petri Nets (PNs) in his Ph.D. thesis presented at the Technical University of Darmstadt, Germany (PETRI, 1962). Petri nets are a powerful modeling tool that may be used to represent concurrent, asynchronous, distributed, parallel, deterministic, or stochastic processes. The original theory (place/transition net) was developed as an approach to model and analyze communication systems. PNs define a specification technique that allows a graphical and mathematical representation, and it has analytical mechanisms that enable verification on the properties and correctness of modeled systems. As a mathematical tool, it is possible to set up state and algebraic equations, and other mathematical models governing the behavior of systems. As a graphical tool, PNs may be utilized as a visual representation mechanism, aiding in process modeling and analysis.

PNs are constituted by a set of graphical components as shown in Figure 7. PNs are constituted by a set of places, transitions, tokens, and directed arcs. Basically, PNs are directed models consisting of two types of nodes: places and transitions. Places are represented by circles, whereas transitions are depicted as filled rectangles. Places correspond to states in which the modeled system may be at some point during its operation. Transitions represent events occurrence or actions taken by the system. Tokens are markings that indicate the activation of a particular state as well as available resources. Further, tokens are used in these nets to simulate the dynamic and concurrent systems activities. Tokens (small filled circles) are stored in places, and depending on the model, each place may store an indefinite number of them. The number of tokens that may be stored in a place is in the cardinality of the natural numbers (N). The way which a token set is distributed over the places at a given moment determines the current system state. Directed arcs define the flow of tokens. They can only connect places to transitions or vice versa. Each individual arc has a weight with cardinality N^+ and for default, its weight is one.

The arc weight defines the number of tokens consumed or generated during transition firings. As we can see, visual modeling allows users to have a quick overview of the workings of complex systems as well as their constraints. The way these components are organized defines the system and its behavior.

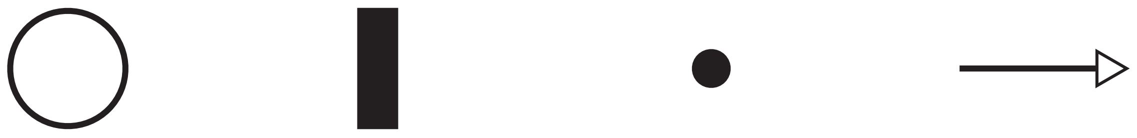
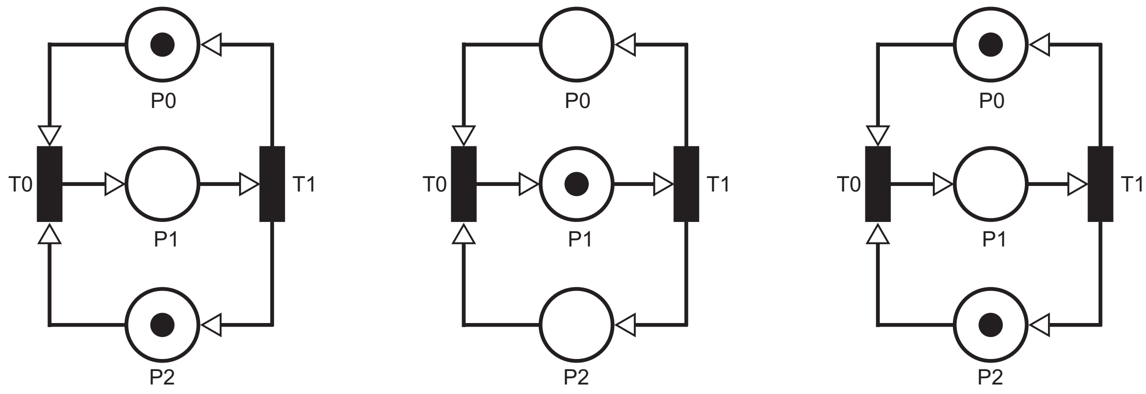

(a) Place (b) Transition (c) Token (d) Arc

Figure 7 – Components of a Petri Net.

The behavior of a PN is defined in terms of a token flow. Tokens are created and destroyed according to the transition firings (GERMAN, 2000). There may be a relationship between places and transitions so that transitions may be enabled and able to fire. The places where the directed arcs exit toward a transition are called input places whilst the places connected to the arcs that exit a transition are called output places. When a transition fires, tokens are consumed from its input places, and new ones may be created at its output places. The execution of an action or event (i.e. transition firing) may depend on certain preconditions. A transition can only be enabled whether the number of tokens available at its input places is equal to or greater than the weight of the arcs connecting each place to it. After transition firing, some places may have their markings changed leading the model to a new state (i.e. postcondition). Transition firing occurs atomically which means it consumes tokens and generates new ones in one single step. Further, it is important to highlight that transitions may fire without the existence of input places connected to them and transitions may not be connected to output places. The order in which transitions are fired occurs non-deterministically. That is, if two or more transitions are enabled in a certain state, then the transition to be fired will be chosen randomly.

Figure 8 contains the graphical representation of a Petri net in two different markings. It consists of three places (P0, P1, and P2) and two transitions (T0 and T1). Figure 8a shows the model in its initial marking. Places P0 and P2 both have one token, which creates the preconditions for the transition T0 to be fired. Transition T0 firing takes the model to its second marking (see Figure 8b). This marking corresponds to a single token in the place P1. In the second marking, transition T0 is no longer enabled and cannot be fired. However, the T1 transition is enabled and may fires. Transition T1 on firing takes the model to the initial marking again. This causes T0 transition to be fired, which repeats the entire firing cycle again (see Figure 8c). Since transitions T0 and T1 are immediate, there is no delay for them to be fired.

Since Carl Petri's work, many representations and extensions have been proposed providing more concise descriptions and describing systems features not observed in previous models (MURATA, 1989). Time representation was introduced in PNs (NOE; NUTT, 1973;



(a) Initial Marking (b) Second Marking (c) Initial Marking Again

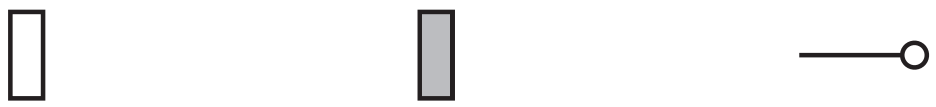
Figure 8 – Example of a Petri Net

MERLIN; FARBER, 1976). More specifically, event time may be represented in an interval, deterministic, non-deterministic or stochastic way (RAMCHANDANI, 1974; MERLIN; FARBER, 1976; MARSAN; CONTE; BALBO, 1984). As extensions to traditional PNs we can cited the Colored Petri Nets (CPNs) (JENSEN, 2009), inhibitor arcs, object petri nets, time and timed Petri nets, stochastic Petri nets (SPNs), among others.

2.3.4 Stochastic Petri Nets

Stochastic Petri Nets (SPNs) are an extension of Petri Nets (PNs). Stochastic modeling is a widely used approach in the performance evaluation process (MOLLOY, 1981). SPNs let the use of PNs in the performance evaluation of systems and processes. SPNs associate a stochastic delay to each timed transition. Thus, PNs become probabilistic, being described by a stochastic process. Timed transitions store the time to perform an activity. The period in which a timed transition remains enabled corresponds to the duration of a activity. Transition firing represents the end of the activity. On the other hand, immediate transitions represent instantaneous activities which means they do not have time associated with themselves. Besides, they have higher firing priority than timed transitions. The need to represent both timed and immediate activities within a single formalism gave rise to an extension of SPNs called Generalized Stochastic Petri Nets (GSPNs). GSPNs allow the association of timed and immediate transitions on the same model (MARSAN; CONTE; BALBO, 1984), making possible the modeling of systems with times, and logical conditions. In the literature the acronym SPN is often used to represent the entire family of SPN derived models, such as GSPN (GERMAN, 2000). In this work, from here down we have used the acronym SPNs when referring to GSPNs.

SPNs have introduced new components and rules to the tradicional PNs. Figure 9 exhibits the new components introduced into SPN formalism. Timed transitions are depicted as hollow or gray rectangles. White transitions represent activities with exponential times (see Figure 9a). Other probability distributions other than exponential are represented graphically as gray color transitions (see Figure 9b). An inhibitor arc is a special arc type that depicts a circle-headed at one edge, instead of an arrow (see Figure 9c). Inhibitor arcs are usually used to disable transitions if the corresponding input place contains at least as many tokens as the cardinality of the corresponding inhibitor arc (MARSAN et al., 1994).



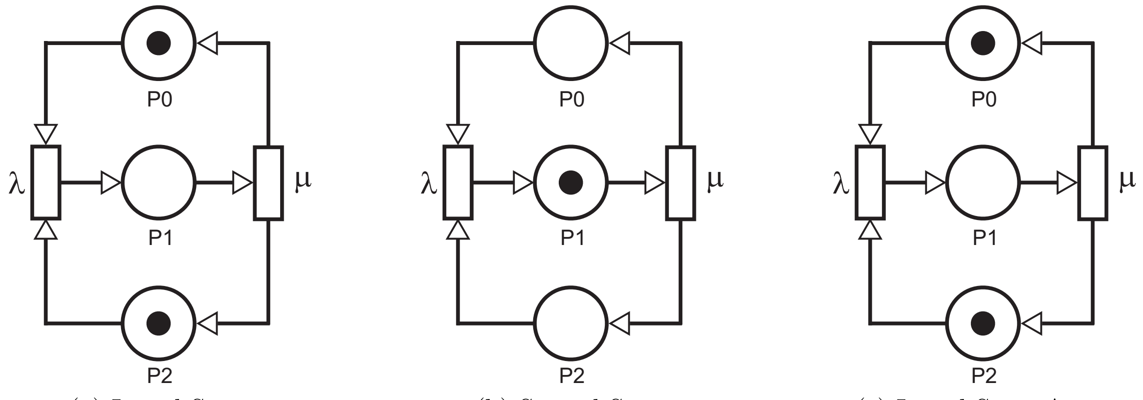
(a) Exponential Timed Transition (b) Non-Exponential Timed Transition (c) Inhibitor Arc

Figure 9 – SPN Components.

There are some properties that increase modeling power. It is possible to associate priorities, weights and enabling functions with immediate transitions (HERZOG, 2001; MARSAN et al., 1994). Transitions with higher priority level fire first. The weight defines the probability at which a immediate transition will be fired when more than one of them have the same priority level and is enabled in the same marking. Enabling functions are logic expressions related to the net's marking and places. Even when there are tokens in sufficiente numbers at its input places, an immediate transition only may fires when its enabling function is evaluated as true (MARSAN et al., 1994). On the other hand, it is possible to associate guard expressions with timed transitions (HERZOG, 2001; MARSAN et al., 1994). Guard expressions have the same concept of the enabling functions, but they can only be associated with timed transitions. Enabling functions are adopted in this work.

The addition of timed transitions introduced the concept of multiple enabling degrees and firing semantics. Enabling degree is related to the maximum number of tokens that make a timed transition enabled considering for it tokens available in its input places. Firing semantics define the way in which a transition fires when its enabling degree is greater than one. In other words, a firing semantic defines the number of tokens that will be processed in parallel. There are three types of them defined as Single Server Semantic (SSS), Infinite Server Semantic (ISS), and K-Server Semantic (KSS). SSS defines that timed transitions can only be enabled and fired by the processing of one token in each input place at a time. That is, there is no token parallel processing in SSS. The time counting will be restarted after each transition firing. Transitions with ISS assigned to them process the entire set of tokens available in their input places simultaneously. Finally, when using KSS, transitions process a set of tokens in parallel up to the maximum degree of parallelism defined. System analyst should pay attention to the assignment of firing semantics to each timed transition because they have a direct impact on the metric calculation.

The proposals regarding performance evaluation aimed at an equivalence between SPN and CTMC (GERMAN, 2000). SPNs with a finite number of places and transitions may be isomorphic for continuous time Markov chains and, hence, they can provide performance measures (GERMAN, 2000). In order to obtain an equivalence between SPNs and CTMCs, it was necessary to introduce temporal specifications such that the future evolution of a model, given the present marking, is independent of its marking history. For that aim, all time values assigned to all timed transitions must be exponentially distributed. In other words, the reachability graph of the state space related to an SPN model is converted to a CTMC for performance evaluation. In this case, each reachable tangible marking in an SPN model is equivalent to one state in a CTMC. Figure 10 depicts a SPN with two reachable states and Figure 11 shows the CTMC related to the reachability graph of this SPN. The SPN contains only exponential transitions. As we can see, the CTMC in question contains two states S1 and S2, which represent the initial state and the second one, respectively. Therefore, SPNs may be converted to CTMCs, which means that SPNs may be resolved to reach the desired performance or reliability results (MOLLOY, 1982), (MARSAN et al., 1994), (TRIVEDI, 2001), (MARSAN, 1990). If non-poli-exponential distributions (BOLCH et al., 2006), (SOUSA et al., 2014) are adopted, the SPN can only be evaluated through simulation.



(a) Initial State (b) Second State (c) Initial State Again

Figure 10 – Example of a basic SPN

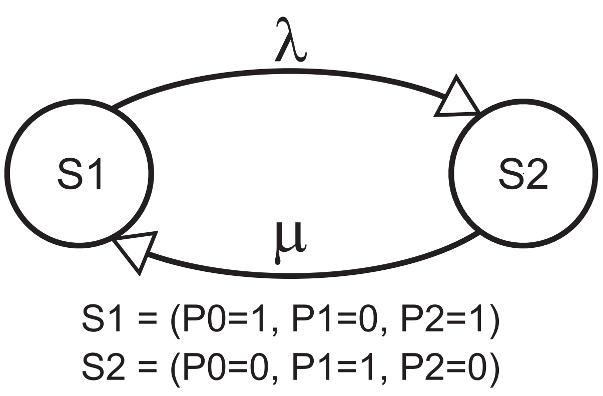


Figure 11 – CTMC Related to the State Space of the SPN Presented in Figure 10

Through numerical evaluation it is possible to obtain more precise results. However, there are limitations for its usage. The first limitation is that the time necessary to metrics computation could be very large when during numerical evaluation occurs state space explosion. The second one is that the probability distribution associated with all timed transitions times must be exponential. A technique called moment matching could be used when using non-exponential times. Moment matching adds new places and transitions to represent the poly-exponential distribution which more approximate the non-exponential distribution evaluted. However, as demonstrated in Section 2.3.5, the technique may lead to the occurrence of state space explosion (TUFFIN; HIREL; TRIVEDI, 2007).

Simulation is another way to compute model metrics. Simulation techniques do not generate and numerically evaluate the CTMC related to the SPN state space. Thus, they are a viable option when occurs state space explosion during model numerical evaluations. Further, when using simulation it is possible to evaluate SPN models with non-exponential timed transitions. Metrics computed by simulation techniques are obtained within a confidence interval whilst metrics obtained by numerical analysis are more accurate (TUFFIN; HIREL; TRIVEDI, 2007). Simulation methods are a good option when there are limitations to use numerical evaluations.

There are some tools that make be able the construction of SPN models as well as metrics computation such as the Mercury, TimeNet, GreatSPN or SHARPE (SILVA et al., 2015; GERMAN et al., 1995; GREATSPN, 2004; TRIVEDI; SAHNER, 2009). These tools generate the state space of the evaluated SPN model and create the corresponding CTMC enabling numerical evaluations. In addition, they implement simulation techniques for allowing model evaluations when numerical evaluation execution is not possible. The most simple SPN models may have a large state space. System analysts may model the whole systems directly using CTMC. It means that in some situations, it is a very difficult task to model all possible states in which a system may be as well as the transition rates between them. Errors during manual CTMC definition are avoided when using a stochastic modeling tool.

Despite the simplicity of SPNs, they are a powerful formalism. Using SPNs it is possible to model complex activities as concurrency, resource sharing and synchronization in a easy way. Figure 12 shows an example of an SPN model using all SPN components. There are difficulties in modeling synchronization and resource sharing activities when using queuing networks (KLEINROCK, 1975; BOLCH et al., 2006). However, SPNs have the power to represent queues similar to queuing networks and offer more descriptive power compared to them (MARSAN et al., 1994).

2.3.5 Phase-type Approximation Technique

A number of approximate analysis techniques are based on matching moments of continuous time phase-type distributions. Systems with non-Markovian properties may be ana-

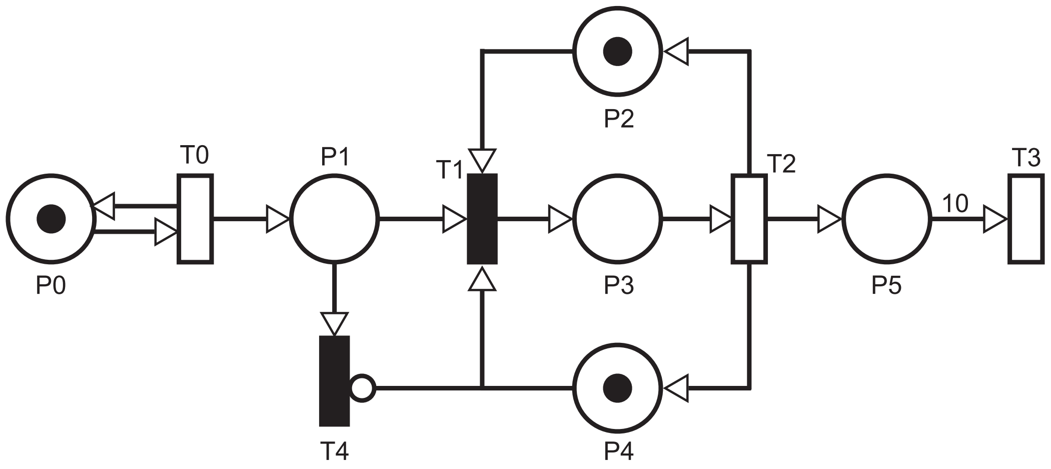


Figure 12 – An Example of an SPN model

lyzed numerically through phase-type approximation technique (DESROCHERS; AL-JAAR; SOCIETY, 1995; MALHOTRA; REIBMAN, 1993). Phase-type approximation technique allows to represent unknown or non-exponential distributions using poly-exponential distributions such as Erlang, Hyperexponential and Hypoexponential (TRIVEDI, 2001).

The technique receives as input the inverse of the coefficient of variation (CV) corresponding to the evaluated empirical probability distribution (DESROCHERS; AL-JAAR; SOCIETY, 1995). The CV is defined as the ratio between the standard deviation σ and the mean μ of the empirical distribution (see Equations 2.6 and 2.7). It is important to highlight that numerical evaluation may not be possible when using poly-exponential distributions. As already mentioned in this work, it is necessary to add new places and transitions in the SPN model to represent the selected poly-exponential distribution. Thus, phase-type increases the state space of a model. When state space explosion occurs, simulations should be adopted. The evaluation of CV lets to choose the most suitable poly-exponential distribution.

$$CV = \frac{\sigma}{\mu} \tag{2.6}$$

$$\frac{1}{CV} = \frac{\mu}{\sigma} \tag{2.7}$$

Erlang distribution can be used to represent an activity when the inverse of the coefficient of variation related to it is an integer number other than one. Erlang distribution is represented in an SPN model by the addition of the Erlang SPN subnet depicted in Figure 13 in order to replace a non-exponential timed transition. Equation 2.8 obtains the value of γ assigned as the multiplicity of the output arcs of the transition T1 and the place P2. The firing rate to be assigned to the new exponential timed transition T2 is calculated by Equation 2.9 (DESROCHERS; AL-JAAR; SOCIETY, 1995).

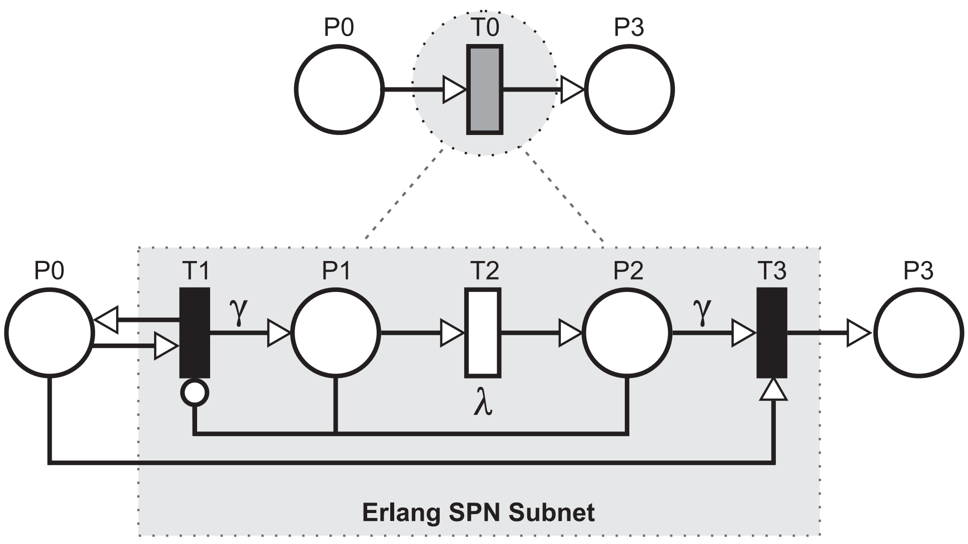


Figure 13 – Erlang Distribution SPN Subnet

$$\gamma = \left(\frac{\sigma}{\mu}\right)^{-2} \tag{2.8}$$

$$\lambda = \frac{\gamma}{\mu} \tag{2.9}$$

Hyperexponential distribution can be used to represent an activity when the inverse of the coefficient of variation obtained is a number less than one. Hyperexponential distribution is represented in an SPN model by the addition of the Hyperexponential SPN subnet depicted in Figure 14 in order to replace a non-exponential timed transition. The firing rate to be assigned to the new exponential timed transition T3 is calculated by Equation 2.10, and the weights assigned to the two new immediate transitions T1 and T2 are calculated by Equations 2.11 and 2.12 (DESROCHERS; AL-JAAR; SOCIETY, 1995), respectively.

$$\lambda = \frac{2\mu}{\mu^2 + \sigma^2} \tag{2.10}$$

$$w_1 = \frac{2\mu^2}{\mu^2 + \sigma^2} \tag{2.11}$$

$$w_2 = 1 - w_1 \tag{2.12}$$

Hypoexponential distribution can be used to represent an activity when the inverse of its coefficient of variation is a non-integer number greater than one. Hypoexponential distribution is represented in an SPN model by the addition of the SPN subnet depicted in Figure 15 in order to replace a non-exponential timed transition. Equation 2.13 obtains the number of phases represented by γ assigned as the multiplicity of the output arcs of the transition T2 and the place P3. The firing rate to be assigned to the new transitions

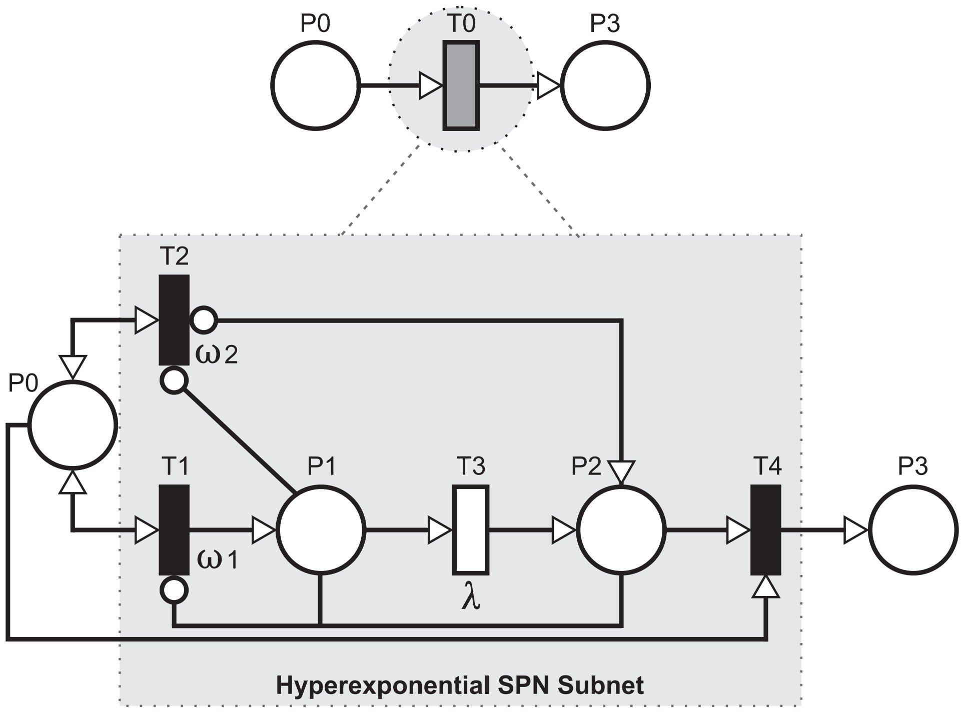


Figure 14 – Hyperexponential Distribution SPN Subnet

is calculated by Equations 2.14 and 2.15 (DESROCHERS; AL-JAAR; SOCIETY, 1995). The first exponential transition (T2) in the subnet receives the firing rate λ_1 obtained by Equation 2.14, while the second timed transition (T3) receive the rate λ_2 obtained by Equation 2.15. The values of μ_1 and μ_2 considered by those equations are obtained by Equations 2.16 and 2.17.

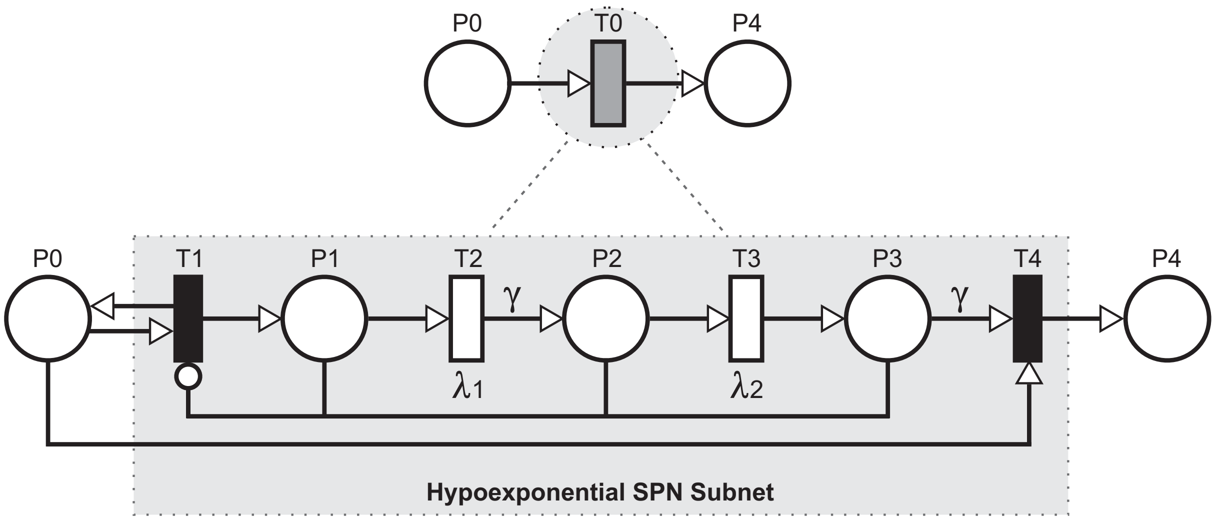


Figure 15 – Hypoexponential Distribution SPN Subnet

$$\left(\frac{\mu}{\sigma}\right)^2 - 1 \leq \gamma < \left(\frac{\mu}{\sigma}\right)^2 \tag{2.13}$$

$$\lambda_1 = \frac{1}{\mu_1} \tag{2.14}$$

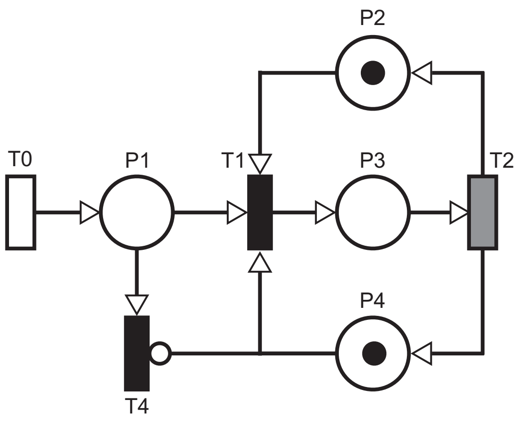
$$\lambda_2 = \frac{1}{\mu_2} \tag{2.15}$$

$$\mu_1 = \frac{\mu \pm \sqrt{\gamma(\gamma+1)\sigma^2 - \gamma\mu^2}}{\gamma+1} \tag{2.16}$$

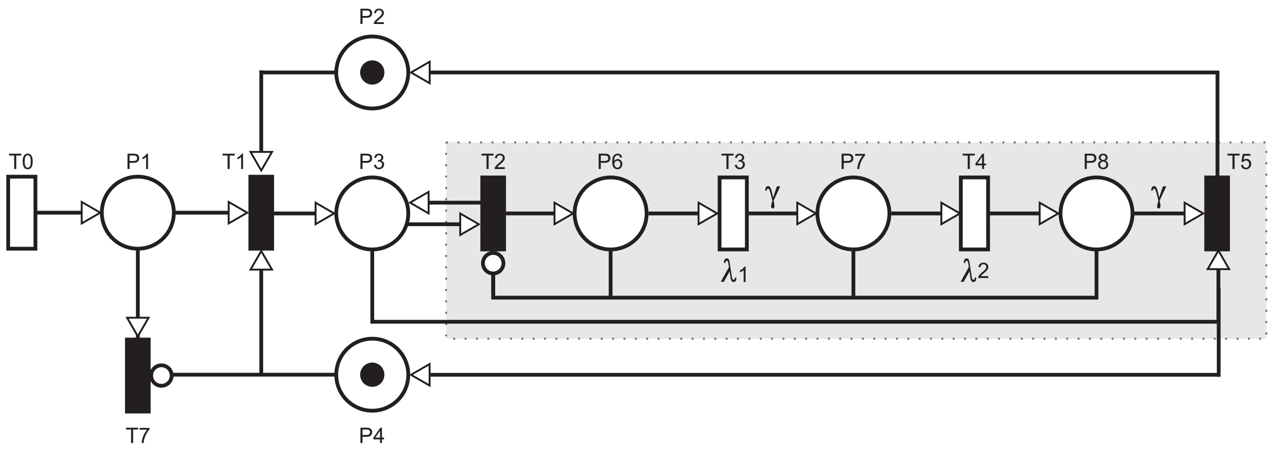
$$\mu_2 = \frac{\gamma\mu \pm \sqrt{\gamma(\gamma+1)\sigma^2 - \gamma\mu^2}}{\gamma(\gamma+1)} \tag{2.17}$$

Now let us see how the change occurs in a real SPN model to represent a poly-exponential approximation of a non-exponential timed activity by adding an SPN subnet related to the chosen approximate distribution. Figure 16a demonstrates an SPN model with a non-exponential timed activity represented by transition T2. Figure 16b demonstrates the same model with transition T2 replaced by an Hypoexponential subnet. Now the changes enable the numerical evaluation of the model.

A few stochastic modeling software may support analysts in order to choose the poly-exponential distribution that best fits their empirical distribution. The software Mercury evaluates the most appropriate poly-exponential distribution based on the mean and standard deviation of the empirical distribution (SILVA et al., 2015). In addition, Mercury demonstrates how the SPN subnet related to the most suitable poly-exponential distribution should be constructed in the SPN model of the user as well as it shows the values of each parameter to be assigned in the subnet components. Mercury supports five poly-exponential distribution that are Erlang, Hyperexponential, Hypoexponential, Cox-1, and Cox-2.



(a) An SPN Model with a Non-Exponential Timed Activity.



(b) An SPN Model with a Hypoexponential Subnet.

Figure 16 – An SPN Model with a Non-Exponential Timed Activity Represented by a Hypoexponential Distribution.

2.4 Face Detection and Recognition

One of the most remarkable abilities of human vision is the ability for face recognition. This ability is important for several aspects of our social life, and together with related abilities, such as estimating the expression of people with which we interact, has played an important role in the course of evolution (BRUNELLI; POGGIO, 1993). For many years, several different techniques have been proposed for computer recognition of human faces (KANADE; COHN; TIAN, 2000) (RICKMAN; STONHAM, 1992) (COHEN et al., 2003) since other methods do not offer the same reliability in the biometric personal identification field for example.

There is an obvious and strong need for user-friendly systems that can secure our assets and protect our privacy, without losing our identity in a sea of numbers. At present, one needs a personal identification number to withdraw money from an automated banking machine, a password for a computer, a dozen others to access some Internet services, and so on. Although reliable methods of biometric personal identification exist, such as fingerprint analysis and iris scans, these methods rely on the cooperation of the participants, whereas a personal identification system based on analysis of frontal or profile images of the face is often effective without the participant's cooperation or knowledge (ZHAO et al., 2003).

Table 3 lists some of the applications of face recognition.

Table 3 – Typical Applications of Face Recognition (ZHAO et al., 2012)

Areas	**Specific Applications**
Entertainment	Video game, virtual reality, training programs
	Human-robot-interaction, human-computer-interaction
Smart cards	Drivers' licenses, entitlement programs
	Immigration, national ID, passports, voter registration
	Welfare fraud
Information security	TV Parental control, personal device logon, desktop logon
	Application security, database security, file encryption
	Intranet security, internet access, medical records
	Secure trading terminals
Law enforcement and surveillance	Advanced video surveillance, CCTV control
	Portal control, postevent analysis
	Shoplifting, suspect tracking and investigation

A general statement of the problem of face recognition can be formulated as follows: given images of a scene, identify or verify one or more persons in the scene using a stored database of faces. The solution to the problem involves segmentation of faces (face detection) from cluttered scenes and extraction of features from the facial regions for recognition. In this work we consider face detection and extraction in the same phase, since face extraction is responsible for capturing the specific facial characteristics. We call the entire process, including all three steps, the face recognition process (ZHAO et al., 2003).

The face detection determines the potential locations of the human faces within an image (e.g.: Figure 17). In this work we have utilized the Haar Features and Haar Classifiers (VIOLA; JONES, 2004) to perform face detection. This decision was motivated by their widespread adoption for a vast range of computer vision applications (TANG et al., 2010).

This iterative approach begins with fairly primitive classifiers that group potential face candidates based on a small number of features. These simple classifiers in this initial stage have low computational complexity but must operate on a large amount of data, and they produce a large number of face candidates. The algorithm then progressively eliminates some of these candidates by using increasingly more sophisticated classifiers based on additional features at successive stages of the detection pipeline, such that the final stage outputs the detected faces with high confidence (SOYATA et al., 2012). Although the number of remaining candidates is significantly less at each successive stage, the complexity of the calculations increases at almost the same rate, and thus the overall computational complexity of each pipeline stage of this detection algorithm stays somewhat constant (SOYATA et al., 2012).

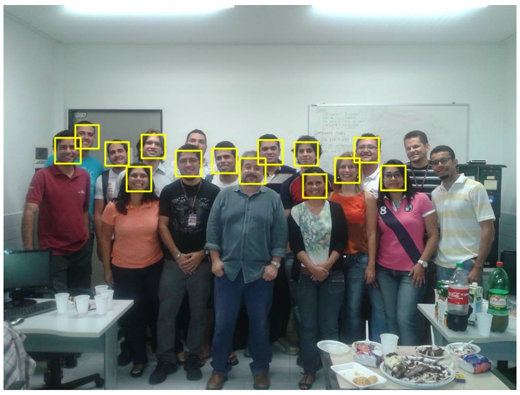


Figure 17 – Face Detection: This picture was processed by Haar Features and Classifiers technique.

The face recognition determines the match-likelihood of each face to a template element from a database. The potential faces determined in the previous face detection phase are then recognized. We have employed the widely accepted Eigenfaces approach (TURK; PENTLAND, 1991). This process extracts the relevant information in a face image, encodes it, and compares the encoded face image with a database of models, similarly encoded. A simple approach to extracting the information contained in an image of a face is to somehow capture the variation in a collection of face images (called training images) and use this information to encode and compare individual face images.

In mathematical terms, the objective of Eigenfaces approach is to find the principal components of the distribution of faces, i.e., the eigenvectors of the covariance matrix of the set of face images. Let $\Gamma = (\Gamma_1, \Gamma_2, ..., \Gamma_M)$ be the set of M face images used as a database of face models, where each Γ_i is a vector of N pixel values constituting a single face image. The "average face" $\Psi = \frac{1}{M}\sum_{i=1}^{M}\Gamma_i$ is computed from this database. The difference from each face image to the average face is $\Phi_i = \Gamma_i - \Psi$. The covariance matrix of the face images is computed as:

$$C = \frac{1}{M}\sum_{i=1}^{M}\Phi_i^T\Phi_i = AA^T, \tag{2.18}$$

where $A = [\phi_1, \phi_2, ..., \phi_n]$.

The eigenvectors of the covariance matrix C can be thought of as a set of features that together characterize the variation between face images. Each image location contributes more or less to each eigenvector, so that we can display the eigenvector as a sort of ghostly face that we call an eigenface. Some eigenfaces were generated using the image depicted in Figure 17, and the result is shown in Figure 18. It is important to highlight that these eigenfaces are just examples, because in practice hundreds of faces are used to build a training set. In the actual recognition phase, a new face undergoes a pattern matching to the eigenfaces in the training images database.

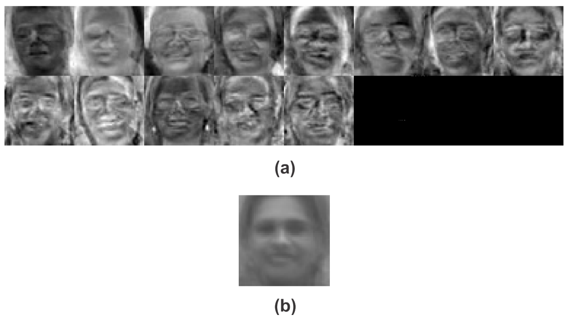


Figure 18 – (a) Thirteen eigenfaces calculated considering the faces of Figure 17; (b) The average face.

Each face image in the training set can be represented exactly in terms of a linear combination of the eigenfaces. The number of possible eigenfaces is equal to the number of face images in the training set. However the faces can also be approximated using only the “best" eigenfaces - those that have the largest eigenvalues, and which therefore account for the most variance within the set of face images. The primary reason for using fewer eigenfaces is computational efficiency.

Chapter 3 Related Work

“ *Weak people never give way when they ought to.* ”

Cardinal de Retz, *1613-1679*

The first papers in MCC had the objective of optimizing the offloading process itself. They focused on improving the offloading techniques by monitoring the mobile device, the application, and the network conditions. Many offloading frameworks have tackled mobile device constraints by offloading as much as possible heavy tasks obeying context factors (KRISTENSEN, 2010; CUERVO et al., 2010; KEMP et al., 2012; KOSTA et al., 2012; SOYATA et al., 2012; RAHIMI et al., 2012; CORDESCHI et al., 2015; CORDESCHI; AMENDOLA; BACCARELLI, 2015; BACCARELLI et al., 2016; CHANG et al., 2017). The number of works addressing context-aware offloading optimization is very large. Once the benefits of these frameworks became widely acknowledged by the research community, a new research trend appeared: MCC infrastructure planning (GABNER et al., 2011; PARK et al., 2011; PANDEY; NEPAL, 2012; OLIVEIRA et al., 2013; CHEN; WANG; PEDRAM, 2014). The scope of this field is to obtain an intelligent use of limited cloud resources by applying sophisticated system evaluation techniques.

Formal methods have been applied in diverse computer areas by evaluating system performance and assisting software engineers with architecture planning. Most of them have dedicated to evolve what it is called Software Performance Engineering (SPE) (HERZOG, 2001). SPE is a systematic, quantitative approach to constructing software systems that meet performance requirements, classified as real-time or responsive systems. SPE uses model predictions to evaluate trade-offs in software functions, hardware size, quality of results, and resource requirements. MCC has presented the need for applying SPE methods requiring to reach higher quality levels. For this reason, the current work focuses on MCC infrastructure planning applying SPE methods.

As observed in (SILVA et al., 2016), the main metrics used to evaluate the MCC using stochastic models are Reliability, Availability, Energy, and Execution Time. Reliability is defined as the probability that a device will perform its intended functions satisfactorily for a specified period of time under specified operating conditions (ARAUJO et al., 2014). Since the performance of a system usually depends on the performance of its components, the reliability of the whole system is a function of the reliability of its components (KUO; ZUO, 2003). Availability is defined as the probability that the system is operating properly at any given time (OLIVEIRA et al., 2013). Availability is the vital metric for nowadays systems; near 100 % availability is becoming mandatory for both users and service providers. High availability is an important feature for MCC applications given that the cloud–dependency can introduce unexpected failures (OLIVEIRA et al., 2013).

Execution Time and Energy are the most utilized metrics when evaluating the MCC systems. Computing speeds of mobile devices do not increase at the same rate as servers' performance (NIMMAGADDA et al., 2010). This is due to several constraints, including: Form Factor—as users want devices that are smaller and thinner and yet with more computational capability. Power consumption-insofar the current battery technology constrains the clock speed of processors. As the clock speed is increased, the power consumption is increased too. As a result, it is difficult to offer long battery lifetimes with high clock speeds (NIMMAGADDA et al., 2010). Therefore, energy and execution time will continue to be an MCC concern in long term.

Although Reliability, Availability, and Energy are very important metrics, our work focuses on Execution Time. For some computational tasks, it is possible to save battery when we reduce the processing time. Ding *et al.* (DING; YANG, 2018) investigates existing modeling and corresponding analyses methods for representing MCC systems based on formal methods. This paper provides an analysis and comparison of formal methods from the aspects of modeling capacity and related analysis techniques. Authors have argued that PN and its variations are appropriate modeling paradigms for representing mobile systems, taking into account their main characteristics. More specifically, the three basic characteristics of MCC systems considered by the authors are concurrency, interaction, and mobility.

Sousa *et al.* (SOUSA et al., 2014) have used stochastic modeling to represent systems deployed in the cloud. More specifically, the authors evaluated a Virtual Learning Environment - VLE deployed in a private cloud. Using the SPN-based modeling strategy proposed by the authors, it is possible to evaluate the performance and cost of system configurations. The strategy considers the arrival rate and evaluates the response time for each hardware and software configuration and related costs. This work does not consider the use of elastic mechanisms in the infrastructure to support an increase in demand.

Silva *et al.* (SILVA et al., 2018) propose an SPN-based modeling strategy to represent method call executions of mobile cloud systems. The approach enables a designer to plan and optimize MCC environments in which SPNs represent the system behavior and estimate the execution time of parallelizable applications. The approach provides graphs depicting Throughput, MTTE, and CDFs. Authors evaluated a mobile cloud facial recognition system and an image processing Android application for color reduction. However, neither networking requirements nor data traffic generated by the MCC application was considered in the modeling strategy proposed by authors. In addition, the authors did not consider on the cloud side arrival rates, scaling policies, allocation of simultaneous jobs in each virtual machine nor their related costs. The work considered that each virtual machine can only process one user request at a time.

Elasticity in multi-tier cloud applications was analyzed by some studies (see (HAN et al., 2014; AL-DHURAIBI et al., 2018; HOROVITZ; ARIAN, 2018; AL-FAIFI et al., 2018; DUPONT et al., 2015; ASLANPOUR; GHOBAEI-ARANI; TOOSI, 2017)).

Dupont *et al.* (DUPONT et al., 2015) evaluated elasticity strategies in the cloud computing. The work considered the time required to start elastic resources in the cloud. It is important because when the time to launch resources is known, it is possible to define more effective scaling strategies. What makes it possible for the system to not react to unexpected workload spikes. Authors also discussed the waste of the partial usage of contracted resources due to the billing cycles' granularity of existing pricing models. For this, authors argued that the software layer can take part in the elasticity process as the overhead of software reconfigurations can be usually considered negligible if compared to infrastructure one. This approach allowed redefining the elasticity configurations in order to provide sufficient computing resources considering the demand.

Campos *et al.* (CAMPOS et al., 2015) evaluated the elasticity mechanism in a private cloud. Authors have used a CTMC to represent the autoscaling process and to evaluate its performance. The work considered the VM type, VM image size, and probability that the VM will be in the cache. The strategy proposed by authors allows checking the impact of every parameter on the system response time and pointing out effective ways for improvement of autoscaling performance. Authors have used a full factorial DoE to compute the effect, relevance, and interactions of the factors on the total time for instantiation. The strategies of this work may be used for tunning private cloud systems and speed up the time required to get a new VM instance running.

Monetary cost evaluation based on stochastic models may support companies in planning their applications on the public cloud. A very few papers have proposed strategies based on stochastic models in order to support the cost evaluation of resource consumption on public clouds. These works addressed the monetary cost of using VMs.

Fe *et al.* (FE et al., 2017) propose an SPN-based model to assist in video transcoding system planning on public clouds. The model proposed by authors takes as input the autoscaling configuration parameters and the time between user requests. Using their model it is possible to compute throughput, mean response time, and cost of the cloud computing infrastructure for supporting a system configuration. However, authors considered only the cost related to the use of VM instances. In addition, the model represents only one type of VM instance running on the infrastructure. On the other hand, our SPN-based modeling strategy considers the use of a large number of VM instance types running on the same infrastructure. Using different types of VM instances in the same infrastructure may deliver further cost savings for cloud customers. Besides the cost for using instances, our strategy also considers the data traffic generated in a given period and its related costs. Data traffic plays an important role in a cost evaluation process for some types of applications. Our strategy supports analysts to perform more accurate cost analyzes for a set of system configurations that may be deployed in a public cloud. Using it, it is possible to represent more accurately real-world applications.

Ribas *et al.* (RIBAS et al., 2015b; RIBAS et al., 2015a) proposed a Coloured Petri Net (CPN)-based model that represents the use of public clouds spot instances pricing scheme in order to save costs. This work focused on modeling and reducing cost of elasticity of cloud services. The proposed model models the use of on-demand instances in the autoscaling process. Using this model it is possible to calculate the monthly cost for using a set of instances and saving offers by using Spot instances (SPIs). The model considers a set of reserved instances that will remain always up and a set of autoscaled instances to handle the increase in demand. Authors proposed a set of autoscaling policies that may offer cost savings when using cloud resources. Authors identified that SPIs could help reduce cost when compared to on-demand and reserved instances in an auto-scaling process.

Guerfel *et al.* (GUERFEL; SBAÏ; AYED, 2018) proposed a CPN-based model to support the modeling of elasticity strategy. The objective of the approach is to make it possible to find the most cost-effective elasticity strategy considering SLA constraints. For that aim, this work considers two factors in order to guide the definition of elasticity strategies. The proposed approach considers thresholds and the cost of each option when applying elasticity actions. More specifically, authors have considered the maximum and minimum number of user demands that each service can hold and the cost gained when applying the elasticity strategy. However, CPN tool does not allow us to check specific properties and result can be obtained only through simulation.

Silva *et al.* (SILVA et al., 2015) presented an approach to represent Graphic Processing Unit (GPU) parallel processes deployed in a public cloud by using SPNs. Using this approach, it is possible to simulate GPU executions and compute and plot cumulative distribution functions (CDFs). The modeling strategy allows to calculate the probabilities of satisfying user requirements when using GPUs with different number of cores. This work conclude that, based on the evaluated workload, AWS's customers may reduce their infrastructure costs by opting for less powerful GPUs instances, while still satisfying application's requirements in terms of execution time.

Some works have evaluated facial recognition systems in the context of mobile cloud computing (CIDON et al., 2011; CUERVO et al., 2010; KEMP et al., 2012; KOVACHEV; YU; KLAMMA, 2012; KWON; TILEVICH, 2012; SOYATA et al., 2012; HUANG; WANG; NIYATO, 2012; WANG; DONYANAVARD; CHENG, 2012; ZHANG et al., 2012).

According to Dey *et al.* (DEY et al., 2013), an efficient scheduling algorithm in mobile cloud must consider simultaneously both, communication and computation requirements. He also concludes with experiments that there is a tradeoff between satisfying response time for different user requests and maximizing system capacity. They tried to solve this problem by ranking the current capacity of heterogeneous access networks (like macrocell, microcell, carrier WiFi, public WiFi, etc.) and different clouds. In other words, for each network and cloud it assigns a utilization percentage. The problem with this strategy is the low level of resources granularity, not considering the machines as a resource unit, which could lead to a more efficient scheduling.

Evolving this approach, Soyata *et al.* (SOYATA et al., 2012) have implemented a software called MOCHA to improve mobile cloud face recognition simulating public clouds and a cloudlet. A cloudlet is a resource-rich computer or cluster of computers with fast Internet and available for use by nearby mobile devices (SATYANARAYANAN et al., 2009). MOCHA has demonstrated that a cloudlet as a unique server enhances face recognition. It redistributes the load to remote machines and ranking them by their round-trip time (RTT). Although MOCHA has presented an efficient behaviour, some improvements are still feasible.

Silva *et al.* (SILVA et al., 2017) have implemented an approach called SmartRank to improve the performance of mobile cloud face recognition. SmartRank is a scheduling approach that perform load partitioning and offloading. It has a scheduler algorithm that take into account multiple metrics and assign weights to them. SmartRank intends to minimize the response time of mobile applications by using cloud computing with heterogeneous communication latencies and compute power in terms of CPU and memory. Authors have applied the approach for suuport a face recognition process based on cloudlet federation and resource ranking based on balanced metrics. The approach considers as metrics the CPU utilization, round-trip time (RTT), processing time, and transmission time. The smart scheduling algorithm ranks cloud servers and distributes pictures of human faces among them. All VMs running the facial recognition system, and a dynamic distribution of faces occurs based on the VMs' ranking. Authors have used DoE and analytical modeling through a CTMC for performance evaluate purposes.

Although many studies focus on the optimization of the offloading process, in the recent literature, few studies have addressed stochastic performance modeling in the context of MCC infrastructure planning. In addition, the tradeoff between performance, data traffic and use of VM instances — and their related costs — for offloading on public clouds have been ignored by these studies. Table 4 presents the main works related to our proposal. Unlike these works, we offer an approach to support at design time the evaluation of performance, data traffic, use of VM instances, scaling policies, and related costs. It may be adapted and used in conjunction with a context-aware offloading optimization approach to providing on-the-fly performance evaluations. Mainly, our work advances the related works in the following aspects: (i) an SPN-based modeling strategy that represents applications installed on user devices and the use and sharing of the BW; (ii) an SPN-based modeling strategy that represents heterogeneous and elastic MCC infrastructures on public clouds that considers both the use of RIs and ODIs as well as data traffic; (iii) and cost models supported by the proposed stochastic models that consider data traffic and use of VMs.

Table 4 – Related Work Comparison

Related Work	**Data Traffic**	**Elasticity**	**Face Recognition**	**Financial Cost**	**Infrastructure Planning**	**Use of VMs**	**Models**	**Offloading Planning**	**Performance Evaluation**	**Public Cloud**
(SOUSA et al., 2014)				x	x		x		x	
(CAMPOS et al., 2015)		x					x			
(DUPONT et al., 2015)		x		x		x			x	x
(RIBAS et al., 2015b)		x		x	x	x	x			x
(SILVA et al., 2015)				x	x	x	x	x	x	x
(FE et al., 2017)		x		x	x	x	x		x	x
(SILVA et al., 2017)			x		x		x	x	x	
(DING; YANG, 2018)							x	x	x	
(GUERFEL; SBAÏ; AYED, 2018)		x				x	x		x	
(SILVA et al., 2018)			x		x	x	x	x	x	
Our Work	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**	**x**

Chapter 4 Methodology

“ *Experience shows that, if one foresees from far away the designs to be undertaken, one can act with speed when the moment comes to execute them.* ”

Cardinal Richelieu, *1585-1642*

In this chapter, we present two methodologies for supporting MCC service providers when designing their MCC services. The first methodology aims to support performance prediction of MCC applications installed on user devices. Making it possible to choose suitable offloading scenarios. The second one aims to support the planning of MCC systems in public clouds. As we will see, some steps are common between the two methodologies. Besides that, we present concepts, resources, tools, techniques, and methods for supporting the evaluation processes. Section 4.1 describes in detail all the steps of the methodology for evaluating MCC applications' offloading. Section 4.2 describes in detail all the steps of the methodology for planning MCC systems in public clouds.

4.1 Methodology for Evaluating MCC Applications Offloading

In this section, we present the methodology for supporting performance evaluation of MCC applications installed on mobile devices in order to find the most suitable scenario for offloading data and code. The methodology comprises a set of well-defined steps for supporting the evaluation process. The steps of the methodology are as follows: *understanding the MCC application*, *parameters definition*, *metrics definition*, *stochastic modeling*, *model validation*, *model-experiment refinement*, *scenario generation*, *model generation*, and *metrics evaluation.*

4.1.1 Steps of the Methodology

In this subsection, we present detailed information about the process of evaluating MCC applications installed on user devices. This process supports the decision-making process related to the offloading scenario to be adopted. Figure 19 depicts the MCC perspective considered by this methodology. Using the methodology, service providers are able to decide "*where*" to execute the application's workload. Figure 20 depicts the steps that developers must follow in order to evaluate their MCC applications. *"Offloading Scenario Evaluation Process"* is a subprocess that is considered after the modeling strategy is validated.

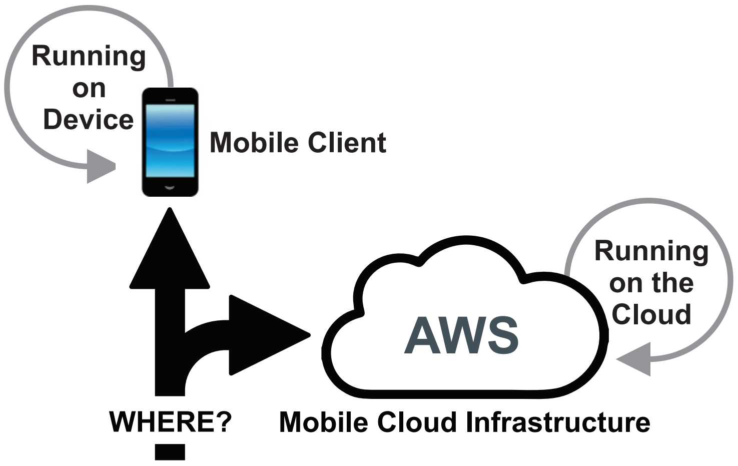


Figure 19 – Decisions About "*Where*" to Process the Application's Workload

Understanding the MCC Application: Developers should understand the mobile application itself, its requirements, and how users can interact with it when designing MCC applications. Understanding the application also includes understanding the source code to find out how the workload can be divided and distributed. Developers should identify more intensive components in processing and understand how they interact with each other. Some applications have performance requirements defined in SLAs. Thus, a developer needs to identify which parts are best suited for remote processing in order to achieve the desired performance levels.

Let us now describe the performance evaluation of MCC applications executing on mobile devices. Developers need to evaluate the performance of each part of the application that is suitable to be processed in the cloud. To do this, it is necessary to perform experiments in order to collect the processing time. Developers must instrument the source code of their applications to register the time required to process some tasks. In addition to processing time, developers may consider other metrics, such as CPU, memory, and energy consumption. Below we describe how developers may evaluate power consumption and CPU and memory usage on mobile devices.

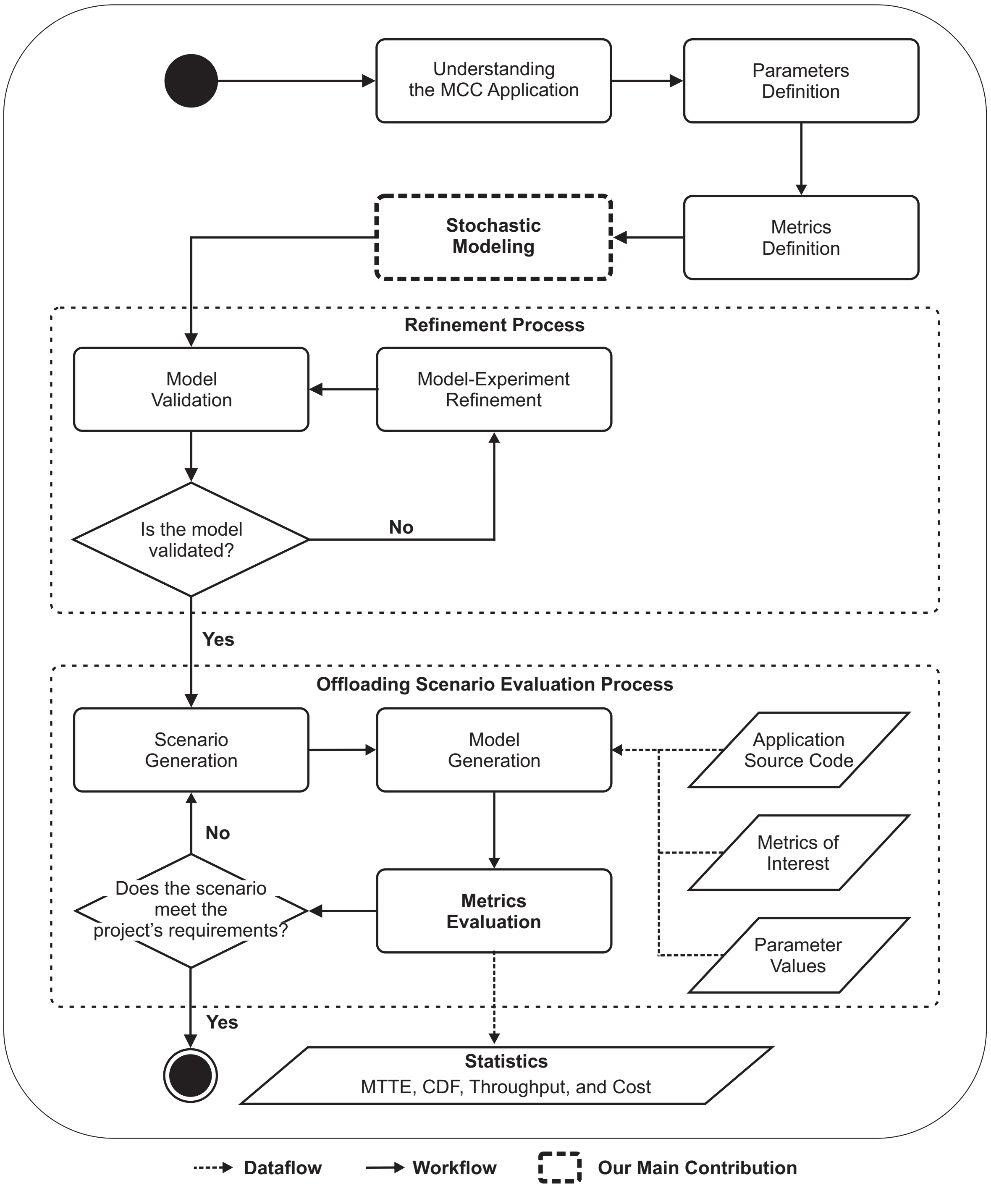


Figure 20 – Steps of the Methodology for Evaluating MCC Applications' Offloading

Power Consumption Evaluation. Regarding power consumption, there exist hardware profiles that allow an user to record the total energy consumed of a device, such as Watts Up (HIRST et al., 2013). There are also some software-level energy profilers that record the energy consumption per application, such as eDoctor (MA et al., 2013), and PowerTutor (ZHANG et al., 2010; POWERTUTOR, 2018). PowerTutor is a mobile application that monitors the power consumed by some components of the device, such as CPU and display, and by mobile apps running on the device. It provides the developer with a log file containing detailed data about the monitoring session and allows it to analyze the impact of each offloading scenario on energy efficiency. Figure 21 demonstrates an example of power consumption measured by PowerTutor while the mobile device was running a specific workload.

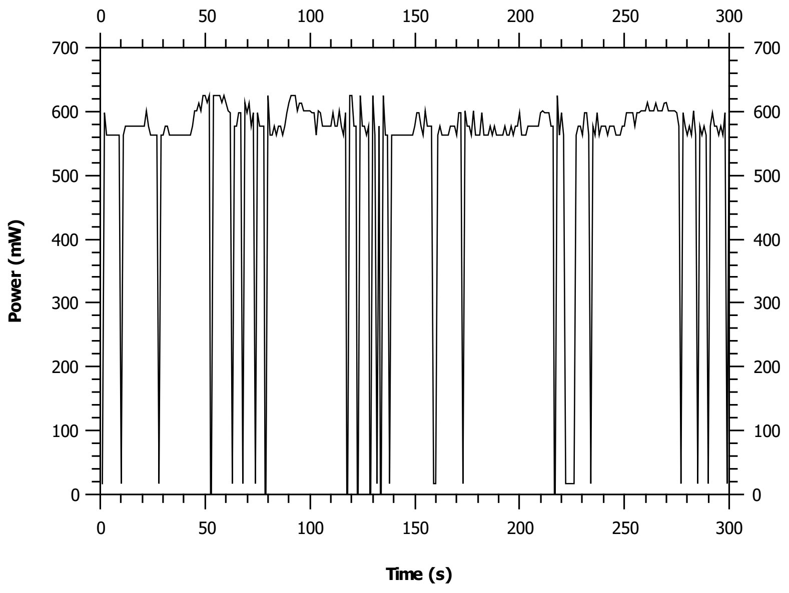


Figure 21 – Mobile Device Energy Profiling

CPU and Memory Consumption Evaluation. On an Android system, some statistics about CPU load and other system resources are available in the files in the */proc* directory. Developers may need to read these files to get the metrics they want when developing applications for this operating system. Directory */proc* is an interface to access data in the kernel of the operating system. The */proc* interface provides detailed information about processes and some system resources. For example, when reading the */proc/stat* file, an analyst may monitor the CPU of the mobile device when executing a workload and find out the time spent in the CPU and percentage of CPU usage. Developers can identify the CPU usage executing different types of workloads. In addition, by reading the */proc/meminfo* file, a developer get statistics related to memory usage, such as total and free memory. Another option for an Android application is to use the *java.lang.Runtime* class to collect some memory information. Figures 22 and 23 depict CPU and memory usage by considering a face recognition workload running on four different facial databases.

Parameters Definition: In this step, we define the parameters that will support the modeling process. Each parameter is a characteristic of the mobile application

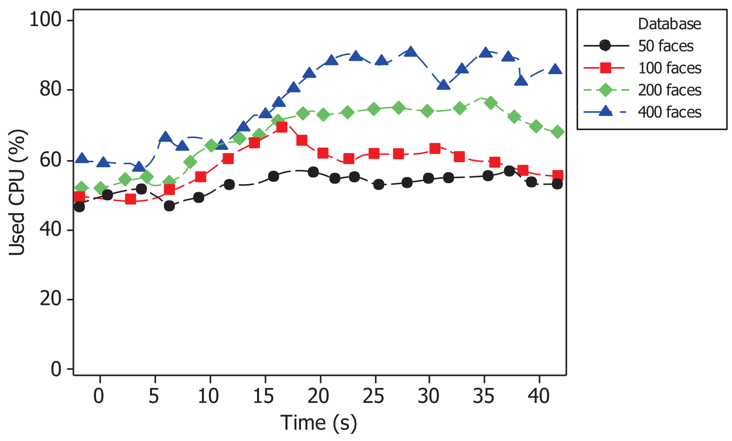


Figure 22 – Mobile Device CPU Profiling

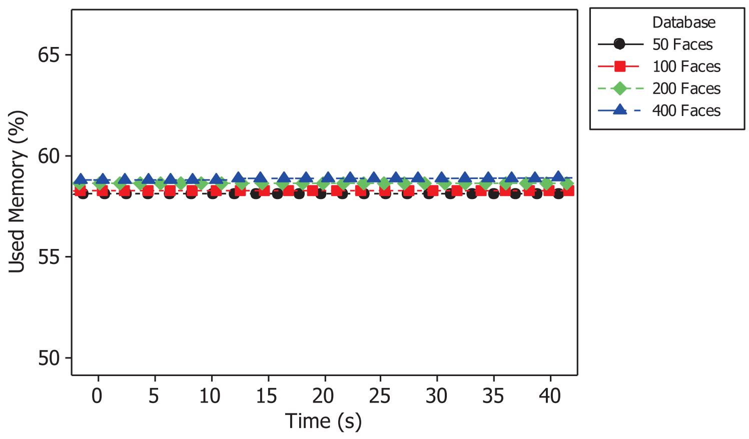


Figure 23 – Mobile Device Memory Profiling

or its environment that affects its execution. This work considers as parameters of an MCC application running on mobile devices the bandwidth available for supporting offloading operations, local and remotely processing times, data traffic and application source code itself.

Developers may use DoE technique to generate scenarios to execute the application's workload considering the parameters defined here (JAIN, 1990; MONTGOMERY, 2017). DoE makes it possible to evaluate the state space of possible scenarios considering the input parameters and, based on this evaluation process, an appropriate scenario may be found. Analysts may choose partial, fractional factorial, or full factorial designs to evaluate their systems. Most of the parameters defined in this chapter fall into two categories which are measurement and configuration parameters.

Measurement parameters represent the time spent to perform some activity. Developers need to know the times required to perform some system activities and their probability distributions. These times define an empirical distribution. Analysts must perform some experiments in order to collect the measurement parameters values. Our modeling strategy represents these parameters as timed transitions.

On the other hand, some configuration parameters have an upper and lower limit with regard to them. They represent resources available in the cloud or in the mobile device. On the device side, this work considers bandwidth as the only configuration parameter. For example, the bandwidth may assume a minimum and maximum value, and performance may be evaluated by varying this parameter.

Metrics Definition: Metrics definition is an important step in the methodology. The metrics that are chosen in this step support developers on the decision-making process to choose an appropriate scenario for offloading the mobile app's workload. There exists a large number of metrics that developers may use to support the offloading decision process. This work considers performance and costs metrics. More specifically, the performance metrics we consider on mobile devices are mean time to execute (MTTE), throughput (TP), and cumulative distribution function (CDF).

Stochastic Modeling: Our SPN-based modeling strategies can represent the mobile device and the cloud processing the MCC workload, and the communication process between both sides. Using our models, it is possible to estimate the performance of the MCC application. The mobile device modeling strategy can represent the application's source code and the device executing the app. In addition to representing it by sending the workload for remote processing through the offloading technique and receiving results. For this purpose, in order to estimate the communication time our strategy considers the bandwidth allocated for offloading operations, as well as the volume of data that the app needs to transmit.

Parameters and metrics defined in the last steps determine how the models will be generated. Our mobile device modeling strategy represents the only configuration parameter (i.e. bandwidth) as logic expressions for estimating transitions delays. The structure of the model depends on the input parameters and it impacts on the definitions of metrics. For example, the modeling strategy represents each part of the application with its own set of places, arcs, tokens, and transitions in the model. The definition for calculating metrics depend on the generated models since parameters values can change the model structure. In other words, parameters define the model structure and the model structure defines how desired metrics are obtained.

Model Validation: Analysts should validate their models to make sure the metrics obtained by them represent the results that the actual system presents by having the same parameters and processing the same workload. Obviously, since we are evaluating a stochastic system, the metric obtained from the model may differ from the metric of the actual system within a predefined limit. Taking this into account, model validation considers a confidence interval regarding the metric evaluated.

There exist some statistical methods that may be used to validate stochastic models. In this work, we consider a non-parametric method called Bootstrap (GONZALEZ-RODRIGUEZ; COLUBI; GIL, 2012; GINE; ZINN, 1990). The first step in validating a model is the definition of a system configuration that both the model and actual system receive. After that, the analyst runs a specific workload both on the model and actual system, and during this process, the desired metrics are collected. An analyst may consider that a model represents an actual system when the results presented by both sides are within the confidence interval.

Methods for Supporting Model Evaluations. Once the analyst generates the models, it is necessary to choose the appropriate method to evaluate them and get the desired metrics. When using SPN or CTMC models, it is possible to evaluate them considering a transient or stationary perspective. An analyst must calculate time-dependent metrics using transient evaluation. On the other hand, for non-time dependent metrics, a stationary analysis obtains the metrics in a steady state. In this work, we have considered three performance metrics evaluated through stationary analysis, which are throughput (TP), mean time to execute (MTTE), and mean response time (MRT). On the other hand, cumulative distribution function (CDF) is a transient metric and it depends on a specific time parameter t defined by the evaluator.

As we pointed out in section 2.3.4, there exists two ways to compute metrics. Analysts can evaluate their models through numerical analysis or simulation. Before that, the analyst needs to configure the parameters in the model in order to calculate the desired metrics. The numerical analysis of an SPN model corresponds to the analysis of the underlying CTMC regarding to its state space. Numerical analysis provides more accurate results (TRIVEDI, 2001; BOLCH et al., 2006; TUFFIN; HIREL; TRIVEDI, 2007). On the other hand, there may be some problems when using numerical analysis. The two main problems regarding numerical evaluations are state space explosions and use of non-exponential times in any transition (TUFFIN; HIREL; TRIVEDI, 2007; TRIVEDI, 2001; BOLCH et al., 2006). State space explosions may occur when the model represents a large number of parameters and each of them can have a large number of possible values. Another problem is the memory available for supporting the software that performs metrics computation. A stochastic evaluation software, such as Mercury, generates the CTMC relative to the SPN state space and maintains the whole CTMC in the system memory throughout the computation. This may cause memory failures considering models with large state spaces. Among other factors, an analyst should use simulation technique when the time required to numerically evaluate a model is prohibitive (TRIVEDI, 2001). It is important to highlight that simulation computes metrics within a specific confidence interval. Thus, numerical analysis is the only recommended method for problems that require an exact solution.

Figure 24 depicts the decision process regarding the definition of the most appropriate method for metrics calculation. An analyst may use moment matching technique in order to represent a non-exponential distribution in which it approaches some poly-exponential distribution. Based on the mean and standard deviation of a non-exponential distribution, it is possible to evaluate the possibility of using a poly-exponential distribution such as Hyper-exponential, Erlang or Hypo-exponential (see Section 2.3.5).

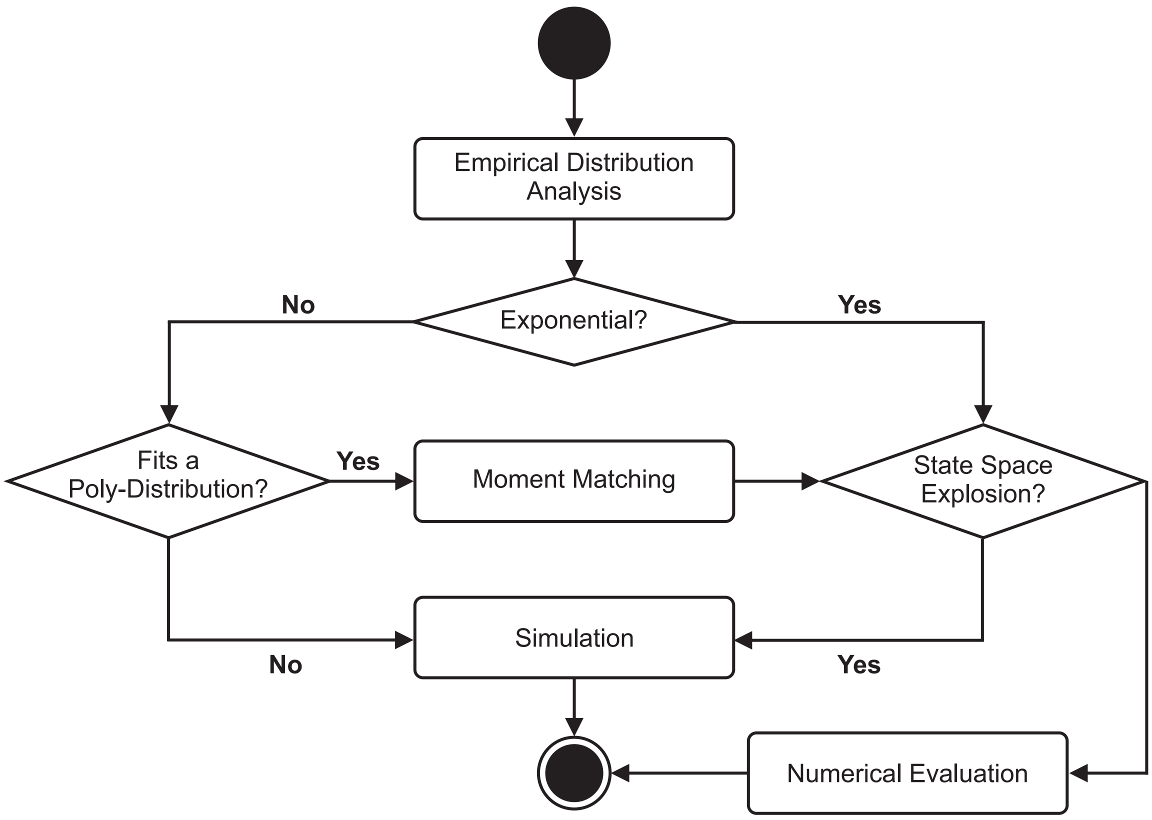


Figure 24 – Decision Process for Metrics Evaluation

Model-Experiment Refinement: When metric values differ, it means that it is necessary to refine the model to more accurately represent some system activities or, in some cases, refine the experiment run on the actual system in order to collect more detailed data. After that, the experiment is executed again and the metric values are compared. This process is repeated until the evaluated metric is statistically equal, that is, within the confidence interval defined by the analyst. Validating a model allows an analyst to check the behavior of complex scenarios by using it. This provides confidence in the values obtained from them.

Scenario Generation: The next steps of our methodology describe the activities to find an appropriate scenario for offloading data and code. They characterize the

Offloading Scenario Evaluation Process. The purpose of this process is to find out an offloading scenario that meets the project's requirements. An offloading scenario defines where each part of the mobile application should be executed. Parameter variations characterize this process. In the current step, developers define the minimum and maximum values that the only configuration parameter can take. That is, the minimum and maximum actual bandwidth allocated for data transmission operations. Thus, bandwidth varies from the minimum to the maximum value and the evaluator may obtain the impact of this variation on the performance metrics, considering each offloading scenario. A set of parameters and their values characterize a scenario and each scenario impacts on how the analyst should refine the models for evaluating metrics. The next step receives as input the scenario selected in the current step for supporting the modeling refinement.

Model Generation: It is a difficult task to evaluate all possible offloading scenarios through experiments. Considering this, the efforts to find out an appropriate set of parameters values may be high, and an analyst may spend a lot of time and money in this process. To avoid this, the process of evaluating scenarios allows evaluating, in a short period of time, a myriad of scenarios using only stochastic models. Once the modeling strategy is statistically validated, models can be refined to represent a large number of scenarios. This step receives as input the desired metrics and a set of parameters that represent a scenario to be evaluated in order to generate the model.

Metrics Evaluation: In this last step, an analyst analyzes the metrics provided by the model considering the current set of parameters values selected in the step *"Scenario Generation"*. Service providers may have statistics related to execution time and throughput. These results guide them in planning how to configure the offloading. If the metrics values do not meet the project's requirements, a new offloading scenario must be generated; after that, a new refined model is generated and metrics are evaluated again.

4.2 Methodology for Planning MCC Systems in Public Clouds

In this section, we present the methodology for supporting performance evaluation of MCC systems to be deployed in public clouds in order to find the most suitable configuration. The methodology comprises a set of well-defined steps for supporting the evaluation process. The steps of the methodology are as follows: *understanding the MCC system*, *parameters definition*, *metrics definition*, *stochastic modeling*, *model validation*, *model-experiment refinement*, *configuration generation*, *model generation*, and *metrics evaluation.*

4.2.1 Steps of the Methodology

In this subsection, we present detailed information about the process of supporting the planning of MCC systems in public clouds, in order to support the decision-making process related to the deployment configuration to be adopted. Figure 25 depicts the steps that developers must follow in order to evaluate their MCC systems. *"Configuration Evaluation Process"* is a subprocess that is considered after the modeling strategy is validated.

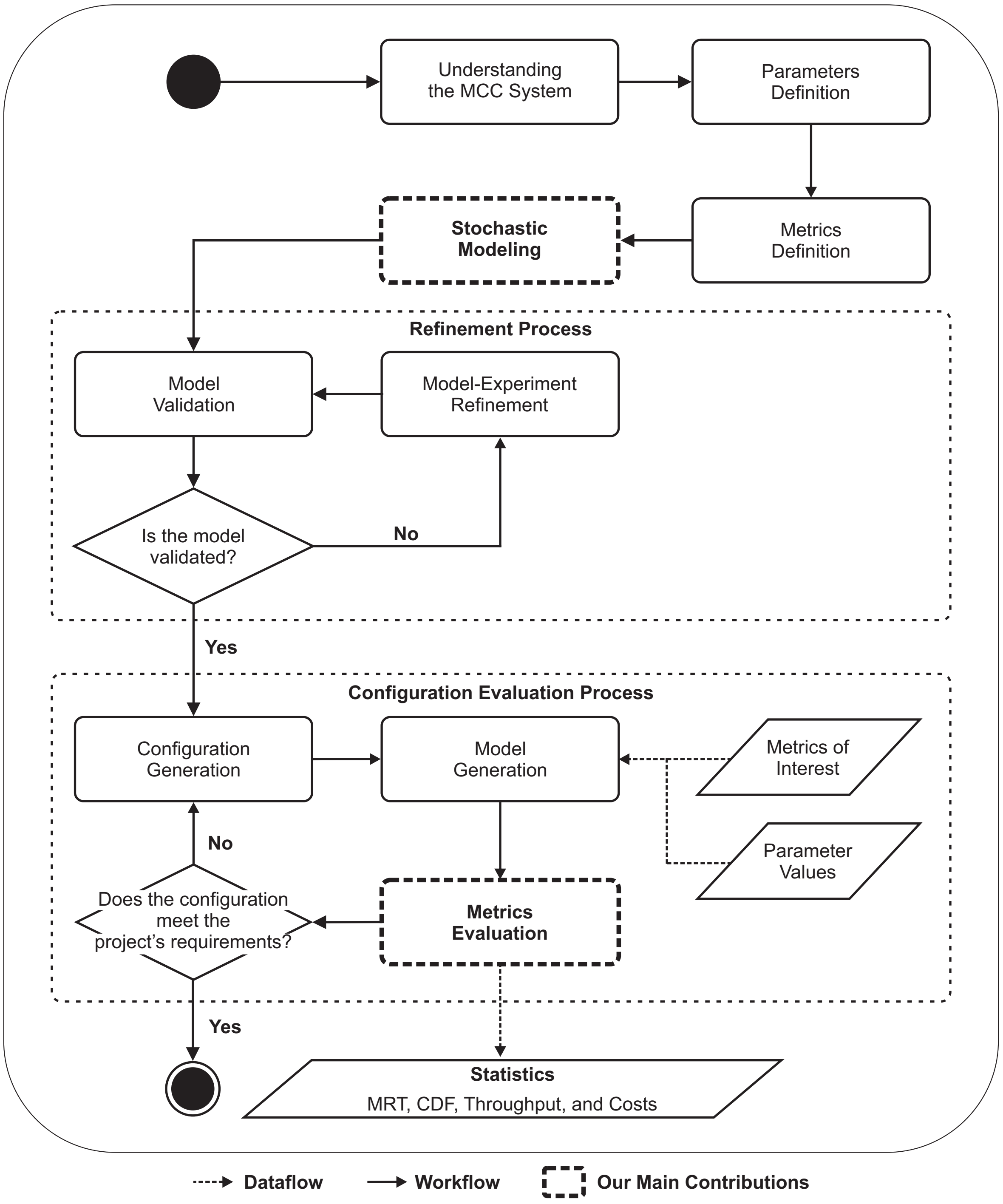


Figure 25 – Steps of the Methodology for Planning MCC Systems in Public Clouds

Understanding the MCC System: Offloading planning may be a difficult task when considering demands generated by mobile users and the parts of the MCC application that are suitable for remote processing. Developers need to analyze resource consumption in the cloud, taking into account the performance requirements and user demand for each offloading scenario. The purpose of offloading planning is to meet the performance expectations defined in SLAs while minimizing resource consumption in the cloud.

The demand for some types of applications may vary and the system needs to maintain its performance even when there is an increase in demand. There must be a set of remote resources to support the normal demand and it is possible to define some thresholds to scale the system out/in. The workload defines how an application scales in the cloud, and the thresholds are specific to each application. Each workload is more intensive in the use of some computational resource. For example, video transcoding and face recognition processing is more CPU intensive (FE et al., 2017; SILVA; MACIEL; MATOS, 2015). Perhaps for these workloads the level of CPU utilization on a machine may set the scaling thresholds. On the other hand, other categories of applications can be more memory intensive, and monitoring memory consumption in VM instances may define when and how the application scales. And, that is why developers need to understand the system workload so that thresholds must be set properly. Developers may set thresholds based on various criteria such as CPU usage, memory, I/O operations, queue size (GALANTE; BONA, 2012). Some public clouds provide APIs for supporting the definition of customized scaling policies. This work considers a customized reactive policy based on the queue size and available processing capacity for request processing.

By combining different VM instance types, simultaneous jobs per instance and scaling thresholds, it is possible to offer different response times for each offloading scenario. However, each of these variables affects resource consumption on the cloud side as well as the cost that the MCC service provider pays to an IaaS provider. The response time of the system deployed in the cloud affects the execution time of the mobile app running on the user device. Sometimes, the service provider needs to allocate the maximum number of user requests in a VM instance to save money. It is necessary to evaluate how computing resource consumption in the VM instance occurs in order to set the maximum number of simultaneous requests to be processed by it. To do this, the analyst needs to monitor the system and collect some metrics related to resources consumption.

This step finishes with the analysis of all collected data. Developers at the end of this step need to know the mobile application itself, its demand, and which parts are best suited to process them in the cloud. Developers have to know how they can split their applications and the possible scenarios to execute the system. Evaluating the workload running in the cloud and on the mobile device supports the definition of some parameters. For example, when evaluating the cloud side, it is necessary to know the impact on the service time to simultaneously process a different number of requests in the VM instances. More detailed analysis in the system logs may provide important information for supporting the definition of the system parameters.

Remote Resource Consumption Evaluation. Now let us describe the performance evaluation of the system in the cloud. We have used JMeter (JMETER, 2018; HALILI, 2008) to generate external requests. JMeter is an open-source tool specialized in load and functional tests. The goal of using JMeter is to simulate requests made by mobile users. Developers may set some request rate and JMeter generates requests accordingly. In addition, we have used Nigel's Monitor (NMon) software to monitor system resources (NMON, 2018). NMon is a computing resource monitoring tool for AIX and Linux operating systems. NMon can monitor a large number of computational resources and provides detailed reports related to the collected data. Developers may find out the most intensive features in each VM instance type in the infrastructure by analyzing the NMon logs. Each instance might handle more than one request at a time. In such a case, the developer may increase the number of concurrent requests on each instance in each experiment that she runs to discover the application thresholds in relation to resource consumption and processing time. The number of concurrent requests affects response time and, as a consequence, it impacts on the total time to process the mobile application.

Example of Deployment Configurations for an MCC System. There may be a large number of offloading scenarios depending on the number of parts the developers split their MCC applications. In the same way, each scenario can have a large number of configurations in the cloud for supporting offloading. These configurations correspond to the number of VM instances available to process external requests. Table 5 demonstrates an example of offloading scenarios for an MCC application, and Table 6 demonstrates configurations that a developer could adopt for deploying the MCC system in a cloud considering the scenario #1 depicted in Table 5. As we can see, there may be a large number of configurations for deploying an MCC system taking into account each offloading scenario.

Parameters Definition: This work considers as parameters of an MCC system to be deployed in a public cloud its capacity in terms of the number of requests that may be there at a time, arrival rates, number and types of VM instances, number of simultaneous requests that each instance can process, maximum number of on-

Table 5 – Example of Offloading Scenarios

Possibility	***m1()***	***m2()***
Scenario #1	cloud	cloud
Scenario #2	mobile	mobile
Scenario #3	mobile	cloud
Scenario #4	cloud	mobile

Table 6 – Example of Deployment Configurations for an MCC System

Configuration	**Instance Type 1**	**Instance Type 2**
	t2.micro	**t2.small**
#1	1	3
#2	3	4
#3	3	2
#4	2	2
#5	4	1
#6	3	3

demand instances (ODIs) that the system may use, time spent to launch ODIs, stepsizes and thresholds for scaling the system.

Analysts must perform experiments to collect the values of the measurement parameters. These parameter values depend on the workload running on the system and the state of the cloud provider. Some software for load testing and resource consumption monitoring in VM instance support these evaluations, such as JMeter (JMETER, 2018; HALILI, 2008) and NMon (NMON, 2018) respectively.

On the other hand, configuration parameters represent resources available in the cloud. This work considers as configuration parameters in the cloud the system capacity, number and types of VM instances, maximum number of ODIs that the system may use, stepsizes and thresholds for scaling the system in/out. The value of some parameters must be carefully selected. For example, the thresholds for scaling the system in need to be smaller than the thresholds for scaling the system out. Otherwise, the MCC system will never terminate unused ODIs running in the infrastructure. As we can deduce, in addition to affect the modeling process, each parameter also impacts performance and costs.

Metrics Definition: The metrics that are chosen in this step support MCC service providers on the decision-making process to choose an appropriate configuration for deploying their systems in the cloud. We consider performance and cost metrics. The performance metrics are mean response time (MRT), TP, and CDF. Cost metrics represent the cost of using resources in the cloud. We consider the cost of using VM instances and data traffic. The cost of using instances corresponds to the cost of using RIs and ODIs. The cost of data traffic depends on how much data the MCC system sends to mobile users.

Stochastic Modeling: SPN makes easy the representation and evaluation of queuing systems (SAHNER, 1996; TRIVEDI, 2001; JOHN, 2006; GERMAN, 2000). It is important to highlight that an analyst may use our cloud modeling strategy to represent the system running in a public or private cloud. However, in order to support resource consumption estimates in a highly distributed scalable environment, we consider the MCC system deployed in a public cloud.

Parameters and metrics defined in the last steps determine how the models will be generated. Our modeling strategy represents the configuration parameters as places, tokens, enabling functions, arcs multiplicity, or logic expressions for estimating transitions delays. It represents each set of VM instances with its own set of places, arcs, tokens, and transitions in the model. In addition, each group of elastic instances also has its own set of SPN components.

An analyst using SPN-based modeling may define some logic in the model regarding the system behavior. That is, in addition to the inherently stochastic behavior of the model in relation to transitions delays, priorities and weights; logical expressions defined as enabling functions, guard expressions or arc multiplicities also affect the model behavior. We have used the software Mercury (SILVA et al., 2015) to support us during the definitions and validations phases of our modeling strategies. Mercury is an integrated environment for supporting performance and dependability metrics evaluation of general systems and it allows users to define their models graphically. When defining an SPN model, Mercury allows users to see the tokens flow, supporting accurate analysis and avoiding errors in the modelling definition. However, analysts may use others software other than Mercury to support their evaluations such as TimetNet (GERMAN et al., 1995) or SHARPE (TRIVEDI; SAHNER, 2009).

Model Validation: We may use JMeter to generate workload in the actual system deployed in a public cloud in order to accurately obtain the desired metrics. The obtained metrics support the model validation process. For further information about this process, see the step *Model Validation* in section 4.1.

Model-Experiment Refinement: For sake of conciseness, see section 4.1. The same information available in the step *Model-Experiment Refinement* of the first methodology also applies here.

Configuration Generation: The next steps of our methodology characterize the *Configuration Evaluation Process*. The purpose of this process is to find out a configuration for deploying the MCC system in the public cloud that meets performance within the resource consumption levels expected by the MCC provider. In the current step, developers define the value that each configuration parameter can take. For example, the number of RIs and ODIs, or the stepsize associated with an instance type for each scale-out request. We consider the use of DOE technique adopting a full factorial planning. That is, we evaluate all possible configurations within the state space of the evaluated parameters. The individual variation of each parameter can provide an estimate of the impact of the parameter on the evaluated metric. The next step receives as input the configuration generated in the current step for supporting the modeling refinement.

Model Generation: This step receives as input the metrics of interest and a set of parameters that represent the evaluated configuration in order to generate the SPN model. Once the analyst generates the model, metrics may be evaluated to support the decision-making process.

Metrics Evaluation: In this last step, an analyst analyzes the metrics provided by the model. The performance metrics evaluated on the cloud side are MRT, CDFs, and throughput. Here, we also consider resource consumption in the cloud. MCC service providers may have statistics related to execution time, resource consumptions, and costs when resolving the stochastic models. These results guide them in planning how to deploy their MCC systems in the cloud. If the obtained metrics do not meet the performance and/or resource consumption level expected by the service provider, a new configuration must be generated considering the state space of the possible parameter values. After that, a new refined model is generated and metrics are evaluated again.

Chapter 5 Remote MCC Architecture

“*One should not be too straightforward. Go and see the forest. The straight trees are cut down, the crooked ones are left standing.*”

Kautilya, Indian philosopher, *third century B.C.*

A basic MCC architecture deployed in a public cloud for supporting offloading generated by mobile users needs to receive the requests, process them, and lastly send the results to the mobile devices. Figure 26 depicts the MCC architecture we consider in this work. Basically, the architecture consists of one front-end instance and a set of RIs and ODIs for service processing. The front-end is a RI and it performs some tasks. As we may presume, the front-end communicates with the outside world and receives requests sent by mobile users. RIs compose the set of VM instances that are always available for service processing and they handle the normal workload of the system. On the other hand, the ODIs compose the group of autoscaled instances that the MCC system may request to the cloud when there is an increase in the expected number of request in it. More specifically, the system requests ODIs to the cloud manager when it reaches any threshold for scaling out. The cloud charges for the use of RIs per year and for the use of ODIs per hour of use. An MCC service provider may use ODIs to allow their systems to handle transient increases in user demand to comply with performance requirements and by paying for the period of time the VM instance was in use. Some works have adopted a similar architecture in their evaluations (YANG et al., 2015; LIN et al., 2013; FE et al., 2017). As we may presume, this architecture can support a large number of application types.

Our architecture can represent a heterogeneous environment in relation to the types of VM instances running on it. As mentioned above, the architecture may comprise a set of RIs and ODIs. Each type of VM instance defines a group of service instances. It means that each group of service instances must consist of RIs and ODIs of the same type. The number of instances in each group may change considering the number of requests in the system.

Figure 26 demonstrates an infrastructure with two groups of service instances depicted with gray circles. **Group 1** comprises instances of type *t2.small* and **Group 2** comprises intances of type *t2.medium*. Each instance running on the system has a maximum number of parallel requests that it can process at the same time. This maximum number of parallel requests assigned to each instance defines the processing capacity of a instance type. Each group has its own processing capacity in relation to the number of requests it can process per unit of time. The sum of the processing capacity of each VM instance in a group defines the capacity of this group.

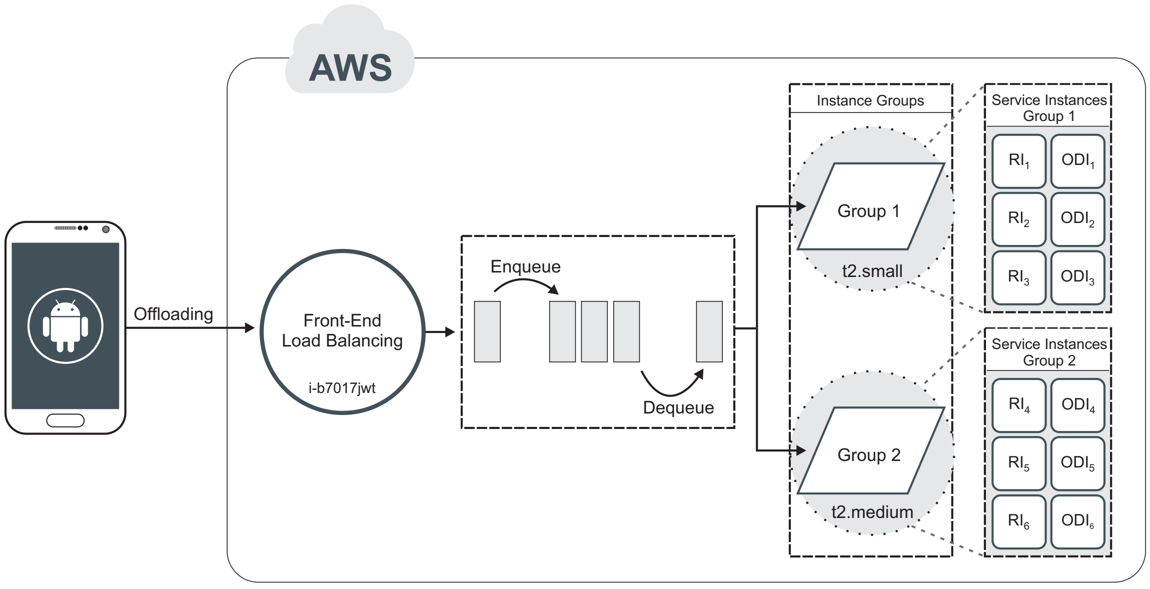


Figure 26 – Remote Architecture for Deploying an MCC System in a Public Cloud

IaaS public cloud providers provide virtual machines with different computing power. Table 7 describes the types of VM instances we consider in this work. As we can see, each type offers different processing capabilities, such as the number of vCPUs, memory, storage, network capacity, among others. The more powerful the VM instance, the higher the price charged by the provider for it. The provider may charge different prices for an type of instance. More specifically, contracting modalities provided by the cloud provider determines the price that customers pay. AWS provides families of different instances that are appropriate for multiple sets of workloads based on computing, memory, network, and storage requirements. For example, some families provide VM instances with a large amount of memory, while others provide instances with graphics processing capabilities (i.e. instances with GPU capabilities). We have considered in this dissertation VM instances of the *t2 family*. The *t2 instance family* is appropriate for most application workloads. However, the architecture and strategies proposed here may be applied considering any instance families.

Our architecture comprises a set of VM instances to handle a specific MCC workload. However, in a real-world context, a mobile application may offload a variety of workload types for remote processing. In this case, there may be a specific architecture to handle

Table 7 – EC2 Instances (AWS, 2018b)

Model	vCPU	Memory (GiB)	Reserved ($/year)	On-demand ($/hour)
t2.micro	1	1	$ 59.00	$ 0.0116
t2.small	1	2	$ 118.00	$ 0.023
t2.medium	2	4	$ 235.00	$ 0.0464
t2.large	2	8	$ 470.00	$ 0.0928

each workload. That is, each MCC function comprises its own set of RIs and ODIs. Figure 26 shows an MCC architecture for supporting only one offloadable function and Figure 27 demonstrates an MCC system deployed on the cloud for supporting two distinct offloadable functions. As we can see, each architecture comprises two types of instances running on it. However, in a real-world context, the number of instance types in the infrastructure can be large.

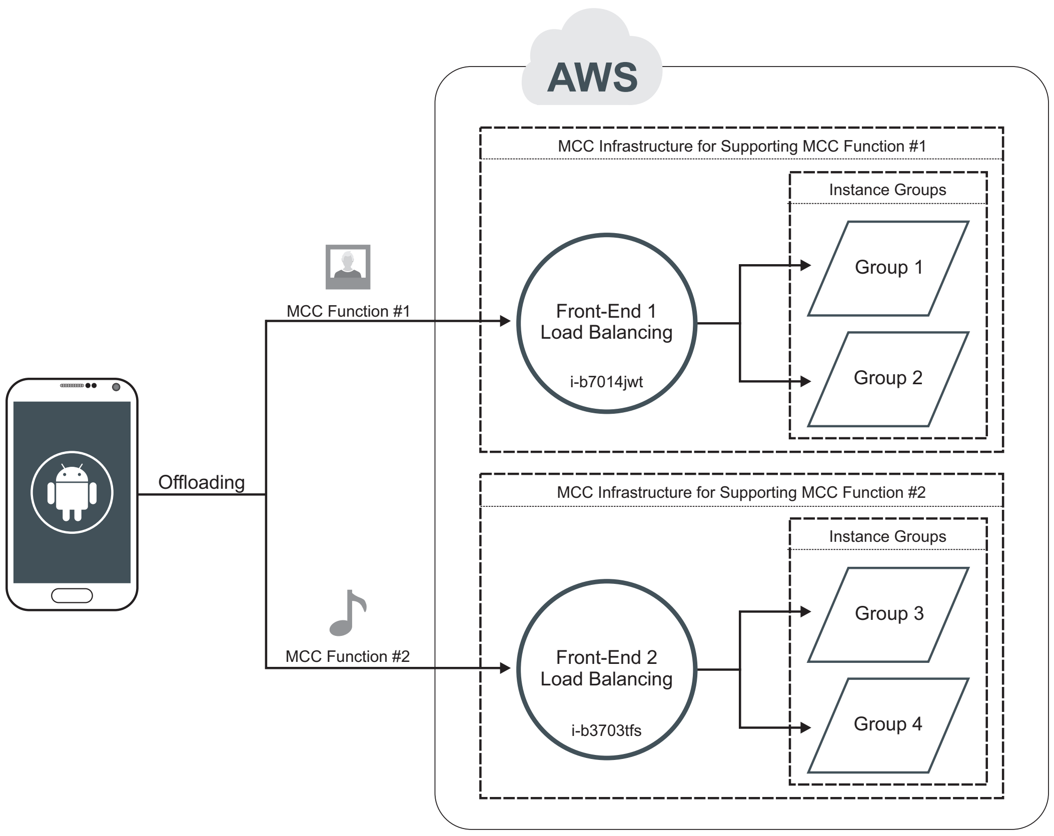

AWS
MCC Infrastructure for Supporting MCC Function #1
Instance Groups
Group 1
Group 2
Front-End 1 Load Balancing
i-b7014jwt
MCC Function #1
Offloading
MCC Infrastructure for Supporting MCC Function #2
Instance Groups
Group 3
Group 4
Front-End 2 Load Balancing
i-b3703tfs
MCC Function #2

Figure 27 – Remote Architecture for Supporting Two MCC Functions

In the following, we provide a more detailed description of our architecture. The front-end machine performs essential activities. It receives requests sent by mobile users, forwards them to receive processing, control the queue, performs load balancing, monitors the MCC system and predefined thresholds, and performs scale in/out requests to EC2. An application running on a mobile device offloads a task to the remote MCC infrastructure. The front-end receives the external request and forwards it to any service instance available on the system. Likewise, the front-end forwards requests to the queue when there is no processing capability available to manipulate them. The architecture adopts a first come first serve policy (FCFS) to manage requests in the queue. When a request that was receiving service on any VM instance leaves the system, another request waiting in the queue is forwarded to the available instance. Each MCC system may have a capacity related to the number of jobs in it. The purpose of the capacity parameter is to prevent the expected number of requests in the system grows and the response time exceeds the desired value. The system may reject requests when no buffers are available when they arrive. The front-end may perform scaling requests to the cloud manager when the system reaches predefined thresholds. We consider a horizontal scaling approach to scale the systems (AL-DHURAIBI et al., 2018). The system adds ODIs when it reaches any scaling out threshold. Likewise, the system removes ODIs running in any service group when it reaches a scaling in threshold. More specifically, the MCC system sends asynchronous scaling requests to EC2 and AWS scales the system accordingly (see Figure 28). Each request for scaling out may have a stepsize associated with it that defines the number of ODIs that the cloud needs to insert. Our work considers a reactive scaling approach (BISWAS et al., 2015; ASSUNCAO et al., 2016; LORIDO-BOTRAN; MIGUEL-ALONSO; LOZANO, 2014). The size of the queue, the processing capacity of the VM instances and the number of requests receiving service are the factors that our work adopts to define the scaling thresholds. Figure 29 shows an UML sequence diagram representing the system behavior when a request arrives. As we can see, in addition to forwarding the request, the front-end verifies whether the system has reached a threshold for scaling out. If so, front-end sends requests to EC2 for launching ODIs by considering predefined stepsizes.

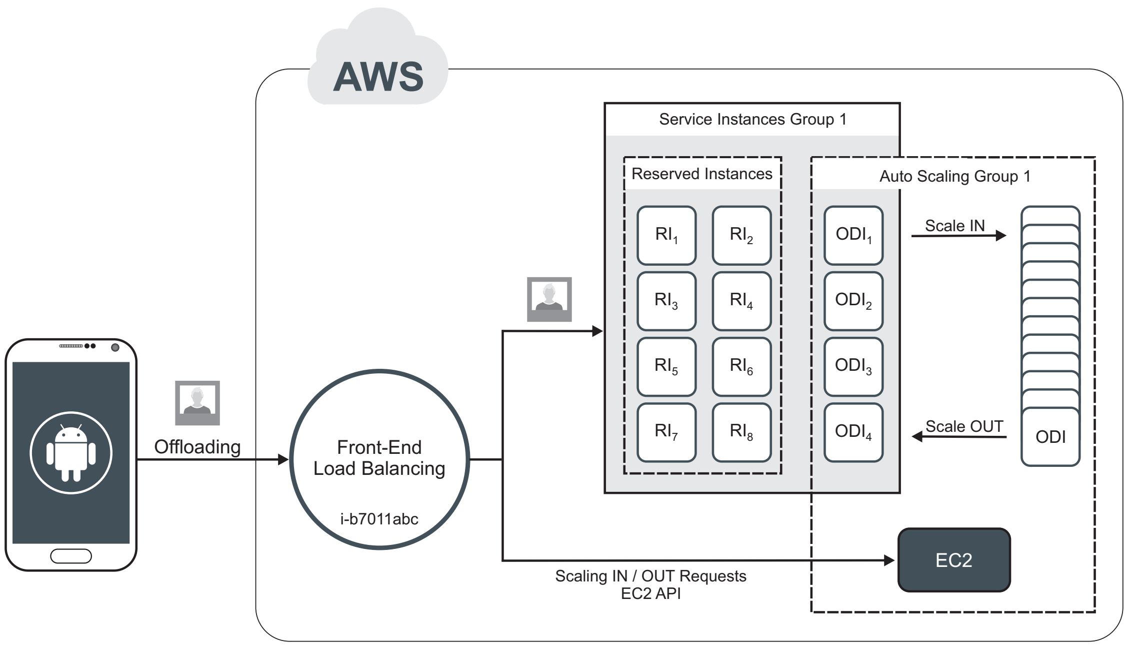


Figure 28 – Requests for Scaling Operations Sent by the Front-end to AWS EC2

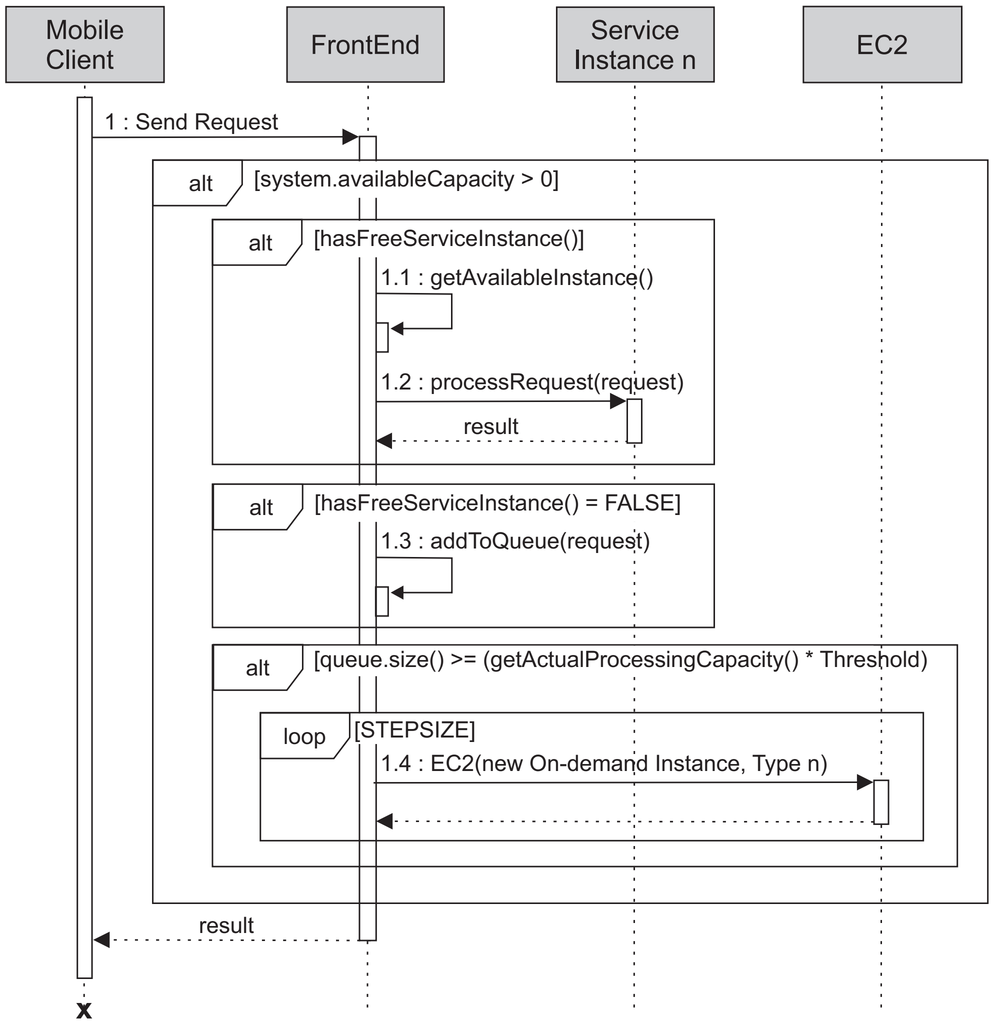

Mobile Client
FrontEnd
Service Instance n
EC2
1 : Send Request
alt
[system.availableCapacity > 0]
alt
[hasFreeServiceInstance()]
1.1 : getAvailableInstance()
1.2 : processRequest(request)
result
alt
[hasFreeServiceInstance() = FALSE]
1.3 : addToQueue(request)
alt
[queue.size() >= (getActualProcessingCapacity() * Threshold)
loop
[STEPSIZE]
1.4 : EC2(new On-demand Instance, Type n)
result

Figure 29 – Sequence Diagram Depicting the Front-end Behaviour When a Request Arrives in the System

Understanding and evaluating a system are key factors for deploying it in a cloud. The following we describe how analysts may deploy their MCC applications in our architecture. The first step in deploying an MCC application in the cloud is to understand its behavior. Analysts need to evaluate the resource consumption of their applications. Each application has its own patterns in resource consumptions. The main computational resources that most systems consume are CPU and memory. However, some applications may be more intensive in other resources such as network or storage. The main idea behind the study of this behavior is to define the system thresholds. Thresholds regarding the number of concurrent jobs that the system can process in a single VM instance and thresholds that define when to scale.

Analysts need to evaluate the performance of their applications on each type of instance they want to use in their infrastructure. The application processing a single request and a set of them at the same time. The analyst varies the number of simultaneous executions and evaluates the time to process each of them. It is important to understand the impact on the processing time of a single request, considering a different number of concurrent requests running on the evaluated VM instance. The processing time tends to increase as high the number of concurrent tasks in the same machine. This time tends to grow linearly for some applications, while for others it may have nonlinear growth. The number of concurrent tasks plays an important role in defining the performance of the system as well as the cost to maintain the infrastructure in the cloud. An analyst needs to consider the trade-off between performance and cost. The use of powerful VM instances to process a single request at a time may be disadvantageous from a financial point of view. On the other hand, allocating a large number of requests in one VM instance to process all of them at one time may decrease the total cost, but may not meet SLA performance requirements.

There exist tools that analysts can use to evaluate their applications. We have used JMeter and NMon software to support our evaluations (JMETER, 2018; NMON, 2018). An analyst sets a specific arrival rate and JMeter generates requests accordingly. JMeter generates requests that the front-end machine handle them and from there the system processes them. Making it possible to evaluate performance in the cloud. NMon is another tool we have used. NMon is a monitoring tool that collects a large number of operating system and computer resource metrics. NMon supports analysts to define thresholds related to the number of concurrent processing in each instance type. An analyst needs to execute a different number of request simultaneously and evaluate the resource consumptions for each execution as well as the processing time of a single request in each bulk requests execution. Some applications may have a limit regarding the number of concurrent executions. This limit defines when the system crashes for not having more resources available for supporting its execution or the processing time of a single request exceeds the minimum performance levels. The task of the analyst is to find out the limit in order to make it possible to define the number of concurrent tasks in each type of VM instance that allows to comply with the SLA, minimizing the consumption of resources. An analyst should consider during this evaluation all of instance types that will compose her infrastructure. Each instance type has its own limits regarding the number of concurrent processing in it.

Service providers using a public cloud may set autoscaling thresholds based on the resource consumption or the state of their applications. Amazon Cloud Watch provides several features for load balancing and scaling in/out an application based on predefined thresholds, resources monitoring in each VM instance and in system logs (AWS, 2018a). The cloud system monitors the customer's application during its execution, and when the application reaches any predefined thresholds, it performs some actions according to the threshold, such as inserting or removing VM instances. Some clouds provide their customers with tools that make it easy to configure these parameters. Users can set scaling thresholds based on computational resources such as CPU usage, memory, I/O operations, and more. Additionally, they can set thresholds based on the number of requests in the system and in the queue. By controlling the number of requests in the system, the queue size, and the VM instances for requests processing, it is possible to ensure the performance required by adding ODIs to handle increase in demand or remove them when it decreases. While these tools make it easy for users to define their autoscaling strategies, on the other hand, they do not offer flexibility to define custom policies. Some providers, such as AWS, offer APIs that allow their customers to scale their applications based on custom rules. AWS offers the EC2 API and this API makes it easy to define customized scaling rules. In this case, the customer's application monitors itself and performs actions based on predefined custom rules. The application scales itself through API calls when reaching any predefined threshold.

5.1 Parameters Definitions

We describe below the parameters for performance and resource consumption evaluations. We consider parameters related to the MCC application running on mobile devices and on a public cloud. These parameters provide input values to support evaluations considering our proposed approach.

5.1.1 Device Parameters

We describe below the parameters related to the MCC application running on mobile devices. These parameters support performance evaluation through our approach.

- **Source Code.** Analysts may use the source code itself to define where the system will process each class or method call. They may adopt another abstraction to support the decision-making process on where to process each part of their applications. For example, they may represent their systems as functions or modules and use the chosen abstraction to decide where to process the parts.

- **Offloading Scenario.** The offloading scenario defines where the mobile application processes its workload. The application processes the workload on the user device or can offload the workload or part of it to a remote MCC infrastructure.

- **Processing Time.** This parameter defines the average time to process a workload on the user device or in the cloud. When the application offloads a workload to the cloud, the response time of the cloud becomes the processing time during performance evaluation on the mobile device. An analyst can estimate the processing time on a mobile device, when the system transfers the workload to the cloud, with the response time of the remote infrastructure and the time to send and receive data. The communication time can be estimated with the actual bandwidth for the offloading operation and the amount of data transferred.

- **Bandwidth.** In this work, we consider that an MCC service provider deploys its MCC system in a public cloud and this requires an Internet connection to transfer data and code between the mobile application and the remote system. The bandwidth allocated for offloading operations becomes a critical aspect that affects the performance of the MCC application. The higher the actual bandwidth, the shorter the time to transfer a volume of data.

- **Data Volume.** This parameter defines the amount of data the system transfers in each offloading operation. The data volume includes data and codes that an application sends and receives to support this operation. An analyst can estimate the communication time having the actual bandwidth and the volume of data.

5.1.2 Cloud Parameters

We describe below the parameters related to the MCC system running in the cloud. These parameters support the evaluation of performance and resource consumption through our approach.

- **Arrival Rate.** The arrival rate is related to the offloading generated by users. This parameter corresponds to the interarrival time between requests arrival in the system.

- **System Capacity.** This parameter sets the maximum number of requests that can be in the system at a time. If the analyst increases the system capacity and there is no increase in the processing capacity, it means that the response time may increase. The parameter is also related to the probability that the system will reject external requests when there is no available buffer in the system.

- **Types of Instances.** The type of VM instance defines the number of concurrent requests that an instance may process per unit of time, and this number affects the processing time of a single request on the instance. An MCC system may use different types of instances in its infrastructure. In this case, each instance in the system is in its own group in relation to its types. The parameter impacts the time required to process requests as well as to start the autoscaled instances.

- **Reserved Instances (RIs).** This parameter defines the types and quantities of RIs that the MCC service provider contracts. These VM instances will always be available regardless of whether or not there is a workload on the system. Although reserved instance contracts are longer, they are cheaper. These instances will be used to meet normal system demand.

- **On-demand Instances (ODIs).** This parameter defines the number and types of ODIs that the MCC system may use to meet a transient increase in demand. By using ODIs, the MCC service provider only pays for the VM instances that it uses.
- **Processing Capacity.** This parameter defines the number of simultaneous requests that one VM instance can process at a time. The higher this number, the longer it takes to complete the processing of each request.
- **Time to Process Requests.** This parameter represents the time to process a single request on an instance, considering a predefined number of concurrent requests being processed at the same time. That is, this parameter represents the service time related to a request.
- **Autoscaling Groups.** ODIs allow the MCC service provider to pay for computing capacity during the period of use with no long-term commitments. Each instance type has its own auto-scaling instance group. The system adds or removes ODIs for request processing when it reaches any scaling out/in threshold.
- **Step Size.** The step size defines the number of ODIs to be launched in each scaling out request. Each instance type may have its own step size.
- **Scaling Thresholds.** This parameter defines when the system adds or removes ODIs for request processing. The queue size and processing capacity of the MCC infrastructure are the key parameters that will determine when to add or remove extra processing power to handle the increase in demand. When the number of requests in the queue increases and no increase occurs in computational resources, the response time may increase. In such a case, it is necessary to add extra computational capacity to reduce or maintain response time.
- **Time to Launch On-demand Instances.** This parameter represents the time the cloud spends to start and insert an ODI into a group of service instances after the MCC system executes a scale out request. The MCC system itself, the operating system, the instance type, and the state of the cloud provider impacts in this time. The cloud must load the MCC system in the operating system during the instance launching process. This time may have a large variation in each scale-out request.
- **Data Volume.** This parameter defines the data volume the MCC system transfers in each offloading operation. The data volume comprises data and codes that the MCC system sends and receives to support this operation. The volume of data to be transferred from the cloud to mobile devices determines the cost of data traffic over a given period for supporting an offloading scenario.

- **Costs.** The costs associated with each feature that the MCC system uses in the cloud. In our case, we have considered the cost of using VM instances and data traffic.

5.2 Performance Metrics in the Cloud

Some strategies proposed in this work aim to find a configuration for the remote MCC system so that SLA performance requirements are met whereas minimizing the use of resources in the public cloud. For that aim, this work considers the mean response time and throughput of the remote MCC system as cloud performance metrics.

5.2.1 Mean Response Time (MRT)

The MRT determines the time a user request spends on the remote MCC system. MRT generally increases as user requests in the system increase. Using Little law (LITTLE, 1961) it is possible to obtain MRT. Little law is a powerful tool for showing the performance of a queue system over time. This law relates the mean number of requests in the MCC system with the arrival rate (AR) and the average time a request spent in it (i.e. MRT), as demonstrated by Equation 5.1. By derivation, we have Equations 5.2 and 5.3. Equation 5.2 obtains MRT and Equation 5.3 obtains the arrival rate. Little law may be applied as long as the number of requests arriving at the system is equal to the one completing service. In other words, it means that the MCC system does not create new internal requests and no user request is lost inside it. Even in MCC systems that reject some user requests because there is no available capacity, or requests are lost due to problems in the mobile network, the Litle law may be used because, once the request enters the system, it is processed. In such a case, the analyst must adjust the arrival rate to remove the requests lost. The effective arrival rate is the arrival rate that represents only the requests that enter the system.

$$Requests = AR \times \ MRT \tag{5.1}$$

$$MRT = \frac{Requests}{AR} \tag{5.2}$$

$$AR = \frac{Requests}{MRT} \tag{5.3}$$

5.2.2 Throughput

Throughput corresponds to the rate at which the remote MCC system processes user requests. We may say that the throughput is the number of requests processed per unit of time. This rate generally increases as the number of incoming requests increases initially. However, throughput stops increasing after a certain number of arriving requests. The nominal capacity of an MCC system represents the maximum throughput when the system is under ideal workload (JAIN, 1990). In this work, the throughput of an MCC system in the cloud corresponds to the number of requests processed in a given period, considering all service instances in the system. Higher throughput is considered better, but the cost may be high when the system is on a public cloud. Equation 5.4 demonstrates how to obtain throughput.

$$TP = \frac{Requests}{Time} \tag{5.4}$$

Chapter 6 Modeling MCC Applications

“ *Space we can recover, time never.* ”

Napoleon Bonaparte, *1769-1821*

In this chapter, we present an SPN-based modeling strategy that represents MCC applications running on mobile devices. It may represent the structure of the application's source code. Representing the source code enables the software engineer to access a more accurate result. In addition, this modeling strategy represents the use and sharing of the network bandwidth (BW) available for supporting offloading operations as well as the effect of BW variation on the metrics. In this way, making possible to represent the communication time to transfer data and code.

Companies in some situations need to balance system performance, resource consumptions and financial costs to find the most appropriate strategy that meets the requirements of their projects. As the user base of an offloadable application grows, the higher may be the consumption of some remote resources. The amount of money that a company must pay to an IaaS cloud provider may grow significantly. Awareness of the related cost is a major element in the choice of appropriate strategies. Application's performance and data traffic are key elements in defining an offloading scenario.

Most public cloud providers (Amazon[1], Google[2], Microsoft[3]) charge their customers for data traffic. More specifically, by the outbound data traffic to the Internet. The proposed approach for estimating data traffic evaluates from mobile users' perspective. More precisely, this chapter seeks to answer the following questions:

1. How to calculate the MTTE, CDF and throughput of a set of method-calls — that may represent a system functionality — using SPNs?

[1] AWS: https://aws.amazon.com
[2] Google Cloud Platform: https://cloud.google.com
[3] Azure: https://azure.microsoft.com

2. How to estimate the impact of the available bandwidth and its variation on the throughput, MTTE, and CDF?

3. How to estimate the data volume that will be transferred during the offloading process of a set of method-calls for a given period of time over a public cloud?

4. How to estimate the monetary cost of transferring a data volume over a public cloud?

6.1 Networking Aspects

Network performance is a key factor that has a direct impact on the performance of the MCC application. However, it is a difficult task to estimate precisely at design time the network conditions in which applications will be used. As we have already mentioned, our approach is not a context-aware offloading approach. It aims to support developers at design time. Thus, aspects related to context-aware approaches as network congestion are not addressed in our approach. Our work considers the actual TCP throughput between the device and cloud for tasks offloading. Our approach considers the effect of network bandwidth on the applications' performance.

An application may have many users and the users' network has different conditions from each other. Therefore, developers may establish network requirements for their applications in which the expected and minimum bandwidths are defined. In addition, when planning their MCC applications, developers should consider that the expected bandwidth allocated for their applications may vary within a specified limit. For example, if the application detects that the available bandwidth is less than the lower-limit set value, then the processing is performed locally. Considering the actual TCP throughput and the number of bytes to be transferred it is possible to estimate the communication time (CT) for data transfer using Equation 6.1 (MATHIS et al., 1997; PADHYE et al., 1998). Varying the available bandwidth within the defined limits, it is possible to take a myriad of evaluations for supporting both the offloading decision process and MCC infrastructure planning.

$$CT = \frac{datasize}{BW} \tag{6.1}$$

Our solution may be adapted considering strategies proposed by other authors. When adapting our strategy to work in a real-time context, developers may implement an approach to estimate the actual bandwidth available during the application execution. For example, based on the actual available bandwidth estimated and the performance prediction performed using the strategy proposed in this work, mobile apps may decide how the processing will be executed. In this context, there are many works with the aim of evaluating MCC offloading traffic considering the networking aspect and resulting network-induced constraints (BACCARELLI et al., 2016; CORDESCHI et al., 2015; CORDESCHI; AMENDOLA; BACCARELLI, 2015; CHANG et al., 2017). Cordeschi *et al.* (CORDESCHI et al., 2015) (CORDESCHI; AMENDOLA; BACCARELLI, 2015) uses an optimization scheme to solve the well-known resource management problem. Authors proposed a reliable adaptive resource management controller for vehicular access networks in order to provide reliability guarantees to traffic considering the inherent mobility and fading induced changes. The strategy proposed by the authors scale energy and bandwidth consumptions with the dynamic demands where performance are enhanced through data traffic offloading to the local or remote cloud.

6.2 Execution Time (MTTE and CDF)

MTTE corresponds to the average time to finalize the processing of a set of method-calls. Figures 30 and 31 present an example of SPNs for computing MTTE. MTTE is the expected time to reach an absorbing state. A state of an SPN is absorbing whether it is impossible to leave it (i.e., *P(#FINISH = 1)*). It means a deadlock marking has been reached. MTTE is based on the probability that the processing of a system functionality has been completed. MTTE is the average time for a number of tokens to reach the place *FINISH* given they were in place *START* at time instant zero. SPN models can be evaluated either by numerical methods or by simulation (NELSON, 2013).

Figures 30 and 31 demonstrate a simple representation using SPN of one functionality with only one method-call. Let us first look at the SPN representation that corresponds to the local method-call (see Figure 30). The SPN model comprises three places and two transitions. The first transition (*trigger_time*) is immediate. It means that the transition has zero as its delay value. The second transition (*processing_time*) is a General Time High-level Transition. It represents the time to processing the respective method-call.

The SPN pattern that represents an offloadable method-call has two new places and two new transitions (see Figure 31). Transition *offloading_time* represents the time spent to execute offloading. Transition *receiving_time* represents the time spent to receive the result sent by the cloud. These transitions are depicted by a gray rectangle, and the model is later refined by assigning probabilistic distribution parameter values to respective transitions. If, on the one hand, such transition is refined by poly-exponential distributions (DESROCHERS; AL-JAAR; SOCIETY, 1995), (SOUSA et al., 2014), (SILVA et al., 2014), the SPN can be evaluated either by numerical analysis or by simulation. On the other hand, simulation should be carried out.

The models evolved by transformation of high-level transitions into exponentially distributed timed transitions. It allows assigning average delays to respective timed transitions. Such transformation of transitions and delays assignments allow the SPN models to be solved. From here, for simplicity, all timed transitions will have exponential enabling times. However, they may adopt other probabilistic distributions as well as deterministic values.

Moment matching (DESROCHERS; AL-JAAR; SOCIETY, 1995) could also be applied to obtain poly-exponential distributions (ARAUJO et al., 2011), (SILVA et al., 2014), (COSTA et al., 2015) (see Section 2.3.5). By adopting moment matching, the planner may estimate what exponential-based probability distribution best fits the mean. Additionally, moment matching generates more accurate models, which can still be numerically evaluated. If none poly-exponential distributions are adopted, simulations should also be adopted.

Such SPN patterns may originate other models to evidence data dependency between method-calls of any source code arrangement. The SPN modeling pattern evolved to calculate MTTE of distinct scenarios. Algorithms 1, 2 and 3 demonstrate three types of method-call combinations. Figures 33b, 33d, and 33f demonstrate how the original SPN model evolved to calculate MTTE of the three types of method-call combinations. The pattern embraces general features common in concurrent systems. The place *SYSTEM_INACTIVE* when having one token means that the system is idle. The timed transition *T0* receives the delay to start the processing of method-calls when there is a token in the place *SYSTEM_INACTIVE*. When there is no delay, *T0* becomes an immediate transition. In a real-world context, multiple combinations can be derived taking into account the application's method-calls and the modeling patterns presented in Figures 30 and 31.

Algorithm 1 Three Sequential Method-Calls

```
1: function $rootMethod(input1)$
2:     $result1 \leftarrow performTask1(input1)$          ▷ m_call_1
3:     $result2 \leftarrow performTask2(result1)$        ▷ m_call_2
4:     $result3 \leftarrow performTask3(result2)$        ▷ m_call_3
5:     return $result3$
6: end function
```

Algorithm 2 Two Sequential Method-Calls and One in Parallel.

```
1: function $rootMethod(input1, input2)$
2:     $result1 \leftarrow performTask1(input1)$          ▷ m_call_1
3:     $result2 \leftarrow performTask2(result1)$        ▷ m_call_2
4:     $result3 \leftarrow performTask3(input2)$         ▷ m_call_3
5:     return $result2, result3$
6: end function
```

The mean processing time and communication time of each evaluated method are the base for MTTE calculation. In the models presented, the *processing_time_m1* and *processing_time_m2* transitions receive the mean processing times of the methods *m_call_1* and *m_call_2*, respectively. If the application's method under analysis is an offloadable

Algorithm 3 Three Parallel Method-Calls

```
1: function $rootMethod(input1, input2, input3)$
2:     $result1 \leftarrow performTask1(input1)$        ▷ m_call_1
3:     $result2 \leftarrow performTask2(input2)$        ▷ m_call_2
4:     $result3 \leftarrow performTask3(input3)$        ▷ m_call_3
5:     return $result1, result2, result3$
6: end function
```

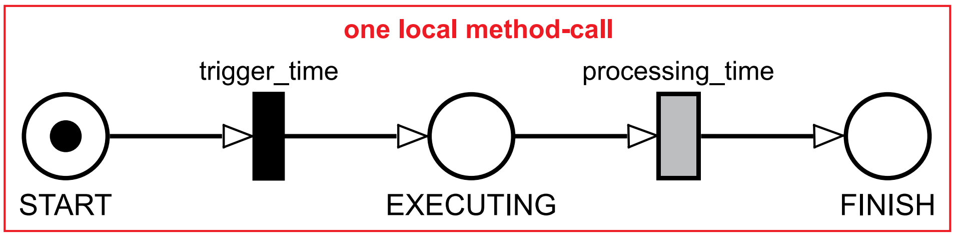


Figure 30 – Basic SPN Representation of One Application with Only One Local Method-Call Using Absorbing State

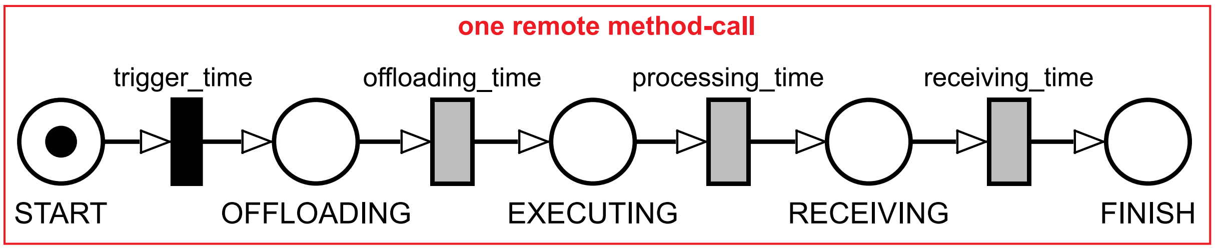


Figure 31 – Basic SPN Representation of One Application with Only One Offloadable Method-Call Using Absorbing State

method, it is necessary to obtain the number of bytes transferred to send tasks and to receive the remote results.

In this work, we consider that there is a specific bandwidth allocated to offloading operations, as well as to receive the remote results. More specifically, the developer should consider bandwidth variation for a more accurate estimate (see Section 6.1). Equations 6.2 and 6.3 consider the probability of there being tokens in the *offloading* place (variable O_{mj}) as well as in the *receiving* place (variable R_{mj}) for other method-calls other than mi, respectively. $BW_{offloading}$ represents the actual bandwidth allocated for tasks offloading – in bits/s. $BW_{receiving}$ represents the actual bandwidth allocated to receive the remote results – in bits/s. Thus, if other methods are using the allocated bandwidth, the bandwidth allocated for the operation (i.e. offloading or receiving) is divided among the methods that are using the network for the same operation.

Equations 6.4 and 6.5 calculate the communication time to transfer an amount of data taking into account the actual bandwidth allocated to the evaluated method mi. $datasize\_o_{mi}$ represents the amount of data transferred to offload the method mi – in bits. $datasize\_r_{mi}$ represents the amount of data received as the result of the remote processing of the method-call *mi* – in bits. Equations 6.4 and 6.5 are assigned as the mean delay value of transitions *offloading_time_mi* and *receiving_time_mi* of the method-call *mi*, respectively. In the evaluation process, developers must convert Equations 6.4 and 6.5 to the syntax of the stochastic evaluation program used.

$$BWO_{mi} = \left(\frac{1}{1+\sum_{j\neq i}^{n} P\left\{O_{mj} > 0\right\}}\right) \times \left(\frac{BW_{offloading}}{1000}\right) \tag{6.2}$$

$$BWR_{mi} = \left(\frac{1}{1+\sum_{j\neq i}^{n} P\left\{R_{mj} > 0\right\}}\right) \times \left(\frac{BW_{receiving}}{1000}\right) \tag{6.3}$$

$$offloading\_time_{mi} = \frac{datasize\_o_{mi}}{BWO_{mi}} \tag{6.4}$$

$$receiving\_time_{mi} = \frac{datasize\_r_{mi}}{BWR_{mi}} \tag{6.5}$$

After that, a transient analysis on the model obtains the MTTE. A tool such as Mercury, TimeNet, GreatSPN or SHARPE may execute this analysis (SILVA et al., 2015), (GERMAN et al., 1995), (GREATSPN, 2004), (TRIVEDI; SAHNER, 2009). When MTTE is obtained through numerical evaluation, these tools generate the state space of the SPN model and create the corresponding CTMC. Figure 32 presents a CTMC that represents the elapsed time to finish the processing of an offloadable method-call. The calculations described in the Section 2.3.2 are performed to obtain this transient metric.

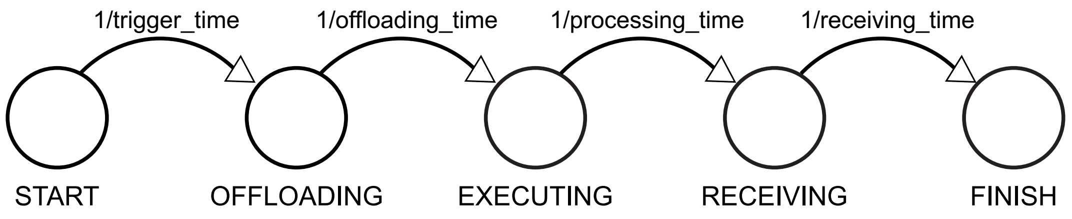
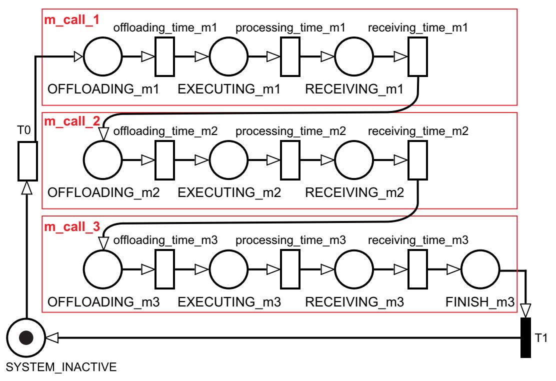


Figure 32 – CTMC of an Application with Only One Offloadable Method Call With Absorbing State

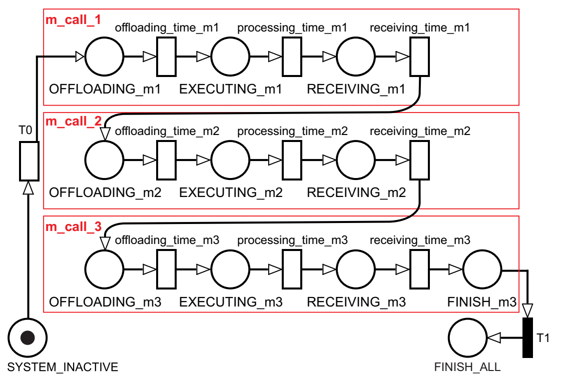
We have adopted SPN as modeling formalism due to its greater descriptive power in relation to Markov chains (MARSAN et al., 1994). Using SPN it is possible to make a high-level graphical representation of method-calls and their possible synchronizations. However, developers may model their systems using CTMC, but this approach may lead to a model with a large number of states. Thus, depending on the size of the system, it may become impracticable to directly model all method-calls, their synchronizations and transitions rates using CTMC.

6.2.1 Cumulative Distribution Functions (CDFs)

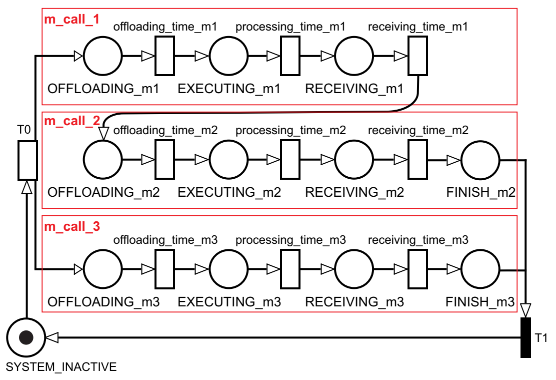
Application developers and service providers willing to plan and design an MCC environment should be aware at when their applications are more likely to finish execution.



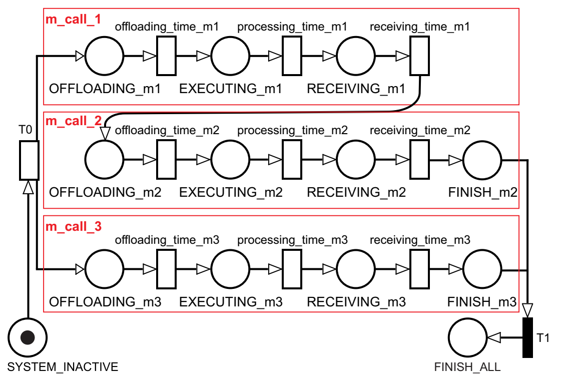
(a) SPN without Absorbing State Used to Calculate Throughput of the *Application_A* (Three Sequential Method-Calls.



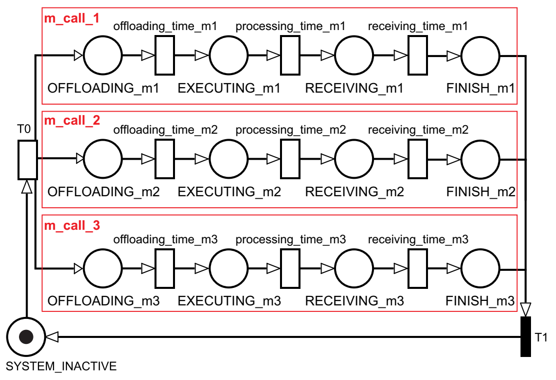
(b) SPN with Absorbing State Used to Calculate MTTE and CDF of the *Application_A* (Three Sequential Method-Calls).



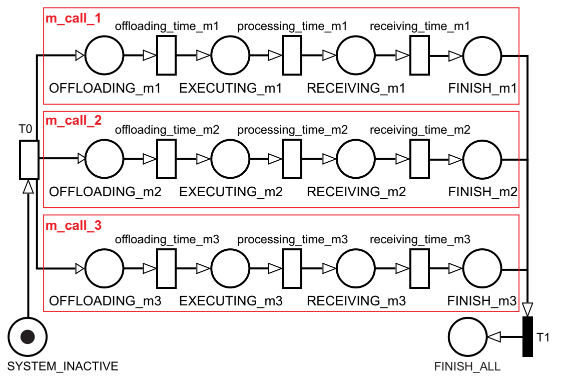
(c) SPN without Absorbing State Used to Calculate Throughput of the *Application_B* (Two Sequential Method-Calls and One in Parallel).



(d) SPN with Absorbing State Used to Calculate MTTE and CDF of the *Application_B* (Two Sequential Method-Calls and One in Parallel).



(e) SPN without Absorbing State Used to Calculate Throughput of the *Application_C* (Three Parallel Method-Calls).



(f) SPN with Absorbing State Used to Calculate MTTE and CDF of the *Application_C* (Three Parallel Method-Calls).

Figure 33 – SPNs Representing Offloadable Method-Calls

Cumulative Distribution Functions (CDFs) may indicate such a moment through the maximum probability of absorption. CDFs are associated with a specific probability distribution. In this work, the probability distribution is related to the probability of finishing the application execution within a specified time. It is obtained through transient evaluation generating probabilities with time tending to one value t. In other words, developers compute the probability to absorption in $[0, t)$, through transient evaluation, where $F(t)$ approaches 1. CDFs may also indicate the maximum probability of an application's processing to be completed within a given time interval. Regarding CDFs, non-negative random variables may have the probability distributions defined in terms of its probability density function (see Equation 6.6).

$$F(t) = \int_0^t f(x)dx \tag{6.6}$$

CDFs allows a broader cost-effective architectural simulation of mobile cloud systems. Analyzing the hypothetical example of CDF line plot in Figure 34, two types of interpretation may be traced:

- *Probability of Finishing Execution Before Time t*: Considering one specific execution time point t, the graph returns the probability $P(T < t)$ of finishing the execution before such time for each application. The probability of finishing the execution before 150 ms is equal to 66 % for *Application A*, 95 % for *Application B*, and 98 % for *Application C*.

- *Probability Interval*: The developer may obtain the probability of finishing the execution between a time interval $(t_1,\ t_2)$, which is calculated as $P(t_1 < T < t_2) = P(T < t_2) - P(T \leq t_1)$. Thus, we can calculate the probability of finishing the execution between 20 ms and 80 ms resulting in 9.2 % for *Application A*, 47 % for *Application B*, and 65 % for *Application C*. As expected, *Application C* has higher probability of satisfying the constraint.

  If we relax the timing constraints and increase the time interval between 100 ms and 200 ms, the probability of finishing the execution with *Application A* becomes higher. Specifically, for each application we obtain the following values: 68 % for *Application A*, 29 % for *Application B*, and 16 % for *Application C*.

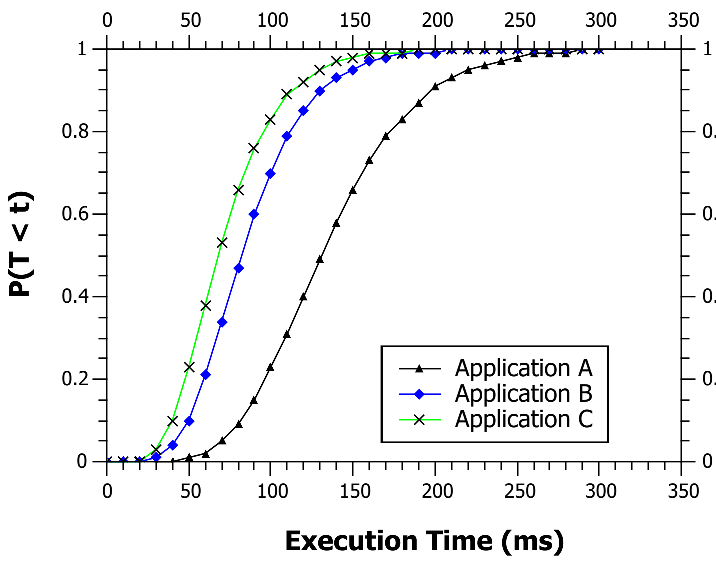


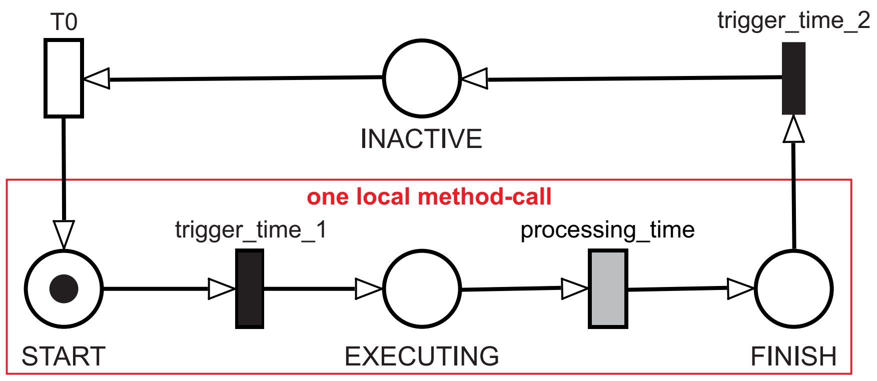
Figure 34 – Example of *CDF* based on SPN

6.3 Throughput

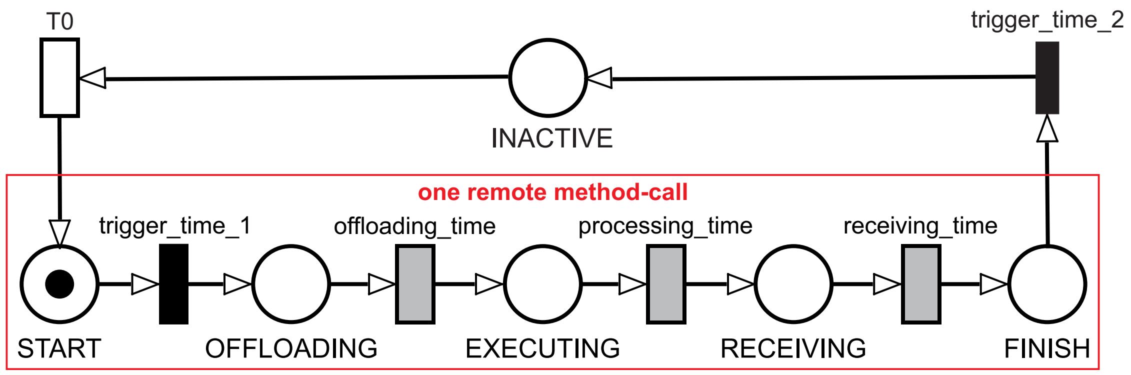
The throughput (TP) represents the number of executions per unit of time of a set of method calls. Tp is obtained by computing the expected value of tokens at a place, multiplied by the inverse of the transition delay (MACIEL et al., 2011). For that aim, the SPN model presented in Figures 30 and 31 evolved. Now, as illustrated in Figures 35a and 35b, the models need two transitions to allow them to return to the initial state when workload execution is complete. Such SPN pattern may be extended to evidence the method-calls data dependency of any application (see Figures 33a, 33c, and 33e).

The throughput may be calculated considering two possibilities: Single Server Semantics (SSS) and Infinite Server Semantics (ISS). In the SSS, the flow of tokens will occur in series, regardless of the degree of the transition activation. In the ISS, every set of tokens of the enabled transition is processed simultaneously. Equation 6.7 calculates throughput according to SSS, and Equation 6.8 to ISS. The variable i represents the weight of the arc that connects the place *INACTIVE* to the subsequent transition *T0*. The variable i may vary until N, where N is the highest enabling degree of the subsequent transition at the place marking $m(INACTIVE) = i$.

The throughput is defined by the delay between each request (transition *T0*) as well as the delays assigned for others transitions in the SPN model. The available bandwidth and the number of bytes to be transferred represent the transfer time. Thus, if there is a change in the actual bandwidth, there may be a change in the throughput. If the delay between requests is small (*T0*), the impact on the throughput may be high. On the other



(a) SPN without Absorbing State Used to Calculate Throughput of an Application With One Local Method-Call.



(b) SPN without Absorbing State Used to Calculate Throughput of an Application With One Offloadable Method-Call.

Figure 35 – SPN Representation of an Application with Only One Method-Call without Absorbing State.

hand, if the delay between requests is high, the impact on the throughput may be small.

$$Tp = P(m(INACTIVE) >= i) \times \frac{1}{Time} \tag{6.7}$$

$$Tp = \left(\sum_{i=1}^{N} P(m(INACTIVE) = i) \times i \right) \times \frac{1}{Time} \tag{6.8}$$

6.4 Costs for Data Transfer

Public clouds charge their customers for data transfers by considering “utilization ranges" as illustrated in Tables 8, 9 and 10 — corresponding to Amazon, Azure, and Oracle price tables[4], respectively. The price charged per gigabyte transferred decreases as the volume increases over a period. The providers mentioned above charge their customers for outgoing data traffic to the Internet. Therefore, developers should only consider the bytes transferred from the remote MCC infrastructure to mobile devices.

[4] The practiced values by the IaaS cloud providers may change over time.

Table 8 – Amazon EC2 Prices per Transferred Bytes (AWS, 2017)

Data Transfer OUT To Internet	**Price/GB**
First 10 TB / month	\$0.09
Next 40 TB / month	\$0.085
Next 100 TB / month	\$0.07
Next 350 TB / month	\$0.05

Table 9 – Oracle Cloud Prices per Transferred Bytes (ORACLE, 2017)

Data Transfer OUT To Internet	**Price/GB**
First GB / month	Free
Next 9.999 TB / month	\$0.12
Next 40 TB / month	\$0.09
Next 100 TB / month	\$0.07
Next 350 TB / month	\$0.05

Using the Equation 6.9 it is possible to obtain the total transferred bytes (TTB) for each evaluated offloading scenario. To obtain the TTB is necessary multiplying: (*i*) the throughput(*Tp*); (*ii*) the evaluation period in milliseconds (*Time*); (*iii*) the volume of data transferred in each request (*Bytes*); and finally (*iv*) the number of users (*Users*).

$$TTB = Tp \times Time \times Bytes \times Users \tag{6.9}$$

Now, using Equation 6.10 it is possible to obtain the financial cost. First, it is necessary to split the number of total bytes according to the utilization range of the price table used. After that, it is necessary multiplying the number of bytes consumed in each "utilization range" by its respective price. Companies may adjust the cost calculation according to the data traffic charges policy of their IaaS cloud provider.

$$Cost = \sum_{ur=1}^{n} TTB_{ur} \times PricePerGB_{ur} \tag{6.10}$$

Table 10 – Microsoft Azure Prices per Transferred Bytes (AZURE, 2017)

	Zone 1	Zone 2	Zone 3	DE (trustee)
Data Transfer OUT To Internet	**Price/GB**	**Price/GB**	**Price/GB**	**Price/GB**
First 5 GB / month	Free	Free	Free	Free
5 GB - 10 TB / month	\$0.087	\$0.138	\$0.181	\$0.100
Next 40 TB / month	\$0.083	\$0.135	\$0.175	\$0.095
Next 100 TB / month	\$0.070	\$0.130	\$0.170	\$0.080
Next 350 TB / month	\$0.050	\$0.120	\$0.160	\$0.057

6.5 Model Validation

Many aspects may interfere in the similarity between the model results and the reality, such as connection with bad quality. To reduce the influence of errors (e.g., noise) in the measuring process, a statistical technique called Bootstrap was utilized to validate the models (EFRON; TIBSHIRANI, 1993; SILVA et al., 2014).

We implement and analyze an image processing mobile application following the principles of method-call computation offloading (KOSTA et al., 2012). The implementation uses a simple client-server architecture with remote method invocation (RMI).

Application A resides on the mobile device (see Algorithms 4, 5, and 6). If the method-call is offloaded to a remote server (lines 2 to 4), it means that the mobile application makes image processing calls to the server by passing one input (original images). In this case, the app connects to one virtual machine and then calls the method *reduceColor* in the server side. Thereafter, the processed image returns to the user device.

Both client and server side adopt the Open Source Computer Vision Library (OpenCV) (OPENCV, 2018) and *JavaCV* (JAVACV, 2018b). We have implemented the computing vision example of Picture's Colour Reduction (JAVACV, 2018a). This example transforms images by decreasing the number of colors depending on the picture's size.

As infrastructure, the private cloud Eucalyptus 3.4.0.1 (NURMI et al., 2009) was used with two physical machines (one node and one controller). The physical machines have the following configuration: Intel Core i7-3770 3.4 GHz CPU, 4 GB of RAM DDR3, and 500 GB SATA HD. An Ethernet network is adopted to connect the physical servers through a single switch and one VM of type *m1.medium* (1 CPU, 512MB of RAM, and 10GB Disk). At the mobile device side, a Samsung Galaxy Note 4 was used running Android 5.1.1

Algorithm 4 Application A - Three Sequential Method-Calls

```
1: function $processImages(img1)$
2:     $result1 \leftarrow reduceColor(img1)$                ▷ m_call_1
3:     $result2 \leftarrow reduceColor(result1)$             ▷ m_call_2
4:     $result3 \leftarrow reduceColor(result2)$             ▷ m_call_3
5:     return $result3$
6: end function
```

Algorithm 5 Application A - Two Sequential Method-Calls and One in Parallel.

```
1: function $processImages(img1, img2)$
2:     $result1 \leftarrow reduceColor(img1)$                ▷ m_call_1
3:     $result2 \leftarrow reduceColor(result1)$             ▷ m_call_2
4:     $result3 \leftarrow reduceColor(img2)$                ▷ m_call_3
5:     return $result2, result3$
6: end function
```

Algorithm 6 Application A - Three Parallel Method-Calls

```
1: function $processImages(img1, img2, img3)$
2:     $result1 \leftarrow reduceColor(img1)$                ▷ m_call_1
3:     $result2 \leftarrow reduceColor(img2)$                ▷ m_call_2
4:     $result3 \leftarrow reduceColor(img3)$                ▷ m_call_3
5:     return $result1, result2, result3$
6: end function
```

Lollipop. Only the essential system processes were running on it during the experiments.

First, through controlled experiments, we have monitored the actual bandwidth for downloading and uploading data. More specifically, we send and receive data to a remote server on a local network in order to obtain the average actual bandwidth. We performed a set of 100 data transfer operations, 50 for data download and 50 for data upload. For each operation we have considered Equation 6.11 to obtain the TCP throughput. At the end of the process, the testbed collected the mean values of them.

$$BW = \frac{datasize}{CT} \tag{6.11}$$

After, the testbed executed all method-calls local and remotely (using one VM as offloading target). Through a controlled experiment, the processing time for each method-call were collected (*m_call_1*, *m_call_2* and *m_call_3*). The experiment executed and monitored 50 times each scenario. At the end of the process, the testbed collected the mean values of them.

Now, 50 executions have been performed, capturing the total execution time for each execution considering each evaluated scenario (see Figures 33b, 33d and 33f). In order to validate an SPN model, the metric extracted from it should be inside the bootstrapped confidence interval. Table 11 presents the results comparison. The results show that the MTTE extracted from the models (Model column) remains inside the respective confidence interval. Therefore, the experiments provided evidence that our SPN modeling strategy is reliable.

Table 11 – SPN Model Validation Using Bootstrap Technique

Model	**MTTE - Model**	**MTTE - Experiment**	**CI** ($B\alpha/2$)	**CI** ($B[1-\alpha/2]$)
A[1]	22,731	22,947	22,680	23,208
B[2]	287,914	287,055	281,803	291,352
C[3]	26,147	25,761	25,223	26,300
D[4]	26,999	27,003	26,932	27,085
E[5]	110,853	111,021	110,275	111,424
F[6]	59,856	59,711	55,696	63,634

[1] Model with Three Sequential Method-Calls with Remote Processing (see Algorithm 4 / Figure 33b).

[2] Model with Three Sequential Method-Calls with Local Processing (see Algorithm 4).

[3] Model with Three Parallel Method-Calls with Remote Processing (see Algorithm 6 / Figure 33f).

[4] Model with Three Parallel Method-Calls with Local Processing (see Algorithm 6).

[5] Model with Two Sequential Method-Calls and One in Parallel with Remote Processing (see Algorithm 5 / Figure 33d).

[6] Model with Two Sequential Method-Calls and One in Parallel with Local Processing (see Algorithm 5).

Chapter 7 Modeling MCC Systems in the Public Cloud

It must be considered that there is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things.

Niccolò Machiavelli, *1469-1527*

In this chapter, we present an SPN-based modeling strategy that makes it possible to represent remote MCC systems deployed in a heterogeneous infrastructure in a public cloud with variable processing and buffers capacities, variable demands, scaling policies, and a large number of VM instances running in the same infrastructure.

Public cloud providers charge their customers for resources consumed during a period. The amount of money that a company must pay to an IaaS cloud provider may grow significantly. Awareness of the related cost is a major element in the choice of appropriate strategies for deploying the system. This work considers as remote resources the use of VM instances and data traffic for supporting an MCC system deployed on a public cloud. These are the two basic resources for supporting a remote MCC system. As the number of users of an MCC system grows, the greater may be the consumption of VM instances and data traffic generated for supporting the increase in demand. Chapter 6 considered the data traffic generated by offloading operations from the perspective of the mobile app running on users devices. In this chapter, unlike that, we consider the data traffic generated by the remote MCC infrastructure to mobile devices. Companies in some situations need to evaluate the trade-off between some factors in order to find the most appropriate configuration that meets the requirements of their projects. The strategies proposed here support MCC service providers in the process of finding the most appropriate configuration for deploying their system in the cloud considering the possible offloading scenarios. Thus, making possible to comply with the performance requirement of SLAs within an acceptable cost.

More precisely, this chapter seeks to answer the following questions:

1. How to represent an elastic MCC infrastructure deployed on a public cloud using SPNs in order to support performance, data traffic, and cost evaluations?

2. How to calculate MRT, CDF, and throughput of a remote configuration for deploying an MCC system — that represent a set of RIs, ODIs, and scaling thresholds — using SPNs?

3. How to estimate the impact of the system parameters and their variations on the evaluated metrics?

4. How to estimate the data volume that will be transferred from the MCC infrastructure to mobile devices for a given period of time?

5. How to estimate the monetary cost of using a set of VM instances and transferring a data volume for supporting an offloading scenario during a given period?

7.1 Modeling Elastic MCC Systems

Analysts may adopt stochastic modeling to verify the impact of a set of settings on the metrics of their systems. By adopting modeling, developers may perform performance evaluation by considering a set of configurations in a model that represents the actual system. After, the actual system receives the most appropriate configuration. It saves time and money in some situations. A specific configuration for an application running on a public cloud is a variable that affects the cost of maintaining the underlying infrastructure over a period of time. It also impacts the overall system performance. This work considers a set of parameters and values for each of them in which each possible combination of them defines an application configuration in the cloud. We consider as parameters the types and number of RIs and ODIs to be used, scaling thresholds, stepsizes, system capacity, arrival rate, maximum number of jobs per VM instance (γ_n), maximum number of n-type ODIs ($MNODI_n$) for provisioning and costs. Although costs are not system parameters, they are parameters evaluated in the decision-making process. The costs considered are related to the cost of data traffic and use of instances. The time and the effort required to evaluate some metrics using modeling is smaller than evaluating each possible parameter setting in an actual system.

Let us describe some parameters in more detail. The instance type and the maximum number of concurrent jobs per VM instance (γ_n) affect the service time and response time. Autoscaling service can insert ODIs in the system when the system reaches any configured thresholds. Thus, the system can handle an increase in user requests without degrading performance. In this work, thresholds for scaling operations evaluates the number of requests in the system. The parameter "*scaling out threshold*" (SOT) sets the threshold for scaling out. The parameter "*scaling in threshold*" (SIT) sets the threshold for scaling in. Here, scaling out and scaling in are conditions that determine whether the system may insert or remove ODIs, respectively. SOT is a condition that evaluates whether there are conditions to insert ODIs. On the other hand, SIT is a condition that evaluates whether the system may remove ODIs. There may be a limit on the number of ODIs that the system may insert in order to ensure that a financial cost is not exceeded. Therefore, there exists a parameter ($MNODI_n$) that sets the maximum number of n-type ODIs the system may use at a time. On the other hand, the NRI_n parameter defines the number of n-type RIs in use. The type of VM instance, the operating system and the MCC system itself impact the time it takes for an ODI to become available and ready to process new requests. The time necessary to start on-demand resources may be long in some situations (MAO; HUMPHREY, 2012). The *service instances* (SI) are another parameter of our modeling strategy. SIs represents the set of n-type ODIs and RIs that are running on the system to process external requests. The *stepsize* parameter (ω_n) defines the number of ODIs the cloud starts when the system reaches a scaling out threshold. Each type of ODIs may have its own stepsize assigned to it. As we will see, each parameter takes an important role in the performance evaluation of the MCC system.

There are so many parameters and there may be a large set of possible values for each parameter. Analysts need to adjust their system and verifying the effect of each change on the performance when evaluating a set of configurations in an actual system. It is a difficult task to evaluate and find out the most appropriate configuration for a system by evaluating a large set of possible configurations in the actual system. By using our model, an analyst can evaluate the performance of his system with less effort in order to choose the most appropriate setting for the actual system running in the cloud.

Our cloud model represents the architecture presented in Chapter 5. Figure 36 demonstrates a cloud model for supporting two types of VM instances running in the same MCC infrastructure. However, by using our modeling strategy it is possible to represent an unlimited number of VM instances of different types through a refinement process. In addition, the model may represent the autoscaling mechanism for different types of instances. Our model makes it possible to obtain performance metrics, such as waiting time, service time, response time, throughput. An MCC service provider may evaluate other metrics such as the mean number of tasks in the system, data traffic, ODIs consumption estimation, and monetary costs related to resource consumption to maintain a system configuration in the cloud. Using a large number of instance types in the same model influences the time it takes to calculate the metrics. The higher the number of VM instances, the longer it takes to calculate the metrics. In this work, from here down, we use the terms requisitions, requests and jobs interchangeably.

Two SPN-based subnet composes the model (see Figure 36). The *Processing Subnet* represents the arrival and processing of outer requests. On the other hand, the *Autoscaling Subnet* represents the autoscaling mechanism. Our model has three type of timed parameters that are the arrival rate, the processing time for each instance type (i.e service time), and the time for an n-type ODI becomes available on the system for requests processing. Table 12 demonstrates the description of the model variables and components and Table 13 shows the attributes of the transitions. Hereafter we describe in detail the structure of the model and how it may be refined for supporting various instance types and represents more accurately other characteristics of the system.

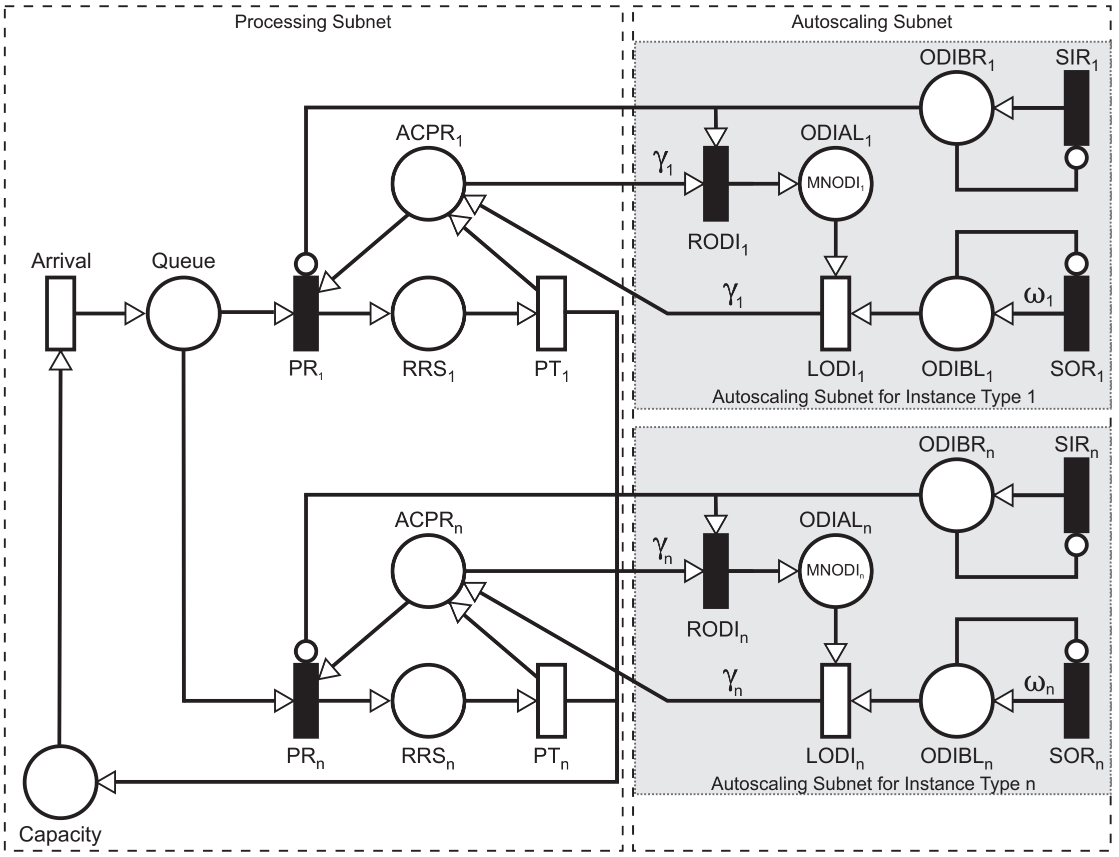


Figure 36 – Public Cloud MCC Infrastructure Model

The most common arrival process is the so-called Poisson arrivals, which simply means that the interarrival times are independently and identically distributed (IID) (JAIN, 1990). The transition *Arrival* is an exponential timed transition and it receives the mean delay related to the interarrival time of the evaluated system. This work considers only independent arrival times which are exponentially distributed. However, analysts may use other probability distribution related to the interarrival time other than exponential. For that aim, analysts need to refine the model when metrics need to be numerically evaluated and an expolynomial-phase-type-distribution must represent an interarrival time distribution (see Section 2.3.5). The model receives a set of new places and transitions related

Table 12 – Components of the SPN-based Modeling Strategy

Component	Description
γ_n	Maximum number of simultaneous jobs for each n-type VM instance
ω_n	Stepsize for launching n-type on-demand instances in each scale out request
ACPR$_n$	Available capacity to process requests in all n-type VM instances
Arrival	Interarrival time
Capacity	Capacity of the system
LODI$_n$	Launch n-type on-demand instances
MNODI$_n$	Maximum number of n-type on-demand instances that the system may use
NRI$_n$	Number of n-type reserved instances running on the system
ODIAL$_n$	N-type on-demand instances available to be launched
ODIBL$_n$	N-type on-demand instances being launched to be inserted in the system
ODIBR$_n$	N-type on-demand instances being removed from the system
PR$_n$	Process a request waiting in the queue in an n-type VM instance
PT$_n$	Processing time on an n-type VM instance
Queue	Requests waiting for service
RODI$_n$	Remove a n-type on-demand instance running on the system
RRS$_n$	Requests that receive service in any n-type VM instance
SIR$_n$	Scaling in request
SIT$_n$	Scaling in threshold associated with a n-type instance group
SOR$_n$	Scaling out request
SOT$_n$	Scaling out threshold associated with a n-type instance group

Table 13 – Transitions Attributes

Transition	Type	Server Semantic	Weight	Priority	Enabling Function
Arrival	Timed	Single Server	-	-	-
$LODI_n$	Timed	Infinite Server	-	-	-
PR_n	Immediate	-	1	1	-
PT_n	Timed	Infinite Server	-	-	-
$RODI_n$	Immediate	-	1	1	-
SIR_n	Immediate	-	1	1	Yes
SOR_n	Immediate	-	1	1	Yes

to a specific poly-distribution, replacing the transition *Arrival*. This refinement process may be applied to any timed transition in which the moment-matching technique may be applied 2.3.5. When, however, moment matching cannot be used, the related exponential transition becomes a general high-time transition (represented by a gray rectangle) and it receives a non-exponential probability distribution. Considering this, only by using simulation technique it is possible obtain the desired metrics. Transition *Arrival* firing creates a token in the place *Queue*. It means that a request has just entered the system and is waiting in the queue for receiving service. It is important to emphasize that our cloud model does not consider the inherent delay related to data transfer activities as well as packet loss regarding the use of mobile networks. The model considers the effective arrival rate; that is, it considers only the requests that are actually entering the system and will receive service.

The tokens in the place *Queue* represent requests waiting to receive service at a given time. The waiting time corresponds to the time interval between the time a request arrives and the time it begins to receive service. The service discipline is the order in which requests waiting in the queue receive service. Our model adopts a service discipline known as FCFS policy (JAIN, 1990). That is, the system processes requests taking into account the order in which they arrive. The system forwards external requests to the queue in the case where there is no computational capacity available to process them at the moment one of them arrives. Figure 37 describes the sequence diagram related to the method-calls executed by the system running in the front-end machine when an external request arrives at it. Obviously, we describe our system here to offer more insight to analysts. However, analysts need to adapt the strategies presented here by taking their systems into account. As we can see in Figure 37, immediately after the arrival of a request, the front-end verifies through the method-call *system.hasAvailableCapacity()* whether there is any buffer available to it accept the incoming request. If there is capacity to accept the request then the front-end verifies through the method-call *hasFreeServiceInstance()* if there is computational capacity available in any SIs to process the request. If the answer is positive, the system forwards the request to the SI that have available capacity to process it through the *processRequest()* method-call. Otherwise, the system forwards it to the queue through the *addToQueue()* method.

Our model can represent in the same MCC infrastructure for a specific workload more than one instance type for request processing. From a financial point of view, the use of two or more instance types may offer a reduction in costs. Each instance type defines an instance group and it comprises its own set of places, transitions, and arcs. We consider that each instance group comprises instances of the same type, with the same system configuration and running the same number of concurrent processing at a time. Figure 38 demonstrates the subnet of our model representing an instance type for request processing. Two transitions, two places, and arcs connecting them compose the subnet. It is important to highlight that each set of VM instances is composed of instances of the same type. The n-index in all component names of our model represents the index associated with a specific set of instances composed of instances of type n. As we can see in Figure 36, the model represents a system with two types of SIs. Our cloud model receives the set of components of the subnet presented in Figure 38 for each instance type

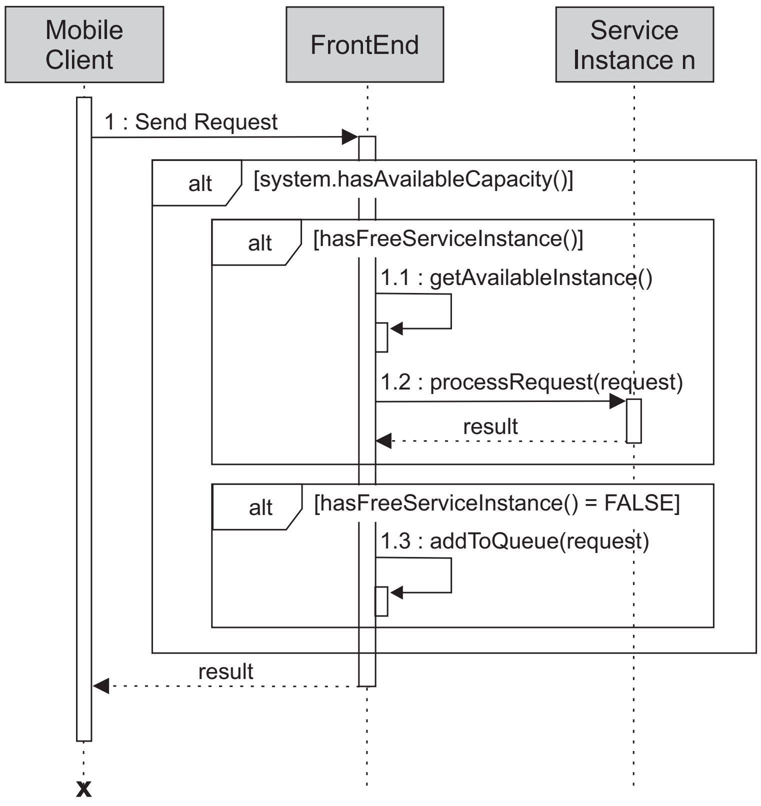

Mobile Client
FrontEnd
Service Instance n
1 : Send Request
alt
[system.hasAvailableCapacity()]
alt
[hasFreeServiceInstance()]
1.1 : getAvailableInstance()
1.2 : processRequest(request)
result
alt
[hasFreeServiceInstance() = FALSE]
1.3 : addToQueue(request)
result

Figure 37 – Sequence Diagram of the Front-End System Depicting Some Activities

in the MCC infrastructure. In addition, one output arc connects the *Queue* place to each new transition PR_n, as well as the transition PT_n to the place *Capacity*. We can see that whenever the system completes a job processing it restores its capacity. Analysts need to refine our cloud model by adding components to each instance type used in their MCC infrastructure.

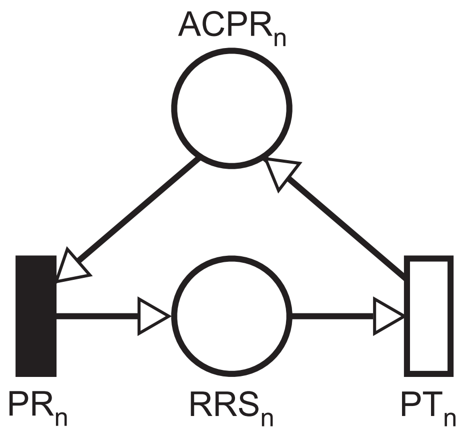


Figure 38 – SPN Subnet Representing a Type of Service Instances

The firing of transition PR_n represents the admission of a request that is waiting in the queue to be processed in an n-type VM instance. This transition has no delay assigned to itself. PR_n on firing represents the forwarding of the oldest queued request to any n-type instance when capacity becomes available for processing. Transition PR_n has two input places. Therefore, for PR_n to be fired, at least one token in the queue and one token in $ACPR_n$ is required. Tokens in $ACPR_n$ represent the processing capacity currently available to process new requests. In this work, the processing capacity represents the number of jobs that can be processed in the system simultaneously at a given moment. It is important to realize that there is no waiting time if, at the moment a request arrives, there are no queued requests and there is available processing capacity in any SI to process it. We consider that PR_n transitions have the same priority and weight between them. In spite of this, modelers may assign other weights and priorities for each of them in order to more accurately represent their systems. For example, an analyst may want to give preference to the requests to be processed in a given instance type when two or more instance types are available to process them. Let us consider the standard scenario in which PR_n transitions have the same weight and priority between them. It means that when two or more transitions PR_n are enabled at any given time, the type of n-type instance responsible for processing the request are randomly chosen with equal probabilities between them. When PR_n transition fires a token is consumed from each input place $Queue$ and $ACPR_n$, and another one is created in its unique RRS_n output place. The system starts processing the request immediately after the token is created in RRS_n place.

Now let us describe how the model represents the processing resources in the system. The relation simultaneous jobs per VM instance define the processing capacity of an instance. Thus, the maximum number of concurrent jobs that the system is able to handle at any time represents the total processing capacity. The presence of tokens in RRS_n place represents external requests being processed (i.e, receiving service). The number of VM instances of type n in use multiplied by the maximum number of jobs (γ_n) that each of them may process simultaneously determines the processing capacity for each set of n-type instances. In its initial state, the model receives the total processing capacity in each place $ACPR_n$ in relation to the total capacity for each type n. In other words, each token in all $ACPR_n$ places represents each job that the system can handle. That is, in the initial state, the sum of the number of tokens in each $ACPR_n$ place ($\sum_{type=1}^{n} ACPR_{type}$) represents the processing capacity of the system. The γ_n variable stores the maximum number of concurrent jobs that a n-type VM instance can execute. Analysts must perform experiments to find out the most appropriate value for γ_n variable, as we have already shown in Section 4.2.1. Later we will see that two arcs of the model receive the value of γ_n as its multiplicity. The number of tokens in all places $ACPR_n$ and RRS_n defines the processing capacity of the whole system at any time for stationary or transient analysis purposes. Equation 7.1 obtains the processing capacity (PC) of the MCC system at any time. In the same way, we may obtain the total number of service instances (TNSI) in the system using Equation 7.2. Developers need to pay attention to evaluating their application to set the processing limit for each VM instance.

$$PC = \sum_{type=1}^{n} \#ACPR_{type} + \#RRS_{type} \tag{7.1}$$

$$TNSI = \sum_{type=1}^{n} \frac{\#ACPR_{type} + \#RRS_{type}}{\gamma_{type}} \tag{7.2}$$

This work adopts the number of concurrent jobs as the abstraction related to the consumption of computational resources in the VM instances. Each type of application has its own pattern of computational resources consumption. Here, the computational resources are those that define the configuration of an instance such as vCPU, memory, storage, networking. There are types of applications that need more memory to satisfy their running requirements. For these applications, the greater the number of parallel executions, the higher the memory consumption. Similarly, other applications may only consume more CPU time for code processing. As an example we can cite image processing applications (SILVA et al., 2015). Obviously, there may be a limit related to the parallel code execution for a system. A system may fail after a specific limit has been exceeded. The performance of the system may degrade as the number of parallel executions increases. Computer system CPUs use Round-Robin algorithm with a fixed quantum (JAIN, 1990). It means that every process running on a computer system has a fixed fraction of time that is divided between them. The time window for workload processing is assigned to each process in a circular way, one process at a time. For a high-level system representation as well as high-level metrics computation, as is the purpose of our work, it is impracticable to represent the computer system at a low level. More specifically, it is a difficult task to represent the behavior of low-level computational resources to evaluate high-level application metrics. In addition to the fact that operating system processes are running in the background in the computer system, which makes this task more complex. Thus, this work abstracts the computational resources consumption in terms of the number of threads of an MCC system that are being executed in the VM instance at a given moment. As each request processing runs in its own threads, the term threads and jobs are interchangeable in this work.

The number of code that is running in parallel can affect the performance of an application to process a particular job. The task of the analysts is to find out the most appropriate number of parallel execution for their applications so that the desired performance is satisfied and at the same time there is an appropriate use of the computational resources. Using powerful instances to process only one job at a time can be a waste of money. Developers may need to allocate the maximum number of parallel jobs running on an instance when using a public cloud to achieve the desired performance within the defined financial cost.

Transition PT_n receives the mean processing time to process a job in an n-type VM instance. Transition PT_n represents the service time and the service time depends on the number of requests that each service instance is able to process concurrently. As already mentioned, each VM instance can process a fixed number of parallel requests determined by the γ_n parameter. The parameter γ_n remains constant while the system is up and running. We consider that, as the system process a request, it leaves the system and a new one in the queue receives the service right away. It is important to highlight that the processing time assigned to PT_n take in account the average time for processing one job when there are γ_n jobs executing in the VM instance at the same time. Each PT_n transition has infinite server semantics (ISS) associated with it as SIs may process requests concurrently. ISS enables the concurrent processing of an entire set of tokens that is enabling a timed transition (TRIVEDI, 2001; GERMAN, 2000). However, in our model, there are limits regarding the number of tokens that may be enabling a PT_n transition. The number of n-type SIs in the system multiplied by the γ_n parameter determines the maximum enabling degree of a PT_n transition at a moment. Thus, the initial number of tokens in each $ACPR_n$ place defines the initial enabling degree of the related PT_n transition. As we will see later, the maximum enabling degree of each PT_n may changes when ODIs are inserted or removed from the system.

The *Capacity* place represents the capacity of the MCC system in the cloud. This place represents the maximum number of requests that may be in the system at a time. It is important to highlight that there is a subtle difference between processing capacity and system capacity in this work. Processing capacity defines the maximum number of requests that may receive service at a given time. On the other hand, the system capacity includes requests waiting in the queue for service, as well as the ones that are receiving service. In other words, the system capacity defines the number of jobs that can be in the system at a time. An analyst may increase the capacity of the system to handle an increase in request rate without having to contract new VM instances for supporting the new demand. However, in this case, the waiting time tends to increase and, as a consequence, there will be both an increase in response time and a decrease in throughput. As we can see, system capacity is a key variable that affects all system performance and the cost of maintaining the infrastructure in a public cloud. Developers may change the capacity of their systems to verify the impact of each change on the metrics evaluated. Place *Capacity* is the unique input place of the *Arrival* transition and the unique output place of PT_n transitions. The firing of the *Arrival* transition depends on the existence of tokens in the place *Capacity.* It means that user requests are lost when the number of jobs in the system is equal to the initial number of buffers in *Capacity* (i.e. there are no tokens in *Capacity* place). Each PT_n transition when firing creates one token in the place *Capacity.* Thus, the system restores its capacity to handle new external requests when it finishes the processing of jobs. Our model represents finite buffer systems that are always stable since requests in the system never exceed the number of buffers.

Heretodown we do not have depicted the representation of the scaling mechanism by our model. We describe in the next paragraphs the scaling mechanism and how the system inserts and removes resources for request processing. This work adopts a reactive approach to activate the scaling mechanism (GALANTE; BONA, 2012; BISWAS et al., 2015; ASSUNCAO et al., 2016). There exists some strategies that an analyst can adopt to define when the system adds or removes resources. Most cloud providers monitor some metrics such as CPU, memory usage, response time, queue size to define when applications should be scaled. For example, AWS and Microsoft Azure provide tools to enable their customers to set up scaling policies based on the parameters mentioned above. In addition, developers also have the option of implementing a custom scaling policy. A custom implementation would scale resources through calls to API provided by the cloud provider. For that aim, taking into account some custom rules defined by the developer. CPU and memory usage and any other computational resource can be controlled by limiting the number of concurrent processes running in the VM instance. As we pointed out in Chapter 5, the front-end machine is also responsible for monitoring the state of the system and scaling the system accordingly. In this work, the size of the queue is the factor that defines whether and when to scale the system. We adopted a reactive approach that monitors the number of jobs in the system queue and the processing capacity available in the system at a given moment.

Let us see how the model represents the process for adding ODIs to the system. When the system reaches a SOT, it scales. More specifically, the system adds ODIs for service processing when it reaches a predefined condition for this action. The system adds ODIs to handle an increase in the number of requests in the queue in order to maintain the expected performance defined in SLAs. In our architecture, the front-end controls the scaling process. The front-end sends a command to the cloud manager by using the API for the addition of ODIs when the system reaches any SOT and there is at least one ODI available for insertion.

Place $ODIAL_n$ represents the number of n-type ODIs available for scaling out the system and each token in it represents one non-used ODI. Variable $MNODI_n$ defines the maximum number of ODIs for an n-type service instance group that the system may use. Place $ODIAL_n$ receives the value of this variable as its initial value. Public clouds allow customers to provisioning resources without limits. However, the higher the resources usages, the higher the monetary value that the service provider pays. In this way, this variable prevents a predefined financial cost from being exceeded. It means that if the system reaches a SOT but there are no tokens in $ODIAL_n$, it will not be able to add ODIs in a service group.

Transition SOR_n represents the process for inserting ODIs in the system. The firing of this transition corresponds to an asynchronous call to the cloud manager in order to insert one or more ODIs in the system. No input places exist for transition SOR_n and it only fires when the system reaches any predefined condition. This transition has an enabling function associated with it to represent the request for addition of n-type ODIs. As we have pointed out, an enabling function is a boolean expression associated with an immediate transition (see Section 2.3.4). In our work, enabling functions define conditions to scale in/out the system. Equation 7.3 demonstrates an example of enabling function associated with an SOR_n transition. As we can see, this condition verifies whether the system has reached a defined SOT, as well as whether there is at least one ODI available for addition in the n-type service instance group. Following this example, if the number of tokens in place *Queue* is greater than or equal to a specific SOT and there is at least one n-type ODI available, the transition can fires. Equation 7.4 sets the threshold to scale the system out. As we can see, it considers the processing capacity available in the system at a given time and multiplies it by a predefined threshold. The *Threshold* parameter in Equation 7.4 is an integer that represents a multiplicative factor. The stepsize associated with the output arc of the transition SOR_n defines the multiplicity of this arc. The ω_n parameter in the model represents the stepsize for an n-type ODIs group. It means that the cloud starts the number of n-type ODIs associated with the stepsize for each scale out request. Transition SOR_n firing creates the number of tokens defined by ω_n in the place $ODIBL_n$.

$$(\#Queue >= SOT) \ \bigwedge \ (\#ODIAL_n > 0) \tag{7.3}$$

$$SOT = \left(\sum_{type=1}^{n} \#ACPR_{type} + \#RRS_{type} \right) \times Threshold \tag{7.4}$$

There may be situations where the stepsize is greater than the number of ODIs available for insertion. Thus, when the SOR_n transition fires, the multiplicity of its output arc performs a check to verify that the stepsize is less than or equal to the number of tokens available in $ODIAL_n$ place. We have defined the verification for the multiplicity condition of the output arc, as shown in Equation 7.5. As we can see, the condition associated with the multiplicity of this arc determines the actual number of tokens that the transition creates in the output place $ODIBL_n$. After that, the effective process to launch the ODIs starts.

$$IF(\#ODIAL_n >= STEPSIZE) : (STEPSIZE) \ ELSE \ (\#ODIAL_n) \tag{7.5}$$

The presence of tokens in the place $ODIBL_n$ enables the $LODI_n$ transition and each token represents one VM instance. This place when having tokens represents that the system starts one or more ODIs. Transition $LODI_n$ has infinite server semantics assigned to itself. It means that the instantiation process of each ODI occurs simultaneously. The delay assigned to an $LODI_n$ transition comprises the time interval between sending a request to the cloud for scaling out and the time point at which the VM instance is available in the system for request processing. This delay is a random variable whose value depends on the instance type, the operating system, the MCC application itself, and the state of the cloud provider. As we can see, this delay is inherently related to an application. It means that two applications may have different instantiation time for their instances, even when they use the same instance type. Each IT has a processing limit related to the maximum number of jobs per unit of time can receive service and the variable γ_n defines that limit. The γ_n variable defines the multiplicity of the unique output arc of the $LODI_n$ transition. Transition $LODI_n$ when firing represents that an ODI is available and ready for processing requests. In addition, when the transition fires, the system's processing capacity increases by γ_n requests. When the system inserts an ODI to a service group, it means that the number of ODIs available to scale the system out decreases. Thus, when $LODI_n$ fires, it consumes one token of place $ODIBL_n$ and one token of place $ODIAL_n$ and creates the number of tokens defined by γ_n in place $ACPR_n$.

Our model considers that the system only executes one scaling out request at a time to avoid the over-provisioning of VM instances. For that aim, an inhibitor arc connects the place $ODIBL_n$ to the transition SOR_n. It means that the system only may execute a new scaling out request after the cloud processes a previous one. However, the MCC system may instantiate more than one instance per request — ω_n parameter. After the cloud process a scaling up request, the MCC system may execute another one, if necessary.

Let us see how our model represents the scaling in process. The first step for scaling in is the existence of conditions for removal ODIs running in the system. The system inserts ODIs to handle an increase in the number of requests in the queue. On the other hand, the system needs to remove the ODIs running in it when the queue size decreases. The objective for scaling in is to adjust the number of service instances taking into account the number of requests and VM instances running in the system. In addition, the scaling in process aims to avoid over-provisioning, and as a result, decreasing the amount of money the service provider pays to an IaaS cloud provider. The system scales in when it reaches any SIT.

The firing of the transition SIR_n represents a request for scaling in. Transition SIR_n only fires when there exists a condition for scaling in associated with its n-type ODIs and the system satisfies this condition. This transition represents the front-end sending a request to the cloud to remove one or more ODIs running in a service group. No input places are associated with the transition SIR_n. Transition SIR_n has an enabling function that verifies whether the system reached a threshold to scale the system in and if there is at least one ODI running on the system. Equation 7.6 demonstrates an example of an enabling function associated with a SIR_n transition. Following this example, it means that the MCC system only executes the request for scaling in when the size of the queue is lower than a predefined SIT and there is at least one n-type ODI running. Equation 7.7 sets the threshold to scale the system in. As we can see, it considers the processing capacity available in the system at a given time and multiplies it by a predefined threshold. The *Threshold* parameter in Equation 7.7 is an integer that represents a multiplicative factor. The firing of this transition creates one token in the $ODIBR_n$ place.

$$(\#Queue < SIT) \ \bigwedge \ (\#ODIAL_n < MNODI_n) \tag{7.6}$$

$$SIT = \left(\sum_{type=1}^{n} \#ACPR_{type} + \#RRS_{type} \right) \times Threshold \tag{7.7}$$

It is important to note that the arc connecting the SIR_n transition to the $ODIBR_n$ place has one as its multiplicity value. Place $ODIBR_n$ has an inhibitor arc connecting it to the transition PR_n. It means our model considers that the system only removes one ODI per scaling in request. The system removes one ODI per request and, after removal, it monitors its state. That is, after removing the VM instance, if the system needs to remove another one, it will execute another request for scaling in. However, a refinement in our model can change this behavior. This refinement consists in changing the multiplicity of the output arc of SIR_n to define the number of VM instances that the system removes in each scaling in request. This arc receives a numeric value or a condition that defines the number of tokens that the SIR_n transition creates in the $ODIBR_n$ place at the time it fires. That is, based on the state of the system, the number of ODIs that the system removes in a single request can be changed by changing the multiplicity of the arc.

We have defined an abstraction to represent inactive service instances in the system because GSPN does not differentiate tokens in the model as CPN does. For this purpose, in our modeling strategy, when the number of tokens in $ACPR_n$ is equal to or greater than γ_n, it represents the system has at least one non-used n-type instance. In this work, an idle VM instance represents an instance that does not process any external requests. Place $ODIBR_n$ has an inhibitor arc connecting it to the transition PR_n. It means that the system does not forward queued requests to the ODI that it must remove. Thus, the system lets the ODI to finish the requests processing, and thereafter, the system removes it from its instance group.

Transition $RODI_n$ when firing removes an ODI running on the system. This transition has two input places that are $ODIBR_n$ and $ACPR_n$. It needs at least one token in $ODIBR_n$ and γ_n tokens in $ACPR_n$ to be able to fires. The γ_n parameter represents the maximum number of concurrent jobs in a single n-type instance. That is, the processing capacity of each n-type instance. This parameter defines the multiplicity of the output arc of the place $ACPR_n$ to the transition $RODI_n$. As mentioned above, the system can only remove an ODI when there are no requests receiving service in the instance. That is, the place $ACPR_n$ must have at least the number of tokens equivalent to γ_n. The firing of the transition $RODI_n$ creates one token in the place $ODIAL_n$ restoring the scaling capacity and decreasing the actual processing capacity of the MCC system.

7.2 Performance Metrics on the Cloud

Performance evaluation is a critical process for determining whether the system performance complies with the requirements defined in SLAs. Analysts can evaluate their systems by taking a long-term or short-term perspective. Short-term perspective corresponds to the behavior of the system during a certain period of time. Long-term perspective corresponds to the behavior of the system over a long period of execution. The transient analysis evaluates the time-dependent behavior of the system. On the other hand, the stationary analysis evaluates the behavior of a system for a long period of execution. The transient analysis is relevant if the short-term behavior is more important than the long-term for an evaluation. Using the proposed model, it is possible to calculate a variety of metrics taking into account a transient or stationary perspective. MRT and throughput are the two key performance metrics when evaluating a system deployed in a cloud.

MRT corresponds to the mean time spent to process an external request, taking into account the time it enters the system and the time it leaves. The MRT calculation considers the Little's law (LITTLE, 1961) (see Section 5.2.1). Little's law considers three parameters for estimating metrics, which are the average number of tasks in the system, arrival rate, and mean response time. There are two requirements for applying Little's law. The first one is that the system is in a stable condition. The second one is that the analyst knows the arrival rate and the average number of jobs in the system (JAIN, 1990). The first requirement means that the arrival rate is less than or equal to the service rate. In other words, the number of incoming requests is less than or equal to the number of jobs that leave the system after they receive service, considering a period of time. Otherwise, the number of requests in the system could increase indefinitely, and the time a single request would remain in the system could be infinite. In this context, a request enters the system and never comes back to the mobile device. Equation 7.8 obtains the expected number of requests waiting to receive service. Equation 7.9 gets the expected number of requests that receive service considering all service instance group in the infrastructure. Finally, Equation 7.10 gets the mean system size (MSS) or average number of jobs in the system. Now we can obtain the MRT of an MCC system configuration. Equation 7.11 computes the MRT taking into account the two requirements above. This equation considers the probability that the system will accept new requests. The firing rate of the transition Arrival (AR) multiplied by the probability that the system can accept new requests represents the effective arrival rate. This effective arrival rate does not consider discarding that may occur when there is no capacity in the system to accept new requests. The expected number of jobs in the system divided by the effective arrival rate give us the MRT.

$$Requests_{queue} = \sum_{i=1}^{n} P(m(Queue) = i) \times i \tag{7.8}$$

$$Requests_{service} = \sum_{type=1}^{n} \left(\sum_{i=1}^{n} P(m(RRS_{type}) = i) \times i \right) \tag{7.9}$$

$$MSS = Requests_{queue} + Requests_{service} \tag{7.10}$$

$$MRT = \frac{MSS}{AR \times (1 - P(m(Capacity) = 0))} \tag{7.11}$$

The throughput represents the rate of processed jobs per unit time. In an SPN model, the throughput evaluation considers the firing rate of a timed transition and the number of tokens in its input places. This transition represents the activity whose rate we need to evaluate. For example, the PT_n transition represents the processing of an external request. Thus, the firing rate of the transition PT_n impacts the throughput of this activity. The throughput for an n-type instance service group corresponds to the product of the firing rate of the PT_n transition and the expected number of tokens in the place RRS_n. The firing semantic, as well as the related probability distribution of a timed transition, change the way an analyst gets the performance of his system. Equation 7.12 calculates throughput according to SSS. Equation 7.13 computes it considering ISS. These equations consider that the transition delay is exponentially distributed. However, the adoption of other probability distributions may change these equations. The MCC infrastructure may have more than one group of n-type service instances running at the same time. Thus, the system throughput is the sum of the throughput of all groups running in the cloud. Finally, the system throughput can be obtained by the Equation 7.14. Little law states that we may obtain TP when we know the MSS — that is, the mean number of requests in the system— and MRT (JAIN, 1990). Thus, Equation 7.15 give us the throughput considering the MSS and MRT of a system configuration. A stationary evaluation in the model obtains these metrics.

$$TP_n = P(m(RRS_n) >= i) \times \frac{1}{PT_n} \tag{7.12}$$

$$TP_n = \left(\sum_{i=1}^{z} P(m(RRS_n) = i) \times i \right) \times \frac{1}{PT_n} \tag{7.13}$$

$$TP = \sum_{n=1}^{z} TP_n \tag{7.14}$$

$$TP = \frac{MSS}{MRT} \tag{7.15}$$

Analysts may evaluate performance metrics other than throughput and MRT when using our SPN-based modeling strategy. In some situations, it is necessary to know the probability of the system completing a set of jobs in a given period, as well as the mean time for the system to process them (MTTA). More specifically, obtaining the CDF related to the probability of the system completing the jobs processing at each point t in the evaluated period. For this purpose, the SPN model must be refined in order to create an absorbing state in the state space. After that, a transient evaluation in the model obtains CDF and MTTA. This analysis can provide a more accurate understanding of the system behavior.

7.3 Costs for Using Resources in the Public Cloud

This work considers as cloud resources for cost evaluation the use of VM instances and data traffic. A public cloud provides instances with different computational capabilities and it can charge different values to use them. AWS charges for the use of RIs take into account a long-term contract. On the other hand, AWS charges for the use of ODIs considering the amount of hours that the system has used them during a period. Our cost model considers the cost of using RIs and ODIs in the same MCC infrastructure. In this case, it is necessary to group the VM instances by their types and contracting model to estimate the costs for each group of instances. The sum of the costs of using VM instances of all groups defines the total costs of using instances over a period of time.

The MCC service provider knows the number of RIs that have been contracted since the beginning of the system operation. Considering this, it is not a difficult task to calculate the total cost of using RIs in an MCC infrastructure over a given period. On the other hand, the use of ODIs over a period of time may vary. The use of ODIs by a system depends on the predefined scaling thresholds, stepsizes, user demands, and the time that the system will be up and running.

The first step in calculating the cost of using ODIs is to estimate their use within the period that the system is running. An MCC service provider can estimate the use of ODIs using our SPN-based modeling strategy through a stationary analysis. This metric is obtained by subtracting the maximum number of ODIs ($MNODI_n$) available to scale the system and the expected number of ODIs not used by the system for each n-type service instance group. Equation 7.16 calculates the expected ODI usage (EODIU) metric. Variable $MNODI_n$ defines the initial number of tokens in place $ODIAL_n$. The expected number of tokens in the place $ODIAL_n$ obtained through a steady-state evaluation correspond to the non-use of ODIs by the system. Thus, we may estimate the cost of using ODIs when the value of the metric $EODIU_n$ is known in advance.

$$EODIU_n = MNODI_n - \left(\sum_{i=1}^{n} P(m(ODIAL_n) = i) \times i \right) \tag{7.16}$$

For this purpose, the next step is to know the duration of time the MCC infrastructure will be up and running. Equation 7.17 calculates the total cost of using ODIs (TCODI) for a given period of time. The analyst must provide this value in hours since the public cloud charges for the number of hours the system used the ODIs. $Cost_n$ corresponds to the hourly price for the n-type ODI evaluated. And finally, $EODIU_n$ defines the expected number of n-type ODIs the system uses. We just multiply all of them to obtain the cost of using ODIs for a group of n-type service instances. The sum of the costs of all instance groups gives us the TCODI.

$$TCODI = \sum_{n=1} EODIU_n \times Cost_n \times Time \tag{7.17}$$

AWS makes possible that a customer contract RIs for long-term use. Customers may choose to contract RIs for a year or more. Obviously, as the usage period is long, it means that the price that AWS charges for them is low considering other contracting models. Equation 7.18 calculates the total cost for using RIs (TCRI). The parameter n defines the type of RIs evaluated. The variable RI_n corresponds to the number of n-type RIs contracted. $Cost_n$ corresponds to the annual price for this type of VM instance. And $Time$ defines the number of years the system will be up in the cloud. If this value is less than one, the analyst should consider one year. We just multiply the values to get the cost for using a set of n-type RIs. The sum of the costs of all groups of RIs gives us the TCRI. Equation 7.19 gives us the total cost of using instances (TCI). This equation considers the cost of using RIs and ODIs.

$$TCRI = \sum_{n=1} RI_n \times Cost_n \times Time \tag{7.18}$$

$$TCI = TCRI + TCODI \tag{7.19}$$

Public clouds charge their customers for outgoing data transfers to the Internet by considering “utilization ranges" as illustrated in Table 14. As we can see, the price decreases as the volume of transferred data increases over a period. The first step in the data transfer costs evaluation is to estimate the volume of data transferred over a given period. For that aim, developers should only consider bytes transferred from the remote MCC infrastructure to mobile devices. Using Equation 7.20 it is possible to find out the TTB. TTB corresponds to the volume of data transferred during a period and, to obtain it, it is necessary multiplying: (*i*) the system throughput (*TP*) — in requests/sec; (*iii*) the period of time the system is up and running (*Time*) — in sec; and finally (*ii*) the amount of data transferred in each request (*Bytes*).

$$TTB = TP \times Time \times Bytes \tag{7.20}$$

Table 14 – Amazon EC2 Prices per Transferred Bytes (AWS, 2017)

Data Transfer OUT To Internet	**Price/GB**
First 10 TB / month	\$0.09
Next 40 TB / month	\$0.085
Next 100 TB / month	\$0.07
Next 350 TB / month	\$0.05

Now, using Equation 7.21 it is possible to obtain the financial cost for data transfer (FCDT). First, it is necessary to split the number of total outgoing bytes according to the utilization range of the price table used (see Table 14). After that, it is necessary to multiply the number of bytes consumed in each “utilization range" by its respective price. The sum of the cost of all ranges used defines the cost for a given period. Companies may adjust the cost calculation according to the data traffic charges policy of their IaaS cloud provider.

$$FCDT = \sum_{ur=1}^{n} TTB_{ur} \times PricePerGB_{ur} \tag{7.21}$$

Finally, using Equation 7.22 it is possible to obtain the total cost (TC) of maintaining an MCC system in the public cloud. Analysts can use these equations to support the financial evaluation of all possible configurations for deploying their MCC systems. First, they choose the configurations that meet the desired performance level defined in the SLAs. After that, they analyze the cost for each of them. Making possible the trade-off between performance and costs in the decision-making process. In addition, analysts may change these equations to perform more accurate financial analysis of their systems.

$$TC = TCI + FCDT \tag{7.22}$$

7.4 Model Validation

We have validated our SPN-based modeling strategy that represents a remote MCC infrastructure using the infrastructure provided by AWS. The validation process consisted of two scenarios and each one having its own configuration. The difference between them lies in the thresholds for scaling the system out, number of simultaneous processing in each VM instance (γ) and request rates. Tables 15 and 16 demonstrate the instance types and parameters we have considered in scenarios #1 and #2, respectively. As we can see, the stepsize (ω), number of RIs and SIT value remain the same for both scenarios. Both the actual system as well as the SPN model were configured with the same parameters and they received the same workloads.

Table 15 – Scenario #1 for Model Validation

	Scenario #1	
	Instance Type 1	**Instance Type 2**
Parameter	**Value**	**Value**
Instance Type	t2.micro	t2.small
RI	1	1
Max ODIs (MNODIs)	3	3
SOT[1]	3	3
SIT[2]	1	1
γ[3]	1	1
ω[4]	1	1

[1] Scaling Out Threshold.

[2] Scaling In Threshold.

[3] Maximum number of simultaneous jobs for each n-type instance.

[4]The stepsize for launching n-type on-demand instances in each scale out request.

Table 16 – Scenario #2 for Model Validation

	Scenario #2	
	Instance Type 1	**Instance Type 2**
Parameter	**Value**	**Value**
Instance Type	t2.small	t2.medium
RI	1	1
Max ODIs (MNODIs)	2	1
SOT[1]	4	4
SIT[2]	1	1
γ[3]	1	2
ω[4]	1	1

[1] Scaling Out Threshold.

[2] Scaling In Threshold.

[3] Maximum number of simultaneous jobs for each n-type instance.

[4]The stepsize for launching n-type on-demand instances in each scale out request.

We have evaluated a heterogeneous MCC infrastructure composed of two types of VM instances. The MCC infrastructure deployed in the cloud is similar to that presented in Chapter 5, as we can see in Figure 39. The only difference between them is the presence of an aditional RI in the infrastructure evaluated here, depicted in Figure 39 as "*JMeter Instance*". This additional RI assumes the role of external users making requests. JMeter was running in this VM instance to generate requests to be handled by the front-end, simulating offloading making by mobile devices. JMeter run requests based on specific exponential request rates configured by us (see Appendix A). JMeter uses threads to run requests in parallel and each thread is supposed to simulate one user request. JMeter and front-end instances are VM instances of type *t2.small*.

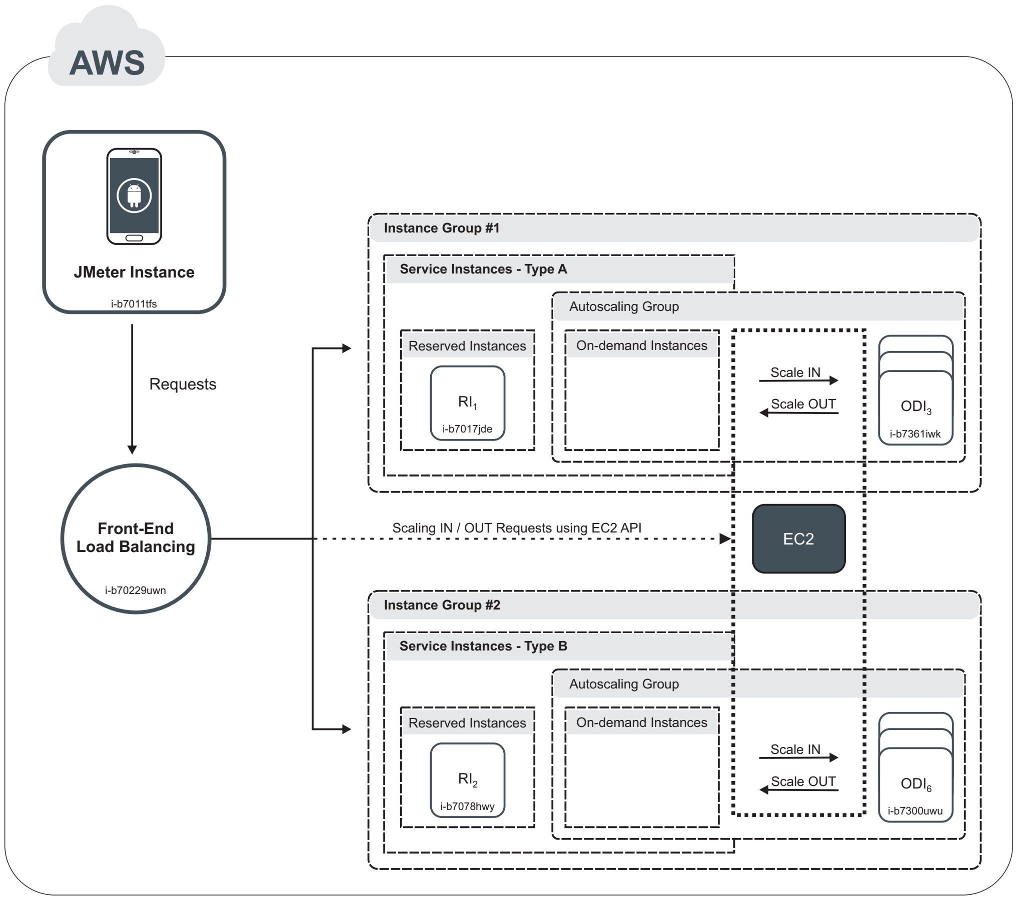


Figure 39 – Remote Architecture for Model Validation

We have implemented an MCC service using a face recognition application as a benchmark. The MCC system has been implemented in Java and it is composed of two subsystems. The first subsystem runs on the front-end and the second one runs on each service instance. The two subsystems communicated with each other via socket messages. Front-end handles the incoming request and performs some actions based on the state of the system, such as queuing the request, forwarding it for receiving service, executing scaling requests to AWS EC2, and sending the request back to the user device after the service has been finished (see Chapter 5). The front-end subsystem performed requests for scaling the system in/out using EC2 API. As all requests pass through the front-end, it knows the state of the MCC system and performs load balancing when distributing the queued request for receiving service among the available service instances — that is, considering γ_n parameter for each instance. Each service instance had OpenCV (OPENCV, 2018) and *JavaCV* (JAVACV, 2018b) for supporting face recognition processing. Basically, mobile users — JMeter — send photos to our face recognition system and the system process the requests. Each JMeter request sent a face image, chosen at random, of a set of face images. The subsystem running in the service instances received the requests forwarded by the front-end and processed them each one in its own thread. The system considered a database of 8,000 faces replicated in each service instance. Figure 40 represents the communication process between JMeter, front-end, and a service instance from the moment JMeter generates a request.

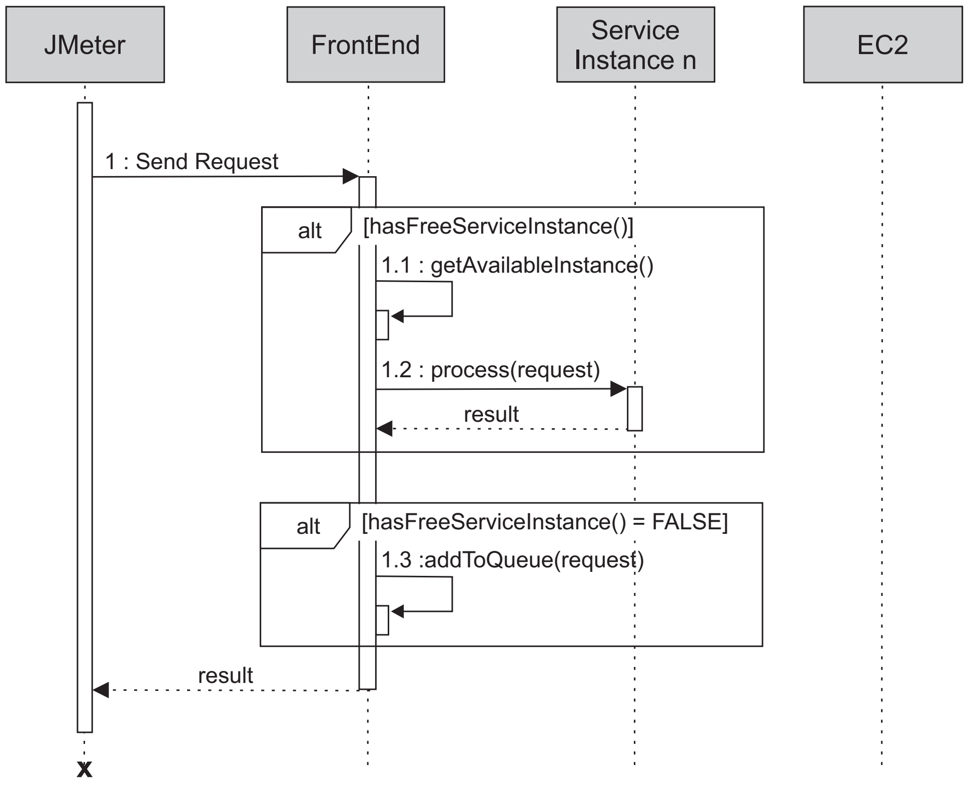


Figure 40 – Sequence Diagram Depicting the Communication Between JMeter, Front-End and a Service Instance

The first step in the validation process was to collect the distribution of times related to the face recognition processing in each VM instance type evaluated. In scenario #1, we consider only one user request being processed in each VM instance (parameter $\gamma = 1$). On the other hand, in scenario #2, instances of type *t2.medium* process two requests at a time (parameter $\gamma = 2$). For this evaluation, we have used JMeter to generate requests in order to collect the time required to process a single request and its distribution, considering the configuration and workload evaluated. JMeter generated 80 requests with one minute delay after the end of each request. It is important to highlight that in scenario #2 the request was duplicated in the service instance of type *t2.medium*, and the instance processed both requests at the same time, since we consider that this type of instance can process two requests at the same time (parameter $\gamma = 2$, see Table 16). Table 17 depicts the mean processing time and probability distribution related to the time for processing each request considering each instance type evaluated. We have used the methods Anderson-Darling and Kolmogorov-Smirnov and they have indicated with 95 % of confidence that there was no evidence to refute that the time distribution related to the face recognition is a Gaussian distribution (ANDERSON; DARLING, 1954; CHAKRAVARTY; LAHA; ROY, 1967).

Table 17 – Mean Processing Time Considering γ Parameter

Instance Type	**Processing Time (ms)**	γ[1]	**Distribution**	**SD**[2]
t2.micro	23,638	1	Gaussian	0.094
t2.small	19,075	1	Gaussian	0.047
t2.medium	19,131	2	Gaussian	0.090

[1] Maximum number of simultaneous jobs.

[2] Standard Deviation.

The second step in the validation process was to collect the time distribution related to the time AWS EC2 spent to launch our ODIs and make them available in our system for requests processing. There are several factors that affect this time and it does not depend only on AWS EC2. Among the factors, we can mention the instance type, AMI, and the state of the AWS EC2 when the system executes a request to scale itself. It is important to highlight that the time to launch an ODI may change abruptly. Sometimes ODIs may become available faster, and at other times, this process may take a long time. That is, depending on the system configuration in relation to the number of ODIs, stepsizes, and scaling thresholds, there may be a considerable difference in relation to the metrics presented by the model and the metrics of the real system, due to the long delay to make an ODI available. Not even AWS guarantees the time to launch an ODI. Thus, users who adopt our modeling strategy should consider this issue.

We have generated 80 requests to AWS EC2 for launching our ODIs *t2.micro*, *t2.small* and *t2.medium.* Table 18 demonstrates the time required to launch our three types of ODIs. The duration of time comprises the instant of time the subsystem running in the front-end sent a request to EC2 to launch an ODI until the time the ODI was available in the system for processing requests. The methods Anderson-Darling and Kolmogorov-Smirnov indicated that there was no evidence to refute that the time distribution related to the time for launching our ODIs follows an Erlang distribution with 95 % confidence. An portion of the algorithm responsible to execute the scaling out requests to AWS EC2 using EC2 API is available in Appendix B.

We have evaluated the two scenarios after collecting the times related to the face recognition processing and launching of ODIs. It is important to highligh that both the model and the actual system were configured with the same parameters (see Tables 15 and 16). For both scenarios we have considered exponentially distributed interarrival times. The

Table 18 – Mean Time to Launch our ODIs

Instance Type	**Lauching Time (s)**
t2.micro	175,06
t2.small	204,57
t2.medium	239,46

default implementation of JMeter does not provide a default function to generate requests considering the exponential distribution. Considering this, we have implemented a JMeter add-on to generate our exponential requests. Appendix A demonstrates the script that adds this functionality to JMeter.

In scenario #1 we have considered an arrival rate of $1/10,000\ ms$. JMeter performed 30 iterations and, at each iteration, 80 requests for facial recognition were performed. The results collected represent 2,400 requests processed for scenario #1. We have monitored the mean response time and the use of on-demand instances. The use of ODIs comprises the time interval that the ODI is available in the system. It is obtained for each instance type dividing the total time of each iteration by the sum of the time interval that each ODI of the same type was available in the system. At the end of each iteration our testbed collected the mean values of them. Firstly, we adopted the stationary simulation to calculate metrics using the model, since the time related to the face recognition processing follows a Gaussian distribution and the time related to the launching of ODIs follows an Erlang distribution. After that, we obtained the metrics adopting exponential transitions in order to verify the possibility of evaluating our case studies through numerical evaluation. Table 19 shows the mean response time and use of ODIs. As we can see, for both types of evaluations, the model presents values within the confidence interval (CI). It means that it is possible to evaluate the desired metrics using numerical evaluation.

Table 19 – Scenario #1 - Validation

Metric	**CI ($B\alpha/2$)**	**CI($B[1-\alpha/2]$)**	**System**[1]	**NA**[2]	**Simul**[3]
Response Time (ms)	57,160	63,611	60,473	62,473	60,661
ODIs Usage (t2.micro)	0.130081	0.169886	0.157543	0.165369	0.132399
ODIs Usage (t2.small)	0.063564	0.127743	0.120569	0.125931	0.088613

[1] Actual System.

[2] Numerical Analysis.

[3] Simulation.

Let us analyze the results for scenario #2. The model validation for #2 considered the configurations presented in Table 16. In scenario #2 we have considered an arrival rate of $1/5,000\ ms$. We are now using VM instances of type *t2.medium* and two simultaneous requests are being processed in each of them (parameter $\gamma = 2$). JMeter performed 30 iterations and, at each iteration, 80 requests for face recognition were performed. The results collected represent 2,400 requests processed for scenario #2. Table 20 shows the mean response time and use of ODIs. As we can see, the model evaluation presents values within the CI.

Table 20 – Scenario #2 - Validation

Metric	**CI(**$B\alpha/2$**)**	**CI(**$B[1-\alpha/2]$**)**	**System**[1]	**NA**[2]	**Simul**[3]
Response Time (ms)	76,370	87,424	82,138	80,332	79,554
ODIs Usage (t2.small)	0.416685	0.587414	0.501754	0.453051	0.451691
ODI Usage (t2.medium)	0.188375	0.248902	0.218816	0.203887	0.191265

[1] Actual System.

[2] Numerical Analysis.

[3] Simulation.

Tables 19 and 20 show that the metrics extracted from the models remains inside the respective CI. Therefore, this validation process indicates that the generated models represents, statically proved, the reality. Figures 41 and 42 depict the mean response time and use of ODIs obtained by simulation, numerical analysis, and the real system considering both scenarios, respectively. Considering the obtained results, we have adopted numerical evaluation to obtain the desired metrics in our case studies.

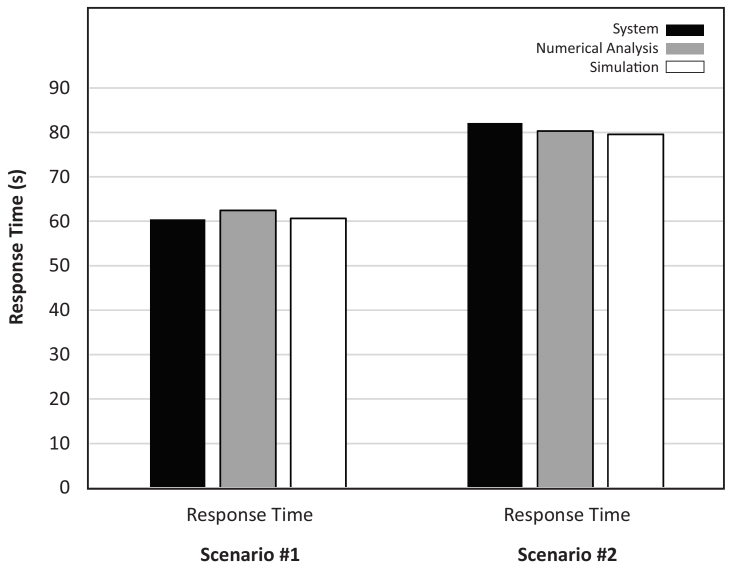


Figure 41 – Response Time for Both Scenarios

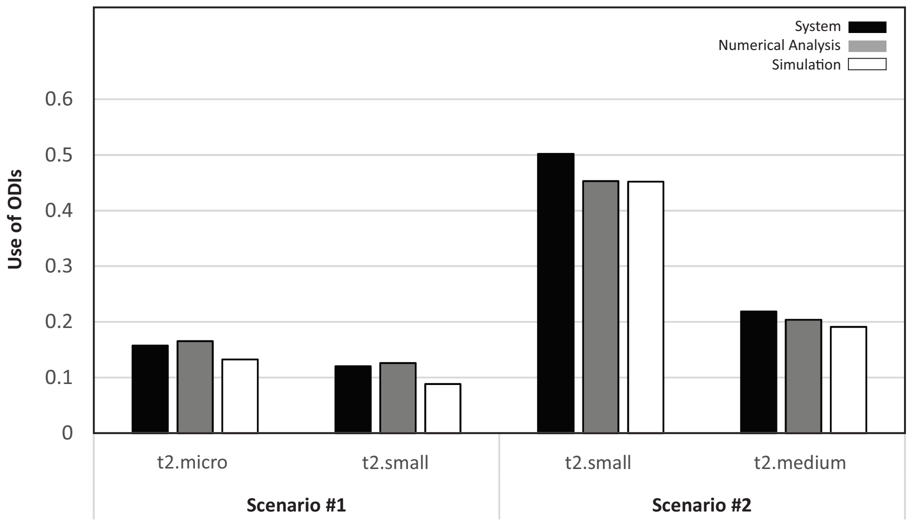


Figure 42 – Use of ODIs for Both Scenarios

Chapter 8 Case Studies

The best strategy is always to be very strong; first in general, then at the decisive point... There is no higher and simpler law of strategy than that of keeping one's forces concentrated... In short the first principle is: act with the utmost concentration.

— On War, *Carl von Clausewitz*, 1780-1831

This chapter presents four case studies to show the applicability of the proposed approach. The first case study evaluates a mobile application for reducing images color. In this case study, we have considered networking requirements, performance, data traffic, and its related costs. The second case study evaluates distinct scenarios for deploying a face recognition system in a public cloud. We evaluated a heterogeneous elastic MCC infrastructure composed of two types of VM instances. We analyzed the costs of using VMs and MRT for each evaluated scenario. In the third case study, we evaluated for a set of configurations the probability of completing the processing of an expected workload in an MCC face recognition infrastructure using CDFs and MTTA. The second and third case studies focused only on the remote infrastructure for supporting offloading. Finally, the fourth case study presents performance, resource consumption, and costs evaluations considering the MCC system deployed in a public cloud and the application running on mobile devices. Networking requirements are considered in this last case study.

8.1 Case Study One: Color Reduction Application

We implement and analyze an image processing mobile application following the principles of method-call computation offloading (KOSTA et al., 2012; CUERVO et al., 2010). The implementation uses a simple client-server RMI architecture. Both client and server side adopt OpenCV (OPENCV, 2018) and *JavaCV* (JAVACV, 2018b). We have implemented the computing vision example of Picture's Colour Reduction (JAVACV, 2018a). This example transforms images by decreasing the number of colors depending on the picture's size. Algorithm 7 demonstrates relevant parts of the evaluated source code.

The evaluation process considered the root method *processImages()* as a benchmark aiming to evaluate the offloading possibilities. *Application_A* resides on the mobile device and its method-calls are dependency free (lines 2 to 3). That is, the methods may be executed in parallel. If the method-call is offloaded to the cloud, it means that the mobile application makes image processing calls to the server by passing one input (original image). In this case, the mobile app connects to one virtual machine and then calls the method *reduceColor* in the server side. Thereafter, the processed image returns to the mobile device. Our case study analyzed and compared all method-calls combinations.

Algorithm 7 Application_A

```
1: function $processImages(img1, img2)$
2:     $a \leftarrow reduceColor(img1)$                ▷ m_call_1
3:     $b \leftarrow reduceColor(img2)$                ▷ m_call_2
4:     return $a$, $b$
5: end function
```

8.1.1 The Evaluation

The case study evaluated distinct offloading scenarios by using the Color Reduction Application. The analysis has focused on the root method *processImages()* of the *Application_A* (see Algorithm 7). As aforementioned, the two method-calls are independent. By making it possible for them to run in parallel on mobile devices or in the cloud. Considering two method-calls and two possibilities as target (device or cloud), four scenarios were exploited, as presented at Table 21. For simplicity, from here down the two method-calls are referenced as *m1()* and *m2()*.

Table 21 – Scenarios to Method-Calls Executions

Scenario	***m1()***	***m2()***
#1	mobile	mobile
#2	mobile	cloud
#3	cloud	mobile
#4	cloud	cloud

The analysis evaluated application's performance and the volume of data traffic generated over a period of one month[1] to 1000 active users and the resulting cost. Each user makes requests at a rate of $1/(25,000,000\ ms)$ which is exponentially distributed.

[1] 30 days

We have used the Amazon's data transfer pricing table. The evaluated SLA establishes a MTTE around 150 s. We have considered that the available bandwidth for offloading (upload) as well as receiving data (download) may vary within a specified limit impacting the desired metrics in each variation (see Table 22). We have calculated the metrics by varying the available bandwidth. Depending on the volume of data sent and received in each user request, a high degree of bandwidth variation may result in a significant impact on evaluated metrics.

Table 22 – Bandwidth Variation (in Megabits/s)

	BW Variation	
Operation	**Minimum BW**	**Expected BW**
Offloading Data	0.5	2.0
Receiving Data	1.0	3.0

As infrastructure, the private cloud Eucalyptus 3.4.0.1 (NURMI et al., 2009) was used with two physical machines (one node and one controller). The physical machines have the following configuration: Intel Core i7-3770 3.4 GHz CPU, 4 GB of RAM DDR3, and 500 GB SATA HD. An Ethernet network is adopted to connect the physical servers through a single switch and one VM of type *m1.medium* (1 CPU, 512MB of RAM, and 10GB Disk). At the mobile device side, a Samsung Galaxy Note 4 was used running the Android 5.1.1 Lollipop. Only the essential system processes were running on it during the experiments.

The testbed executed all method-calls local and remotely (using one VM as offloading target). Through a controlled experiment, the collected input parameters for each method-call were (see Tables 23 and 24): *(i)* the processing time and *(ii)* the number of bytes sent and received. As we can see, processing times are lower when the code is running in the cloud. The experiment executed and monitored 50 times each scenario. At the end of the process, the testbed collected the mean values of them.

Table 23 – Registered Processing Times per Method-Call

Method-Call	**Device (ms)**	**Cloud (ms)**
m1()	81,399	17,256
m2()	131,406	25,476

Table 24 – Transferred Bytes per Method-Call

Method-Call	**Sent Bytes (MB)**	**Received Bytes (MB)**
m1 (Image 1)	2.60	1.07
m2 (Image 2)	4.27	1.68

We generated and refined the SPN models to calculate the desired metrics. We evaluated the four scenarios by combining the four processing time values as well as varying bandwidth for each operation (offloading and receiving data). For this experiment, we considered the maximum number of parallel resources — that is, two target virtual machines.

First, it is necessary to analyze the MTTE of each scenario. Figure 43 shows the SPN model that represents the evaluated application. Transitions *processing_time_m1* and *processing_time_m2* are exponential. They store the mean processing times of the method-calls *m1()* and *m2()*, respectively. Equations 6.4 and 6.5 were used as the mean delay value of the transitions that represent the offloading (upload) and receiving (download) processes, respectively. Table 25 and Figure 44 present the respective MTTEs. Figure 45 shows the impact of the bandwidth variation on the MTTE.

We can extract some conclusions by analyzing the results. Considering that the SLA accepts a MTTE around 150 s, the company may adopt the scenario #2, #3 or #4. In this context, perhaps it has no advantage performing total offloading as in scenario #4. The data volume transferred may be high as the cloud executes all method-calls. Then, the company may decide to execute its mobile application adopting scenarios #2 or #3. Scenario #1 was not impacted by the bandwidth variation, all processing was performed locally. The variation had little effect on the MTTE of the scenario #3. A large portion of the whole execution time was spent on local processing. On the other hand, scenario #4 was the most sensitive in relation to variation. Scenario #4 transferred more data than the other ones. The impact of the bandwidth variation on the performance metrics is high when the amount of data being transferred is large. The performance of scenarios #3 and #4, when executed on the minimum bandwidth requirements, was close to the performance achieved when processing the whole application locally (see Figure 44). In scenarios #2 and #4 the actual bandwidth available for offloading and receiving data had a significant impact on the MTTE of the whole application. In theses scenarios, the higher the available bandwidth, the lower the MTTE. The method *m2()* is the heaviest, so it must be processed in the cloud. If *m2()* is processed locally, maybe it is not advantageous to only perform the offloading of method *m1()* - scenario #3. Therefore, considering a remote infrastructure that attends multiple users, the performance gain is small. Moreover, the company will pay more for offloading only one method-call. Scenario #4 is most suitable but the volume of data transferred in the period is high. Thus justifying the adoption of scenario #2. Developers should carefully evaluate the effect of the available bandwidth variation on the performance metrics.

Service providers should be aware at when their applications are more likely to finish execution. Using CDFs, companies are able to calculate the probability of finishing the application execution within a specified time. For that aim, we have computed CDFs for all evaluated scenarios from $t = 0$ s to $t = 250$ s (see Figure 46). When comparing the CDFs of scenarios #2 and #4 (see Figures 46a and 46b), we can see that there was a great variation in the probability of absorption considering the available bandwidth. According

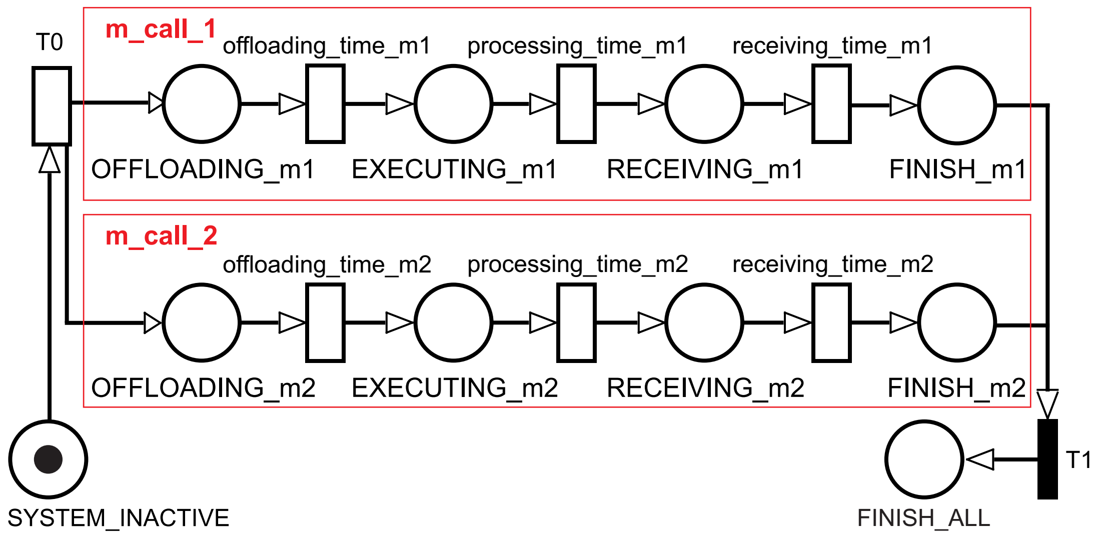


Figure 43 – SPN Model Representing the Method *processImages()*

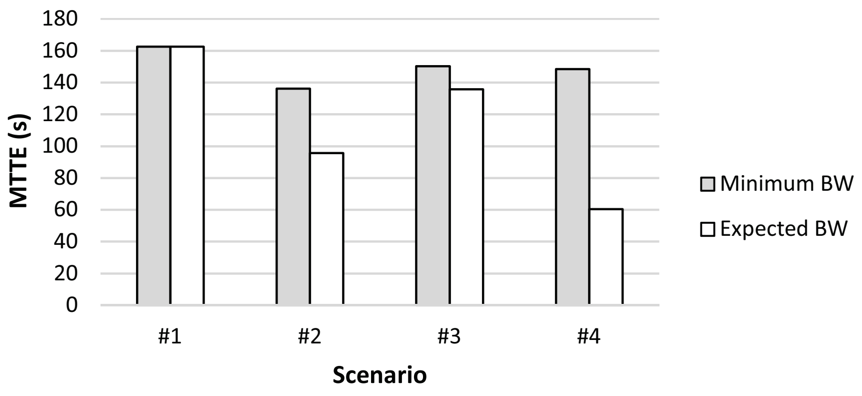
Table 25 – MTTE of the Experiment

	Result (MTTE)	
Scenario	**Minimum Bandwidth (ms)**	**Expected Bandwidth (ms)**
#1	162,544	162,544
#2	136,160	95,608
#3	150,273	135,863
#4	148,577	60,392

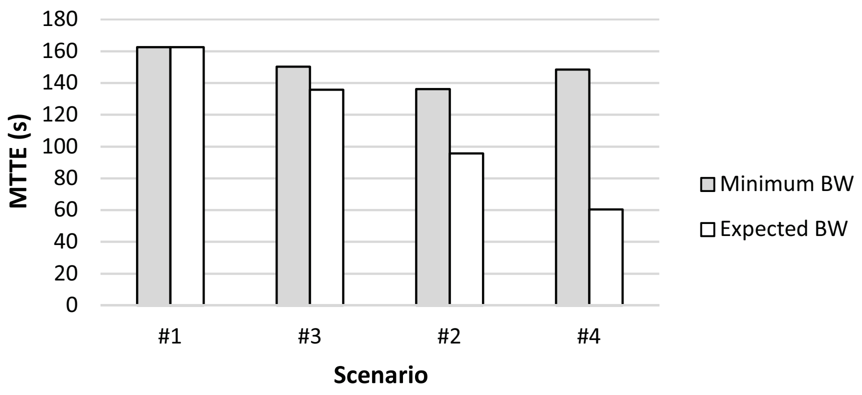
to Figure 46a, the distance between the probabilities of the scenarios #4 and #1 is large in comparison with the distance presented in the Figure 46b. Scenario #4 transfers more data compared to the other ones, thus the bandwidth variation in this scenario has a greater impact on the probability of absorption. When considering only the CDFs of the Figure 46b, the probabilities of absorption are close to each other.

According to Figure 47a, the distance between the probability regarding the scenario #4 is larger than the probabilities regarding #1, #2, and #3. The probability of finishing the execution for the scenario #4 becomes 100 % only after 220 s. Considering all scenarios, in some points, the probabilities for absorption are very close to each other. For example, at some points, scenarios #2 and #3 have similar probabilities of absorption. The difference between their MTTEs is around 40 s (expected BW), and both scenarios only offload one method-call.

Best performance could be observed when adopting the scenario #4 (see Figures 47a and 47b). Scenario #4 has the highest probabilities for finishing execution faster. However, in that scenario more code is offloaded and, consequently, the data traffic may be higher. Thus, it may result in a higher financial cost to the service provider. In the scenario #4, the probability to finish the code execution at 100 s is equal to 89.44 %. The second-best



(a) MTTE Ordered by Scenarios Index



(b) MTTE Ordered by MTTE Values (Expected BW).

Figure 44 – MTTE Results

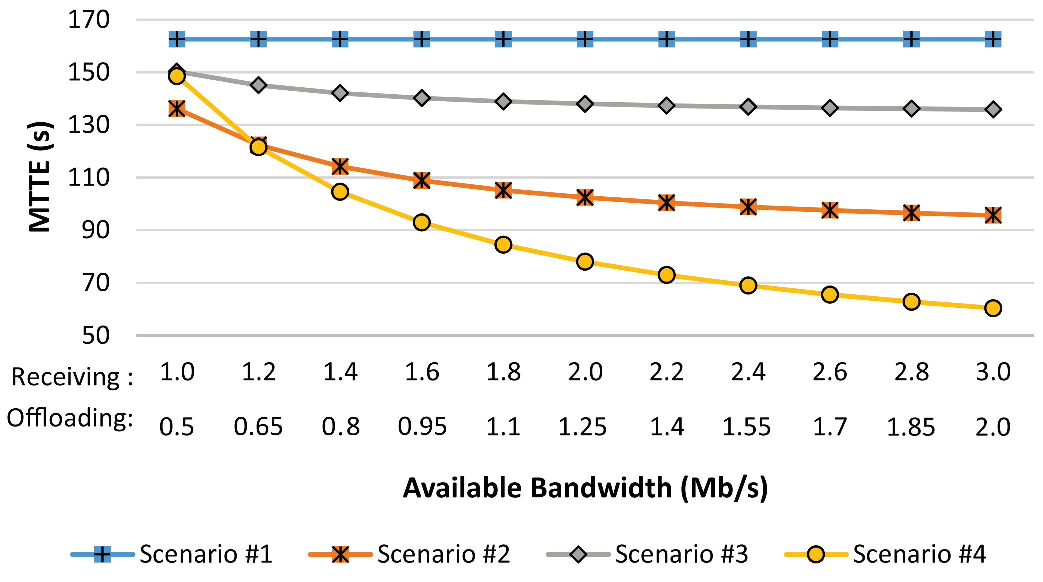
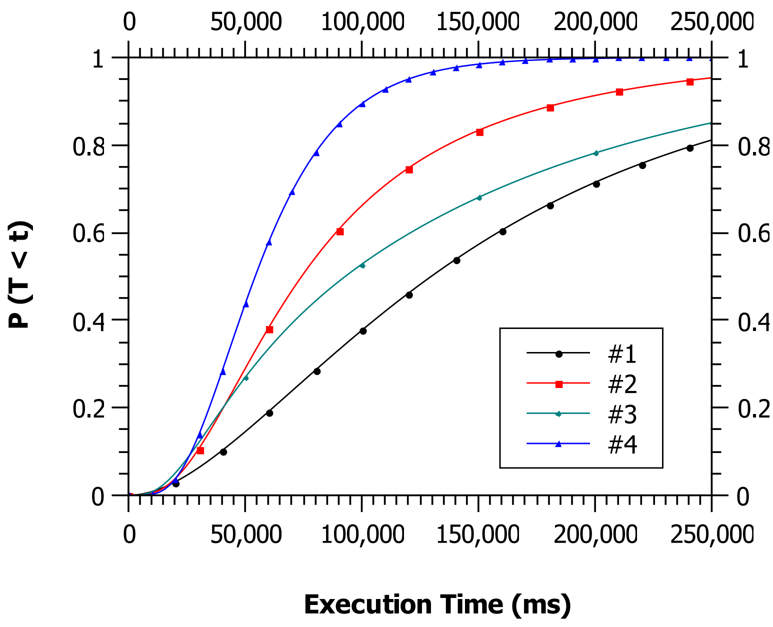
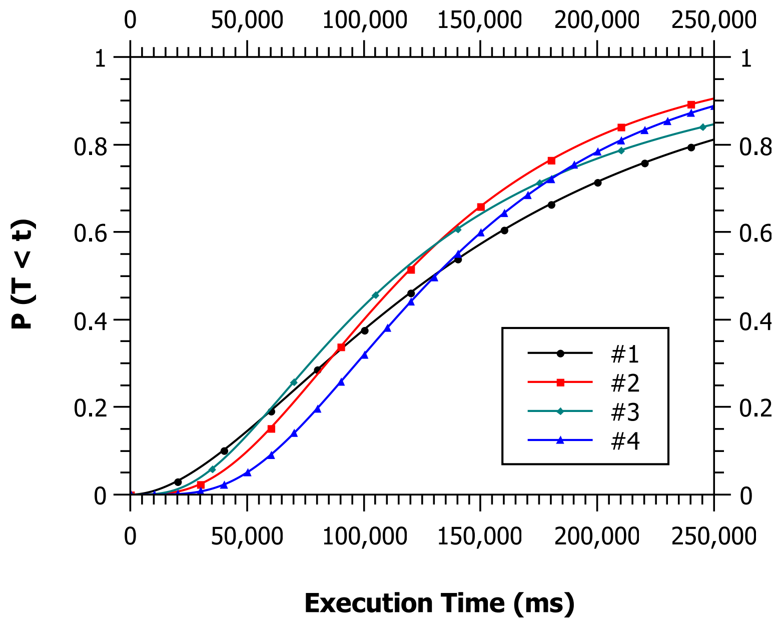


Figure 45 – Available Bandwidth and MTTEs

scenario — scenario #2 — offers 66.14 % of probability to finalize the code execution at 100 s. The difference between their MTTEs is around 35 s, and the probability of absorption is 23.3 % higher for scenario #4. The best and the worst scenario for absorbing



(a) *CDFs* of the Evaluated Scenarios (Expected BW)



(b) *CDFs* of the Evaluated Scenarios (Minimum BW)

Figure 46 – Probability Analysis of the Scenarios based on SPNs

at 100 s — scenarios #4 and #1 — reflect a difference between them of 51.77 %. When evaluating absorbing probabilities considering the minimum bandwidth or bandwidth variation, other considerations may be traced. For example, the absorbing probability of the scenario #4 is the worst when considering the minimum bandwidth and a time interval between 0 s and 130 s.

Now, let us execute an analysis from another point of view. Evaluating the absorbing probability of the faster and the lower scenario that meets the SLA — scenarios #4 and

#3, respectively — considering the expected bandwidth, we can notice a difference of 36.66 % between them to absorbing at 100 s. The difference between their MTTEs is around 75 s. Given that scenario #3 is the most constrained, the service provider should specify the observation of mainly scenario #3 in its Service Level Agreement. If the final user needing the application finishes execution by 100 s, the probability for scenario #3 is always around 52.78 %. Therefore, the service provider could agree to deliver the service by charging low prices due to the offloading scenario limitations.

Table 26 – Absorbing Probabilities at 100 s

	Absorbing Probabilities		
Scenario	**Minimum BW**	**Expected BW**	**Diff**
#1	37.67	37.67	-
#2	39.97	66.14	26.17
#3	43.11	52.78	9.67
#4	31.92	89.44	57.52

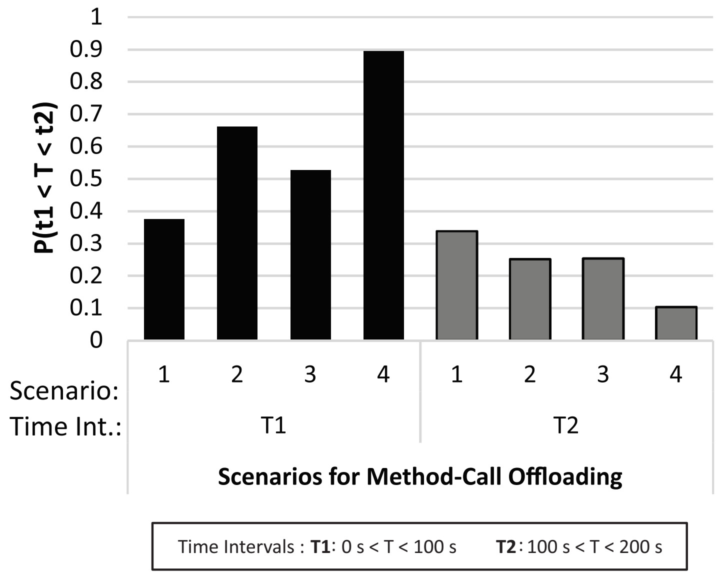
Willing to obtain the probability of absorption, the service provider may consider any time within the range. Probability intervals can also be exploited using CDFs. Aiming to better analyze the applications, Figure 47 depicts the respective probabilities of absorption obeying two intervals and two contexts. These intervals elucidate the cumulative probability considering the difference between two moments. ¨

Considering the context with the expected bandwidth (see Figure 47a), the probability of absorption is high in the T1 range compared to the other one. The mean absorption probability is around 61.51 % and 23.67 % for the T1 and T2 intervals, respectively. As we can see, scenario #4 has the highest probability of absorption in the T1 interval. On the other hand, the scenario #1 has the worst absorption probability in the same interval. With regard to the interval T2, the probability of absorption is higher for the scenario #1 — 33.80 % (see Table 27). Now, the scenario #4 has the lowest absorption probability. Considering the interval T2, maybe it is not suitable to perform the offloading since the scenario #1 has the highest probability when compared to the other one.

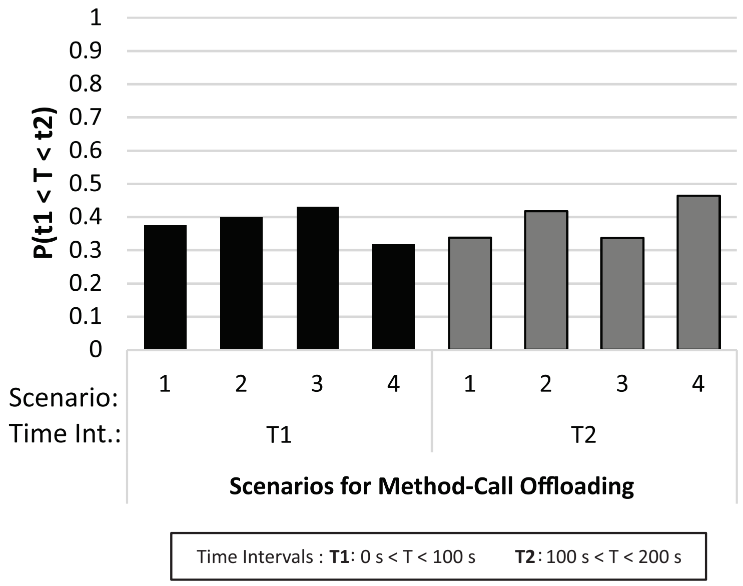
Table 27 – Absorbing Probabilities Considering T2 Interval (Expected BW)

Scenario	**Probability**
#1	33.80
#2	25.16
#3	25.38
#4	10.34

Considering the context with the minimum bandwidth (see Figure 47b), the mean



(a) Probability Interval (Expected BW)



(b) Probability Interval (Minimum BW)

Figure 47 – Probability Intervals

absorption probability is around 38.17 % and 38.92 % for the T1 and T2 intervals, respectively. Now, the probability of absorption is slightly higher in the interval T2. In this context, considering the T1 range, scenario #3 has a higher probability of absorption compared to the other ones. Performing the full-offloading (scenario #4) has the lower probability of absorption in the interval T1. Scenario #2 has similar probabilities in both intervals. Scenarios #1 and #3 have similar probabilities considering only the interval T2. Due to such a myriad of interpretations, the application developer or service provider should also pay attention to probability intervals.

The next step is to identify the expected throughput for each user in each scenario.

For that aim, the SPN model presented in Figure 43 evolved. Now, an arc connects the last transition (*T1*) to the first place (*SYSTEM_INACTIVE*) as illustrated in Figure 48. When the system is not processing any method-calls it corresponds to having a token in the *SYSTEM_INACTIVE* place. The transition *T0* receives the request rate which is equal to $1/(25,000,000\ ms)$. A stationary analysis obtains the throughput using the Equation 8.1. The equation considers both the probability of having tokens at place *SYSTEM_INACTIVE* and the delay between each user request ($request\_time$). Table 28 presents the values obtained.

$$Tp = P(\#SYSTEM\_INACTIVE >= 1) \times \frac{1}{request\_time} \tag{8.1}$$

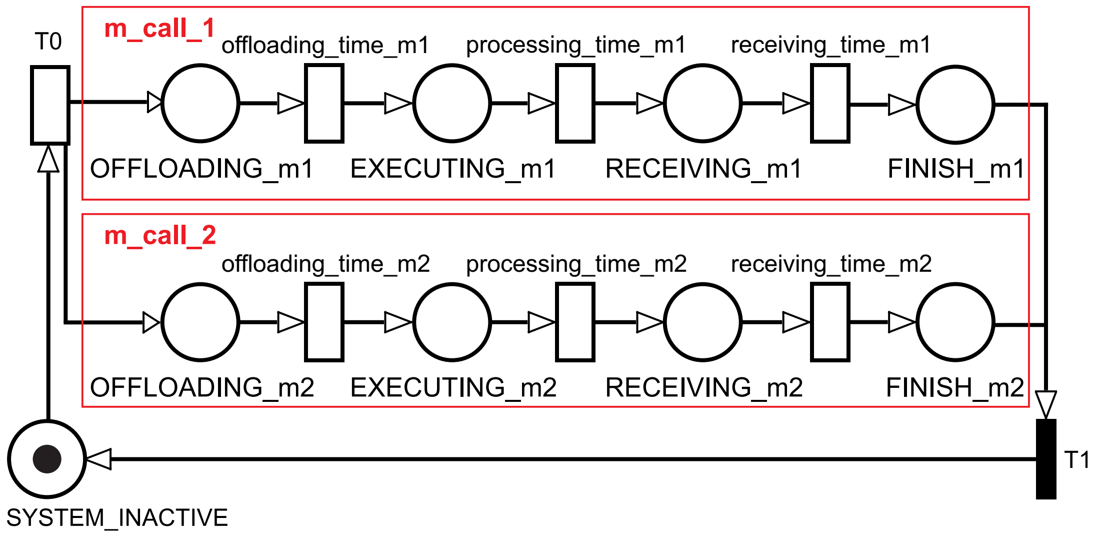


Figure 48 – SPN Used to Calculate Throughput of an Application With Two Parallel Method-Calls

Table 28 – Throughput for Each User in Each Scenario

	Executions/ms	
Scenario	**Minimum Bandwidth**	**Expected Bandwidth**
#1	3.9741×10^{-8}	3.9741×10^{-8}
#2	3.9783×10^{-8}	3.9847×10^{-8}
#3	3.9761×10^{-8}	3.9783×10^{-8}
#4	3.9763×10^{-8}	3.9903×10^{-8}

Now, let us estimate the volume of transferred data generated by each scenario. Data traffic sent from the cloud to device in each request for the method calls *m1()* and *m2()* are respectively: 1.07 MB and 1.68 MB (see Table 24). The combination of these measurements let to obtains the total costs for data traffic considering distinct offloading scenarios. Table 29 presents the number of bytes transferred for processing one user request in each scenario. Table 30 shows the number of requests made by users during the evaluated period. The last column represents the difference between the values obtained with the minimum and expected bandwidth. Using the Equation 6.9 it is possible to find out the TTB. TTB corresponds to the data volume transferred during a period. Table 31 demonstrates the TTB considering the application's throughput. Amazon provides a prices table that defines the cost per transferred gigabyte considering utilization ranges (see Table 8). The cost calculation includes only the outbound data traffic generated by the offloadable methods (see Equation 6.10). Table 32 summarizes the outgoing TTB and its financial cost. Considering the networking requirements, we can see that bandwidth variation had little impact on the amount to be paid for each scenario.

Table 29 – Transferred Bytes for Each Scenario

Scenario	**Sent Bytes (MB)**	**Received Bytes (MB)**	**Total Bytes (MB)**
#1	-	-	-
#2	4.27	1.68	5.95
#3	2.60	1.07	3.67
#4	6.87	2.75	9.62

Table 30 – Executions per Month in Each Scenario

	Executions/month		
Scenario	**Minimum Bandwidth**	**Expected Bandwidth**	**Diff**
#1	103,010	103,010	0
#2	103,118	103,285	166
#3	103,060	103,119	59
#4	103,067	103,430	362

Table 31 – Total Transferred Bytes for Each Scenario

		Total Transferred Bytes (GB) / month		
		Minimum BW	**Expected BW**	**Diff**
Scenarios	**#1**	-	-	-
	#2	599.10	600.06	0.96
	#3	369.95	370.16	0.21
	#4	968.77	972.18	3.41

The costs are inversely proportional to the MTTE, and depending on the application perhaps a lower MTTE is more important than saving money. Figure 49 compares MTTEs and volume of transferred data for all scenarios. In scenario #1 the mobile device processes the two method-calls, whereas all method-calls are offloaded in scenario #4. In scenario #4 the cost is high, and the MTTE is low. In scenario #1, the interpretation is the

Table 32 – Bytes Sent from the Cloud to Mobile Devices and their Costs on AWS

			Minimum Bandwidth	**Expected Bandwidth**	**Diff**
Scenarios	**#1**	**Bytes (GB)**	-	-	-
		Cost ($)	-	-	-
	#2	**Bytes (GB)**	169.15	169.42	0.27
		Cost ($)	15.22	15.25	0.02
	#3	**Bytes (GB)**	108.11	108.18	0.06
		Cost ($)	9.73	9.74	0.01
	#4	**Bytes (GB)**	277.19	278.16	0.98
		Cost ($)	24.95	25.03	0.09

inverse —the MTTE is high, and there is no data traffic cost. The company will pay more for the adoption of the scenario #4. On the other hand, this offloading scenario will offer a higher probability for the application to complete its processing when considering other scenarios. In that case, the higher absorption probability reflects in higher financial cost. It means that the mobile application offloads more code to the cloud and the MCC service provider will pay more for the resource consumption. Consequently, end users may pay more for using the service. The proposed models offer evaluations that can result in decisions that provide money savings and that meet the SLA.

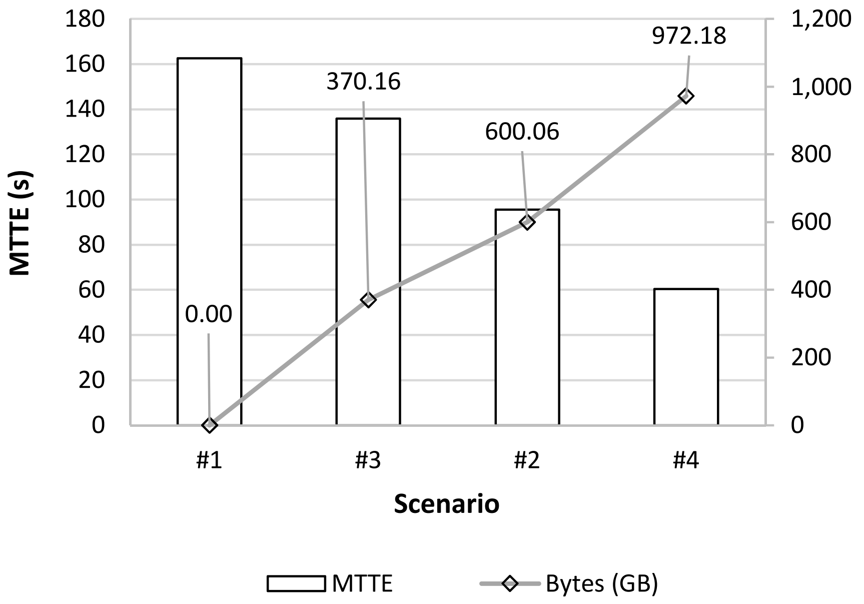


Figure 49 – Comparing MTTEs and Volume of Total Transferred Data (Expected BW)

8.2 Case Study Two: A Heterogeneous MCC Infrastructure in the Cloud

In this case study, we evaluated distinct scenarios for deploying a face recognition system in a public cloud. The purpose of the system is to process face recognition requests sent by remote users. Face recognition determines the match-likelihood of each face to a template element from a database of faces. The widely accepted Eigenfaces method was employed (TURK; PENTLAND, 1991). This method extracts relevant informations in a face image, encodes them, and compares the encoded face with a database of encoded faces called face-space. Algorithm 8 presents the analyzed class *FaceRecognitionService*. The heaviest method, *recognize*, contains two heavy method-calls. The first one, *readFaceBundles*, constructs the face-spaces from a given directory. The second one, *recognizeFace*, performs the comparison between one photo sent by a mobile device and the face-space. This method-call identifies the name of the most similar photo from the face-space and a Euclidean distance in that face.

We have implemented the mobile service in Java language using the face recognition application as a benchmark. The server side adopts OpenCV (OPENCV, 2018) and *JavaCV* (JAVACV, 2018b). Our project uses *OpenCV* as an auxiliary library for processing the faces at the cloud side and the wrapper *JavaCV* to access the *OpenCV*. *OpenCV* and *JavaCV* must be installed in each AMI attached to each SI that composes the infrastructure. It is important to highlight that a system deployed on AWS EC2 may run multiple VM instances on a single AMI when it is necessary multiple instances with the same configuration. The remote system uses a database of 8,000 faces replicated in each SI.

Algorithm 8 FaceRecognitionService

```
1: function $recognize(face)$
2:     ...
3:     $readFaceBundles(database)$
4:     $recognizeFace(face)$
5:     ...
6: end function
```

We have considered a heterogeneous MCC infrastructure composed of two types of VM instances for supporting the face recognition system. We analyzed the costs of using VM instances and the MRT for each scenario. It is important to highlight that the cost of using the front-end instance was not considered in the cost evaluations. Since the infrastructure has only one front-end and it represents a fixed cost (i.e. the front-end is a RI). We have focused on the cost to maintain a set of SIs for requests processing — that represent a scenario for deployment. Table 33 shows the two EC2 instance type that composes all scenarios and the costs of using them as RIs and ODIs. Table 34 demonstrates the parameter values for supporting the generation of scenarios. Scenarios were generated by varying the number of RIs for each type of instance. More specifically, each scenario has at least 1 and at most 4 RIs for each instance type. Table 35 shows the number of RIs for each scenario. The value for scaling out threshold (SOT) is 3 and the value for scaling in threshold (SIT) is 1. It means that the system adds ODIs for service processing when the number of requests in the queue is equal to or greater than four times its processing capacity. As we have stated, the processing capacity refers to the number of requests that the system can process at a time. The processing capacity for each type of VM instance is defined by multiplying the number of SIs that are up and running by the number of parallel requests each instance may process. We get the total processing capacity of the system for a given moment by adding up the processing capacity of each instance type. Considering our case study, the number of SIs running in the system defines its processing capacity at a given time since each VM instance can process only one request at a time (γ parameter). Performing a full factorial design we obtain sixteen scenarios for evaluation.

Table 33 – EC2 Instances Used by all Evaluated Scenarios (AWS, 2018b)

Model	**vCPU**	**Memory (GiB)**	**Reserved (\$/year)**	**On-demand (\$/hour)**
t2.micro	1	1	\$ 59.00	\$ 0.0116
t2.small	1	2	\$ 118.00	\$ 0.023

Table 34 – Parameters for Supporting the Generation of Scenarios

Parameter	**Instance Type 1**	**Instance Type 2**
	Value	**Value**
Instance type	t2.micro	t2.small
RIs	1-4	1-4
Max ODIs (MNODIs)	3	3
SOT[1]	3	3
SIT[2]	1	1
γ_n[3]	1	1
ω_n[4]	1	1

[1] Scaling Out Threshold.

[2] Scaling In Threshold.

[3] Maximum number of concurrent jobs for each n-type instance.

[4]The stepsize for launching n-type on-demand instances in each scale out request.

The analysis evaluated the performance of a remote MCC system and the resulting cost over a period of one year. We have considered that mobile users perform requests at a rate of $1/(5,000\ ms)$ which is exponentially distributed. The evaluated SLA establishes

Table 35 – Evaluated Scenarios

Scenario	**Instance Type 1**	**Instance Type 2**
	t2.micro	**t2.small**
#1	1	3
#2	3	4
#3	3	2
#4	2	2
#5	4	1
#6	3	3
#7	4	3
#8	1	2
#9	1	4
#10	4	2
#11	1	1
#12	2	1
#13	4	4
#14	2	3
#15	2	4
#16	3	1

a maximum MRT around 60 s. The system capacity related to the maximum number of requests in the system consists of 100 requests.

The first step in the evaluation process is to collect the time required to process one face recognition request in each instance type as well as the time the AWS EC2 takes to launch our ODIs and make them available for service processing. As we can see in Table 34, the γ_n parameter for each VM instance has one as its value. It means that each instance in the system must process only one request at a time. In other words, there is no requests processing occurring concurrently in the instances. Through a controlled experiment our testbed collected the required times. The testbed executed all requests remotely using one instance of each type as the target in order to obtain the processing time. The experiment executed and monitored 80 times both the workload in each instance and the requests to EC2 for scaling the system out. Our testbed collected the mean values of them at the end of the process. Table 36 presents the time the EC2 spent to launch our ODIs and make them available in the system. Table 37 shows the mean processing time registered for each instance type to process one request. As we can see, the difference in the processing times in each instance type is about 4 s.

We generated and refined the SPN models to calculate the desired metrics. Figure 50 shows the refined SPN model that represents the evaluated MCC infrastructure to be

Table 36 – Time Required to Launch our ODIs

Instance Type	**Time for Launching (s)**
t2.micro	175.06
t2.small	204.57

Table 37 – Processing Time Registered for Each Instance Type

Instance Type	**Processing Time (ms)**
t2.micro	23,638
t2.small	19,075

deployed in a public cloud. The model considers two types of VM instances running in the same infrastructure. Transition *Arrival* receives the interarrival time — $5,000\ ms$ — and the place *Capacity* receives the capacity of the system — 100 requests. Each parameter γ_n and ω_n receives 1 as value. Transitions PT_1 and PT_2 are exponential. They store the mean processing times to process a single request in the instance type *t2.micro* and *t2.small*, respectively. Places $ACPR_1$, and $ACPR_2$ receives the number of tokens related to the number of RIs of the scenario (i.e. $\gamma_1 = 1$ and $\gamma_2 = 1$). Places $ODIAL_1$ and $ODIAL_2$ receives the number of tokens related to the maximum number of ODIs (MNODIs) the system may use. Transitions $LODI_1$ and $LODI_2$ receive the time to launch one ODI of type *t2.micro* and *t2.small*, respectively. We evaluated the sixteen scenarios by varying the number of RIs (see Table 35).

Let us describe how to obtain the performance metrics for each scenario. Equation 8.2 obtains the expected number of requests waiting to receive service. Equation 8.3 gets the expected number of requests that receive service considering the two SIs group in the infrastructure. Finally, Equation 8.4 obtains the MSS. MSS corresponds to the mean number of requests in the system. Equation 8.5 obtains the MRT of the system. This equation considers the arrival rate (AR) — $1/(5,000\ ms)$ — and the probability of having tokens at the place *Capacity*. That is, the probability of the system being able to handle the request received. Equation 8.6 obtains the use of ODIs by the system. We obtain the metric of *"expected ODI usage"* (EODIU$_n$) by subtracting the maximum number of ODIs from each instance type n that the system may use (i.e. 3 in this case study) by the expected number of tokens in the place $ODIAL_n$. A stationary analysis in the model obtains MSS, MRT, and EODIU$_n$.

$$Requests_{queue} = \sum_{i=1}^{n} P(m(Queue) = i) \times i \tag{8.2}$$

$$Requests_{service} = \sum_{type=1}^{n} \left(\sum_{i=1}^{n} P(m(RRS_{type}) = i) \times i \right) \tag{8.3}$$

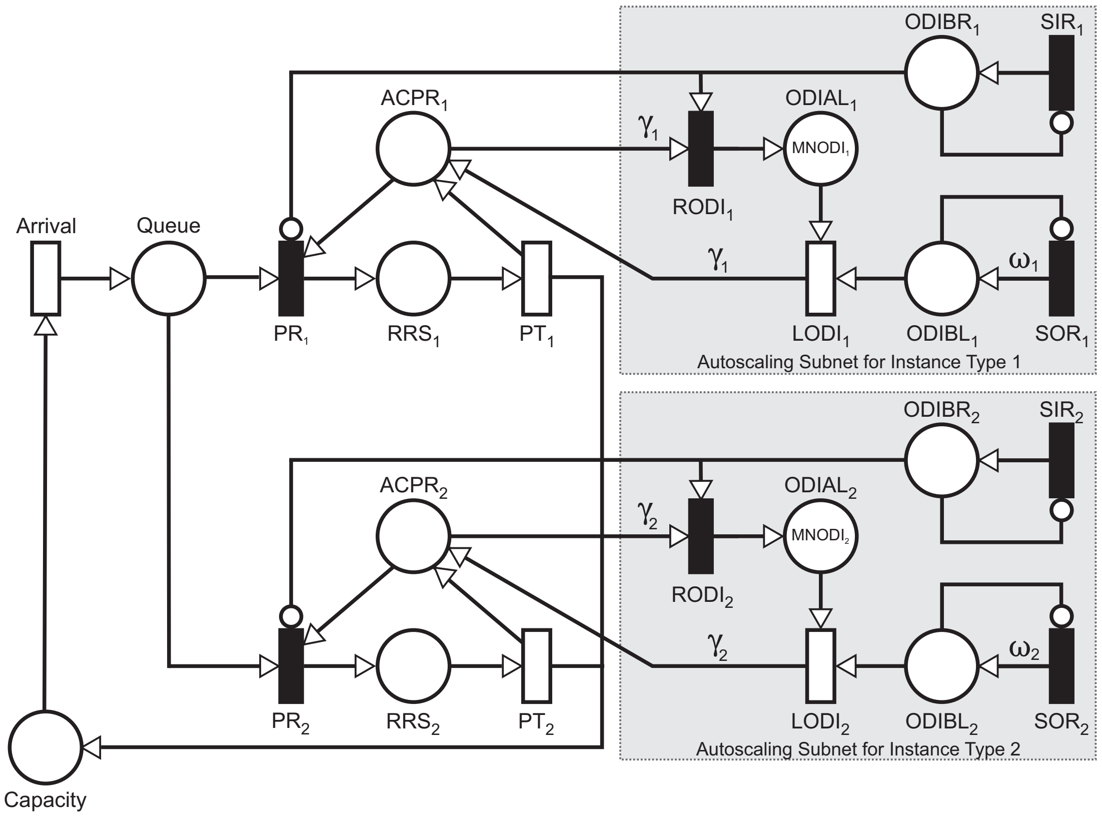


Figure 50 – Public Cloud MCC Infrastructure Model

$$MSS = Requests_{queue} + Requests_{service} \tag{8.4}$$

$$MRT = \frac{MSS}{AR \times (1 - P(m(Capacity) = 0))} \tag{8.5}$$

$$EODIU_n = 3 - \left(\sum_{i=1}^{n} P(m(ODIAL_n) = i) \times i\right) \tag{8.6}$$

Now, using Equation 8.7 it is possible to obtain the total cost of using ODIs of type n (TCODI$_n$). The cost calculation must be performed for each type of ODI in the infrastructure. To obtain $TCODI_n$ it is necessary multiplying: (*i*) the expected ODI usage ($EODIU_n$); (*iii*) the period of time the system is up and running (*Hours*); and finally (*ii*) the price per hour for using the evaluated ODIs ($PricePerHour_n$).

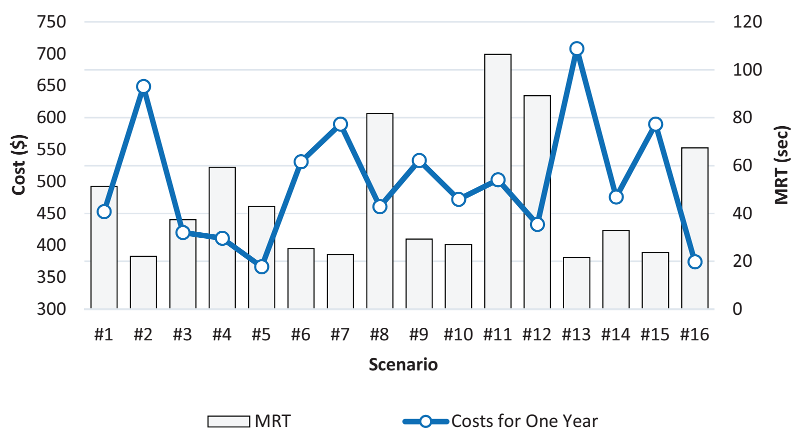
$$TCODI_n = EODIU_n \times Hours \times PricePerHour_n \tag{8.7}$$

Now, let us analyze the MRT and cost for each scenario. Table 50 and Figure 51 present the respective MRTs. We can extract some conclusions by analyzing the results. As we can see in Figures 51b and 51c the costs are not inversely proportional to MRTs. Scenario #11 offers the worst performance (106 s) considering all evaluated scenario and its cost is \$ 30 dollars higher than the cost for scenario #10. However, the difference between the MRTs is almost four times lower for scenario #10 ($\equiv$ 27 s). Scenario #11 is the most intensive scenario regarding the use of ODIs. Table 38 depicts that the use of ODIs composes 64.7 % of the cost for this scenario whereas the use of RIs composes 99.95 % of the cost for scenario #10. Scenario #11 uses only one RI of type *t2.micro* and one *t2.small*, and in scenario #10, the configuration uses six RIs. It means that the use of ODIs makes up most of the cost for scenario #11 because the number of RIs is low considering the workload. In this case, the analyst may add more RIs in the infrastructure to try to reduce the cost. The MRT for scenario #1 is 30 s lower than scenario #8, but both scenarios offer an equivalent cost. The difference between scenario #16 and #11 is almost 40 s and the cost of #11 is $ 118 higher. Let us now evaluate the scenarios that offer the worst and best performance. Considering scenarios #8 and #10, the cost difference is almost $ 10, but the performance of scenario #10 is three times faster than #8. Scenario #1 is twice as fast as scenario #11 and #11 is $ 50 dollars higher than #1. Scenario #11 is intensive in the use of ODIs and the use of ODIs is almost twice as large in comparison to RIs. Scenario #14 is more than three times faster than #11 and the difference between the cost is $ 25. Scenario #13 uses eight RIs in its configuration and the use of RIS composes the entire cost for instance usage. It is important to highlight that the time required to launch ODIs may degrade the system performance — scenario #11. The use of ODIs for a long period of time may offer a higher cost compared to the use of RIs. Our models enable an analyst to perform a myriad of performance and cost evaluations.

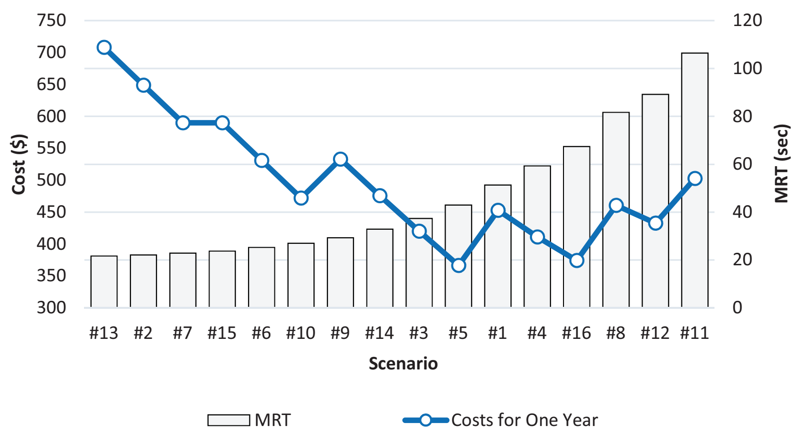
Table 38 – MRT and Costs for Each Scenario

Scenario	**MRT (ms)**	**Monetary Costs for One Year**		
		On-demand $	**Reserved $**	**Total $**
#1	51,350	39.75	413	452.75
#2	22,128	0	649	649.00
#3	37,373	7.00	413	420.00
#4	59,247	57.13	354	411.13
#5	42,985	12.39	354	366.39
#6	25,251	0	531	531.00
#7	22,896	0	590	590.00
#8	81,716	165.70	295	460.70
#9	29,317	2.19	531	533.19
#10	26,998	0.25	472	472.25
#11	106,380	325.80	177	502.80
#12	89,205	196.85	236	432.85
#13	21,676	0	708	708.00
#14	32,893	3.91	472	475.91
#15	23,746	0	590	590.00
#16	67,424	79.10	295	374.10

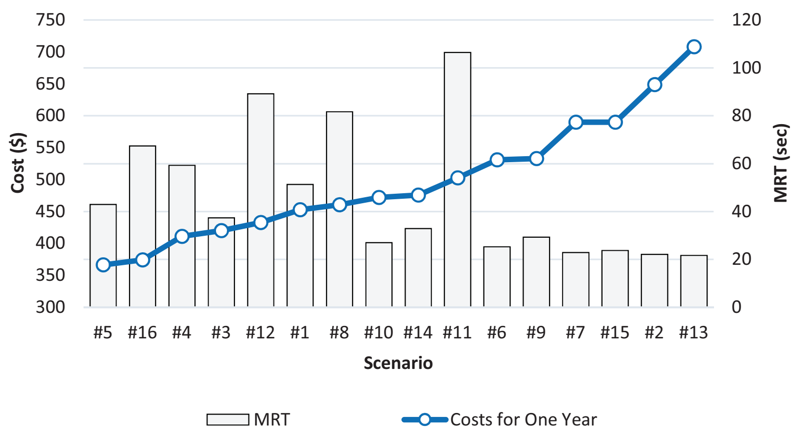
Considering that the SLA accepts a maximum MRT around 50 s, the company may



(a) MRTs and Costs Ordered by Scenarios Index



(b) MRTs and Costs Ordered by MRTs Values



(c) MRTs and Costs Ordered by Costs Values

Figure 51 – MRTs and Costs Results for all Scenarios

adopt the configuration #2, #3, #5, #6, #7, #9, #10, #13, #14 or #15 as its remote MCC configuration. Table 39 shows the scenarios that comply with the SLA performance. Figure 52 compares MRTs and costs for those scenarios that have attended the SLA. Looking at the graph we can see that the tendency is that as the cost decreases while the MRT increases. Scenario #13 has eight RIs, whereas scenario #5 has five ones. In scenario

#13 the cost is high, and MRT is low. In scenario #5, the interpretation is the inverse — MRT is high, and cost is low. Perhaps there may be no advantage in choosing scenario #1 when considering scenarios #7 and #13. The difference between their MRTs and costs are 1 s and $ 118 (in one year), respectively. On the one hand, a response time of one second more may do not negatively impacts on users' QoE, and a small IT company may save one hundred and twenty dollars — in one year, approximately. This difference may become significant when the developer takes into account other resources consumed, such as data traffic. It is important to highlight that both scenarios do not have ODIs consumption — #7 and #13. It means that the number of RIs is sufficient to process all requests without the need for requesting additional processing capacity. In other words, we can say that this configuration prevents the MCC system from reaching the predefined SOT. Scenarios #2 and #7 have similar performance (22 s) because both scenarios have seven RIs in their infrastructure, but the difference between their cost is almost $ 60 dollars for one year. The *t2.small* instances cost twice the value of the *t2.micro* instances and, considering our workload, the performance gain is only 4 seconds for a *t2.small* instance. Scenarios #10 and #14 have equivalent costs and the difference in their MRTs is almost 6 s. For both scenarios, the use of RIs composes almost 100 % of the costs. It means that the number of RIs supports the workload well and the system does not reach the defined threshold. The same occurs with scenarios #6 and #9. The difference between their MRTs are 4 s and both have equivalent costs. As the performance of the two types of instances — *t2.micro* and *t2.small* — was similar considering our workload we highlight that all configurations that have at least five RIs have reached the performance defined in the SLA.

Table 39 – MRT and Cost of Scenarios that Comply with the SLA Orderered by MRT Values

Scenario	**MRT (ms)**	**Monetary Costs for One Year**		
		On-demand $	**Reserved $**	**Total $**
#13	21,676	0	708	708.00
#2	22,128	0	649	649.00
#7	22,896	0	590	590.00
#15	23,746	0	590	590.00
#6	25,251	0	531	531.00
#10	26,998	0.25	472	472.25
#9	29,317	2.19	531	533.19
#14	32,893	3.91	472	475.91
#3	37,373	7.00	413	420.00
#5	42,985	12.39	354	366.39

Let us execute another evaluation from another perspective. The approach highlighted a difference around $ 340 dollars in one year when we evaluate the most economical and the most expensive scenario that meets the SLA – #5 and #13. The company will pay more for the adoption of the scenario #13. On the other hand, this scenario will offer a

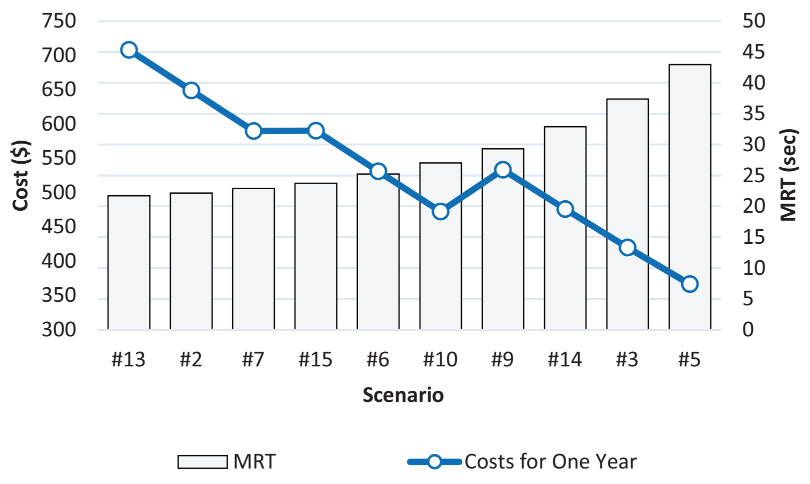


Figure 52 – MRT and Cost of Scenarios that Comply with the SLA

higher probability for the MCC system to complete the request processing within 50 s as defined in the SLA. In that case, the higher probability may reflect in higher financial cost and depending on the application perhaps a lower MRT is more important than saving money. As a result, end users may pay more for using the MCC service. Projecting this amount over a period of three years we obtain a difference around $ 1,000 dollars.

Let us consider a MRT less than 30 s. Then, the company may decide to deploy its MCC system adopting scenarios #2, #6, #7, #9, #10, #13 and #15. In another context the company when comparing the MRT of scenarios #10 and #13, since these scenarios have similar performances, it may choose the scenario #10. Therefore, considering a remote MCC infrastructure that needs to attend multiple users, the performance gain is minimal by comparing these scenarios (#10 and #13). Moreover, the company will pay more for gain 5 s — almost $ 235 dollars more.

Figure 53 depicts that the costs of using ODIs are proportional to MRTs. It is important to highlight that for all scenarios that comply with the SLA — 50 s — the consumption of ODIs was close to 0 —#2, #3, #5, #6, #7, #9, #10, #13, #14 and #15. In other words, the higher the MRT, the higher the consumption of ODIs. The cost for the fastest scenario #13 is composed only by the use of RIs. On the other hand, in the slowest scenario #11 the use of ODIs composes 64.79 % of its total cost. Perhaps the time spent to launch the ODIs may degrade the system performance. An MCC service provider needs to pay attention to this. Considering this, an analyst may increase the stepsize (ω_n) in order to add more ODIs at the same time. Figure 54 depicts the relation between MRTs and costs for the use of RIs. As we can see, the cost of RIs tends to decrease the higher the system response time. The number of RIs to support the expected workload in the infrastructure is small when the cost of using ODIs is high. Figure 55 depicts the relation between MRTs and the cost of RIs and ODIs for all scenarios. We may notice that there is a tendency for MRT to increase as the use of ODIs increases.

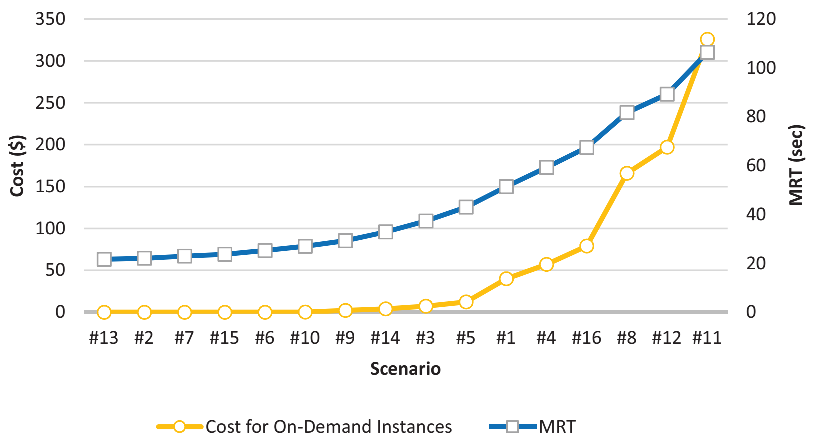


Figure 53 – Comparing MRTs and Costs for On-Demand Instances

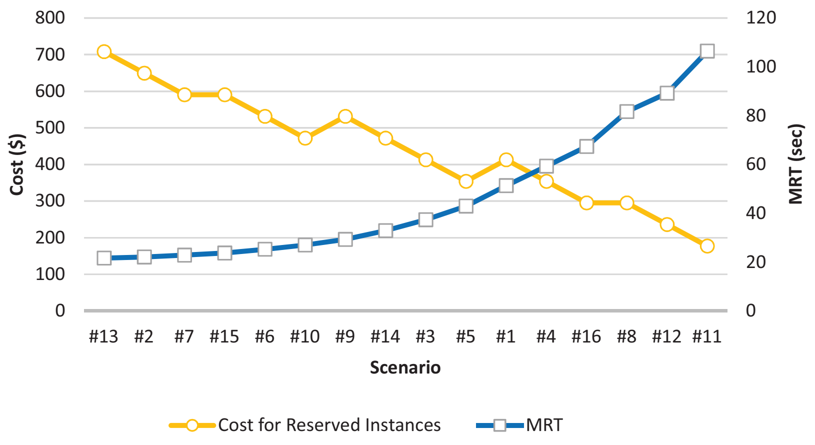


Figure 54 – Comparing MRTs and Costs for Reserved Instances

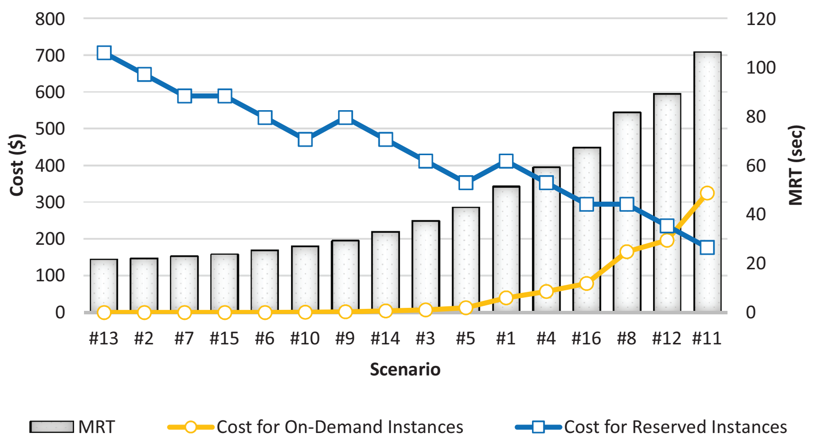


Figure 55 – Comparing MRTs and Costs for ODIs and RIs

Our SPN-based modeling strategy makes it possible to evaluate the impact of the threshold change on performance and cost metrics. Let us now investigate whether the change in the SOT may impact the MRTs and costs by considering our workload. For that aim, we have changed the SOT from 2 to 5 and evaluated its impact on MRT and costs for all scenarios. Table 40 shows the performance and costs by considering the upper and lower SOTs (i.e. 2 and 5). Figure 57 highlights the impact on performance for all scenarios considering the four SOT value. As we can see, for some scenarios, changing the threshold may vary the MRT. However, for others, the performance remains the same. For example, the MRTs for scenarios #2, #6, #7, #9, #10, #13 and #15 remain the same or change slightly. On the other hand, there exist some scenarios in which the change of threshold had a significant impact on the MRT at a low cost. Figure 56 depicts the performance difference for scenarios where the difference in MRT is greater than 5 seconds considering the upper and lower SOTs.

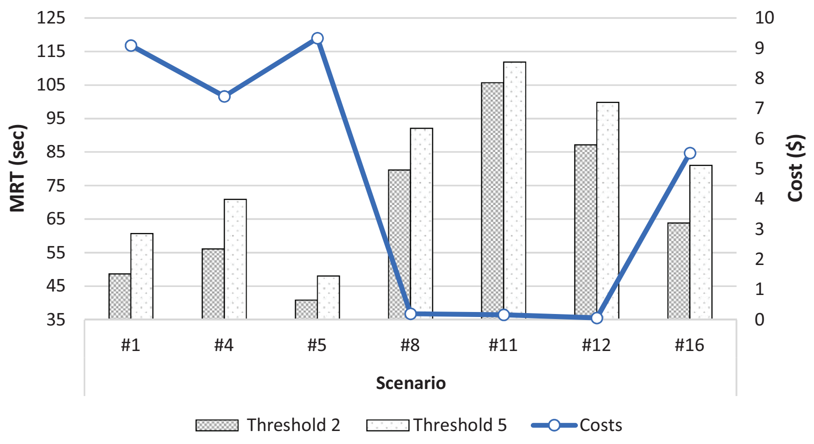


Figure 56 – Difference in MRTs and Costs Considering Thresholds for Scaling Out

Let us take some considerations. The performance for scenario #1 complies with the SLA — 50 s — when we adopt the SOT with value 2. However, by adopting other thresholds, the MRT increases and performance becomes out of the SLA. More specifically, changing SOT from 2 to 5 the MRT becomes 12 s higher and it only decreases the cost in $ 9 dollars. That is, an economy of $ 0.75 dollar for a month is too small and the gain of 12 s in MRT represents a great impact on performance. Changing SOT from 5 to 2 for scenario #12 the MRT decreases — 12.5 s —- and the costs remains the same —- $ 0.06 a year. The same occurs in scenario #8 by decreasing SOT from 5 to 2. It represents a gain of 12.4 s in performance and the cost only increases by $ 0.19. The most impactful scenario is the #16. The MRT increase in 17 s and the cost only decrease by $ 5 per year when we change SOT from 2 to 5. The threshold changing may only impact on performance when the evaluate scenarios are intensive in using ODIs. For example, the cost of using ODIs in scenarios #2, #6, #7, #10, #13, and #15 is less than $ 1 dollar. For these scenarios, the performance remains the same when changing the threshold. Based on our models, an analyst may perform a myriad of evaluations.

We evaluate a simple infrastructure for supporting an MCC system with only a few VM instances. However, in a real-world context, an MCC system can consist of hundreds

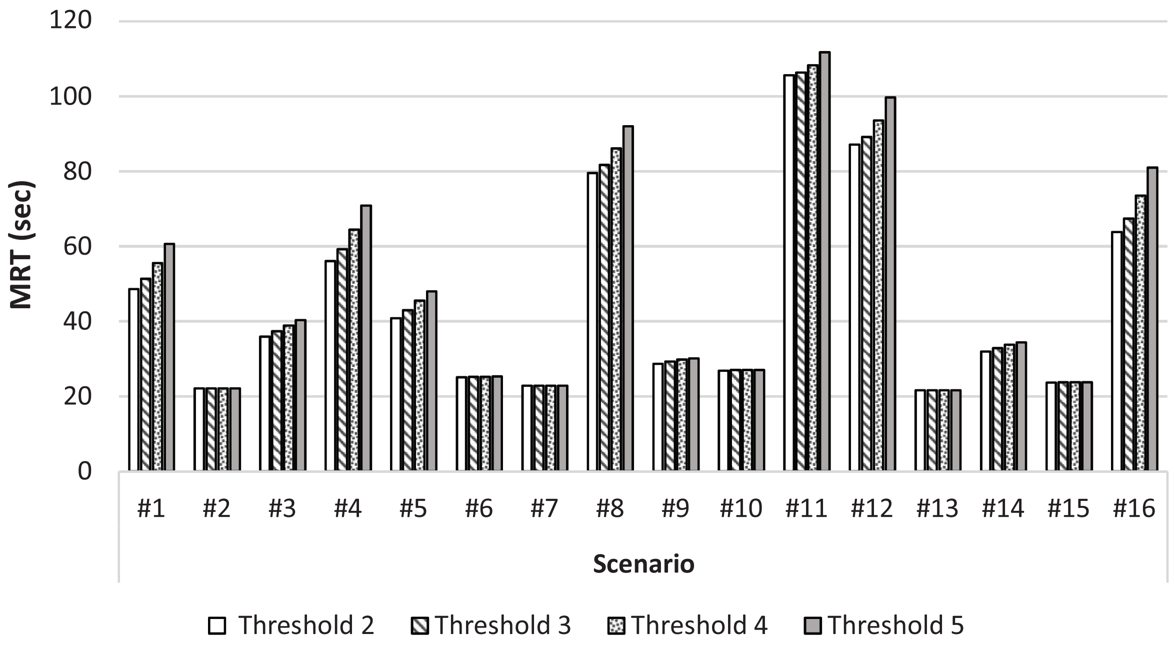


Figure 57 – MRTs and Thresholds for Scaling Out

Table 40 – MRTs and Costs Considering Two Thresholds for Scaling Out

Scenarios	**Threshold 2**		**Threshold 5**		**DIFF**	
	MRT (ms)	**Cost ($)**	**MRT (ms)**	**Cost ($)**	**MRT (ms)**	**Cost ($)**
#1	48,642	455.42	60,703	446.34	12,060	9.08
#2	22,126	649.00	22,128	649.00	0,002	0
#3	35,971	422.79	40,294	416.64	4,323	6.15
#4	56,078	413.31	70,907	405.90	14,829	7.40
#5	40,858	369.85	48,046	360.52	7,189	9.33
#6	25,137	531.46	25,259	531.00	0,122	0.45
#7	22,892	590.00	22,896	590.00	0,004	0
#8	79,648	460.79	92,104	460.59	12,456	0.19
#9	28,732	534.90	30,072	531.46	1,341	3.43
#10	26,847	472.79	27,084	472.00	0,237	0.79
#11	105,684	502.67	111,855	502.83	6,171	0.16
#12	87,175	432.89	99,792	432.96	12,617	0.06
#13	21,676	708.00	21,676	708.00	0	0
#14	31,986	478.10	34,372	473.16	2,385	4.94
#15	23,699	590.27	23,763	590.00	0,065	0.27
#16	63,823	375.83	81,015	370.30	17,192	5.53

or thousands of VM instances. If the MCC service provider does not properly evaluate the trade-off between performance and costs, it may result in financial losses and performance degradation. Projecting this amount over a period of one year or more we may obtain a difference of thousands of dollars.

8.3 Case Study Three: Evaluating Absorbing Probabilities in the Cloud

MCC service providers should be aware at when their systems are more likely to finish execution. As we mentioned in this work, companies may calculate the likelihood of completing the mobile application execution or a batch of requests on a remote MCC infrastructure within a specified period when using CDFs. MTTA is another metric evaluated when a service provider needs to know the average time to conclude processing considering a specific system configuration and an expected workload. For this purpose, we need to refine our original SPN-based model that represents the remote MCC infrastructure, as shown in Figure 58. After that, an evaluator can configure in the model the system parameters related to a specific configuration and calculate CDF and MTTA.

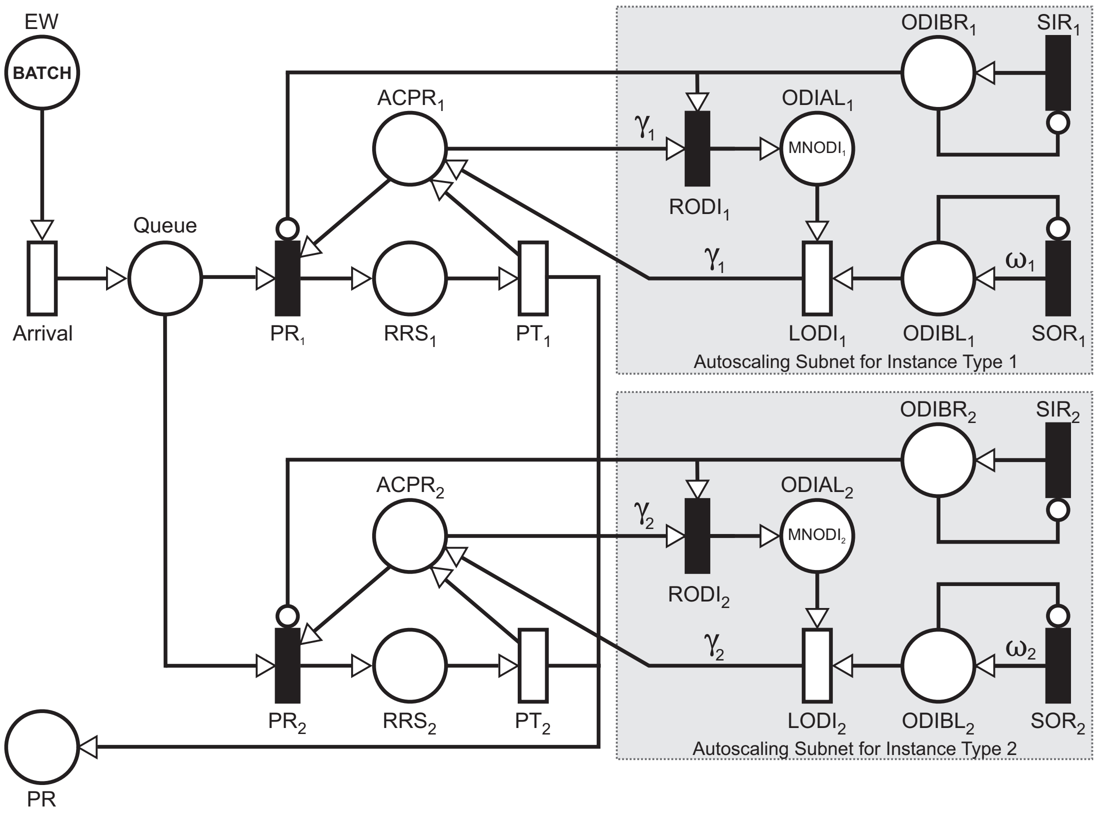


Figure 58 – Public Cloud MCC Infrastructure Model Refined to Calculate CDFs and MTTAs

Let us see the changes we made in the original model for supporting transient evaluations. Now the model has a new place named *EW* (Expected Workload). This place receives the number of tokens related to the number of requests to be processed by the evaluated MCC system considering a specific configuration. As we can see, the place *EW* is the only input place of the *Arrival* transition. Place *Capacity*, in our original model, becomes the place *PR* (Processed Requests) and there is no output arc connecting it to any transition. Place *PR* must be the output place for all PT_n transitions. It means that requests are processed and stored in this new place. A new variable named *BATCH* needs to be defined and it receives the number of expected requests the system needs to process. Thus, the place *EW* receives the value of the variable *BATCH* as its initial number of tokens. After all these structural changes, an analyst needs to define the parameters of the evaluated scenario in the model and perform a transient evaluation on it in order to obtain MTTA and CDFs. A transient evaluation obtains the CDFs based on the time interval defined by the evaluator and evaluating the metric *P{#PR = BATCH}*. During the metric evaluation, the place *PR* represents the absorbing state in the underlying CTMC when the number of tokens in it is equal to the value of the variable *BATCH*. This metric evaluates the probability for the *PR* place to have the number of tokens defined in the *BATCH* variable as time passes within the predefined time interval. After all the steps mentioned above, an analyst may compute transient metrics considering a myriad of scenarios.

We evaluated distinct scenarios for deploying a face recognition system in a public cloud. We have considered a heterogeneous MCC infrastructure composed of two types of VM instances. The face recognition system uses a database of 8,000 faces replicated in each service instance. The first step in the evaluation process is to collect the time required to process one face recognition request in each instance type as well as the time the AWS EC2 takes to launch our ODIs and make them available for service processing. As we can see in Table 41, the γ_n parameter for each VM instance has one as its value. It means that each instance in the system must process only one request at a time. Through a controlled experiment our testbed collected the required times. The experiment executed and monitored 80 times both the workload in each instance and the requests to EC2 for scaling the system out. Our testbed collected the mean values of them at the end of the process. Table 42 shows the mean processing time registered for each instance type to process one request. Table 43 presents the time the EC2 spent to launch our ODIs and make them available in the system.

We have computed CDFs, MTTAs and costs by taking into account five configurations for deploying the system (see Table 41 and Figure 59). The evaluation considered the time required to process a set of 2,000 requests. The evaluated SLA establishes a maximum average processing time around 3,000 s (50 minutes). First, we have computed the probabilities from t = 2,000 s to t = 4,000 s for all scenarios (see Figure 59a). After, as shown by Figure 59b, the probabilities were computed from t = 2,400 s for t = 3,000 s only for the scenarios that meet the SLA's performance — #1, #2, #3 and #5. Table 44 shows the mean time to process the 2,000 requests and the related cost for using RIs over a year considering all scenarios. Figure 60 depicts the relation between MTTAs and costs.

According to Figure 59a, the distances between the probabilities regarding the scenario #2 is larger than the probabilities regarding the scenarios #1, #3, #4 and #5. The probability of finishing the processing for the scenarios that attend the SLA — #1, #2,

Table 41 – Remote MCC Infrastructure Scenarios for Absorbing Probabilities Evaluation

		Instance Type 1				Instance Type 2				SOT[2]	SIT[3]	ω[4]
		Type	RIs	ODIs	γ_1[1]	Type	RIs	ODIs	γ_2[1]			
	#1	t2.medium	1	1	1	t2.large	4	1	1			
	#2	t2.medium	3	1	1	t2.large	3	1	1			
Scenario	#3	t2.medium	4	1	1	t2.large	1	1	1	4	1	1
	#4	t2.medium	1	1	1	t2.small	8	1	1			
	#5	t2.micro	5	1	1	t2.small	8	1	1			

[1] Maximum number of concurrent jobs for each n-type VM instance.

[2] Scaling Out Threshold.

[3] Scaling In Threshold.

[4]The stepsize for launching n-type on-demand instances in each scale out request.

Table 42 – Processing Time Registered for Each Instance Type

Instance Type	Processing Time (ms)
t2.micro	23,638
t2.small	19,075
t2.medium	10,752
t2.large	8,761

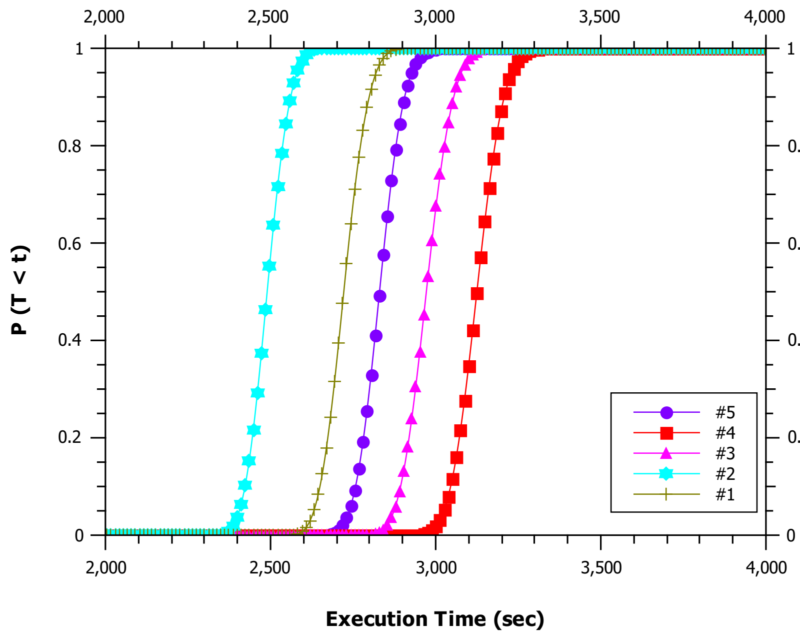
Table 43 – Time Required to Launch our ODIs

Instance Type	Time for Launching (s)
t2.micro	175.06
t2.small	204.57
t2.medium	239.46
t2.large	318.34

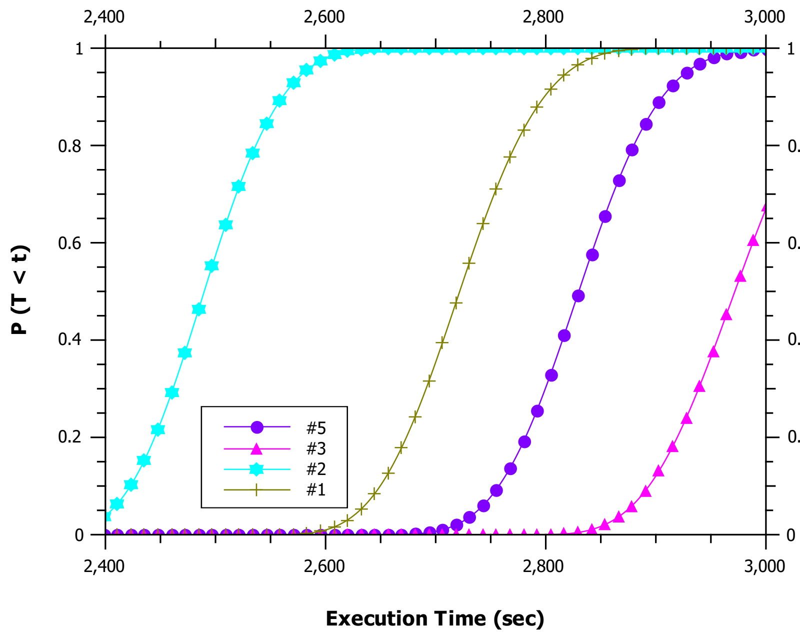
Table 44 – MTTA and Cost for each Scenario

Scenario	MTTA (s)	Cost ($)
#1	2,739	2,115
#2	2,505	2,115
#3	2,990	1,179
#4	3,134	1,239
#5	2,834	1,410

#3 and #5 — becomes 100 % for all of them only after 3,000 s, whereas for the scenario that does not attend the SLA — #4 —, this happens around 3,400 s. Table 45 shows the probability for absorption at 3,000 s considering all evaluated scenarios. As we can see, the system is capable of completing workload processing at 3,000 s considering scenarios



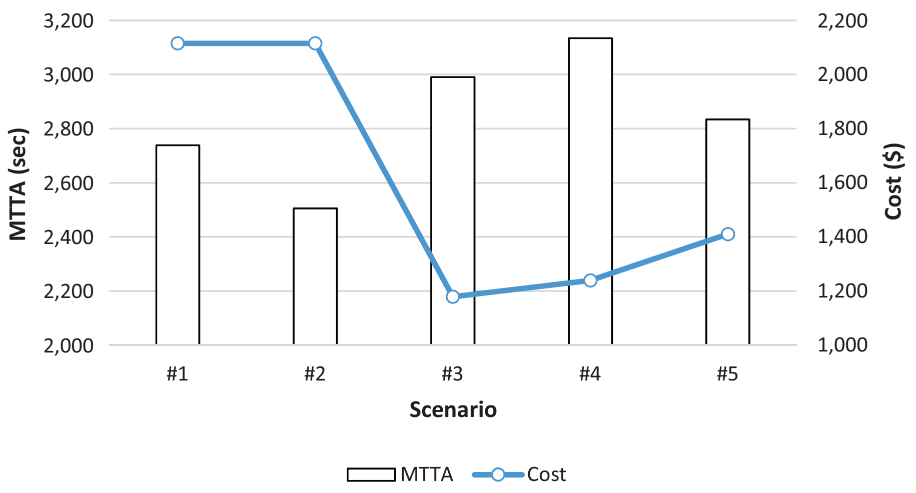
(a) CDFs of the Evaluated Scenarios



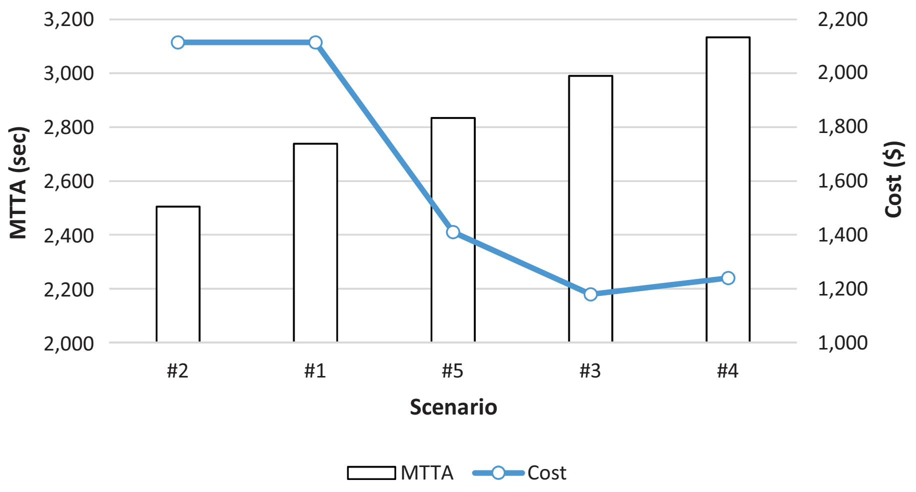
(b) CDFs of the Scenarios that Meet SLA Performance Terms

Figure 59 – Probability Analyses of the Scenarios Based on SPNs Transient Evaluations

#1 and #2. Scenario #5 completes processing of the entire workload 100 s after scenarios #1 and #2 have ended. The cost for scenario #3 is \$ 60 dollars less than scenario #4, but the MTTA for #3 is 144 s smaller than #4 (see Table 44). And scenario #3 offers a probability of 100 % for absorption at 3,200 s, while scenario #4 offers 89.74 %. In other words, the performance of scenario #3 is better and its cost is lower. The best and worst case scenario for absorbing at 3,000 s - #2 and #4 — reflects a difference between them of 98.26 %.



(a) MTTA and Cost Ordered by Scenario Index



(b) MTTA and Cost Ordered by MTTA Value

Figure 60 – MTTA and Cost for all Scenarios

Table 45 – Absorbing Probabilities at 3,000 s

Scenario	Probability
#1	100
#2	100
#3	66.96
#4	1.74
#5	99.76

Best performance could be observed when adopting the scenario #1 and #2 (see Figure 60). Figure 59 depicts that scenario #2 has the highest probabilities for finishing execution faster and scenario #1 has the second highest one. Scenario #2 uses six powerful RIs (three *t2.medium* and three *t2.large*) and scenario #1 uses five ones (one *t2.medium* and four *t2.large*). On the other hand, it is important to highlight that scenario #3 uses five powerful RIs (four *t2.medium* and one *t2.large*) and it has the fourth highest probabilities for finishing execution faster. Hence we may say that scenarios that use *t2.large* instances in more numbers offer the best processing times — #1 and #2 (see Table 41). In other words, it means that the use of *t2.large* instances in a larger number may help decrease all processing time. Thus, it may result in a higher financial cost to the MCC service provider. In scenarios #1 and #2, the probability to finish service for all requests at 3,000 s is equal to 100 %. They have similar cost, but there exists a difference between their MTTAs around 234 s. Let us consider the probability for absorption at 2,6000 s. Although scenario #2 offers a probability of 100 % for absorption at 2,600 s, the second best scenario —#1— offers only 31.78 %. Scenario #5 offers the third least time to process the workload at 3,000 s. This scenario uses thirteen cheaper RIs and it costs $ 705 less than scenarios #1 and #2. Scenario #5 offers a savings of 33 % over the scenarios #1 and #2 in one year. Scenario #5 absorbs at 3,100 s and its MTTA is 96 s higher than scenario #1. It means that, considering our workload, the use of less powerful instances, but in a large number, may allow the system to comply with the SLA at a low cost.

Now, let us execute an analysis from another point of view. Evaluating the absorbing probability of the faster and the lower scenario that meets the SLA — scenarios #2 and #3 — we can notice a difference of 33 % between them to absorbing at 3,000 s. The difference between their MRTs is about 485 s (i.e. more than 8 min) and scenario #2 is able to complete the entire workload processing at 3,000 s. However, the scenario #3 offers savings of $ 936 per year. Given that scenario #3 is the most constrained, the MCC service provider should specify the observation of scenario #3 in its SLA. If an external customer of the MCC service provider requires the MCC system to complete the workload processing at 3,000 s, the probability for scenario #3 will always be around 67 %. Therefore, the service provider could agree to deliver the service by charging low prices due to the limitations of the system configuration. MCC service providers may perform accurate performance and cost analyzes evaluating MTTAs and using CDFs.

Willing to obtain the probability of absorption, the MCC service provider may consider any time within the range. Probability intervals can also be exploited using CDFs. Aiming to better analyze the remote MCC infrastructure, Figure 61 depicts the respective probabilities obeying six intervals (see Table 46). These intervals do not elucidate the cumulative probability starting from zero but rather the difference between two moments.

The probability of absorption is greater in the T4 range compared to the other intervals (see Table 47). Table 48 depicts the probability of absorption for all scenarios with regard to the interval 2,700 s to 2,800 s. As we can see, the probability of absorption is higher for the scenario #1 in that interval — 54.94 %. The scenario #5 has the second highest absorption probability in the T4 interval — 29.59 %. With regard to the interval T5 (see Figure 61), the probability of absorption is higher for the scenario #5 — 57.82 %. Now, the scenario #1 has the third highest absorption probability — 9.58 %. Scenario #3 has

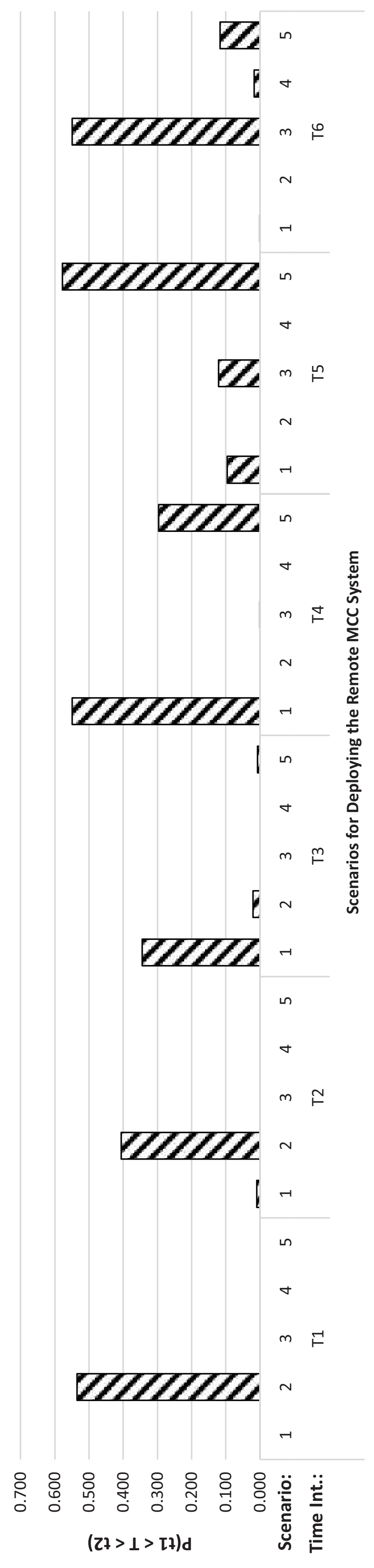


Figure 61 – Probability Intervals

Table 46 – Time Intervals Analyzed

Time Interval (s)	
T1	2,400 - 2,500
T2	2,500 - 2,600
T3	2,600 - 2,700
T4	2,700 - 2,800
T5	2,800 - 2,900
T6	2,900 - 3,000

the highest absorption probability with regard the T6 interval — 54.89 %. Due to such a myriad of interpretations, the application developer or MCC service provider should also pay attention to probability intervals. Perhaps by considering a time interval for absorption rather than a specific time, more cost-effective configurations may be available in order to support the expected workload.

Table 47 – Mean Absorbing Probabilities Related to Six Intervals

Time Interval	Absorbing Probability
T1	10.69
T2	8.30
T3	7.44
T4	16.90
T5	15.89
T6	13.66

Table 48 – Absorbing Probabilities with T4 Interval

Scenario	Probability
#1	54.94
#2	0
#3	0
#4	0
#5	29.59

8.4 Case Study Four: Deployment Planning of MCC Systems Considering Networking Requirements

In this case study, we evaluate an MCC service composed by an app that runs in mobile devices and a system deployed on the cloud for supporting offloading generated by the mobile app. Networking requirements for supporting offloading to the remote infrastructure are considered in this last case study. We consider a mobile cloud facial recognition service that follows the principles of method-call computation offloading (KOSTA et al., 2012). The remote face recognition system uses a database of 8,000 faces replicated in each service instance. Users offload human faces to the remote MCC infrastructure for face recognition. Taking this into account, it is necessary to find a suitable configuration for supporting offloading in the cloud. Evaluating the trade-off between performance and costs in the decision-making process.

Now, let us describe the essential parts of the source code evaluated. Algorithm 9 presents the analyzed class *FaceRecognitionService* that runs in the cloud. The heaviest method, *recognize*, contains two heavy method calls (lines 3 and 4) that perform the face recognition. On the other hand, Algorithm 10 demonstrates the relevant part of the source code that runs in the mobile device. *Mobile App* resides in the mobile device and it has only one offloadable method call (line 3). The method *recognize* is an offloadable method. As the mobile app offloads the method call to the cloud, it means that the mobile app does facial recognition request to the remote MCC infrastructure by passing one argument (that is, one human face). In this case, the app connects to the front-end machine and performs the offloading of the workload.

Algorithm 9 FaceRecognitionService

```
1: function $recognize(face)$
2:     ...
3:     $readFaceBundles(database)$
4:     $recognizeFace(face)$
5:     ...
6: end function
```

Algorithm 10 Mobile App

```
1: function $processFace(face)$
2:     ...
3:     $result \leftarrow recognize(face)$                    ▷ m_call_1
4:     ...
5:     return $result$
6: end function
```

We have considered a heterogeneous elastic MCC infrastructure composed of two types of VM instances. We analyzed the costs of using VM instances and data traffic and some performance metrics for each scenario. We evaluate the performance of the MCC system considering the remote infrastructure and the actual BW available for supporting offloading operations. Our mobile app sends photos for face recognition to a remote infrastructure and, after that, it receives the processing result containing data about the face recognized by the system. It is important to highlight that in the cost evaluation the cost of using the front-end instance was not considered. We have considered only the cost of using VM instances that compose a specific remote configuration for service processing. Table 49 demonstrates the four EC2 instance type that composes all scenarios and the costs of using them as RIs and ODIs. Table 50 shows the cost per gigabyte transferred considering the AWS price table.

Table 49 – EC2 Instances Used by all Evaluated Scenarios (AWS, 2018b)

Model	vCPU	Memory (GiB)	Reserved ($/year)	On-demand ($/hour)
t2.micro	1	1	$ 59.00	$ 0.0116
t2.small	1	2	$ 118.00	$ 0.0230
t2.medium	2	4	$ 235.00	$ 0.0464
t2.large	2	8	$ 470.00	$ 0.0928

Now, let us describe the parameters for supporting scenarios generation. The analysis considered a period of one year and a request rate of $1/(2,500\ ms)$. On device side the mobile app has a single scenario composed by an offlodable method-call, represented here as the *recognize()* method (see Algorithm 10). On the other hand, in the cloud side we have defined eight scenarios for evaluation and Table 51 depicts them. The remote scenarios correspond to the configuration regarding the number of VM instance for supporting the MCC system, the number of simultaneous requests in each instance (γ), and scaling policy to be adopted. Each scenario is composed by two type of VM instances and each type has a fixed number of RIs as well as one ODI available for use. As we can see, as the number of ODIs is one for all scenarios then the MCC system only start one ODI per scaling out request (i.e. stepsize ω defined as 1). The parameters γ corresponds to the number of parallel request running in a single instance. Only scenario #5 processes more than one request per instance at a time ($\gamma = 2$). The system capacity related to the maximum number of requests in the system at a moment consists of 100 requests (i.e. 100 tokens in the place *Capacity*). The MCC system needs to transfer data in and out for supporting the offloading operation. Table 52 shows the amount of transferred data in each user request. As we can see, regardless of the remote scenarios, the amount of data transferred does not vary. As we mentioned in this work, the BW may impact the performance when the volume of data to be transferred is high or the available BW is small. We have considered that the available BW for offloading (upload) as well as receiving data (download) may

Table 50 – Amazon EC2 Prices per Transferred Bytes (AWS, 2017)

Data Transfer OUT To Internet	Price/GB
First 10 TB / month	$ 0.090
Next 40 TB / month	$ 0.085
Next 100 TB / month	$ 0.070

vary within a specified limit impacting the metrics in each variation (see Table 53). We have calculated the metrics on device side by varying the available bandwidth. Depending on the volume of data sent and received in each user request, a high degree of bandwidth variation may result in a significant impact on evaluated metrics.

Table 51 – Scenarios for Deploying the MCC System in the Cloud

	Instance Type 1				Instance Type 2						
	Type	**RIs**	**ODIs**	γ_1[1]	**Type**	**RIs**	**ODIs**	γ_2[1]	**SOT**[2]	**SIT**[3]	ω[4]
Scenarios #1	t2.medium	1	1	1	t2.large	1	1	1	4	1	1
#2	t2.medium	3	1	1	t2.large	2	1	1			
#3	t2.medium	4	1	1	t2.large	1	1	1			
#4	t2.medium	2	1	1	t2.large	2	1	1			
#5	t2.medium	1	1	2	t2.large	1	1	2			
#6	t2.medium	1	1	1	t2.micro	4	1	1			
#7	t2.medium	2	1	1	t2.small	2	1	1			
#8	t2.micro	4	1	1	t2.small	4	1	1			

[1] Maximum number of concurrent jobs for each n-type instance.

[2] Scaling Out Threshold.

[3] Scaling In Threshold.

[4]The stepsize for launching n-type on-demand instances in each scale out request.

Table 52 – Bytes Transferred in Each Request

Operation	**Transferred Data**
Offloading	512 KB
Receiving Result	1 MB

Table 53 – Bandwidth Variation (in Megabits/s)

	BW Variation (Mb/s)	
Operation	**Min BW**	**Expected BW**
Offloading Data (UP BW)	0.1	1.0
Receiving Data (DW BW)	0.1	1.0

The first step in the evaluation process is to collect the time required to process face recognition requests in each instance type as well as the time the AWS EC2 takes to launch our ODIs. Through a controlled experiment our testbed collected the required times. The testbed executed all requests remotely using one instance of each type as the target in order to obtain the processing time (i.e. service time). The experiment executed and monitored 80 times both the workload in each instance and the requests to EC2 for scaling the system out. Our testbed collected the mean values of them at the end of the process. Table 54 presents the time the EC2 spent to launch our ODIs. Table 55 shows the service time registered for each instance type to process γ_n requests simultaneously. Figure 62 shows the service time for each instance type considering a variable (γ_n) number of requests being processed simultaneously in each of them. As we can see, the more simultaneous requests per VM instance, the higher the time necessary to process each request.

Table 54 – Time Required to Launch our ODIs

Instance Type	**Time for Launching (s)**
t2.micro	175.06
t2.small	204.57
t2.medium	239.46
t2.large	318.34

Table 55 – Service Time

Instance			γ^1 **(ms)**		
Type	**1**	**2**	**3**	**4**	**5**
t2.micro	23,638	48,123	75,608	104,568	133,039
t2.small	19,075	35,052	53,936	70,73	86,802
t2.medium	10,752	19,131	27,018	33,428	39,673
t2.large	8,761	15,154	21,155	26,170	30,960

[1] Maximum number of simultaneous jobs for each n-type instance.

Now, it is necessary to analyze some metrics in the cloud for each scenario. We have considered the same cloud model presented in the Section 8.2. No changes in the model occur for metrics computation and we only set the values of each parameter in the model. Equation 8.8 obtains the expected number of requests waiting to receive service. Equation 8.9 gets the expected number of requests that receive service considering the two SI group in the infrastructure. Equation 8.10 obtains the MSS. Equation 8.11 obtains the MRT of the system. This last equation considers the AR — $1/(2,500\ ms)$ — and the probability of having tokens at the place *Capacity*. That is, the probability of the system being able to handle the request received. Equation 8.12 obtains the waiting time in the queue. This equation considers the expected number of tokens in the place *Queue* and the effective arrival rate. Finally, Equation 8.13 obtains the use of ODIs. As we can see in Table 51 all scenarios can use only one ODI for each instance type in the infrastructure. Table 56 presents some performance metrics and utilization of ODIs for each scenario. A

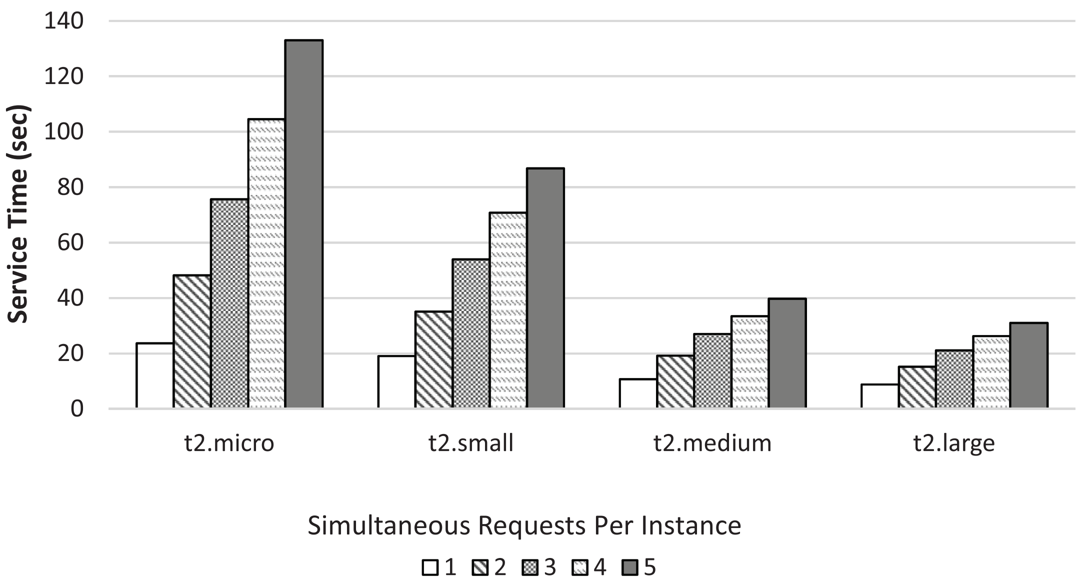


Figure 62 – Service Time for Each Instance Type

stationary analysis in the model obtains these performance metrics.

$$Requests_{queue} = \sum_{i=1}^{n} P(m(Queue) = i) \times i \tag{8.8}$$

$$Requests_{service} = \sum_{type=1}^{n} \left(\sum_{i=1}^{n} P(m(RRS_{type}) = i) \times i \right) \tag{8.9}$$

$$MSS = Requests_{queue} + Requests_{service} \tag{8.10}$$

$$MRT = \frac{MSS}{AR \times (1 - P(m(Capacity) = 0))} \tag{8.11}$$

$$WT = \frac{\sum_{i=1}^{n} P(m(Queue) = i) \times i}{AR \times (1 - P(m(Capacity) = 0))} \tag{8.12}$$

$$EODIU_n = 1 - \left(\sum_{i=1}^{n} P(m(ODIAL_n) = i) \times i \right) \tag{8.13}$$

Now, we need to evaluate MTTE for each MCC scenario considering bandwidth requirements. MTTE represents the time required to execute a workload of a mobile app. The app may process it locally or remotely. MTTE comprises the communication time and the time a remote system spent to process a request when the mobile app offloads a task. Figure 63 depicts the SPN model that represents our mobile app executing in user devices. This model represents only one offloadable function running in the device. More specifically, it represents a face recognition request to the MCC system deployed in the cloud. Transition *processing_time* receives the MRT related to a remote MCC configuration

Table 56 – Performance Metrics and ODIs Utilization for all Scenarios

	Performance Metrics			ODIs Usage	
Scenarios	MRT (ms)	Waiting Time (ms)	MSS	IT1[1]	IT2[2]
#1	138,197	128,536	55	0.9222	0.8980
#2	14,678	4,815	6	0.0022	0.0016
#3	16,770	6,492	7	0.0050	0.0039
#4	28,380	18,715	11	0.0805	0.0576
#5	119,842	102,881	47	0.7251	0.6407
#6	172,201	154,544	68	0.9599	0.9679
#7	100,252	86,474	40	0.7328	0.7689
#8	70,965	49,841	28	0.2849	0.2646

[1] Instance Type 1.

[2] Instance Type 2.

evaluated. That is, *processing_time* represents the time it takes for a remote infrastructure to process a user request. Considering the remote processing time, volume of data transferred in each request and available bandwidth, we may estimate the performance of the mobile application. The SPN-based model considers the Equations 8.14 and 8.15 as the mean delay value of the transitions representing the offloading (upload process) and result receiving (download process) processes, respectively. $BW_{offloading}$ represents the actual bandwidth allocated to offloading a request (upload) — in bits/s. $BW_{receiving}$ represents the actual bandwidth allocated to receive a remote processing result (download) — in bits/s. *datasize_o* represents the amount of data the mobile application transfers to offload a request — in bits. *datasize_r* represents the amount of data that the remote MCC system transfers to the mobile device upon completion of processing of the request — in bits. Table 57 and Figure 64 show the respective MTTEs considering all remote scenarios for deployment the MCC system.

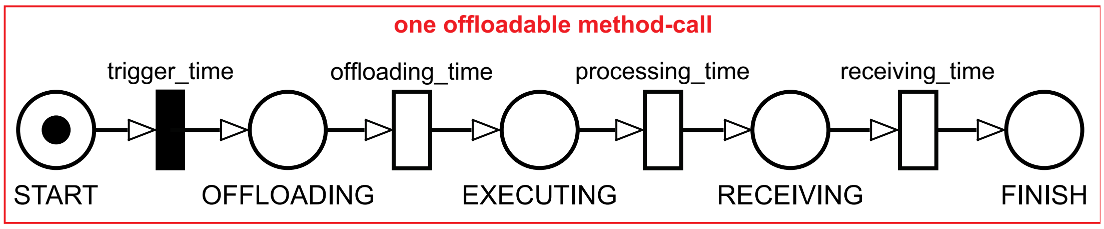


Figure 63 – SPN with Absorbing State Used to Calculate MTTE of an Application With Only One Offloadable Method-Call

$$offloading\_time = \frac{datasize\_o}{BW_{offloading}} \tag{8.14}$$

$$receiving\_time = \frac{datasize\_r}{BW_{receiving}} \tag{8.15}$$

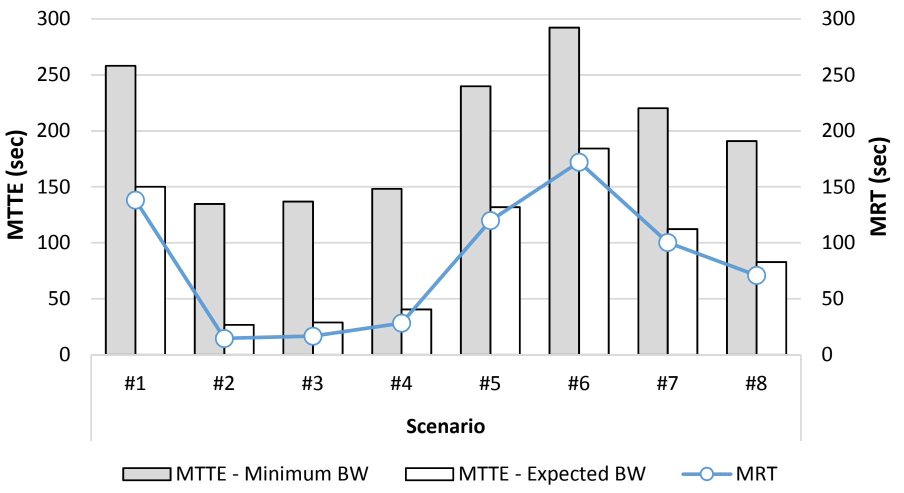
Table 57 – MTTEs and Related Metrics for all Scenarios

		MTTE (ms)		MRT (ms)	Communication Time (ms)		
		Min BW	Exp BW[1]		Min BW	Exp BW[1]	Diff
Scenarios	#1	258,197	150,197	138,197	120,000	12,000	108,000
	#2	134,678	26,678	14,678			
	#3	136,770	28,770	16,770			
	#4	148,380	40,380	28,380			
	#5	239,842	131,842	119,842			
	#6	292,201	184,201	172,201			
	#7	220,252	112,252	100,252			
	#8	190,965	82,965	70,965			

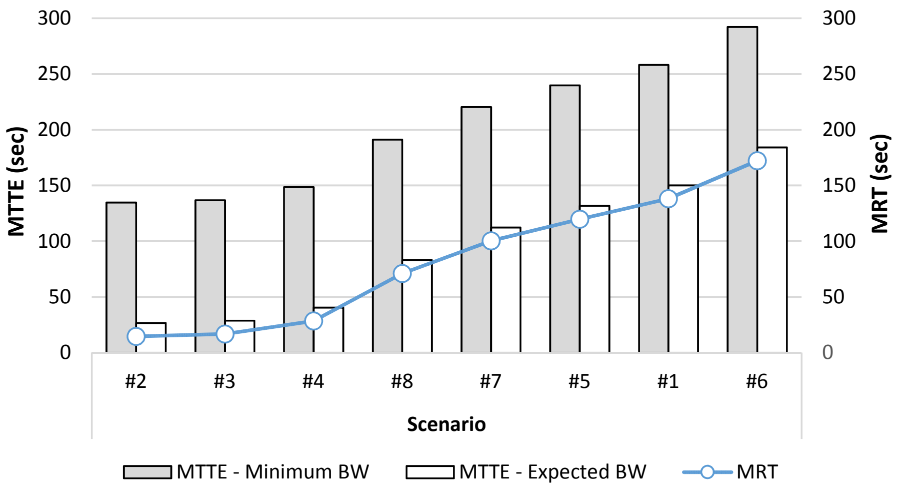
[1] Expected BW.

We can extract some conclusions by analyzing the results. MTTE is close to MRT for all evaluated scenarios when the connection between the MCC infrastructure and the mobile device corresponds to the expected actual BW. It is important to highlight that the amount of data transferred is equal for all scenarios. Considering this, the remote MCC system spends 12 s to send the result to the mobile device when it finishes the request processing. MRT is the main aspect that influences the mobile app performance when we consider only the expected BW. However, in a real-world context, BW may vary. The performance of a mobile app may degrade based on the network conditions. In the scenario #1, the MTTE with the min BW represents 800 % of the MRT. In other words, the time the mobile app spent to send and receive data corresponds to eight times the time the remote system needs to process the request. More specifically, the MCC system spends 199 s to finish data traffic. The difference between the MTTE of the best and worse scenarios — #2 and #6 — corresponds to 157 s. As we can see, BW may degrade the performance of the mobile app. Analysts may choose the MRT that offers the best times taking into account the BW may vary. As the amount of data is equal for all scenarios the difference in sec between the MTTE with the minimum and expected BW is 108 s. It means that the MTTE may vary 108 s considering the minimum and expected BW. Let us now considers the scenarios with the minimum BW. The best scenario #2 spend 89 % of its execution time in data transfer tasks. On the other hand, scenario #6 spend 41 % of its execution in data transfer tasks. When operating with the expected BW it is possible to choose remote MCC configuration less expensive. It means configurations that may offer MRT high. Let us now considers the scenarios with the expected BW. The best scenario #2 spend 44 % of its execution time in data transfer tasks. On the other hand, scenario #6 spend only 6 % of its execution in data transfer tasks.

Let us now analyze the impact of the BW variation on MTTEs considering two network requirements for the application. Figure 65a shows how MTTEs changes when BW varies from 0.1 to 1.0 Mb/s. Figure 65b shows how MTTEs changes when BW varies from



(a) MTTEs and MRTs Ordered by Scenarios Index

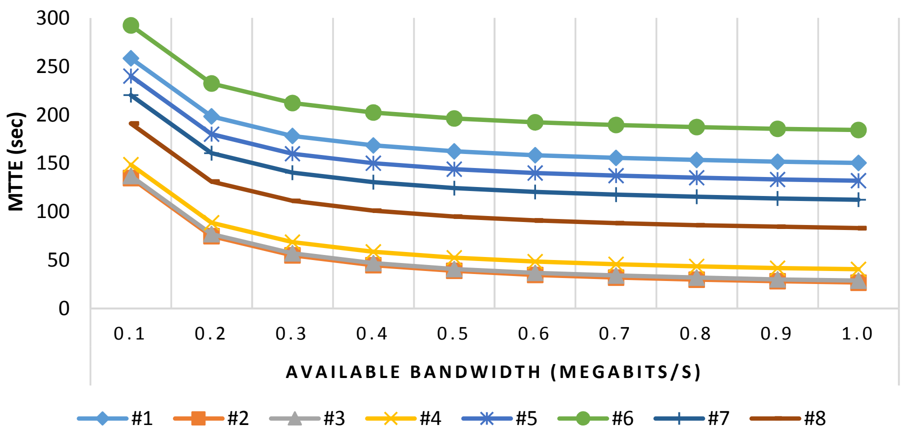


(b) MTTEs and MRTs Ordered by MTTE Values

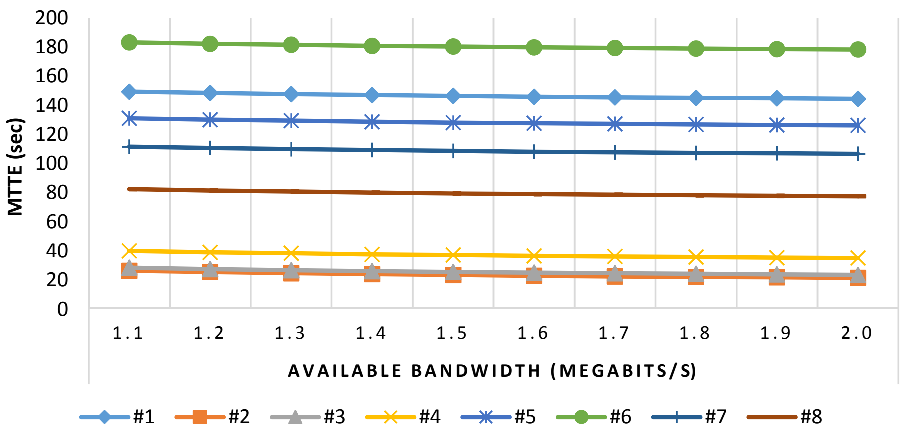
Figure 64 – MTTEs and MRTs Results

1.1 to 2.0 Mb/s. It is important to highlight that Figures 65a and 65b demonstrate the BW variation considering the changing of the download and upload bandwidth at the same time. In other words, both bandwidth types assume the same value represented on the x-axis in the above-mentioned figures. As we can see, the BW variation from 0.1 to 1.0 had a substantial impact on the MTTEs. The impact on MTTEs of the BW variation considering the upper and lower limits aforementioned is 18 s for all scenarios. The greatest impact on MTTEs occurs when BW ranges from 0.1 to 0.4 Mb/s. MTTEs varies 60 sec when BW varies from 0.1 to 0.2 Mb/s. MTTEs varies 20 sec when BW varies from 0.2 to 0.3 Mb/s. MTTEs varies 10 sec when BW varies from 0.3 to 0.4 Mb/s. After that, the impact of BW variation on MTTEs tends to decrease. An MCC service provider needs to pay attention to this. We can see that BW requirements from 0.1 to 0.4 Mb/s can degrade the app performance. Considering this, the service provider may define in the SLA a network requirement with a maximum lower limit around 0.4 or 0.5

Mb/s, for example. On the other hand, the BW variation from 1.1 to 2.0 had little effect on the MTTEs of the scenarios (see Figure 65b). The impact on MTTEs of the BW variation considering the upper and lower limits aforementioned is 4.9 s for all scenarios. It is important to highlight that this behavior depends on the volume of data transferred. Figure 66 depicts the impact of the bandwidth variation on the MTTEs considering the worst BW condition and expected BW condition. Developers should carefully evaluate the effect of the available BW variation on the performance metrics in order to define network requirements.



(a) MTTEs Considering BW Variation from 0.1 to 1.0 Mb/s



(b) MTTEs Considering BW Variation from 1.1 to 2.0 Mb/s

Figure 65 – MTTEs Considering BW Variation

The next step in the evaluation process is to identify the expected TP for each MCC system configuration in the cloud. TP corresponds to the number of user requests the remote MCC system process per unit of time. The arrival rate, system capacity, number of SIs, and service time affect the system throughput. The TP for an n-type instance service group corresponds to the product of the firing rate of the PT_n transition and the expected number of tokens in the place RRS_n (see Equation 8.16). Thus, the system throughput is the sum of the throughput of all instance groups running in the cloud (see

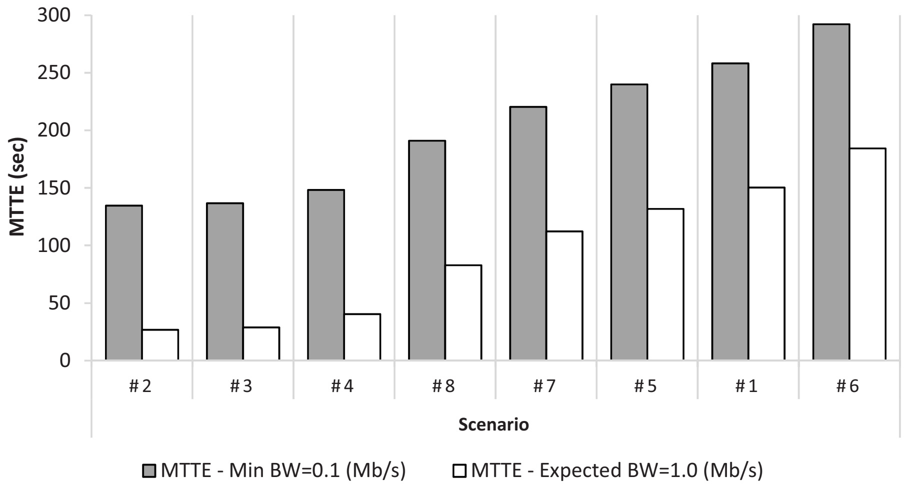


Figure 66 – MTTEs Considering the Minimum and Expected BW for all Scenarios (Ordered by MTTE Values)

Equation 8.17). Little law states that we may obtain TP when we know the MSS — that is, the mean number of requests in the system— and MRT (see Equation 8.18) (JAIN, 1990). A stationary analysis in the model obtains TP using Equations 8.17 or 8.18. TP supports the analyst to estimate the volume of data traffic during a specific period.

$$TP_n = \left(\sum_{i=1}^{z} P(m(RRS_n) = i) \times i \right) \times \frac{1}{PT_n} \tag{8.16}$$

$$TP = \sum_{n=1}^{z} TP_n \tag{8.17}$$

$$TP = \frac{MSS}{MRT} \tag{8.18}$$

Now, let us estimate the volume of transferred data generated by each scenario. As we have stated, Table 52 presents the number of bytes transferred for processing one user request. Using Equation 8.19 it is possible to find out the *TTB*. TTB corresponds to the data volume transferred during a period and, to obtain it, it is necessary multiplying: (*i*) the system throughput (*TP*) — in requests/sec; (*iii*) the period of time the system is up and running (*Time*) — in sec; and finally (*ii*) the volume of data transferred in each request (*Bytes*).

$$TTB = TP \times Time \times Bytes \tag{8.19}$$

Now, using Equation 8.20 it is possible to obtain the financial cost for data transfer (FCDT) (see Section 7.3). The cost calculation should only include the outbound data traffic generated by the remote infrastructure out to mobile devices. We have considered that the data traffic sent from the cloud to a mobile device is 1 MB for each user request considering all evaluated scenarios (see Table 52). For sake of conciseness, we have defined only the same amount of transferred data for all scenarios. However, in a real-world context, each scenario may transfer a different number of bytes. The amount of data that a system transfers over a period allows the service provider to estimate the FCDT considering an MCC scenario for deployment.

$$FCDT = \sum_{ur=1}^{n} TTB_{ur} \times PricePerGB_{ur} \tag{8.20}$$

Table 58 summarizes the throughput, the number of requests per one month, the TTB and its related cost for one month and one year for all scenarios. As we can see, the difference between the cost for each scenario is only a few dollars. In our case study, it means that the data traffic cost has an insignificant impact on the cost evaluation process when deciding the most appropriate configuration since the volume of transferred data in each request is the same for all scenarios. However, an analyst may trace decisions when comparing MRT, MTTE, and cost of using VM instances.

Table 58 – Throughput, Data Traffic and its Related Costs for all Scenarios

				Total Outgoing Transferred Bytes			
				Month		**Year**	
		Tp / sec	**Req / month**	**TB**[1]	**Cost ($)**	**TB**[1]	**Cost ($)**
Scenarios	**#1**	0.395415059	1,024,916	0.98	90.08	11.89	1,086
	#2	0.399999895	1,036,800	0.99	91.12	12.03	1,098
	#3	0.399999813	1,036,800	0.99	91.12	12.03	1,098
	#4	0.399999768	1,036,799	0.99	91.12	12.03	1,098
	#5	0.396200287	1,026,951	0.98	90.26	11.92	1,088
	#6	0.392362291	1,017,003	0.97	89.39	11.80	1,078
	#7	0.399166600	1,034,640	0.99	90.94	12.00	1,096
	#8	0.399987918	1,036,769	0.99	91.12	12.03	1,098

[1] Terabyte.

Figure 67 depicts the relation between the use of ODIs and RIs for all scenarios. Scenarios #2 and #6 are the best and worst scenario, respectively. MRT for scenario #2 is almost 12 times lower than MRT for #6. On the other hand, the costs of using RIs in scenario #2 is 3.5 times higher then #6. The difference in their costs is about $ 690 dollars a year. The use of RIs accounts for most of the total cost of using VM instances in #2. In this scenario, the system only uses ODIs for a few hours per year. Scenario #6 offers the worst performance and is intensive in the use of ODIs. The cost of using ODIs in #6 is a bit high when compared to the costs of using RIs in the same scenario. The cost to use ODIs in #6 is 222 times greater than #2. The higher the cost of using RIs, the smaller the MRT —#2, #3, #4, #8. The best scenarios #2 #3 #4 #8 are not intensive in the use of ODIs (see Figure 70). The system performance worsens as the use of ODIs increases — #1, #5, #6, and #7. Given this context, an MCC service provider may contract more RIs to handle the incoming requests. Perhaps the time it takes to launch an ODI by the cloud provider may degrade performance while increasing the cost. In scenarios #5, #1, and #6, the cost of using ODIs is high than the cost of using RIs. Scenario #6 is the most intensive in using ODIs (see Table 56). However, the cost of using ODIs in #6 is small compared with the scenarios #5 and #1. This is because #6 uses one *t2.micro* ODI that is cheaper - $ 0.01 cent per hour - while #5 and #1 are intensive in the use of one *t2.large* ODI — $ 0.09 cents per hour. Considering our case study, the tendency is for MRT to increase as the cost per use of ODIs increases.

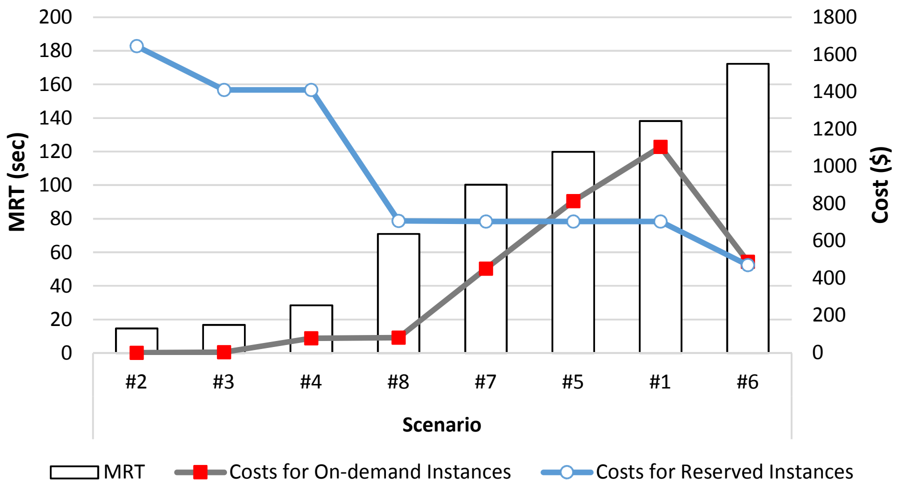


Figure 67 – MRTs and Costs of Using VM Instances

Figure 68 depicts the relation between the costs of data traffic and use of VM instances. As we can see, the cost of data traffic is similar for all scenarios. The cost of data traffic is higher than the cost for using instances in scenarios #8 and #6. Performance in #3 is better than #4, but the cost of using VM instances is higher in #4. Scenario #3 uses four *t2.medium* and one *t2.large* instances, while #4 uses two *t2.medium* and two *t2.large* instances. The performance for processing a single request in *t2.medium* is similar to the performance in *t2.large* — 2 s of difference. However, there exists a difference of $ 74 dollars between them when we compare only the cost of using the instances. One RI of type *t2.medium* costs $ 235 dollars while one RI of type*t2.large* costs $ 470 dollars a year. Scenario #2 uses three *t2.medium* and two *t2.large* instances as RIs. Scenario #3 uses four *t2.medium* instances and only one *t2.large* as RIs. The difference between them is only 2 s, but there exists a difference of $ 232 dollars. It may be more advantageous for an MCC service provider to contract more *t2.medium* than *t2.large* instances. Considering our case study and workload, we may say that there is no advantage in using *t2.large* instances considering both performance and cost.

Figure 69 depicts the relation between MRT and total costs for one year (i.e. costs for data traffic and use of VM instances) (see Table 59). The performance for scenario

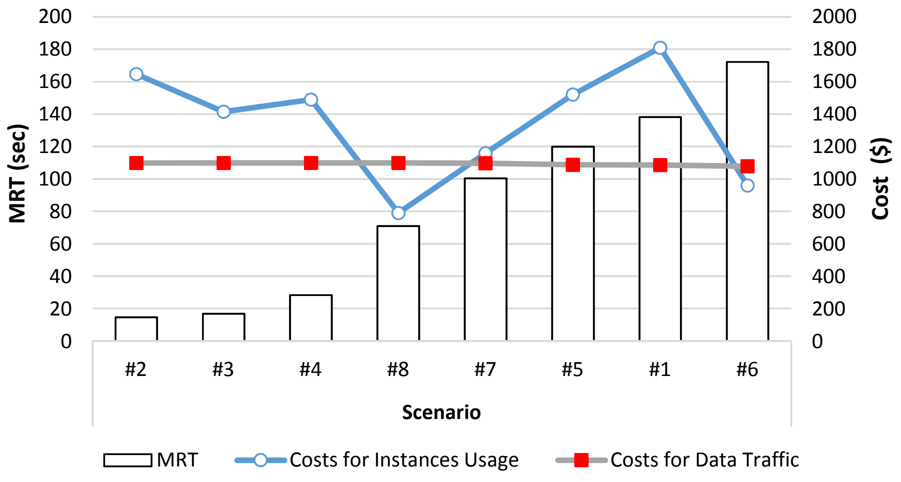


Figure 68 – MRTs and Costs for Data Traffic and use of VM Instances

#8 is 2.5 lower than #6. However, the difference in their costs is approximately $ 149 dollars less for #8. Scenario #8 uses less expensive instances but in a larger number. More specifically, #8 uses four *t2.micro* and four *t2.small* as RIs and one instance of each type as ODI. The utilization of ODIs for scenario #8 corresponds to 27 % and the cost of using them for one year is approximately $ 82 dollars (see Tables 56 and 59). Scenario #1 is the most expensive scenario and consists of one *t2.medium* and one *t2.large* RIs, but this scenario is intensive in the use of ODIs. The utilization of the two ODIs (one *t2.medium* and one *t2.large*) for this scenario corresponds to 90 %. AWS charges $ 0.04 cents per hour to use one *t2.medium* instance as ODI and it charges $ 0.09 cents per hour to use one *t2.large* instance. For this scenario, the cost of using ODIs makes up 63.85 % of the total cost of VM instances for one year (see Table 59). MRT for scenario #4 is 12 s larger than #3, but the difference in their costs is only $ 74 dollars per year. An MCC service provider can lower the MRT in 12 s by choosing scenario #3 with an additional $ 6 monthly. Let us look at the scenarios that offer MRT more than 100 s — #7, #5, #1, #6. Scenario #6 offers the worst performance and is the most intensive scenario in the use of ODIs. However, its cost is small when compared to scenarios #7, #5, #1. Scenario #6 uses one *t2.micro* ODI and one *t2.small* ODI and the cost of using them is only $ 0.01 and $ 0.02 cents per hour, respectively. The use of ODIs is intensive by adopting this scenario but the cost of them per hour is small.

Figure 70 depicts the utilization of ODIs for all scenarios considering the two types of ODIs in the system. The first step in estimating the utilization of ODIs is to obtain the expected use of ODIs in the system. The analyst obtains the expected use of ODIs by subtracting the maximum number of n-type ODIs ($MNODI_n$) by the expected number of tokens in the place $ODIAL_n$. After that, an analyst obtains the utilization of ODIs for a given instance type by dividing the expected use of n-type ODIs by the $MNODI_n$ variable (see Equation 8.21). The expected number of tokens in the $ODIAL_n$ place is obtained

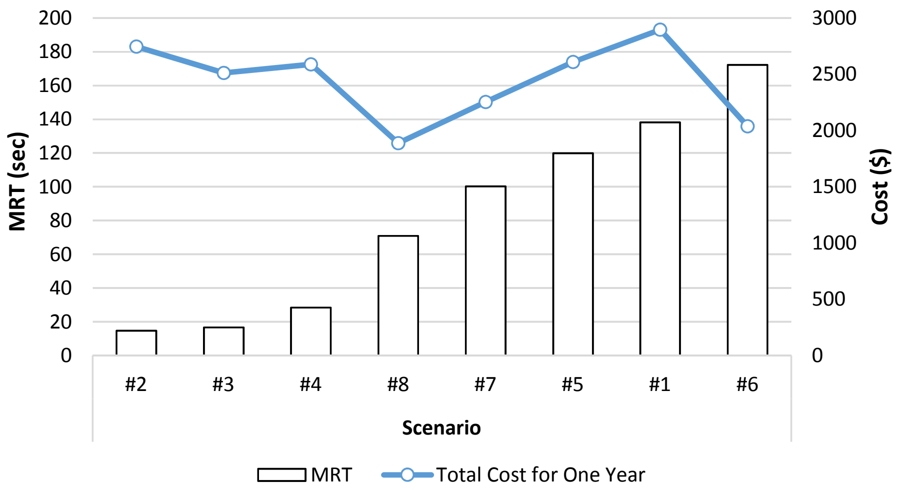


Figure 69 – MRTs and Total Costs for one Year

Table 59 – MRTs and Costs for all Scenarios Considering a Period of One Year

		MRT (ms)	Costs ($) for				
			ODIs	RIs	Use of Instances	Data Traffic	Total
Scenarios	#1	138,197	1,104	705	1,809	1,086	2,895
	#2	14,678	2	1645	1,647	1,098	2,745
	#3	16,770	5	1,410	1,415	1,098	2,513
	#4	28,380	79	1,410	1,489	1,098	2,587
	#5	119,842	815	705	1,520	1,088	2,608
	#6	172,201	488	471	959	1,078	2,037
	#7	100,252	452	706	1,158	1,096	2,254
	#8	70,965	82	708	790	1,098	1,888

through a stationary analysis in the model. This metric *"n-type ODI Utilization"* (ODIU_n) gives us the expected utilization of an n-type ODI. For this case study, the variable $MNODI_n$ takes 1 as value. Using Equation 8.21 it is possible to obtain the utilization considering more than one ODIs of the same type in the system. As we can see, scenarios #1, #6, and #7 are the most intensive in the use of ODIs. The utilization of the two ODIs types corresponds to 90 % for scenario #6. The cost of using one RI of type *t2.medium* for one year is \$ 235. The cost of using one RI of type *t2.micro* corresponds to \$ 59 dollars for the same period. The sum of them corresponds to \$ 294 dollars per year. Table 59 demonstrates that the cost of using one *t2.micro* and one *t2.medium* as ODIs during one year corresponds to \$ 488 dollars for scenario #6. A difference of \$ 194 compared to using them as RIs. In this context, an analyst may contract them as RIs in order to save money. Considering our SPN-based modeling strategies, a myriad of scenarios can be taken by an MCC service provider for supporting accurate analyzes. Making possible to evaluate the trade-off between performance and costs and to choose the best scenarios considering the requirements and limitations of the project.

$$ODIU_n = \frac{MNODI_n - (\sum_{i=1}^{z} P(m(ODIAL_n) = i) \times i)}{MNODI_n} \tag{8.21}$$

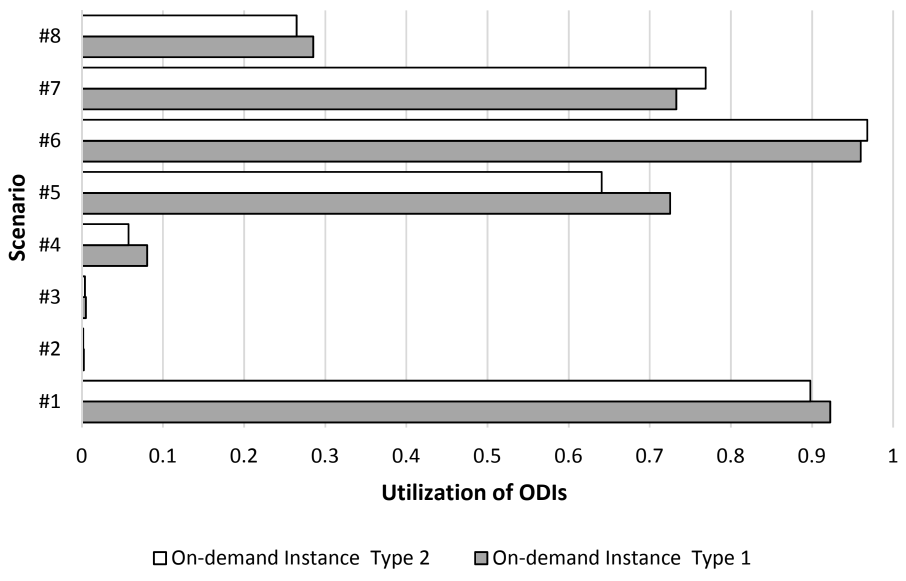


Figure 70 – Utilization of ODIs for all Scenarios

Chapter 9 Conclusion

We have proposed two Stochastic Petri Net (SPN)-based modeling strategies to represent MCC systems running both on mobile devices as well as on a remote infrastructure deployed in a public cloud. Through our modeling strategies, it is possible to evaluate performance and requests making by mobile users at application design time considering mobile devices and a remote infrastructure available for supporting offloading. In addition, our models can represent the communication process between both sides. Four performance metrics are supported: *Mean Time to Execute* (MTTE), *Mean Response Time* (MRT), *Cumulative Distribution Function* (CDF), and *Throughput*. The metrics related to the mobile device are MTTE, CDF, and throughput. The metrics related to the cloud are MRT, CDF, and throughput. MTTE comprises the time to process the application running on a mobile device. On the other hand, MRT comprises the amount of time the remote MCC infrastructure spends to process a request sent by a mobile device. Our approach considers a large number of parameters and each parameter is a characteristic of the MCC system or its environment that affects the system execution. This work considers as parameters of an MCC system to be deployed in a public cloud its capacity in terms of the number of requests that may be there at a time, arrival rates, number and types of VM instances, number of simultaneous requests that each instance can process, maximum number of available on-demand instances (ODIs), time spent by the cloud to launch the system's ODIs, thresholds for scaling the system in/out, and stepsizes. On the other hand, as parameters on the device side, this work considers the application's source code, bandwidth (BW), and local and remotely processing times. Public clouds charge for resource consumption and it is necessary to use resources appropriately. Our SPN-based cloud modeling strategy makes it possible to represent remote MCC systems with variable processing and buffers capacities, variable demands, scaling thresholds, stepsizes, and a large number of VM instances running in the infrastructure. By combining different VM instance types, simultaneous jobs per instance, stepsizes and scaling thresholds, it is possible to offer different response times for each offloading scenario. However, each of these variables affects resource consumption as well as the cost that an MCC service provider pays to an IaaS provider. MCC service providers need to pay attention to the use of elastic resources when using IaaS public clouds. An application may use an unlimited amount of resources and this affects the amount of money that the service provider needs to pay at the end of a period. For that aim, we have proposed an approach that supports the evaluation of resource consumption and costs using SPNs. The cost metrics represent the cost of resource consumption in the cloud. We consider the cost of using VM instances and data traffic. The cost of using VM instances corresponds to the cost of using reserved instances (RIs) and ODIs. The cost of data traffic depends on how much data the MCC system in the cloud sends to mobile users. Each IaaS service provider has its own pricing policy. This work adopted the AWS pricing model. However, another pricing model from a provider other than AWS may be adopted. In addition, our approach considers the available BW to send and receive data. In this way, representing the communication time to transfer data. Our models demonstrate the impact of the available BW variation on the metrics. Thus, considering that a large number of users may use the mobile application with specific network conditions, it allows a more accurate evaluation by developers about the performance of their applications taking into account specific network requirements, offloading scenarios, and remote deployment configurations. We have evaluated four case studies to demonstrate the feasibility of our approach. To validate our device modeling strategy, we evaluated an image processing application. To validate our cloud modeling strategy, we evaluated the deployment of a mobile cloud face recognition system on AWS. Our approach has proven to be feasible and it highlights the most appropriate scenarios for offloading and deploying an MCC system in a public cloud. Using the proposed models enable companies to plan their mobile cloud infrastructures with minimal effort.

9.1 Contributions

Following, we list some contributions of this work:

An SPN-based modeling strategy that represents both applications running on mobile devices and the use of the actual bandwidth available for supporting offloading operations. Making it possible to predict the mobile application's performance by considering specific network requirements and remote configurations for supporting offloading.

An SPN-based modeling strategy that may represent a heterogenous elastic MCC infrastructure considering a set of parameters for deploying an MCC system in public clouds. Our modeling strategy considers thresholds and stepsizes for scaling the system and it supports the definition of scaling policies. Making possible to evaluate system performance and resources consumption for each possible configuration on the remote side. Using our approach, it is possible to represent an MCC infrastructure composed of different types of VM instances. In addition, our approach makes it possible to define scaling policies for each type of VM instances. A heterogeneous infrastructure can deliver cost savings by combining expensive and less expensive instances at the same time. We have validated our modeling strategy through the evaluation of the deployment of a mobile cloud face recognition system on AWS. For other types of workloads, there may be some refinement in the model in order to adapt it to more accurately represent the architecture and system under consideration.

An approach supported by stochastic models for predicting resource consumption and costs in public clouds. Using our approach, it is possible to perform the trade-off between performance and costs when evaluating the most suitable configuration for deploying an MCC system. For that aim, considering the cost of using reserved and on-demand instances and data traffic for a given period. Our case studies depicted that some expensive configurations may offer worse performance than less expensive configurations and that small change in some parameters may have a great impact on performance.

9.2 Limitations and Future Works

Following, we list some limitations and possible future work:

Considering more than one workload in the same infrastructure. Our approach considers only one type of workload running in the same infrastructure. It means that the MCC service provider needs to maintain an infrastructure for each workload type. An analyst may evolve our models in order to represent two or more types of workloads in the same infrastructure.

Evolving our models considering a context-aware offloading approach. Making possible the on-the-fly performance predictions and the choice of the best offloading scenario for the moment. Our approach is not a context-aware offloading approach. Thus, network congestion is not addressed in our approach and it is a limitation of our work. However, our solution may be adapted considering strategies proposed by other authors. Transforming the ideas presented herein in a context-aware approach.

Using optimization algorithms. Combining our modeling strategies with optimization algorithms to automate the evaluation of the state space of the evaluated parameters. Using optimization algorithms makes it possible to evaluate the state space of a set of parameters and to find the most appropriate scenarios for offloading data and code and deploying an MCC system in the public cloud. For that aim, considering performance and financial requirements.

Evaluating other consumable resources in the public cloud. In this work, we have considered as consumable resources on the cloud the use of VM instances and volume of data traffic. However, developers may evolve our approach to represent other services and resources not supported by it.

Evaluating other types of workloads. In this work, we evaluate a mobile face recognition system deployed on a public cloud. Perhaps for some type of workloads, there may be necessary to refine the modeling strategy in order to represent more accurately other aspects of the evaluated system. Considering this, the effectiveness of our approach may be evaluated considering other types of workloads.

Using a hybrid infrastructure. As some public clouds APIs is compatible with other cloud systems, for example, Eucalyptus (NURMI et al., 2009) and OpenNebula (MILOJIčIć; LLORENTE; MONTERO, 2011), MCC systems may be deployed in private and public infrastructures at the same time. Thus, our cloud modeling strategy may evolve to represent parts of the MCC system deployed in a public cloud and other ones in a private infrastructure. For example, sensitive data may be stored on private infrastructure, while a public cloud provides processing power to handle the system's workload. Using a hybrid infrastructure may save money while meeting strict system security requirements.

Evaluating other metrics other than performance, such as dependability metrics. Using other metrics other than performance metrics may allow an MCC service provider to more accurately evaluate other characteristics of its system. For example, using dependability metrics make it possible to evaluate the system's ability to perform in the presence of faults. In this context, analysts may evaluate the cost of using redundant VM instances to ensure a specific level of availability or reliability for their systems.

References

ABOLFAZLI, S.; GANI, A.; CHEN, M. Hmcc: A hybrid mobile cloud computing framework exploiting heterogeneous resources. In: *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. [S.l.: s.n.], 2015. p. 157–162.

AKHERFI, K.; GERNDT, M.; HARROUD, H. Mobile cloud computing for computation offloading: Issues and challenges. *Applied Computing and Informatics*, v. 14, n. 1, p. 1 – 16, 2018. ISSN 2210-8327. Available at: <http://www.sciencedirect.com/science/article/pii/S2210832716300400>.

AL-DHURAIBI, Y.; PARAISO, F.; DJARALLAH, N.; MERLE, P. Elasticity in cloud computing: State of the art and research challenges. *IEEE Transactions on Services Computing*, v. 11, n. 2, p. 430–447, March 2018. ISSN 1939-1374.

AL-FAIFI, A. M.; SONG, B.; HASSAN, M. M.; ALAMRI, A.; GUMAEI, A. Performance prediction model for cloud service selection from smart data. *Future Generation Computer Systems*, v. 85, p. 97 – 106, 2018. ISSN 0167-739X. Available at: <http://www.sciencedirect.com/science/article/pii/S0167739X18300141>.

ANDERSON, T. W.; DARLING, D. A. A test of goodness of fit. *Journal of the American Statistical Association*, Taylor Francis, v. 49, n. 268, p. 765–769, 1954.

ARAUJO, C.; KOSTA, S.; VAZ, F.; COSTA, I.; MACIEL, P.; SILVA, F. Supporting availability evaluation in MCC-based mHealth planning. *Electronics Letters*, Institution of Engineering and Technology (IET), v. 52, n. 20, p. 1663–1665, sep 2016. Available at: <https://doi.org/10.1049/el.2016.1652>.

ARAUJO, C.; MACIEL, P.; ZIMMERMANN, A.; ANDRADE, E.; SOUSA, E.; CALLOU, G.; CUNHA, P. Performability modeling of electronic funds transfer systems. *Computing*, Springer Vienna, v. 91, n. 4, p. 315–334, 2011. ISSN 0010-485X. Available at: <http://dx.doi.org/10.1007/s00607-010-0121-0>.

ARAUJO, J.; SILVA, B.; OLIVEIRA, D.; MACIEL, P. Dependability evaluation of a mhealth system using a mobile cloud infrastructure. In: IEEE. *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on*. [S.l.], 2014. p. 1348–1353.

ASLANPOUR, M. S.; GHOBAEI-ARANI, M.; TOOSI, A. N. Auto-scaling web applications in clouds: A cost-aware approach. *Journal of Network and Computer Applications*, v. 95, p. 26 – 41, 2017. ISSN 1084-8045. Available at: <http://www.sciencedirect.com/science/article/pii/S1084804517302448>.

ASSUNCAO, M. D. de; CARDONHA, C. H.; NETTO, M. A.; CUNHA, R. L. Impact of user patience on auto-scaling resource capacity for cloud services. *Future Generation Computer Systems*, v. 55, p. 41 – 50, 2016. ISSN 0167-739X. Available at: <http://www.sciencedirect.com/science/article/pii/S0167739X15002794>.

AWS. *Amazon EC2 Pricing*. 2017. <https://aws.amazon.com/ec2/pricing/>. Accessed: 2017-05-22.

AWS. *Amazon CloudWatch Documentation*. 2018. <https://docs.aws.amazon.com/cloudwatch/index.html#lang/en_us>. Accessed: 2018-12-16.

AWS. *Amazon Web Services (AWS)*. 2018. <https://aws.amazon.com/>. Accessed: 2018-12-16.

AWS. *Case Studies & Customer Success*. 2018. <https://aws.amazon.com/solutions/case-studies/>. Accessed: 2018-12-29.

AZURE. *Bandwidth Pricing Details*. 2017. <https://azure.microsoft.com/en-us/pricing/details/bandwidth/>. Accessed: 2017-05-22.

BACCARELLI, E.; CORDESCHI, N.; MEI, A.; PANELLA, M.; SHOJAFAR, M.; STEFA, J. Energy-efficient dynamic traffic offloading and reconfiguration of networked data centers for big data stream mobile computing: review, challenges, and a case study. *IEEE Network*, v. 30, n. 2, p. 54–61, March 2016. ISSN 0890-8044.

BALAN, R. K. *Simplifying cyber foraging*. [S.l.]: School of Computer Science, Carnegie Mellon University, 2006.

BISWAS, A.; MAJUMDAR, S.; NANDY, B.; EL-HARAKI, A. An auto-scaling framework for controlling enterprise resources on clouds. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)(CCGRID)*. [s.n.], 2015. v. 00, p. 971–980. Available at: <doi.ieeecomputersociety.org/10.1109/CCGrid.2015.120>.

BOLCH, G.; GREINER, S.; MEER, H. de; TRIVEDI, K. S. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. New York, NY, USA: Wiley-Interscience, 2006. ISBN 0-471-56525-3.

BRUNELLI, R.; POGGIO, T. Face recognition: features versus templates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 15, n. 10, p. 1042–1052, Oct 1993. ISSN 0162-8828.

BUSINESS OF APPS. *Pokémon GO Revenue and Usage Statistics (2017)*. 2017. <http://www.businessofapps.com/pokemon-go-usage-revenue-statistics/>. Accessed: 2017-06-23.

BUSINESS STANDARD. *Apple's iPhone top smartphone brand in Q4, Samsung leader in 2017*. 2018. <https://www.business-standard.com/article/companies/apple-s-iphone-top-smartphone-brand-in-q4-samsung-leader-in-2017-118020200576_1.html>. Accessed: 2018-12-29.

CAMPOS, E.; MATOS, R.; MACIEL, P.; COSTA, I.; SILVA, F. A.; SOUZA, F. Performance evaluation of virtual machines instantiation in a private cloud. In: *2015 IEEE World Congress on Services*. [S.l.: s.n.], 2015. p. 319–326. ISSN 2378-3818.

CARDELLINI, V.; PRESTI, F. L.; NARDELLI, M.; RUSSO, G. R. Decentralized self-adaptation for elastic data stream processing. *Future Generation Computer Systems*, v. 87, p. 171 – 185, 2018. ISSN 0167-739X. Available at: <http://www.sciencedirect.com/science/article/pii/S0167739X17326821>.

CASSANDRAS, C. *Introduction to discrete event systems*. New York, N.Y: Springer Science+Business Media, 2008. ISBN 978-0387333328.

CHAKRAVARTY, I. M.; LAHA, R. G.; ROY, J. D. *Handbook of methods of applied statistics*. New York, NY: McGraw-Hill, 1967. Available at: <http://cds.cern.ch/record/109749>.

CHANG, Z.; ZHOU, Z.; RISTANIEMI, T.; NIU, Z. Energy efficient optimization for computation offloading in fog computing system. In: *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. [S.l.: s.n.], 2017. p. 1–6.

CHEN, S.; WANG, Y.; PEDRAM, M. Optimal offloading control for a mobile device based on a realistic battery model and semi-markov decision process. In: *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. [S.l.: s.n.], 2014. p. 369–375. ISSN 1092-3152.

CHEN, T.; BAHSOON, R.; YAO, X. A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 51, n. 3, p. 61:1–61:40, Jun. 2018. ISSN 0360-0300. Available at: <http://doi.acm.org/10.1145/3190507>.

CHUN, B.-G.; IHM, S.; MANIATIS, P.; NAIK, M.; PATTI, A. Clonecloud: Elastic execution between mobile device and cloud. In: *Proc. of the Sixth Conf. on Computer Systems*. New York, NY, USA: ACM, 2011. (EuroSys '11), p. 301–314. ISBN 978-1-4503-0634-8. Available at: <http://doi.acm.org/10.1145/1966445.1966473>.

CHUNG, K. L. Book review: Stochastic processes. *Bulletin of the American Mathematical Society*, American Mathematical Society (AMS), v. 60, n. 2, p. 190–202, mar 1954. Available at: <https://doi.org/10.1090/s0002-9904-1954-09801-4>.

CIDON, A.; LONDON, T. M.; KATTI, S.; KOZYRAKIS, C.; ROSENBLUM, M. Mars: Adaptive remote execution for multi-threaded mobile devices. In: *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*. New York, NY, USA: ACM, 2011. (MobiHeld '11), p. 1:1–1:6. ISBN 978-1-4503-0980-6. Available at: <http://doi.acm.org/10.1145/2043106.2043107>.

CISCO. *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016–2021 White Paper*. 2019. <https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-738429.html>. Accessed: 2019-03-12.

CNBC. *If Amazon's cloud goes down, the internet would be in trouble, says Reddit's Alexis Ohanian*. 2017. <https://www.cnbc.com/2017/06/30/alexis-ohanian-if-amazon-cloud-breaks-the-web-would-be-in-trouble.html>. Accessed: 2018-12-29.

COHEN, I.; SEBE, N.; GARG, A.; CHEN, L. S.; HUANG, T. S. Facial expression recognition from video sequences: temporal and static modeling. *Computer Vision and Image Understanding*, v. 91, n. 1, p. 160 – 187, 2003. ISSN 1077-3142. Special Issue on Face Recognition. Available at: <http://www.sciencedirect.com/science/article/pii/S107731420300081X>.

CORDESCHI, N.; AMENDOLA, D.; BACCARELLI, E. Reliable adaptive resource management for cognitive cloud vehicular networks. *IEEE Transactions on Vehicular Technology*, v. 64, n. 6, p. 2528–2537, June 2015. ISSN 0018-9545.

CORDESCHI, N.; AMENDOLA, D.; SHOJAFAR, M.; BACCARELLI, E. Distributed and adaptive resource management in cloud-assisted cognitive radio vehicular networks with hard reliability guarantees. *Vehicular Communications*, v. 2, n. 1, p. 1 – 12, 2015. ISSN 2214-2096. Available at: <http://www.sciencedirect.com/science/article/pii/S2214209614000436>.

COSTA, I.; ARAUJO, J.; DANTAS, J.; CAMPOS, E.; SILVA, F. A.; MACIEL, P. Availability evaluation and sensitivity analysis of a mobile backend-as-a-service platform. *Quality and Reliability Engineering International*, p. n/a–n/a, 2015. ISSN 1099-1638. Available at: <http://dx.doi.org/10.1002/qre.1927>.

CUERVO, E.; BALASUBRAMANIAN, A.; CHO, D.; WOLMAN, A.; SAROIU, S.; CHANDRA, R.; BAHL, P. Maui: Making smartphones last longer with code offload. In: *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*. New York, NY, USA: ACM, 2010. (MobiSys '10), p. 49–62. ISBN 978-1-60558-985-5. Available at: <http://doi.acm.org/10.1145/1814433.1814441>.

DESROCHERS, A.; AL-JAAR, R.; SOCIETY, I. C. S. *Applications of petri nets in manufacturing systems: modeling, control, and performance analysis*. IEEE Press, 1995. ISBN 9780879422950. Available at: <https://books.google.it/books?id=mL1TAAAAMAAJ>.

DEY, S.; LIU, Y.; WANG, S.; LU, Y. Addressing response time of cloud-based mobile applications. In: *Proceedings of the First International Workshop on Mobile Cloud Computing & Networking*. New York, NY, USA: ACM, 2013. (MobileCloud '13), p. 3–10. ISBN 978-1-4503-2206-5. Available at: <http://doi.acm.org/10.1145/2492348.2492359>.

DING, Z.; YANG, R. Modeling and analysis for mobile computing systems based on petri nets: A survey. *IEEE Access*, v. 6, p. 68038–68056, 2018. ISSN 2169-3536.

DUPONT, S.; LEJEUNE, J.; ALVARES, F.; LEDOUX, T. Experimental analysis on autonomic strategies for cloud elasticity. In: *2015 International Conference on Cloud and Autonomic Computing*. [S.l.: s.n.], 2015. p. 81–92.

EFRON, B.; TIBSHIRANI, R. *An Introduction to the Bootstrap*. [S.l.]: Chapman and Hall, 1993.

ERL, T. *Cloud computing : concepts, technology, & architecture*. Upper Saddle River, NJ: Prentice Hall, 2013. ISBN 978-0133387520.

FE, I.; MATOS, R.; DANTAS, J.; MELO, C.; MACIEL, P. Stochastic model of performance and cost for auto-scaling planning in public cloud. In: *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. [S.l.: s.n.], 2017. p. 2081–2086.

FORBES. *'Pokémon GO' Servers Down: Weekend Brings Seeming Crush Of Players*. 2016. <https://www.forbes.com/sites/davidthier/2016/07/16/login-crash-pokemon-go-servers-down/>. Accessed: 2018-12-29.

GABNER, R.; SCHWEFEL, H.-P.; HUMMEL, K.; HARING, G. Optimal model-based policies for component migration of mobile cloud services. In: *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*. [S.l.: s.n.], 2011. p. 195–202.

GALANTE, G.; BONA, L. C. E. d. A survey on cloud computing elasticity. In: *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*. Washington, DC, USA: IEEE Computer Society, 2012. (UCC '12), p. 263–270. ISBN 978-0-7695-4862-3. Available at: <http://dx.doi.org/10.1109/UCC.2012.30>.

GERMAN, R. *Performance Analysis of Communication Systems with Non-Markovian Stochastic Petri Nets*. New York, NY, USA: John Wiley & Sons, Inc., 2000. ISBN 0471492582.

GERMAN, R.; KELLING, C.; ZIMMERMANN, A.; HOMMEL, G. Timenet-a toolkit for evaluating non-markovian stochastic petri nets. In: *Proceedings 6th International Workshop on Petri Nets and Performance Models*. [S.l.: s.n.], 1995. p. 210–211. ISSN 1063-6714.

GHANBARI, H.; SIMMONS, B.; LITOIU, M.; ISZLAI, G. Exploring alternative approaches to implement an elasticity policy. In: *2011 IEEE 4th International Conference on Cloud Computing*. [S.l.: s.n.], 2011. p. 716–723. ISSN 2159-6190.

GINE, E.; ZINN, J. Bootstrapping general empirical measures. *The Annals of Probability*, JSTOR, 1990. Available at: <https://www.jstor.org/stable/2244320>.

GONZALEZ-RODRIGUEZ, G.; COLUBI, A.; GIL, M. A. Fuzzy data treated as functional data: A one-way anova test approach. *Computational Statistics & Data Analysis*, v. 56, n. 4, p. 943 – 955, 2012. ISSN 0167-9473. Available at: <http://www.sciencedirect.com/science/article/pii/S0167947310002586>.

GORDON, M. S.; JAMSHIDI, D. A.; MAHLKE, S.; MAO, Z. M.; CHEN, X. Comet: Code offload by migrating execution transparently. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012. (OSDI'12), p. 93–106. ISBN 978-1-931971-96-6.

GREATSPN. *GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets*. 2004. <http://www.di.unito.it/~greatspn/index.html>. Accessed: 2017-07-28.

GUERFEL, R.; SBAÏ, Z.; AYED, R. B. Model checking of cost-effective elasticity strategies in cloud computing. In: *Service-Oriented Computing – ICSOC 2017 Workshops*. Cham: Springer International Publishing, 2018. p. 80–92. ISBN 978-3-319-91764-1.

HALILI, E. *Apache JMeter : a practical beginner's guide to automated testing and performance measurement for your websites*. Birmingham, U.K: Packt Pub, 2008. ISBN 9781847192950.

HAMILTON, J. *Time series analysis*. Princeton, N.J: Princeton University Press, 1994. ISBN 9780691042893.

HAN, R.; GHANEM, M. M.; GUO, L.; GUO, Y.; OSMOND, M. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems*, v. 32, p. 82 – 98, 2014. ISSN 0167-739X. Special Section:

The Management of Cloud Systems, Special Section: Cyber-Physical Society and Special Section: Special Issue on Exploiting Semantic Technologies with Particularization on Linked Data over Grid and Cloud Architectures. Available at: <http://www.sciencedirect.com/science/article/pii/S0167739X12001148>.

HAVERKORT, B. R. Markovian models for performance and dependability evaluation. In: _____. *Lectures on Formal Methods and Performance Analysis: First EEF/Euro Summer School on Trends in Computer Science Bergen Dal, The Netherlands, July 3–7, 2000 Revised Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. p. 38–83. ISBN 978-3-540-44667-5. Available at: <https://doi.org/10.1007/3-540-44667-2_2>.

HERZOG, U. Formal methods for performance evaluation. In: _____. *Lectures on Formal Methods and Performance Analysis: First EEF/Euro Summer School on Trends in Computer Science Bergen Dal, The Netherlands, July 3–7, 2000 Revised Lectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. p. 1–37. ISBN 978-3-540-44667-5. Available at: <https://doi.org/10.1007/3-540-44667-2_1>.

HIRST, J. M.; MILLER, J. R.; KAPLAN, B. A.; REED, D. D. Watts up? pro ac power meter for automated energy recording. *Behavior Analysis in Practice*, v. 6, n. 1, p. 82–95, Jun 2013. ISSN 2196-8934. Available at: <https://doi.org/10.1007/BF03391795>.

HOROVITZ, S.; ARIAN, Y. Efficient cloud auto-scaling with sla objective using q-learning. In: *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*. [s.n.], 2018. v. 00, p. 85–92. Available at: <doi.ieeecomputersociety.org/10.1109/FiCloud.2018.00020>.

HUANG, D.; WANG, P.; NIYATO, D. A dynamic offloading algorithm for mobile computing. *IEEE Transactions on Wireless Communications*, v. 11, n. 6, p. 1991–1995, June 2012. ISSN 1536-1276.

JAIN, R. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. [S.l.]: John Wiley & Sons, 1990.

JAVACV. *Colour Reduction*. 2018. <http://tinyurl.com/pwq8j44>. Accessed: 2018-12-28.

JAVACV. *JavaCV*. 2018. <https://github.com/bytedeco/javacv>. Accessed: 2018-12-28.

JENSEN, K. *Coloured petri nets : modelling and validation of concurrent systems*. Dordrecht New York: Springer, 2009. ISBN 9783642002847.

JMETER. *Apache JMeter*. 2018. <https://jmeter.apache.org>. Accessed: 2018-12-07.

JOHN, L. *Performance evaluation and benchmarking*. Boca Raton, FL: CRC Press, 2006. ISBN 9780849336225.

KANADE, T.; COHN, J. F.; TIAN, Y. Comprehensive database for facial expression analysis. In: *Proceedings Fourth IEEE International Conference on Automatic Face and Gesture Recognition (Cat. No. PR00580)(FG)*. [s.n.], 2000. v. 00, p. 46. Available at: <doi.ieeecomputersociety.org/10.1109/AFGR.2000.840611>.

KEMP, R.; PALMER, N.; KIELMANN, T.; BAL, H. Cuckoo: A computation offloading framework for smartphones. In: GRIS, M.; YANG, G. (Ed.). *Mobile Computing, Applications, and Services*. Springer Berlin Heidelberg, 2012, (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, v. 76). p. 59–79. ISBN 978-3-642-29335-1. Available at: <http://dx.doi.org/10.1007/978-3-642-29336-8_4>.

KLEINROCK, L. *Queueing systems, vol. 1.* [S.l.]: Wiley, New York, 1975.

KOCJAN, P.; SAEED, K. Face recognition in unconstrained environment. In: SAEED, K.; NAGASHIMA, T. (Ed.). *Biometrics and Kansei Engineering.* Springer New York, 2012. p. 21–42. ISBN 978-1-4614-5607-0. Available at: <http://dx.doi.org/10.1007/978-1-4614-5608-7_2>.

KOSTA, S.; AUCINAS, A.; HUI, P.; MORTIER, R.; ZHANG, X. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: *INFOCOM, 2012 Proc. IEEE.* [S.l.: s.n.], 2012. p. 945–953. ISSN 0743-166X.

KOVACHEV, D.; YU, T.; KLAMMA, R. Adaptive computation offloading from mobile devices into the cloud. In: *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications.* [S.l.: s.n.], 2012. p. 784–791. ISSN 2158-9178.

KRISTENSEN, M. D. Scavenger: Transparent development of efficient cyber foraging applications. In: *2010 IEEE International Conference on Pervasive Computing and Communications (PerCom).* [S.l.: s.n.], 2010. p. 217–226.

KUMAR, K.; LU, Y.-H. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 43, n. 4, p. 51–56, Apr. 2010. ISSN 0018-9162. Available at: <http://dx.doi.org/10.1109/MC.2010.98>.

KUO, W.; ZUO, M. J. *Optimal reliability modeling: principles and applications.* [S.l.]: John Wiley & Sons, 2003.

KWON, Y.; TILEVICH, E. Energy-efficient and fault-tolerant distributed mobile execution. In: *2012 IEEE 32nd International Conference on Distributed Computing Systems.* [S.l.: s.n.], 2012. p. 586–595. ISSN 1063-6927.

LI, Z.; WANG, C.; XU, R. Computation offloading to save energy on handheld devices: A partition scheme. In: *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems.* New York, NY, USA: ACM, 2001. (CASES '01), p. 238–246. ISBN 1-58113-399-5.

LIN, S.; ZHANG, X.; YU, Q.; QI, H.; MA, S. Parallelizing video transcoding with load balancing on cloud computing. In: *ISCAS.* [S.l.: s.n.], 2013. p. 2864–2867.

LITTLE, J. D. C. A proof for the queuing formula:l= λ w. *Operations Research*, Institute for Operations Research and the Management Sciences (INFORMS), v. 9, n. 3, p. 383–387, jun 1961. Available at: <https://doi.org/10.1287/opre.9.3.383>.

LORIDO-BOTRAN, T.; MIGUEL-ALONSO, J.; LOZANO, J. A. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, v. 12, n. 4, p. 559–592, 2014. Available at: <https://doi.org/10.1007/s10723-014-9314-7>.

MA, X.; HUANG, P.; JIN, X.; WANG, P.; PARK, S.; SHEN, D.; ZHOU, Y.; SAUL, L. K.; VOELKER, G. M. eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2013. (nsdi'13), p. 57–70. Available at: <http://dl.acm.org/citation.cfm?id=2482626.2482634>.

MACIEL, P.; TRIVEDI, K. S.; MATIAS, R.; KIM, D. S. Performance and dependability in service computing: Concepts, techniques and research directions. In: _____. [S.l.]: Igi Global, 2011. (Premier Reference Source), chap. Dependability Modeling.

MALHOTRA, M.; REIBMAN, A. Selecting and implementing phase approximations for semi-markov models. *Communications in statistics: stochastic models*, Taylor & Francis, v. 9, p. 473–506, 1993.

MAO, M.; HUMPHREY, M. A performance study on the vm startup time in the cloud. In: *2012 IEEE Fifth International Conference on Cloud Computing*. [S.l.: s.n.], 2012. p. 423–430. ISSN 2159-6190.

MARSAN, M. A. In: ROZENBERG, G. (Ed.). *Advances in Petri Nets 1989*. New York, NY, USA: Springer-Verlag New York, Inc., 1990. chap. Stochastic Petri Nets: An Elementary Introduction, p. 1–29. ISBN 0-387-52494-0. Available at: <http://dl.acm.org/citation.cfm?id=90011.90012>.

MARSAN, M. A.; BALBO, G.; CONTE, G.; DONATELLI, S.; FRANCESCHINIS, G. *Modelling with Generalized Stochastic Petri Nets*. 1st. ed. New York, NY, USA: John Wiley & Sons, Inc., 1994. ISBN 0471930598.

MARSAN, M. A.; CONTE, G.; BALBO, G. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 2, n. 2, p. 93–122, May 1984. ISSN 0734-2071. Available at: <http://doi.acm.org/10.1145/190.191>.

MATHIS, M.; SEMKE, J.; MAHDAVI, J.; OTT, T. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 27, n. 3, p. 67–82, Jul. 1997. ISSN 0146-4833. Available at: <http://doi.acm.org/10.1145/263932.264023>.

MATOS, R.; ARAUJO, J.; OLIVEIRA, D.; MACIEL, P.; TRIVEDI, K. Sensitivity analysis of a hierarchical model of mobile cloud computing. *Simulation Modelling Practice and Theory*, Elsevier, v. 50, p. 151–164, 2015.

MELL, P. M.; GRANCE, T. *SP 800-145, The NIST Definition of Cloud Computing*. Gaithersburg, MD, United States, 2011. Available at: <https://csrc.nist.gov/publications/detail/sp/800-145/final>.

MERLIN, P.; FARBER, D. Recoverability of communication protocols - implications of a theoretical study. *IEEE Transactions on Communications*, v. 24, n. 9, p. 1036–1043, September 1976. ISSN 0090-6778.

MILLER, M. *Cloud Computing: Web-Based Applications That Change the Way You Work and Collaborate Online*. [S.l.]: Que Publishing, 2008. ISBN 9780789738035.

MILOJIčIć, D.; LLORENTE, I. M.; MONTERO, R. S. Opennebula: A cloud management tool. *IEEE Internet Computing*, v. 15, n. 2, p. 11–14, March 2011. ISSN 1089-7801.

MOLLOY, M. K. *On the Integration of Delay and Throughput Measures in Distributed Processing Models*. Phd Thesis (PhD Thesis), 1981. AAI8201138.

MOLLOY, M. K. Performance analysis using stochastic petri nets. *IEEE Trans. Comput.*, IEEE Computer Society, Washington, DC, USA, v. 31, p. 913–917, September 1982. ISSN 0018-9340.

MONTGOMERY, D. *Design and analysis of experiments*. Hoboken, NJ: John Wiley & Sons, Inc, 2017. ISBN 978-1119113478.

MOTOROLA. *Motorola Demonstrates Portable Telephone*. 1973. <https://www.motorola.com/sites/default/files/library/us/about-motorola-history-milestones/pdfs/DynaTAC_newsrelease_73_001.pdf>. Accessed: 2018-12-29.

MURATA, T. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, v. 77, n. 4, p. 541–580, Apr 1989. ISSN 0018-9219.

NELSON, R. *Probability, stochastic processes, and queueing theory: the mathematics of computer performance modeling*. [S.l.]: Springer Science & Business Media, 2013.

NIMMAGADDA, Y.; KUMAR, K.; LU, Y.-H.; LEE, C. Real-time moving object recognition and tracking using computation offloading. In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. [S.l.: s.n.], 2010. p. 2449–2455. ISSN 2153-0858.

NMON. *NMON for Linux*. 2018. <http://nmon.sourceforge.net/pmwiki.php>. Accessed: 2018-12-07.

NOE, J. D.; NUTT, G. J. Macro e-nets for representation of parallel systems. *IEEE Transactions on Computers*, C-22, n. 8, p. 718–727, Aug 1973. ISSN 0018-9340.

NURMI, D.; WOLSKI, R.; GRZEGORCZYK, C.; OBERTELLI, G.; SOMAN, S.; YOUSEFF, L.; ZAGORODNOV, D. The eucalyptus open-source cloud-computing system. In: *CCGRID '09*. [S.l.: s.n.], 2009. p. 124–131.

OLIVEIRA, D.; ARAUJO, J.; MATOS, R.; MACIEL, P. Availability and energy consumption analysis of mobile cloud environments. In: IEEE. *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*. [S.l.], 2013. p. 4086–4091.

OPENCV. *OpenCV library*. 2018. <http://opencv.org/>. Accessed: 2018-12-28.

ORACLE. *Cloud Storage Pricing*. 2017. <https://cloud.oracle.com/en_US/storage/pricing>. Accessed: 2017-05-22.

PADHYE, J.; FIROIU, V.; TOWSLEY, D.; KUROSE, J. Modeling tcp throughput: A simple model and its empirical validation. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 28, n. 4, p. 303–314, Oct. 1998. ISSN 0146-4833. Available at: <http://doi.acm.org/10.1145/285243.285291>.

PANDEY, S.; NEPAL, S. Modeling availability in clouds for mobile computing. In: *2012 IEEE First International Conference on Mobile Services*. [S.l.: s.n.], 2012. p. 80–87. ISSN 2329-6429.

PARK, J.; YU, H.; CHUNG, K.; LEE, E. Markov chain based monitoring service for fault tolerance in mobile cloud computing. In: *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*. [S.l.: s.n.], 2011. p. 520–525.

PATHAK, A.; HU, Y.; ZHANG, M.; BAHL, P.; WANG, Y. PDF. *Enabling automatic offloading of resource-intensive smartphone applications*. [S.l.]: docs.lib.purdue.edu, 2011.

PETRI, C. *Kommunikation mit Automaten*. Rhein.-Westfäl. Inst. f. Instrumentelle Mathematik an der Univ. Bonn, 1962. (Schriften des Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn). Available at: <https://books.google.it/books?id=BIGuGwAACAAJ>.

POKEMONGO. *Pokémon Go*. 2018. <http://www.pokemongo.com/>. Accessed: 2018-12-28.

POWERTUTOR. *A Power Monitor for Android-Based Mobile Platforms*. 2018. <http://ziyang.eecs.umich.edu/projects/powertutor/>. Accessed: 2018-12-28.

RAHIMI, M. R.; VENKATASUBRAMANIAN, N.; MEHROTRA, S.; VASILAKOS, A. V. Mapcloud: Mobile applications on an elastic and scalable 2-tier cloud architecture. In: *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*. Washington, DC, USA: IEEE Computer Society, 2012. (UCC '12), p. 83–90. ISBN 978-0-7695-4862-3. Available at: <http://dx.doi.org/10.1109/UCC.2012.25>.

RAMCHANDANI, C. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. Cambridge, MA, USA, 1974.

RIBAS, M.; FURTADO, C.; SOUZA, J. N. de; BARROSO, G. C.; MOURA, A.; LIMA, A. S.; SOUSA, F. R. A petri net-based decision-making framework for assessing cloud services adoption: The use of spot instances for cost reduction. *Journal of Network and Computer Applications*, v. 57, p. 102 – 118, 2015. ISSN 1084-8045. Available at: <http://www.sciencedirect.com/science/article/pii/S1084804515001563>.

RIBAS, M.; FURTADO, C. G.; BARROSO, G.; LIMA, A. S.; SOUZA, N.; MOURA, A. Modeling the use of spot instances for cost reduction in cloud computing adoption using a petri net framework. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. [S.l.: s.n.], 2015. p. 1428–1433. ISSN 1573-0077.

RICKMAN, R. M.; STONHAM, T. J. Coding facial images for database retrieval using a self organising neural network. In: *IEE Colloquium on Machine Storage and Recognition of Faces*. [S.l.: s.n.], 1992. p. 3/1–3/4.

RIM, H.; KIM, S.; KIM, Y.; HAN, H. Transparent method offloading for slim execution. In: *2006 1st International Symposium on Wireless Pervasive Computing*. [S.l.: s.n.], 2006. p. 1–6.

SAARINEN, A.; SIEKKINEN, M.; XIAO, Y.; NURMINEN, J. K.; KEMPPAINEN, M.; HUI, P. Can offloading save energy for popular apps? In: *Proceedings of the Seventh ACM International Workshop on Mobility in the Evolving Internet Architecture*. New York, NY, USA: ACM, 2012. (MobiArch '12), p. 3–10. ISBN 978-1-4503-1526-5. Available at: <http://doi.acm.org/10.1145/2348676.2348680>.

SAHNER, R. *Performance and Reliability Analysis of Computer Systems : an Example-Based Approach Using the SHARPE Software Package*. Boston, MA: Springer US, 1996. ISBN 978-1461360056.

SATYANARAYANAN, M.; BAHL, P.; CACERES, R.; DAVIES, N. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, v. 8, n. 4, p. 14–23, Oct 2009. ISSN 1536-1268.

SHI, C.; HABAK, K.; PANDURANGAN, P.; AMMAR, M.; NAIK, M.; ZEGURA, E. Cosmos: computation offloading as a service for mobile devices. In: ACM. *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing*. [S.l.], 2014. p. 287–296.

SHI, C.; LAKAFOSIS, V.; AMMAR, M. H.; ZEGURA, E. W. Serendipity: Enabling remote computing among intermittently connected mobile devices. In: *Proceedings of the Thirteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*. New York, NY, USA: ACM, 2012. (MobiHoc '12), p. 145–154. ISBN 978-1-4503-1281-3.

SILVA, B.; CALLOU, G.; TAVARES, E.; MACIEL, P.; FIGUEIREDO, J.; SOUSA, E.; ARAUJO, C.; MAGNANI, F.; NEVES, F. Astro: An integrated environment for dependability and sustainability evaluation. *Sustainable Computing: Informatics and Systems*, v. 3, n. 1, p. 1 – 17, 2013. ISSN 2210-5379.

SILVA, B.; MACIEL, P. R. M.; ZIMMERMANN, A.; BRILHANTE, J. Survivability evaluation of disaster tolerant cloud computing systems. In: *Proc. Probabilistic Safety Assessment Management conference 2014 (PSAM 12)*. [S.l.: s.n.], 2014.

SILVA, B.; MATOS, R.; CALLOU, G.; FIGUEIREDO, J.; OLIVEIRA, D.; FERREIRA, J.; DANTAS, J.; LOBO, A.; ALVES, V.; MACIEL, P. Mercury: An integrated environment for performance and dependability evaluation of general systems. In: *Proceedings of Industrial Track at 45th Dependable Systems and Networks Conference, DSN*. [S.l.: s.n.], 2015.

SILVA, B.; TAVARES, E.; MACIEL, P.; NOGUEIRA, B.; OLIVEIRA, J.; DAMASO, A.; ROSA, N. Amalghma - an environment for measuring execution time and energy consumption in embedded systems. In: *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on*. [S.l.: s.n.], 2014. p. 3364–3369.

SILVA, F. A.; KOSTA, S.; RODRIGUES, M.; OLIVEIRA, D.; MACIEL, T.; MEI, A.; MACIEL, P. Mobile cloud performance evaluation using stochastic models. *IEEE Transactions on Mobile Computing*, v. 17, n. 5, p. 1134–1147, May 2018. ISSN 1536-1233.

SILVA, F. A.; MACIEL, P.; ALVES, G.; MATOS, R. A scheduler for mobile cloud based on weighted metrics and dynamic context evaluation. In: *Applied Computing (SAC 2015), Proc. of The 30th ACM/SIGAPP Symposium On*. [S.l.: s.n.], 2015.

SILVA, F. A.; MACIEL, P.; MATOS, R. Smartrank: a smart scheduling tool for mobile cloud computing. *The Journal of Supercomputing*, v. 71, n. 8, p. 2985–3008, Aug 2015. ISSN 1573-0484. Available at: <https://doi.org/10.1007/s11227-015-1423-y>.

SILVA, F. A.; MACIEL, P.; SANTANA, E.; MATOS, R.; DANTAS, J. Mobile cloud face recognition based on smart cloud ranking. *Computing*, v. 99, n. 3, p. 287–311, Mar 2017. ISSN 1436-5057. Available at: <https://doi.org/10.1007/s00607-016-0491-z>.

SILVA, F. A.; RODRIGUES, M.; MACIEL, P.; KOSTA, S.; MEI, A. Planning mobile cloud infrastructures using stochastic petri nets and graphic processing units. In: IEEE. *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. [S.l.], 2015. p. 471–474.

SILVA, F. A.; ZAICANER, G.; QUESADO, E.; DORNELAS, M.; SILVA, B.; MACIEL, P. Benchmark applications used in mobile cloud computing research: a systematic mapping study. *The Journal of Supercomputing*, p. 1–22, 2016.

SOUSA, E. T. G. D.; LINS, F. A. A.; TAVARES, E. A. G.; MACIEL, P. R. M. Performance and cost modeling strategy for cloud infrastructure planning. In: IEEE. *2014 IEEE 7th International Conference on Cloud Computing*. [S.l.], 2014. p. 546–553.

SOYATA, T.; MURALEEDHARAN, R.; FUNAI, C.; KWON, M.; HEINZELMAN, W. Cloud-vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture. In: *Computers and Communications (ISCC), 2012 IEEE Symposium on*. [S.l.: s.n.], 2012. p. 000059–000066. ISSN 1530-1346.

TANG, H.; SUN, Y.; YIN, B.; GE, Y. Face recognition based on haar lbp histogram. In: *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*. [S.l.: s.n.], 2010. v. 6, p. V6–235–V6–238. ISSN 2154-7505.

TRIVEDI, K. S. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. New York: John Wiley and Sons, 2001.

TRIVEDI, K. S.; SAHNER, R. Sharpe at the age of twenty two. *SIGMETRICS Perform. Eval. Rev.*, ACM, New York, NY, USA, v. 36, n. 4, p. 52–57, Mar. 2009. ISSN 0163-5999.

TUFFIN, B.; HIREL, C.; TRIVEDI, K. Simulation versus analytic-numeric methods: a petri net example. In: *Proceedings of the 2nd VALUETOOLS Conference*. [S.l.: s.n.], 2007.

TURK, M.; PENTLAND, A. Face recognition using eigenfaces. In: *Computer Vision and Pattern Recognition Proc. CVPR, IEEE Computer Society Conference on*. [S.l.: s.n.], 1991. p. 586–591. ISSN 1063-6919.

VIOLA, P.; JONES, M. J. Robust real-time face detection. *International Journal of Computer Vision*, v. 57, n. 2, p. 137–154, May 2004. ISSN 1573-1405. Available at: <https://doi.org/10.1023/B:VISI.0000013087.49260.fb>.

VOXMEDIA. *Everything Amazon did in 2018 that you might have missed*. 2018. <https://www.vox.com/the-goods/2018/12/13/18136695/amazon-year-growth-alexa-web-services-kids-tech>. Accessed: 2018-12-29.

WANG, Y.-C.; DONYANAVARD, B.; CHENG, K.-T. T. Energy-aware real-time face recognition system on mobile cpu-gpu platform. In: KUTULAKOS, K. N. (Ed.). *Trends and Topics in Computer Vision*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 411–422. ISBN 978-3-642-35740-4.

YANG, M.; CAI, J.; ZHANG, W.; WEN, Y.; FOH, C. H. Adaptive configuration of cloud video transcoding. In: *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. [S.l.: s.n.], 2015. p. 1658–1661. ISSN 0271-4302.

ZHANG, L.; TIWANA, B.; QIAN, Z.; WANG, Z.; DICK, R. P.; MAO, Z. M.; YANG, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In: *Proceedings of the Eighth IEEE/ACM/IFIP Int. Conference on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2010. (CODES/ISSS '10), p. 105–114. ISBN 978-1-60558-905-3. Available at: <http://doi.acm.org/10.1145/1878961.1878982>.

ZHANG, W.-L.; GUO, B.; SHEN, Y.; LI, D.-G.; LI, J.-K. An energy-efficient algorithm for multi-site application partitioning in mcc. *Sustainable Computing: Informatics and Systems*, v. 18, p. 45 – 53, 2018. ISSN 2210-5379. Available at: <http://www.sciencedirect.com/science/article/pii/S2210537916302244>.

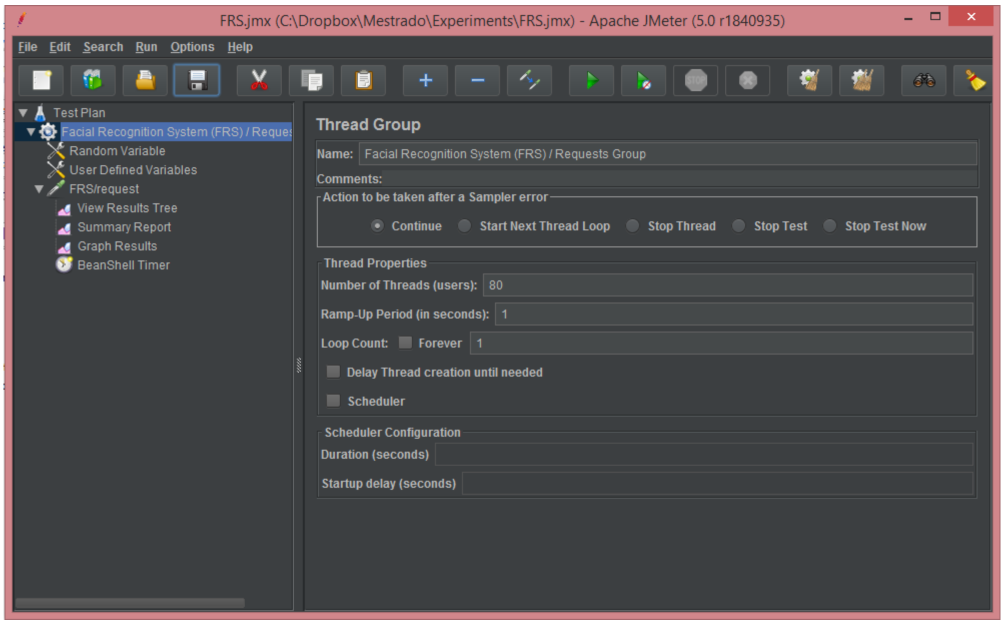
ZHANG, Y.; LIU, H.; JIAO, L.; FU, X. To offload or not to offload: An efficient code partition algorithm for mobile cloud computing. In: *2012 IEEE 1st International Conference on Cloud Networking (CLOUDNET)*. [S.l.: s.n.], 2012. p. 80–86.

ZHAO, B.; XU, Z.; CHI, C.; ZHU, S.; CAO, G. Mirroring smartphones for good: A feasibility study. In: *Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer Berlin Heidelberg, 2012. v. 73, p. 26–38. ISBN 978-3-642-29153-1. Available at: <http://dx.doi.org/10.1007/978-3-642-29154-8_3>.

ZHAO, W.; CHELLAPPA, R.; PHILLIPS, P. J.; ROSENFELD, A. Face recognition a literature survey. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 35, n. 4, p. 399–458, Dec. 2003. ISSN 0360-0300.

APPENDIX A JMETER

JMeter is an open source software, designed to perform load and functional tests. We have used JMeter for supporting us during the validation process of the proposed modeling strategies. Figure 71 depicts JMeter's configuration screen. As we have pointed out in this work, JMeter by default does not support the generation of requests considering exponential times. Considering this, we have implemented a script to support the generation of exponential arrival times. Listening A.1 shows the custom add-on implemented by us. The script has to be attached to a *BS Timer* and, as we can see, it considers a predefined variable named *beforeTime*.


FRS.jmx (C:\Dropbox\Mestrado\Experiments\FRS.jmx) - Apache JMeter (5.0 r1840935)
File Edit Search Run Options Help
Test Plan
Facial Recognition System (FRS) / Reques
Random Variable
User Defined Variables
FRS/request
View Results Tree
Summary Report
Graph Results
BeanShell Timer

Thread Group

Name: Facial Recognition System (FRS) / Requests Group

Comments:

Action to be taken after a Sampler error

[✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 80

Ramp-Up Period (in seconds): 1

Loop Count: ☐ Forever 1

☐ Delay Thread creation until needed

☐ Scheduler

Scheduler Configuration

Duration (seconds)

Startup delay (seconds)

Figure 71 – JMeter's Configuration Screen

```
// Arrival rate (in milliseconds)
double lambda = 1.0 / 10000.0;

```

```
String loc = props.get("beforeTime");

if (loc == null)
  props.put("beforeTime", "0.0");

Double beforeTime = Double.parseDouble(loc);

log.info("### value for the variable 'beforeTime' = " + beforeTime);

double uniform = Math.random();

double exponentialRandomNumber = -Math.log(uniform) / lambda;

log.info("### value for the variable 'exponentialRandomNumber' = " +
    exponentialRandomNumber);

exponentialRandomNumber = exponentialRandomNumber + beforeTime;

// Store the new exponential time generated in the predefined beforeTime variable
     to be used in the next Thread.
props.put("beforeTime", exponentialRandomNumber.toString());

log.info("### Next value for the variable 'beforeTime' = " +
    exponentialRandomNumber);

return (int) exponentialRandomNumber;
```

Listing A.1 – Code to Generate Requests in JMeter Considering Exponential Times

APPENDIX B

AWS EC2 API

```java
import com.amazonaws.auth.AWSStaticCredentialsProvider;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.regions.Regions;
import com.amazonaws.services.ec2.AmazonEC2;
import com.amazonaws.services.ec2.AmazonEC2ClientBuilder;
import com.amazonaws.services.ec2.model.*;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.List;

public class AwsEC2Client
{
    private static AwsEC2Client awsEC2Client = null;
    private static AmazonEC2 client = null;
    private BasicAWSCredentials credentials = null;
    private static final String CREDENTIALS_FILE = "src/credentials.txt";

    private AwsEC2Client()
  {
        loadCredentials();
        client = AmazonEC2ClientBuilder.
                    standard().
                    withCredentials(new AWSStaticCredentialsProvider(credentials)
    ).
                    withRegion(Regions.US_EAST_1).
                    build();
    }

    /**
     * This method lists the instance in a given state
     *
     * @param state The state of the instance
     * Prints the instance id, instance public IP & tags of the instance in a
    given state
```

```
     **/
    public void listInstances(String state)
  {
        List<Reservation> reservations = client.describeInstances().
    getReservations();

        System.out.println("Here is a list of EC2 instances in " + state + "
    state: ");
        for (Reservation reservation : reservations)
    {
            List<Instance> instances = reservation.getInstances();
            for (Instance instance : instances)
      {
                if(state.equalsIgnoreCase(instance.getState().getName()))
        {
                    System.out.printf("Instance Id: %s || Instance Public IP: %s
    || Instance Tags: %s%n",
                            instance.getInstanceId(), instance.getPublicIpAddress
    (), instance.getTags());
                }
            }
        }
    }

    /**
     * This method launches an EC2 instance
     *
     * @param imageId The ID of the AMI
     * @param securityGroup Security Group of AMI
     * @param key_name Name of EC2 key pair
     * @param numOfInstances Maximum number of instances to be launched
     * Launches an EC2 instance with given specification
     **/
    public void launchEC2Instance(String imageId, String securityGroup, String
    key_name, int numOfInstances)
  {
        try
    {
            RunInstancesRequest runInstancesRequest = new RunInstancesRequest();

            runInstancesRequest.
                    withImageId(imageId).
                    withInstanceType(InstanceType.T2Micro).
                    withMinCount(1).
                    withMaxCount(numOfInstances).
                    withKeyName(key_name).
                    withSecurityGroups(securityGroup);

            RunInstancesResult runInstancesResult = client.runInstances(
    runInstancesRequest);

            System.out.println(numOfInstances + " EC2 instance(s) created
    successfully");
        }
    catch (Exception e)
    {
            System.out.println(e.getMessage());
        }
    }

     /**
      * This method starts an EC2 instance
      *
      * @param instanceId The ID of the stopped instance
      * Starts the EC2 instance which is in stopped state
      **/
    public void startEC2Instance(String instanceId)
  {
        try
    {
            StartInstancesRequest startInstancesRequest = new
    StartInstancesRequest();
            startInstancesRequest.withInstanceIds(instanceId);
            StartInstancesResult startInstancesResult = client.startInstances(
    startInstancesRequest);

            System.out.println("Instance started successfully");
        }
    catch (AmazonEC2Exception e)
    {
            System.out.println(e.getErrorMessage());
        }
    }

    /**
     * This method stops an EC2 instance
     *
     * @param instanceId The ID of the instance
     * Stops the EC2 instance
     **/
    public void stopEC2Instance(String instanceId)
  {
        try
    {
            StopInstancesRequest stopInstancesRequest = new StopInstancesRequest
    ();
            stopInstancesRequest.withInstanceIds(instanceId);
            StopInstancesResult stopInstancesResult = client.stopInstances(
    stopInstancesRequest);

            System.out.println("Instance stopped successfully");
        }
    catch (AmazonEC2Exception e)
    {
            System.out.println(e.getErrorMessage());
        }
    }

    /**
     * This method reboots an EC2 instance
     *
     * @param instanceId The ID of the instance
     * Reboots the EC2 instance
     **/
    public void rebootEC2Instance(String instanceId)
  {
        try
    {
            RebootInstancesRequest rebootInstancesRequest = new
    RebootInstancesRequest();
            rebootInstancesRequest.withInstanceIds(instanceId);
            RebootInstancesResult rebootInstancesResult = client.rebootInstances(
    rebootInstancesRequest);

            System.out.println("Instance rebooted successfully");
        }
    catch (AmazonEC2Exception e)
    {
            System.out.println(e.getErrorMessage());
        }
    }

    /**
     * This method provides the complete desciption of EC2 instance
     *
     * @param instanceId The ID of the instance
     * Prints instance id, image id, instance state, instance type, instance
    state & monitoring state
     **/
    public void describeInstance(String instanceId)
  {
        List<Reservation> reservations = client.describeInstances().
    getReservations();

        for (Reservation reservation : reservations)
    {
            for (Instance instance : reservation.getInstances())
      {
                if (instance.getInstanceId().equals(instanceId))
        {
                    System.out.printf("Instance with id %s has the following
    attributes \n" +
                                        "AMI: %s\n" +
                                        "Type: %s\n" +
                                        "Instance State: %s\n" +
                                        "Monitoring state: %s\n",
                                        instance.getInstanceId(),
                                        instance.getImageId(),
                                        instance.getInstanceType(),
                                        instance.getState().getName(),
                                        instance.getMonitoring().getState());
                    return;
                }
            }
        }

        System.out.println("No instance with " + instanceId + " found");
    }

    /**
     * This method starts the monitoring of an EC2 instance
     *
     * @param instanceId The ID of the instance
     * Starts monitoring on the instance with given instance id
     **/
    public void startMonitoringAnInstance(String instanceId)
  {
        try
    {
            MonitorInstancesRequest monitorInstancesRequest = new
    MonitorInstancesRequest();
            monitorInstancesRequest.withInstanceIds(instanceId);
            MonitorInstancesResult monitorInstancesResult = client.
    monitorInstances(monitorInstancesRequest);
            System.out.println("Monitory instance with instance id: " +
    instanceId);
        }
    catch (AmazonEC2Exception e)
    {
            System.out.println(e.getErrorMessage());
        }
    }

    /**
     * This method stops the monitoring of an EC2 instance
     *
     * @param instanceId The ID of the instance
     * Stops the monitoring on the instance with given instance id
     **/
    public void stopMonitoringAnInstance(String instanceId)
  {
        try
    {
            UnmonitorInstancesRequest unmonitorInstancesRequest = new
    UnmonitorInstancesRequest();
            unmonitorInstancesRequest.withInstanceIds(instanceId);
            UnmonitorInstancesResult unmonitorInstancesResult = client.
    unmonitorInstances(unmonitorInstancesRequest);
            System.out.println("Stopped monitoring instance with instance id: " +
     instanceId);
        }
    catch (AmazonEC2Exception e)
    {
            System.out.println(e.getErrorMessage());
        }
    }

    public static AwsEC2Client getEC2Client()
  {
        if (awsEC2Client == null)
    {
            awsEC2Client = new AwsEC2Client();
        }

        return awsEC2Client;
    }

    private void loadCredentials()
```

```java
{
        try
    {
            File file = new File(CREDENTIALS_FILE);
            BufferedReader br = new BufferedReader(new FileReader(file));

            String line;
            List<String> items = new ArrayList<>();
            while ((line = br.readLine()) != null) {
                items.add(line);
            }

            String accessKey = items.get(0);
            String secretKey = items.get(1);

            credentials = new BasicAWSCredentials(accessKey, secretKey);
        }
    catch (Exception e)
    {
            System.out.println(e.getMessage());
        }
    }

public void deleteInstance(Instance instance, AmazonEC2Client ec2)
{
    TerminateInstancesRequest tir = new TerminateInstancesRequest();
    tir.withInstanceIds(instance.getInstanceId());
    ec2.terminateInstances(tir);
}

public List<Instance> searchInstances(String template, String state, String ip,
    AmazonEC2Client ec2)
{
    List<Instance> instances = new ArrayList<Instance>();
    List<Instance> allInstances = listInstances(ec2);

    for(Instance instance: allInstances)
    {
      if (ip == null && instance.getImageId().equals(template) && instance.
    getState().getName().constains(state))
      {
        instances.add(instance);
      }
      else
      if (instance.getImageId().equals(template) && instance.getState().getName()
    .constains(state) && instance.getPrivateIpAddress().contains(ip))
      {
        instances.add(instance);
      }
    }

    return instances;
}


public void instantiateTime(String templateID, string cloudManagerHost, String
    credentialFile) throws InterruptedException, InvalidKeyException,
   NoSuchAlgorithmException, SignatureException, UnsupportedEncodingException
  {
    ArrayList<String> result = new ArrayList<>();

    Thread t;
    Listener listener = new Listener();

    t = new Thread(listener);
    t.start();

    AmazonEC2Cliente ec2 = getEC2(cloudManagerHost, credentialFile);

    ArrayList<String> types = new ArrayList<String>();

    types.add("t2.micro");
    types.add("t2.small");
    types.add("t2.medium");
    types.add("t2.large");

    for(String type: types)
    {
      for (int i = 0; i < 80; i++)
      {
        result.add("creating instance " + type + " " + i + " , " + (new Date()).
   getTime());
        runConversor(templateID, type, ec2);


        listener.accept();


        result.add("receiving instance " + type + " " + i + " , " + (new Date()).
   getTime());
        Instance instance = searchInstances(templateID, "run", null, ec2).get(0);


        result.add("removing instance " + type + " " + i + " , " + (new Date()).
   getTime());
        deleteInstance(instance, ec2);


        Thread.sleep(1000 * 30 * 1);
      }

      Thread.sleep(1000 * 60 * 5);
    }
  }
}
```

Listing B.1 – AWS EC2 API