DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

Huber Flores

Dissertation: Service-oriented and Evidence-aware Mobile Cloud Computing

Abstract

Mobile and cloud computing are two of the biggest forces in computer science. Nowadays, a user either relies on the mobile or the cloud to perform various daily life activities, e.g., social communication, video streaming, gaming, etc. While the cloud provides to the user the ubiquitous computational and storage platform to process any complex tasks, the smartphone grants to the user the mobility features to process simple tasks, anytime and anywhere. Thus, bridging the cloud closer to the mobile user is a reasonable approach to obtain PC-like functionality on the move. Mobile Cloud Computing (MCC) has emerged as the convergence of these two fields.

Specifically, smartphones, driven by their need for processing power, storage space and energy saving are looking towards remote cloud infrastructure. By exploiting the massive cloud capabilities, mobile devices are expecting to find a way to overcome these issues. As a result, the main research question of this work is *how to bring the cloud infrastructure closer to the mobile user?* Naturally, since there are a lot of different architecture designs that can be used to bind mobile and cloud resources, in order to answer this question, we explored different approaches.

A mobile can outsource or offload a task to cloud in order to release the device from doing it. Usually, a task that is outsourced or offloaded is resource-intensive, which means that the device requires considerable computational effort to process it. A task is outsourced at service level (task delegation) and offloaded at code level (computational offloading). Outsourcing a task requires the cloud to be reachable all the time by the device. Thus, network connectivity is mandatory in order to activate the functionality of a mobile app. In contrast, offloading a task allows a mobile app to activate its functionality in presence or not of network communication.

Existent approaches have shown that outsourcing a task enriches a mobile app with sophisticated functionality, but it does not ensure energy saving nor better performance for the app. Since the design and deployment of cloud services mostly follows SOA principles, initially, in this thesis, we investigated how mobile cloud services can be integrated within the mobile apps. We found

out that outsourcing a task to cloud requires to integrate and consider multiple aspects of the clouds, such as resource-intensive processing, asynchronous communication with the client, programmatically provisioning of resources (Web APIs) and cloud intercommunication. To overcome these issues, we proposed a Mobile Cloud Middleware (MCM) framework (multi-cloud middleware approach) that uses declarative service composition to outsource tasks from the mobile to multiple clouds with minimal data transfer.

On the other hand, it has been demonstrated that computational offloading is a key strategy to extend the battery life of the device and improves the performance of the mobile apps. However, computational offloading still faces many challenges pertaining to practical usage. As a result, a mobile app that is instrumented with computational offloading mechanisms can be impacted by a productive or counterproductive effect. Computational offloading is productive when the device saves energy without degrading the normal response time of a mobile application, and counterproductive when the device wastes more energy executing a computational task remotely rather than executing it locally. In this thesis, we also investigated the issues that prevent the adoption of computational offloading, and proposed a framework, namely Evidence-aware Mobile Computational Offloading (EMCO), which uses a community of devices to capture all the possible context of code execution as evidence. By analyzing the evidence, EMCO aims to determine the suitable conditions to offload. EMCO models the evidence in terms of distributions rates for both local and remote cases. By comparing those distributions, EMCO infers the right properties, in which a mobile app has to offload. EMCO shows to be more effective in comparison with other computational offloading frameworks explored in the state of the art.

Finally, beyond the basic benefits of code offloading, we investigated how computational offloading can be utilized to enhance the perception that the user has towards the continuous usage of a mobile application. This perception is modeled as the fidelity, which depicts the time that takes to process a computational task and display its result to the user. Since the cloud provides a vast ecosystem of cloud servers with different computational settings, a task that is offloaded to cloud can be accelerated at multiples levels, which means that the response time of the app can be provided *as a service* based on the suitable and individual perception of each mobile user. Our main motivation behind providing fidelity at multiple acceleration levels is to provide adaptive quality-of-experience (QoE), which can be used as mean of engagement strategy that increases the lifetime of a mobile app. We envisioned that as part of an app released in a store, a counterpart computational service in a cloud also is released by the same party that developed the app to improve the QoE of the users that utilize the app.

Acknowledgements

First and foremost, I would like to thank my CREATOR for giving me once more the strength to fulfill a dream. I would also like to thank to my family for their unconditional support that transcends any distance.

I would like to thank to all the professors that taught me and advised me in becoming a better person and researcher. To **Prof. Marlon Dumas** for helping me to come to Estonia and always being (from the very beginning) a source of inspiration and admiration. To my supervisor, **Prof. Satish Srirama** for his guidance in this journey. To **Prof. Pan Hui** for fostering even further my love for research. To **Prof. Sasu Tarkoma** for enriching my ideas with extraordinary criticism and helping me to improve my creative thinking. To **Prof. Yong Li** for teaching me the importance of shaping my ideas into novel, beautiful, attractive, and consensus stories. To **Prof. Rajkumar Buyya** for his valuable insights about my work. To my opponents, **Prof. Jukka K. Nurminen** and **Prof. Tommi Mikkonen** for their valuable comments that helped me to improve my dissertation. I would also like to express my gratitude to the SIGMOBILE community for showing me the meaning of what is research.

I would also like to express my gratitude to my friends from Mobile Cloud Lab and SyMLab for their friendship and encouragement during my studies, and to all my unconditional friends that always put a smile on my face even in the most complicated times. Specially, I want to thank to Eva Prüüs (and her family), who has always been by my side.

Finally, I want to thank to the University of Tartu and all its academic staff, specially from the Institute of Computer Science. I also want to thank the financial support of the Tiger University Program of the Estonian Information Technology Foundation, the European Regional Development Fund through the EXCS, Estonian Science Foundation grant ETF9287, Target Funding theme SF0180008s12 and the European Social Fund for Doctoral Studies and Internationalisation Programme DoRa.

Contents

List of Figures

List of Tables

Acronyms

AC2DM	Android Cloud to Device Messaging Framework
AI	Aritificial Intelligence
AOSP	Android Open Source Project
API	Application Program Interface
APK	Android Application Package
APNS	Apple Push Notification Service
AOT	Ahead Of Time
CPU	Central Processing Unit
CSV	Comma-Separated Values
D2D	Device to Device
DBMS	Database Management System
DoS	Denial of Service
EC2	Elastic Compute Cloud
EMF	Eclipse Modeling Framework
GCM	Google Cloud Messaging
GEF	Grafical Editing Framework
GLPK	GNU Linear Programming Kit
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IoT	Internet of Things
IM	Instant Messaging
IP	Internet Protocol
JID	Jabber ID
JIT	Just In Time
JSON	JavaScript Object Notation

LB	Load Balancer
LoC	Lines of Code
LP	Linear Programming
MBaaS	Mobile Back-end as a Service
MCC	Mobile Cloud Computing
MPNS	Microsoft Push Notification Service
MQTT	Message Queue Telemetry Transport
OC	Offloading Candidate
OS	Operating System
PaaS	Platform as a Service
PC	Personal Computing
QoE	Quality of Experience
QoS	Quality of Service
REST	REpresentational State Transfer
RSS	Rich Site Summary
RTT	Round Trip Times
S3	Simple Storage Service
SaaS	Software as a Service
SHA	Secure Hash Algorithm
SIP	Session Initiation Protocol
SOA	Service Oriented Architectures
SSL	Secure Sockets Layer
SyncML	Synchronization Markup Language
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VoIP	Voice Over IP
VM	Virtual Machine
XML	Extensible Markup Language
XMPP	eXtensible Messaging and Presence Protocol

Part I

Overview

Chapter 1

Introduction

Mobile and cloud computing are two of the biggest forces in computer science. Nowadays, a user either relies on the mobile or on the cloud to perform computational or data storage tasks [1]. Mobile computing is a form of human - computer interaction, which is extended to fit into human mobility patterns. Mobile computing is potentiated by a distributed infrastructure, which consists of network communication, mobile hardware, and software. With the appearance of smartphones, mobile computing has expanded the horizon to the creation of new devices, communication protocols, software development techniques, business models, etc. Smartphones have pervaded our daily lives through mobile apps, which allow the user to access Internet any time, anywhere and to consume a diversity of services like e-mail, video streaming, image editing, document editing, web browsing, mobile payment, mobile messaging, mobile games, among many others.

On the other hand, cloud computing is a style of computing, in which, typically, scalable resources are provided *as a service* over the Internet to users who need not have knowledge of, expertise in, or control over the infrastructure that supports them. Cloud provisioning can occur at different levels, where each level enables the user to interact with the service at certain granularity. The most common levels are the Infrastructure level (IaaS), the Platform level (PaaS), and the Software level (SaaS). Cloud computing mainly forwards a utility computing model, where consumers pay on the basis of their usage (pay-as-you-go). Moreover, cloud computing promises the ubiquity and availability of virtually infinite resources [2].

Smartphone apps are generally developed following a stand alone or a client-server model. In terms of user functionality, the main difference between them is that a client-server app requires connectivity to a remote source to activate some of its features, e.g. web service. Naturally, the latency in the communication influences the computational effort required by the device to activate these features. Low latency communication can increase energy consumption and degrade the perception that the user has towards the applications. Thus, stand alone apps are preferable to client-server apps (at low latency), because they are not tied to network connectivity, which means that the app functionality is always available in a tolerable manner that does not affect the perception of the mobile user and the device is not overloaded with extra effort to handle the communication. Hybrid apps that merge both models can also be developed. Hybrid apps implement mechanisms for disconnected operations [3], which allow the mobile app to provide stand alone functionality even when the handset is not able to reach the remote system to request information, e.g. e-mail, contacts, docs, etc. Disconnected operations caused by disruption in communication introduce fault tolerance into the mobile systems. In the presence of network communication, the data in the mobile and the remote system is synchronized using protocols like SyncML [4]. Hybrid apps do not augment functionality of the app, but rather allow the user to manage and transport his or her data more easily as these apps also provide a counterpart version for Desktop computers, which can be accessed via Web browser.

However, user's expectation about getting PC-like functionality in the mobile is growing every day. Advances in mobile technologies enable the mobile apps to cope with such computational demands to some extent. Unfortunately, the stand alone operation mode of the mobile devices is constrained by battery life, storage and processing capabilities, which limit the acceptable perceptibility and interactivity of the mobile apps. As a consequence, quality of the results presented to the user can vary abruptly (*fidelity* [5]), which compromises the overall user's experience.

To mitigate the issues of reduced mobile resources, smartphones are looking towards augmenting capabilities via cloud resources [6]. As a result, the domain known as *Mobile Cloud Computing* (MCC) is on the rise [7, 8]. A mobile can outsource or offload a task to cloud in order to release the device from doing it. Usually, a task that is outsourced or offloaded is resource-intensive, which means that device requires considerable computational effort to process it. A task that is outsourced to cloud is time consuming and commonly cannot be processed in the device, e.g. analytics over big sensor data. A task is outsourced to cloud following a SOA (service-oriented architectures) approach using specialized Web APIs that enable the device to control the massive cloud infrastructure at different service levels [6, 9, 10]. In this work, we have defined this access approach to cloud as *delegation schema* [8].

In contrast, a task is offloaded to cloud following a code based approach, where the computational task during runtime is moved from one place (device) to another (surrogate) for its execution. Computational offloading has evolved considerably from its initial notion of cyber-foraging [11]. Cloudlets [3, 12] is one the first proposals in which a low power device through a thin client connects to a rich and nearby server to offload computational tasks. The ultimate goal of the approach is to save energy for the devices. However, the applicability of the approach in high latency networks can cause energy draining rather than energy saving [13, 14, 15], which suggests that the benefits of computational offloading can just be exploited in low latency proximity [3]. To counter this problem, latest works have proposed higher instrumentation of the code [16, 17, 18, 19, 20, 21], e.g. Method, Class, Thread, in order to decrease data transmision to remote cloud. In this work, we also refer to this access strategy to cloud as *offloading schema.*

1.1 Problem Statement

Mobile and cloud computing are converging as prominent technologies that are leading the change to the post personal computing (PC) era. As a result, there is a lot of effort from industry and academy to create innovative solutions that bridge the cloud infrastructure closer to the mobile user [8]. Despite all the previous works [7], MCC is still in its infancy, and thus there are several open issues that need to be answered first before cloud computing can be adopted in the design of future mobile systems [10, 14, 15]. In this work, we ask the main research question, *how to bring the cloud infrastructure closer to the smartphone user?*. This main research question investigates the following problems.

1. *Cloud service integration and composition within smartphone apps* — Generally, service providers in the Internet, e.g. startups, released cloud services that need integration within smartphones apps. Cloud services are accessed via Web APIs, which provide the programmatic routines to configure and consume the service from a mobile client. However, deploying a Web API on a handset is demanding for the mobile operating system when compared with traditional development environments for Web applications. Smartphone introduces many constrains in the development process of an application, e.g. compiler limitations, dependencies, runtime environment incompatibility, etc. Thus, in most of the cases the deployment of a Web API in a mobile application fails. Morever, the problem scales further as the app requires to integrate services from

multiple clouds. Web APIs are not interoperable. Thus, one requires to use a specific Web API from a specific vendor to access a particular service. As a result, mobile clients become thicker and resource consuming. Finally, adapting a Web API requires specialized knowledge of low level programming techniques and most of the solutions are implemented ad hoc. We investigated how to improve the integration of cloud services within the smartphone apps, in short, we answer the question, *how to outsource tasks from the mobile using SOA?*

2. *Invocation of cloud services from the mobile* — Many cloud services require considerable amount of time to process a request, e.g. MapReduce [22], Hive [23], Spark [24], etc. A mobile cannot delegate a task that requires long time waiting as this impacts the user's experience with the app. Moreover, this increases the computational effort of the device to handle the communication. As a result, the device suffers from energy draining. Thus, the problem to overcome is *how to outsource a task that requires long waiting without overloading the communication with constant polling?*

3. *Mobile computational offloading in practice* — Multiple research works have proposed different code offloading strategies to empower the smartphone apps with cloud based resources [17, 18, 19, 20]. Yet, the utilization of code offloading is debatable in practice as the approach has been demonstrated to be ineffective in increasing remaining battery life of mobile devices [13]. The effectiveness of an offloading system is determined by its ability to infer where the execution of code (local or remote) represents less computational effort to the mobile, such that by deciding *what, when, where and how to offload* correctly, the device obtains a benefit. Code offloading is productive when the device saves energy without degrading the normal response time of the apps, and counterproductive when the device wastes more energy executing a computational task remotely rather than executing it locally.

   Current works offer partial solutions that ignore the majority of these considerations in the inference process. Most of the proposals demonstrate the utilization of code offloading in controlled environment by connecting to low-latency nearby servers, e.g. lab setups, and inducing the code to become resource intensive during runtime [25], e.g. passing an input that requires lot of processing. As a result, in practice, in most of the cases, computational offloading is counterproductive for the device [13, 14]. Thus, at this point, the main questions about the strategy are: *can code offloading be utilized in practice,* and *what are the issues that prevent code offloading to yield a positive result?*

4. *Adapting App Quality-of-Experience (QoE)* — Past works modeled QoE in terms of the response time of an app, in this context latency of the response depicts the time that it takes to process a computational task and display its result to the user [5]. It has been demonstrated that an app can be adapted to multiple levels of responsiveness depending on the allocated resources for app's execution [26]. Unlike client-server apps, in which QoE can be improved from the network operator or service provider side, it is difficult to cope with the user's demands in QoE for stand alone apps due to the constrained resources and high diversity of the smartphones. The problem that we overcome is *how to adapt the app to different QoE levels using computational offloading in order to improve user's experience?*.

5. *Scalability of mobile systems supported by cloud* — Generally, systems that follow the design of SOA can properly scale in the cloud for multi-tenancy after the systems are benchmarked based on performance and capacity. In the case of computational offloading, this is an open issue as most of the proposed systems [7] foster one server per each mobile architecture, which is unfeasible if we consider the amount of smartphones nowadays and the provisioning cost of running the server for longer periods. Thus, the question to answer is: *is it possible to provide computational offloading as a service that scales based on incoming load of mobile users?*

1.2 Methodology

The research questions exposed in this work express a need for understanding more about the infrastructural support that cloud computing can provide to the mobile computing architectures. Hence, a design research approach based on architectural modeling and development [27, 28], performance evaluation [29] and capacity planning [30] is taken in order to answer the questions. Naturally, since the design of a mobile architecture influences the quality of experience that a user has towards a smartphone application, quantitative experimental research is conducted [31] in order to determine how cloud infrastructure improves certain aspects of the device, such as energy, performance, usage, etc.

1.3 Contributions

While in principle, computational offloading and task delegation are viable methods to augment the capabilities of mobile devices with cloud power, they focus on different issues and implement different techniques [32]. Consequently, these approaches enrich the mobile applications from different perspectives at diverse computational scales. Our contributions are described as follows:

- *Mobile Cloud Middleware (MCM)* — MCM counters the problems of accessing multiple cloud services through Web APIs and invoking a time consuming operation from a mobile app [10]. The middleware abstracts the Web APIs of multiple clouds at different levels and implements a unique interface that responds, e.g. JSON-based (JavaScript Object Notation), according to the cloud services requested, e.g. REST-based (Representational State Transfer). MCM provides multiple internal components and adapters, which manage the connection and communication between different clouds. Since most of the cloud services require significant time to process the request, it is logical to have asynchronous invocation of the cloud service. Asynchronicity is added to the MCM by using push notification services provided by different mobile application platforms and by extending the capabilities of a XMPP (eXtensible Messaging and Presence Protocol) based on IM (Instant Messaging) infrastructure [33]. Furthermore, MCM fosters a flexible and scalable approach to create hybrid cloud mobile applications based on declarative task composition. Task composition is considered for representing each mobile task to be delegated as a *MCM delegation component*. A composed task is developed graphically in an Eclipse plugin based on user driven specifications and it is modelled as a data-flow structure, where each task depicts a cloud service to be invoked. Once developed, a composed task is deployed within the middleware for execution that is triggered by a single invocation from the mobile. This means that data transmission is minimized in order to decrease computational effort in the device for going cloud-aware. Finally, MCM prototype was extensively analyzed for its performance and scalability under heavy loads, and the details are addressed in Chapter 3.

- *Evidence-aware Mobile Computational Offloading (EMCO)*— EMCO overcomes the challenge to develop a code offloading architecture that can potentiate the sustainability of power-consuming applications in practice. An offloaded task that is unfavorable for the device is the result of a wrong decision process, which tends to get imprecise, based on the scope of observable parameters of the system that the process can consider into its decision [14, 15]. We believe that computational offloading to cloud is possible without a negative impact on the device. We think that a richer context can be expressed in terms of multiple dimensions, e.g. *what, when, where, how, etc.* A dimension defines the properties that a mobile application must meet in order to offload a task. For instance, a face recognition app installed on a device (*which*) offloads a task at code level (*what*) under conditions (*when*) to a remote server of type (*where*) to be executed (*how*) in parallel. Moreover, dimensions can scale to consider other mobility parameters, e.g. *user's location*, which can facilitate the process of pre-caching apps functionality. The key insight of our work is to instrument the computational offloading mechanisms with data analytics in order to reduce the negative impact that arises from offloading to cloud due to wrong decisions, inaccurate code profiling approaches and non-adaptive mechanisms. As a result, we develop a framework, namely, *E*vidence-aware *M*obile *C*omputational *O*ffloading (EMCO), where evidence is history data collected at the cloud from a community of devices about the local and remote execution of a smartphone app [5]. EMCO extracts the dimensions from an analytic process over that information. The purpose of the analysis is to characterize the effect of computational offloading based on all possible contexts captured by the community. Moreover, the characterization is also utilized to identify reusable results of generic code invocations that can be utilized to respond to requests from other apps offloading to cloud. This accelerates even further the response time of the apps. We envision that as a part of the characterization process, pre-cached functionality from the entire mobile application can be requested on demand. In this manner, our cloud assistance approach encourages a system, where the cloud is the expert and mobiles ask the cloud for its expertise. Major modifications are made in EMCO when compared with other frameworks. At mobile app level, we equip our computational offloading mechanism with the ability to feed up from incremental data analytics coming from the cloud. We implement an asynchronous mechanism that uses push notification technologies to deliver the data. The main advantage of this mechanism is fault tolerence for disconnected operations, which means that the data sent from the cloud will reach the device eventually when connectivity is available. At surrogate level, we integrate the Dalvik machine from Android-x86 directly into the server to create a Dalvik-x86 surrogate, which provides better scalability for multi-tenancy. Finally, we developed a use case based on a chess game with artificial intelligence to evaluate the framework. The evaluation shows that the proposed framework is more effective to overcome the limitations that prevent the adoption of code offloading in practice when compared with other solutions.

- *Adaptive App QoE as a Service* — From our study about computational offloading, which is presented in Chapter 4, we identified that besides increasing battery life of the device and improving performance of the apps, computational offloading can be utilized to enhance the perception that the user has towards the continuous usage of the app. Since the cloud provides a vast ecosystem of surrogates with different computational settings, a task that is offloaded to cloud can be accelerated at multiple levels, which suggests that the response time of the app can be provided as a service based on the suitable and individual perception of each mobile user. In other words, continuous computational offloading can smooth the overall execution of a mobile application, which means that a user can obtain better quality of experience, which fits his or her own expectations. The key insight of this work is that by providing multiple levels of acceleration for fidelity, it is possible to create an engagement strategy to foster longer application usage. Adaptation of QoE occurs based on history of app usage by detecting when a user loses interest in using an app. Lose of interest is depicted by session length and frequency of sessions of the app. The goal of the adaptation is to engage the user by providing better QoE and to endure that engagement for longer periods in order to increase the productive and competitive life of the app in the application store. We investigated how to use computational offloading to dynamically adapt the fidelity of smartphone apps to meet QoE requirements of the user. Naturally, to achieve our goal, we redefine current offloading model, where a device is binded to a specific server counterpart in the cloud. The reason is that this model is unfeasible in practice as it requires the cloud server to be active permanently to provision computational resources to a unique device, which means that paid resources are unnecessarily wasted when the server is not handling computational requests. Moreover, we think that the allocation of some high capability servers as surrogates is unrealistic for a single user in long term due to the high provisioning price. As a consequence, we extended our computational offloading framework with the ability to provision continuous computational offloading in multi-tenancy environments. Moreover, our system supports QoS policies to dynamically optimize the number of surrogates needed to handle a specific load of computational offloading requests without affecting the QoE of the active apps offloading to cloud. In this manner, QoS is introduced in the computational service provisioning of the cloud while considering QoE aspects of the mobile user.

1.4 Outline

This thesis is summarized as follows:
Chapter 2: presents a review in MCC, which explores the most relevant works so far in the domain.
Chapter 3: introduces the concept of Mobile Cloud Middleware, goals, architecture design, implementation details and demonstrates MCM in action by presenting several case studies in practice.
Chapter 4: highlights the problems of current computational offloading frameworks and introduces our proposed framework EMCO, which counters the problems of computational offloading in real scenarios.
Chapter 5: explains how to transform a computational offloading architecture into a cloud service that provisions QoE for smartphones apps.
Chapter 6: presents the conclusions of the thesis.
Chapter 7: introduces the future directions that are identified from the thesis.

Chapter 2

A Review of Mobile Cloud Computing

In the emerging ecosystem of mobile cloud computing, a rich mobile app is one that, at low resource consumption, processes huge amounts of information and presents its result to the user at high fidelity levels. While the cloud provides to the user the ubiquitous computational platform to process any task, the smartphone grants the user the mobility features to process a computational task at any time, anywhere. Thus, bridging the cloud closer to the mobile user is a reasonable approach to obtain PC-functionality on the move. In this section, we explore how to bring the cloud to the vicinity of the mobile.

2.1 Service-oriented Mobile Cloud

Cloud computing is changing the way in which services are developed and deployed over the Internet. Service provisioning has never been so easy for the developer, who now does not have to worry about the underlying infrastructure that hosts the service. Cloud computing can be defined as a style of computing, in which, typically, scalable resources are provided *as a service* over the Internet to users who need not have knowledge of, expertise in, or control over the infrastructure that supports them [6]. Cloud computing fosters an *elastic approach*, where the capacity of a service can decrease or augment during runtime and implements a *utility model*, where a user pays the cost of using that service for a specific period of time. The provisioning of cloud services can occur at different levels as shown in Figure 2.1, where each level enables the user to interact with the service at certain granularity.

Basic cloud levels are the Infrastructure level (IaaS), Platform level (PaaS) and the Software level (SaaS). IaaS provides computational and storage services. In the case of computational service, IaaS provisions physical or virtu-

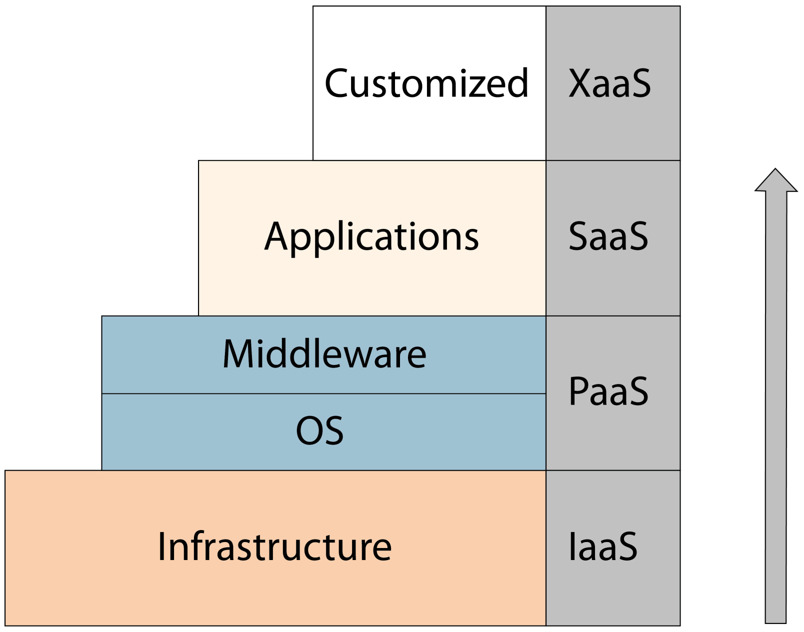


Figure 2.1: Cloud service models

alized servers as instances, which can be accessed via multiple means such as Unix terminal, remote desktop, etc. An instance is associated to a type, which depicts the computational capabilities of a server, e.g. available memory, number of processors, etc. The cost of the instance is proportional to its type[1]. In the case of storage service, IaaS provisions online storage with high replication schema. Generally, replication is used to guarantee availability of the data in the cloud. PaaS provides the development platform, e.g. middleware, which can be used by the software developer to build and deploy custom applications. SaaS delivers end-user applications, which can be accessed via any Web browser. Finally, other levels are customized on top of these basic levels. This customization is denoted as *XaaS*, where *X* represents the customized software that is encapsulated for a particular purpose, e.g. NaaS (Network as a Service), MaaS (Monitoring as a Service), etc.

An application deployed in the cloud as a service follows an SOA design, the principles of which can be exploited to re-configure the service based on demand. Generally, a service is re-configured depending on QoS policies, e.g. response time of a request, which guarantees the conditions of the provisioning, e.g. integrity, availability, reliability, etc. Many different policies can be applied to re-configure a cloud service, e.g. workload of users accessing the service, computational effort to execute the application, etc. This re-configuration process is also known as cloud scaling, which can be of two types, horizontal and vertical. Figures 2.2 and 2.3 show these two types. Horizontal scaling (Figure 2.2) consists of distributing the workload of the service among multiple

[1]http://aws.amazon.com/ec2/instance-types/

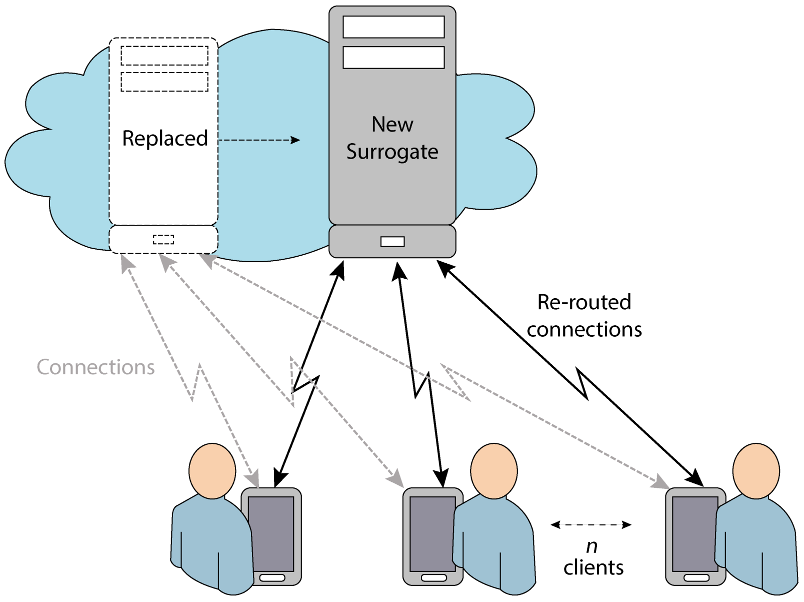


Figure 2.2: Cloud scaling models, vertical scaling

back-end instances running the application. Instances can be added (scaling out) or removed (scaling in) dynamically. This approach requires the utilization of a load balancer as front-end in order to distribute the workload among back-end. Vertical scaling (Figure 2.3) consists of replacing the actual computational capabilities of the instance running the service with higher (scaling up) or lower (scaling down) computational settings.

Scaling is a key feature of cloud computing as it allows the application to adapt to different requirements on the fly, e.g. handling different load of users, improving application performance, etc. For instance, consider a steganography service that receives as parameters two images and it's running in one cloud server. When a user uploads the images, the service encoded/decoded the images into a single result. The response time of the service depends on the computational capacity of the server to process the request. If the response time of the service is too long, the performance of the service can be accelerated by augmenting the number of servers that process the request, in other words, the request is parallelized or the service is scaled.

Accessing the cloud infrastructure from a mobile application requires to assume that network connectivity is always available and there is not disruption in the communication channel. A mobile application can rely on cloud support for the following reasons:

- Activate certain functionality of the mobile application, which is not available when the device is in stand alone mode (online mode), e.g. web service.

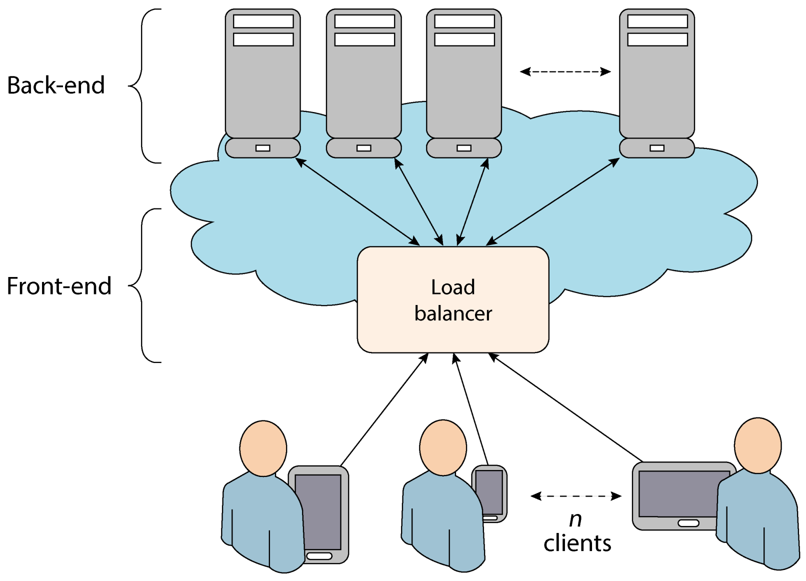


Figure 2.3: Cloud scaling models, horizontal scaling

- Replicate the data stored in the mobile with a storage service in the cloud, e.g. SyncML. Data replication is useful for the device in order to augment storage space, for instance, one picture in the mobile is uploaded to cloud, and then the cloud sends a smaller version of the picture back to the device. A mobile user can utilize this smaller version to access the original picture at any time. Naturally, this feature is optional, it can also be the case that the original picture can be available both in the device and the cloud, but in this case, data replication is just useful to decentralized data from the mobile, so that it can be accessed or synchronized with other devices. Data synchronization is available in both, online and offline operation modes of the device.

- Release the device from computational processing. By relying on an external server which is equipped with a similar runtime environment like the device, it is possible to move a computational task from the device to the external server in order to release the device from processing the task. Naturally, the device has to estimate when to move the task to the external server because it can also be the case that the effort required for the device to process the task externally is higher than processing the task locally.

Depending on the purpose of the mobile application, it has to be adapted to follow a specific access schema. Two different access schemas can be defined: delegation and offloading. The rest of this subsection describes how a smartphone app can implement such schemas to access cloud resources.

2.1.1 Delegation of Mobile Tasks to Cloud

Generally, the integration between mobile and cloud is highly decoupled. This loose integration happens through a mediator (aka middleware) that controls every aspect in the communication and coordinates the attaching/detaching between the back-end infrastructure and the mobile device. A mobile device request access to a cloud resource by sending a request to the middleware, which reads the request and performs the cloud operation. Once the result of the operation is obtained, the middleware encapsulates the result and sends it back to the mobile. Figure 2.4 shows a SOA access schema for mobile cloud. In this delegation schema, the middleware abstracts the complexity of accessing cloud into a single and centralized interface. The middleware implements all the logic to manage network communication with the mobile, e.g. protocol adaptation; and all the routines to manage the invocation of cloud services, e.g. service composition, among others.

Outsourcing a mobile task to a remote server is a common operation that is supported by any mobile platform through various mechanisms, e.g. Web sockets, REST-based requests, etc. Different mobile platforms or versions of the same mobile platform implement different approaches for managing network communication. Usually, network communication between mobile and cloud is synchronous. However, since many cloud services require considerable time to process a task, mobile devices are also equipped to support asynchronous communication. For instance, Android platform level-10 handles REST-based requests synchronously. In contrast, Android platform level-16 and higher forces the developer to extend any network communication with the *AsyncTask Class* running on a different thread so that it will be executed asynchronously in the mobile background. An asynchronous approach is preferable compared to a synchronous one as asynchronous communication guarantees to keep the real-time interactivity of a mobile application and to maintain the normal execution of a mobile application in case an exception arises.

A mobile application that requires resource-intensive functionality of the cloud can benefit from an asynchronous communication in order to delegate and monitor the status of a time consuming task in the cloud. However, this approach introduces several other drawbacks, such as energy consumption (e.g. keeping an open connection while transaction is performed), reliability in the communication (e.g. what happens if the connection fails?) and recoverability of a cloud transaction (e.g. fault tolerance strategies) among others. As a result, middleware support is encouraged in order to reduce the computational and energy overheads in the device [6].

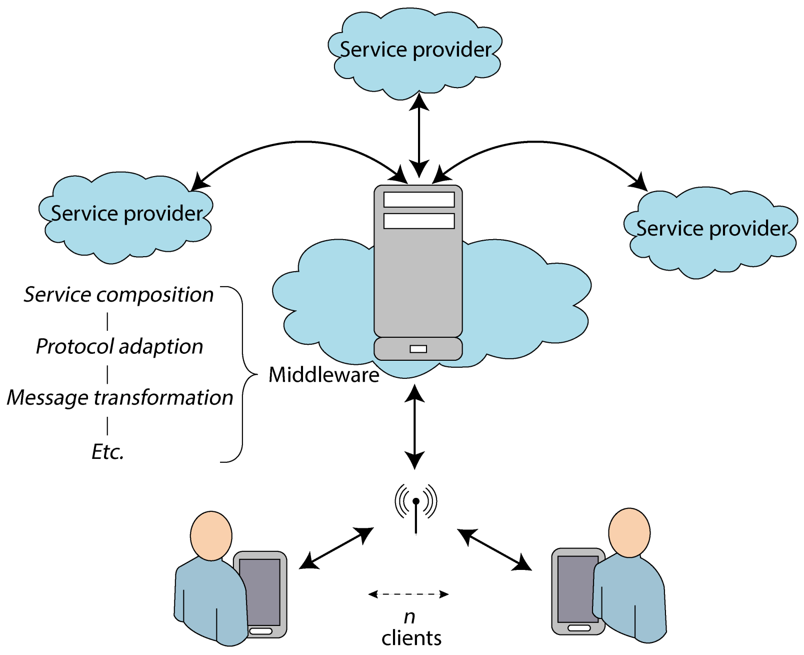


Figure 2.4: Accesing cloud infrastructure via service-oriented middleware

In this context, multiple middleware frameworks have been proposed in the literature. Specifically, middlewares that allow a mobile application to integrate cloud functionality have proposed new architecture designs. Among those, the most relevant can be mentioned. MCCM (Mobile Cloud Computing Middleware) [9], which uses a middleware standing between the mobile and the cloud for the composition of web services into mashups. It handles creating user profiles from the context of the mobile phone, storing the system configuration (pre-defined set of WS which can be consumed from the handset) and the service configuration respectively, for managing existing resources in the cloud. However, such middleware and the API support seems to be tightly coupled in contrast to our proposed solution. Moreover, from studying the applications developed with MCCM we observed that it is not suitable for service invocations that require resource-intensive processing.

μCloud [34] is a middleware framework that models the functionality of a mobile application as graph. Each node in the graph depicts a service, which can be hosted in different providers. The idea is that a graph can be represented as workflow, which can be executed by the mobile application. While μCloud provides a simple prototype of the system, the middleware lets many issues open regarding scalability, offline operation mode of the mobile application, energy consumption of the mobile application, etc. The framework just presents a discussion regarding those issues.

Cloud Agency [35] is another solution that aims to integrate GRID, cloud computing, and mobile agents. Cloud Agency proposes to use GRID as middleware between the mobile and the cloud infrastructure. The specific role of

GRID is to offer a common and secure infrastructure for managing the virtual cluster of the cloud through the use of mobile agents. Agents introduce features that provide the users a simple way for configuring virtual clusters and establish the communication between the middleware and the mobile. However, such solution does not explain how agents enable keeping soft-real time responses when a cloud service is invoked from the mobile or how the communication with different cloud providers is handled by the GRID. Moreover, it does not quantify the effort required by the mobile to access cloud using GRID, e.g. energy. Thus, it is hard to appreciate the benefits of such architecture design.

Unlike previous works, we claim that the integration of cloud services within the mobile applications can happen at different cloud computing levels. Since accessing the massive cloud infrastructure requires considerable computational effort for the device, e.g. time, energy, etc., we provide the architecture design that enables the mobile to reduce the effort of using cloud resources in a hybrid way. Morever, our solution allows the developer to create thin clients, which means that the mobile client is lighter when compared with the thick clients needed in other proposed architectures. Finally, we relied on push technologies to equip the mobile applications with the asynchronous features to handle resource-intensive executions of a remote cloud. The framework is described in Chapter 3.

2.1.2 Cloud Service Integration for Mobile Applications

While cloud infrastructure is programmable through the utilization of Web API, different clouds present different levels of granularity for configuring the cloud resources. Depending on the cloud vendor architecture, a Web API may be used for deploying applications from the scratch (e.g. MapReduce) or for accessing existent functionality, which can be integrated with other software applications. For instance, Amazon API and typica API[2] allow to manage EC2 instances (run scripts, attached volumes, etc.), jetS3t[3] API provides access to S3/Walrus and GData API[4] enables configuring services such as calendar, analytics, etc. Consequently, software applications that require cloud intercommunication are forced to implement multiple Web APIs.

Since a Web API may suffer from disuses, changes or replacements with time due to many reasons such as new Web API releases, improvements, etc., the development of applications becomes tightly coupled and difficult to port,

[2]http://code.google.com/p/typica/
[3]http://jets3t.s3.amazonaws.com/toolkit/guide.html
[4]http://code.google.com/apis/gdata/

to reuse and to maintain. To address most of these problems, several open source projects have been started. For instance, jclouds[5] is a multi-cloud library that claims transparent communication with different cloud providers and the reuse of the source code while working with different services. Jclouds provides a core library, which contains the core functionality and a set of libraries, which handles the communication with any particular cloud. Current version of jclouds supports Amazon, GoGrid, VMWare, Azure and Rackspace. Other projects such as Apache Libcloud[6] and Dasein Cloud API[7] also provide a Web API that abstracts away differences among multiple cloud providers, however currently, jclouds API is the one that supports more cloud vendors. Other projects such as deltacloud[8] focus on managing particular services with the same Web API. For instance, deltacloud allows utilizing the same code routines for starting an instance in Amazon and in Rackspace.

Even though there are many Web APIs that can be used for abstracting the communication with different cloud vendors, most of them are not deployable within a mobile platform due to several drawbacks such as additional dependencies, incompatibility with the mobile platform, integration with the compiler, etc. For instance, the dalvik virtual machine of Android offers just a portion of Java functionality. Consequently, the richness of the language can not be exploited and libraries such as jclouds or typica can not be executed on mobile platforms.

Furthermore, the development of mobile cloud applications using Web APIs dramatically increases the effort of implementing simple operations such as offloading a file to remote storage, etc. For instance, we have ported jetS3t API for Android platform level-10 and its apk file requires approximately 4.55 Mb of storage on the mobile. Moreover, the application uses synchronous communication for delegating data to cloud storage (S3/Walrus). Therefore, the mobile resources get frozen while the transaction is being completed ($\approx$ 6 seconds when uploading a file of 3 Mb using a bandwidth with an upload rate of $\approx$ 1409 kbps). Even more, the source code is not compatible with higher versions of Android. Consequently, the application is not portable and a complete re-implementation is needed for using it within other Android versions.

Currently, Web APIs for mobiles are still under development and often target simple services such as storage. For instance, Amazon provides a stable Web API for accessing S3 for both Android and iOS platforms. Moreover, at the time of writing this thesis, other platforms like Windows Phone are also

[5]http://code.google.com/p/jclouds/
[6]https://libcloud.apache.org/
[7]http://dasein-cloud.sourceforge.net/
[8]http://incubator.apache.org/deltacloud/

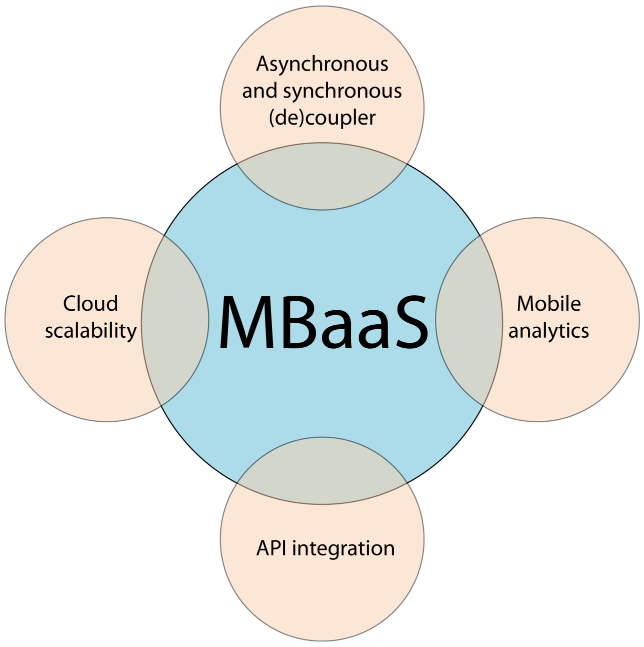


Figure 2.5: Conceptual components of *Mobile Back-end as a Service*

supported, but those Web APIs have just been released as beta version. We have to mention as well that Web APIs from open source projects like JetS3t, get deprecated due to the lack of support to upgrade the libraries to newer versions of the mobile platform.

2.1.3 Challenges and Technical Problems

Cloud infrastructure enters as Mobile Back-end as a Service (MBaaS) into mobile app development to facilitate the process of service integration and reutilization on the fly. However, unlike traditional systems, accessing the cloud from a mobile is complex as the systems that support the mobile communication are constrained by many hardware, software and user's requirements. For instance, just to mention some, an app that implements resource intensive operations is unsuitable as it drains the battery life (hardware). An app, the functionality of which is highly dependent on distributed services is hard to develop and deploy, and thus, the long lifetime of the app cannot be ensured (software). The responsiveness of an app must meet the expectations of the user, otherwise, the app is discarded (user's experience) [29]. Cloud computing can overcome many of these problems by providing suitable access schemas for mobile systems.

As a result, we highlight the most basic components that a MBaaS has to implement. The components are depicted in Figure 2.5. Each component is described as follows:

- ***API integration*** — encapsulates the Web APIs from different services and exposes an interface that can be easily accessed by mobiles. API integration foster the interoperability and integration of the services.
- ***Mobile analytics*** — collects information about app usage in order to identify the most suitable QoE metrics in which an app hast to be executed. The purpose of adapting an app based on QoE metrics is to increase user's satisfaction and engagement.
- ***Cloud scalability*** — ensures the elastic provisioning and dynamic allocation of the back-end to handle heavy load of mobile users and to increase performance of the system.
- ***Asynchronous and synchronous (de)coupler*** — enables the device a flexible coupling and decoupling with the back-end in order to avoid energy draining caused by maintaining an active communication channel.

2.2 Mobile Cloud Offloading

Mobile cloud offloading (aka computational offloading, cyber-foraging) has been re-discovered as a technique that allows to empower the computational capabilities of mobiles with elastic cloud resources. Computational offloading refers to a technique, in which a computational operation is extracted from a local execution workflow, later, that operation is transported to a remote surrogate for being processed, and finally, the result of that processing is synchronized back into the local work flow [15]. Cloudlets [3] is one of the initial works that propose the augmentation of mobile computational resources with nearby servers in proximity, e.g. hot spots. Cloudlets overcome the problem of connecting to high latency remote servers by bridging the cloud infrastructure closer to the mobile user. The motivation of reducing the latency between mobile device and cloud is to enrich the functionality of the mobile applications without degrading its perception and interaction in environments where network communication changes abruptly. Figure 2.6 shows a basic cloudlet architecture. The architecture consists of two parts, a client and a server located in proximity, which means that there is no network hopping between the device and the server. A nearby server is managed by a service provider using virtual machines. A virtual machine is migrated from the cloud of the service provider to the nearby server, so that cloud service provisioning (create, launch or delete) for the mobile can occur from the nearby server. Alternatively, the service provider also can migrate a service to other types of infrastructure,

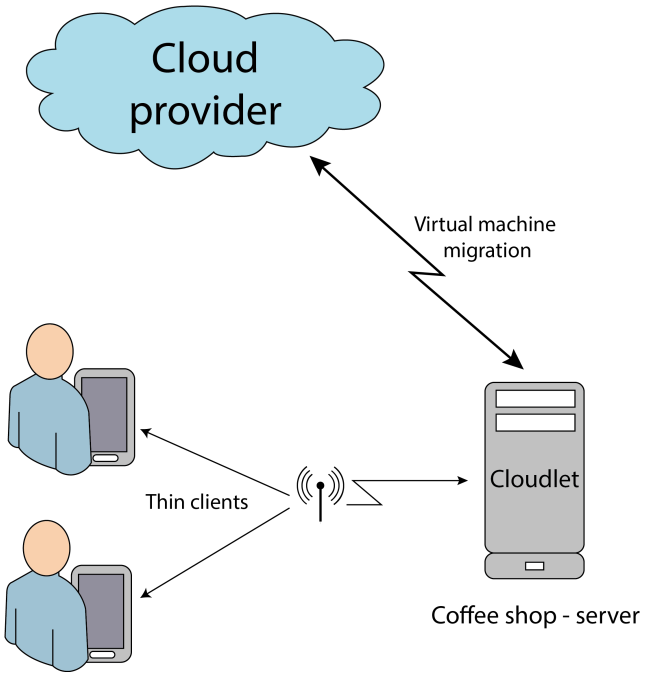


Figure 2.6: Components and functionality of a cloudlet system

e.g. base stations, in order to reduce the communication latency with the device [12].

While a cloudlet overcomes the problems that arise from high communication latency, the deployment of a cloudlet is a complex task, as involves to introduce specialized components or modify existent ones at low level of granularity, e.g., hardware. Thus, its adaptation is neither flexible nor scalable. As a result, many other solutions have been proposed. The goal of these solutions is to optimize the delegation of computational tasks by relying on higher manipulation of the source code of the applications. In this process, computational tasks are delegated to powerful machines at code level as explained in subsection 2.2.1. Notice that a cloudlet also can be equipped with strategies to offload code. However, in this case, a cloudlet inherits the drawbacks of code offloading, which destabilizes its architecture. Thus, a cloudlet cannot guarantee any longer the smooth execution of the application at low resource consumption.

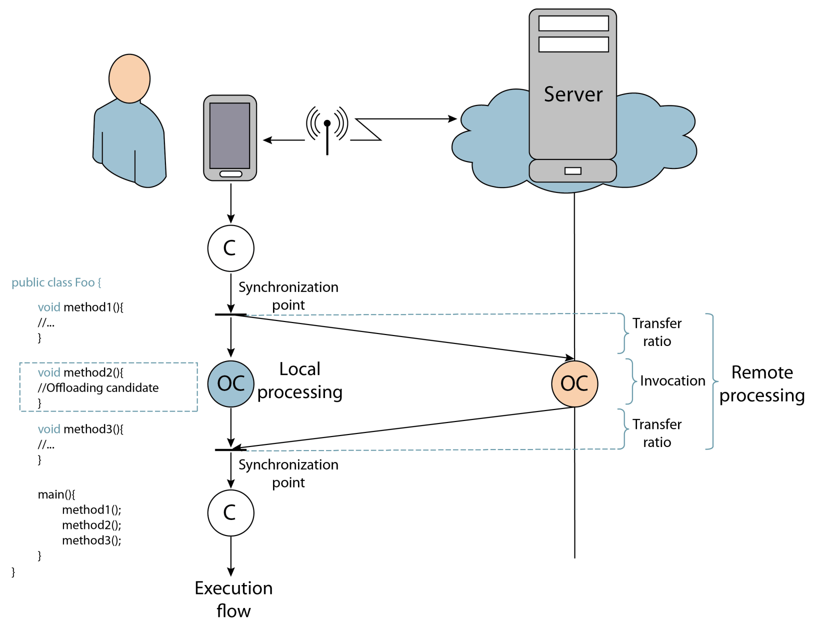
2.2.1 Mobile Code Offloading

Code offloading is the opportunistic process that relies on remote servers to execute code delegated by a mobile device. In this process, the mobile is granted with the local decision logic to detect resource-intensive portions of code, so that in the presence of network communication, the mobile can estimate where the execution of code will require less computational effort (remote or local), which leads the device to save energy [36]. The evaluation of the code requires to consider different aspects, for instance, *what code to offload*, e.g., method name; *when to offload*, e.g. RTT (Round Trip Times) thresholds; *where to offload*, e.g. type of cloud server; *how to offload*, e.g. split code into n processes, etc. Most of the proposals in the field do not cover all these aspects, and thus we describe a basic offloading architecture, which is shown in Figure 2.7. The architecture consists of two parts: a client and a server. The client is composed of a code profiler, system profilers and a decision engine. The server contains the surrogate platform to invoke and execute code. Each component is described in detail as follows:

1. ***Code Profiler*** is in charge of determining *what to offload.* Thus, portions (C) of code —*Method, Thread or Class*—are identified as offloading candidates (OCs). Code partitioning requires the selection about the code to be offload. Code can be partitioned through a diversity of strategies, for instance, a software developer can select explicitly the code that should be offloaded using special static annotations [17], e.g. *@Offloadable, @Remote, etc.*. Other strategies analyze the code implicitly during runtime by an automated mechanism [18]. Thus, once the application is installed in the device, the mechanism selects the code to be offloaded. In order to estimate if a portion of code is intensive or not, the mechanism implements strategies such as static analyzes, history traces, etc. Automated mechanisms are preferable over static ones as they can adapt the code to be executed in different devices. Thus, automated mechanisms overcome the problem of brute-force development in code offloading, which consists in adapting the application every time that is installed in a different device.

2. ***System profilers*** are responsible for monitoring multiple parameters of the smartphone, such as available bandwidth, data size to transmit, energy to execute the code, etc. These parameters influence *when to offload* to cloud. Conceptually, the offloading process is optional, which should take place when the effort required by the mobile to execute the OC is lower in the case of remote invocation than local execution. Otherwise, offloading is not encouraged as excessive amount of energy and time is consumed in transmission of data to cloud.

3. ***Decision engine*** is a reasoner that infers *when to offload* to cloud. The engine retrieves the data obtained by the system and code profilers, and applies certain logic over them, e.g. linear programming, fuzzy logic, markov chains, etc., so that the engine can measure whether the handset



```
public class Foo {

    void method1(){
    //...
    }

    void method2(){
    //Offloading candidate
    }

    void method3(){
    //...
    }

    main(){
        method1();
        method2();
        method3();
    }
}
```

Figure 2.7: A code offloading architecture: components and functionalities

obtains or not a concrete benefit from offloading to cloud. If the engine infers a positive outcome, then the mechanism to offload is activated and the code is invoked remotely, otherwise, the code is executed locally. A mobile offloads to cloud in a transfer ratio that depends on the size of the data and the available bandwidth [36]. Usually, when code offloading is counterproductive for the device, it is due to a wrong inference process, which gets inaccurate based on the scope of observable parameters that the system profilers can monitor [14].

4. ***Surrogate platfrom*** is the remote server located in the proximity of the device or in the cloud, which contains the environment to execute the intermediate code sent by the mobile, e.g. Android-x86. The remote invocation tends to accelerate the execution of code as the processing capabilities of the surrogate are higher than those of most smartphones. In this context, the type of instance that acts as surrogate is important as determines the acceleration in which a computational task is executed, which impacts the overall response time of the app perceived by the user. Higher types of instances can process a task faster than lower ones as higher types can rely on larger memory span and higher parallelization in multiple processors. Finally, the type of instance also defines its capacity to handle multiple code offloading requests at once.

2.2.2 Computational Offloading in the Literature

Table 2.1 describes most relevant proposals in code offloading. The table compares the key features of the offloading architectures, namely the main goal, how code is profiled, the adaptation context, the characterization of the offloading process, and how code offloading is exploited from mobile and cloud perspectives. From the table, the main goal defines what is the actual benefit for using the associated framework. The mechanism used to profile code provides information about the flexibility and integrability of the system. The adaptation context specifies the considerations taken by the system to offload. The characterization means whether the offloading system has a priori knowledge or not about the effects of code offloading for the components of the system. Finally, the exploitation highlights the mobile benefits obtained from going cloud-aware, and the features of the cloud that are leveraged to achieve those benefits. Moreover, we can also observe that currently, most of the effort has been focused on providing the device with an offloading logic based on its local context.

MAUI [17] proposes a strategy based on code annotations to determine which methods from a Class must be offloaded. An annotation is a form of metadata that can be agregated into the source code, e.g. classes, methods, etc. An annotation allows the compiler to apply extra functionality to the code before its called, e.g. override annotations. MAUI uses annotations to identify methods that are resource-intensive for the device. Annotations are introduced within the source code by relying on the expertise of the software developer. Once the code is annotated, MAUI transforms all the annotated methods into an offloadable format. This format equips the methods with RMI capabilities. Since MAUI targets Windows Phones, it is developed using *.NET* framework. Thus, RMI happens by using the WFC (Windows Communication Framework). During application runtime, the MAUI profiler collects contextual information, e.g. energy, RTT, etc., if the MAUI profiler detects a suitable context to offload code, then the execution of the code is delegated to a remote server instead of being performed by the device. While MAUI is successful in saving energy and shortening the response time of the mobile applications, it suffers from many drawbacks. Since MAUI uses code annotations, it is unable to adapt the execution of code in different devices. Thus, the developer is forced to adapt an application to a specific device, which is considered a brute-force approach. Moreover, MAUI suffers from scalability, which means that each mobile that implements MAUI requires to be attached to one specific server acting as a surrogate.

Table 2.1: Code offloading approaches from a mobile and cloud perspectives

Code offloading strategies					**Mobile perspective**	**Cloud perspective**
Framework	*Main goal*	*Code profiler*	*Offloading adaptation context*	*Offloading characterization*	*Applications effect*	*Features exploited (Besides server)*
MAUI [17]	Energy-saving	Manual annotations	Mobile *(what, when)*	None	Low resource consumption, Increased performance	None
Odessa [37]	Responsiveness	Automated process	Mobile	None	Applications are up to 3x faster	None
CloneCloud [18]	Transparent code migration	Automated process	Mobile *(what, when)*	None	Accelerate responsiveness	None
ThinkAir [19]	Scalability	Manual annotations	Mobile + Cloud *(what, when, how)*	None	Increased performance	Dynamic allocation and destruction of VMs
COMET [20]	Transparent code migration (DSM)	Automated process	Mobile *(what, how)*	None	Average speed gain 2.88x	None
EMCO [21]	Energy-saving, Scalability (Multi-tenancy)	Automated process	Mobile + Cloud *(what, when, where, how)*	Based on historical crowdsourcing data	Based on context (Low resource consumption, increased responsiveness, etc.)	Dynamic allocation and destruction of VMs, Big data processing, Characterization-based utility computing
COSMOS [38]	Responsiveness	Manual process	Mobile *what*	None	Increased performance by choosing right surrogate	Resource allocation decided by user
Other works [7]	Responsiveness	Manual annotations	Mobile *what, when*	None	Increased performance	None

Similarly, CloneCloud [18] encourages a dynamic approach at OS level, where a code profiler extrapolates pieces of bytecode of a given mobile component to a remote server. Unlike MAUI, CloneCloud offloads code at *thread level*. CloneCloud uses static analysis to partition code, which is an improvement over the annotation strategy proposed by MAUI. By using a static analyzer, code can be annotated dynamically. Thus, code to offload is adapted based on the type of device without modifying or changing any implementation of the application. However, code profiling is complicate as its execution is non-determistic. Thus, it is difficult to verify the runtime properties of the code, which can cause unnecessary code offloading or even offloading overhead. Moreover, many other parameters also influence when choosing a portion to code to offload, e.g. the serialization size, latency in the network, etc.

COMET [20] is another framework for code offloading, which follows a similar approach as CloneCloud. COMET strategy puts emphasis on how to offload rather than what and when. COMET's runtime system allows unmodified multi-threaded applications to use multiple machines. The system allows threads to migrate freely between machines depending on the workload. COMET is a realization built on top of the Dalvik Virtual Machine and leverages the underlying memory model of the runtime to implement distributed shared memory (DSM) with as few interactions between machines as possible. COMET makes use of VM-synchronization primitives. Multi-thread offloading accelerates even further the execution of applications in which code can be parallelized.

ThinkAir [19] framework is one which is targeted at increasing the power of smartphones using cloud computing. ThinkAir tries to address MAUI's lack of scalability by creating virtual machines (VMs) of a complete smartphone system on the cloud. Moreover, ThinkAir provides an efficient way to perform on-demand resource allocation, and exploits parallelism by dynamically creating, resuming, and destroying VMs in the cloud when needed. However, since the development of mobile application uses annotations, the developer must follow a brute-forced approach to adapt his/her application to a specific device. Moreover, resource allocation in the cloud seems to be static from the handset as the device must be aware of the infrastructure with anticipation. Thus, the approach is neither flexible nor fault tolerant. The scalability claimed by ThinkAir is not multi-tenancy, the system creates multiple virtual machines based on Android-x86 within the same server for code parallelization.

Odessa [37] is a framework that focuses on improving the perception of augmented reality applications, in terms of accuracy and responsiveness. The framework relies on automatic parallel partitioning at data-flow level to improve the performance of the applications, so that multiple activities can be executed simultaneously. However, the framework does not consider dynamic allocation nor cloud provisioning on demand, which is a key point in a cloud environment.

History-based approaches are also proposed to determine what code to offload [39]. However, the weak point of these approaches is the large amount of time required to collect data, which can produce accurate results. Moreover, these strategies are sensitive to changes, which means that when the device suffers drastic changes, e.g. more applications are installed, the history mechanisms need to gather new data to calibrate again. The size of the data collected in the mobile can also be counterproductive for the device as it steals storage space.

COSMOS [38] is a framework that provides code offloading as a service at method level using Android-x86. The framework introduces an extra layer in a traditional offloading architecture to solve the mismatch between how individual mobile devices demand computing resources and how cloud providers offer them. However, it is not clear how the offloading process is encapsulated as SOA. Moreover, the framework is compared with CloneCloud, which is an unfair comparison as CloneCloud mechanisms offload code at thread level. Other frameworks for computational offloading also are proposed [7], but they do not differ significantly from basic implementation or concept.

We claim that the instrumentation of apps alone is insufficient to adopt computational offloading in the design of mobile architectures that relies on cloud. Computational offloading on the wild is shown mostly to introduce more computational effort to the mobile rather than reduce processing load [13]. In this context, CDroid [13] is a framework that attempts to improve offloading in real scenarios. However, the framework focuses more on data offloading than computational offloading. As a result, we propose in this thesis a framework that attempts to overcome the issues of computational offloading in practice. Our framework automates the process of infering the right matching between mobile and cloud considering multiple levels of granularity [21]. The framework is described in Chapter 4.

2.2.3 Challenges and Technical Problems

Computational offloading for smartphones has not changed drastically from its core principles [25]. However, the effectiveness of its implementation in practice shows to be mostly unfavorable for the device outside controlled environments. In fact, the utilization of code offloading in real scenarios shows to be mostly negative [14], which means that the device spends more energy on the offloading process compared to the actual energy that is saved. Consequently, the technique is far away from being adopted in the design of future mobile architectures. Our goal is to highlight the challenges and obstacles in deploying code offloading. The issues are described as follows:

- ***Inaccurate code profiling*** —Code profiling is one of the most challenging problems in an offloading system, as the code has a non-deterministic behavior during runtime, which means that it is difficult to estimate the running cost of a piece of code considered for offloading. A portion of code becomes intensive based on factors [14], such as user input that triggers the code, type of the device, execution environment, available memory and CPU, etc. Moreover, once code is selected as OC, it is also influenced by many other parameters of the system that come from multiple levels of granularity, e.g. communication latency, data size transferred, etc. As a result, code offloading suffers from a sensitive tradeoff that is difficult to evaluate, and thus, code offloading can be productive or counterproductive for the device [40]. Most of the proposals in the field are unable to capture runtime properties of code, which makes them ineffective in real scenarios.

- ***Integration complexity*** —The adaptation of code offloading mechanisms within the mobile development lifecycle depends on how easily the mechanisms are integrated and how effective is the approach in releasing the device from intensive processing. However, implementation complexity does not necessarily correlate with effective runtime usage. In fact, some of the drawbacks that make code offloading to fail are introduced at development stages, for example, in the case of code partitioning that relies on the expertise of the software developer, portions of code are annotated statically, which may cause unnecessary code offloading that drains energy [41]. Moreover, annotations can cause poor flexibility to execute the app in different mobile devices. Similarly, automated strategies are shown to be ineffective and require major low-level modifications in the core system of the mobile platform, which may lead to privacy and security issues.

- ***Dynamic configuration of the system*** —Next generation mobile devices and the vast computational choices in the cloud ecosystem makes the offloading process a complex task as depicted in Figure 2.8. Although the savings in energy that can be achieved by releasing the device from intensive processing, a computational offloading request requires to meet

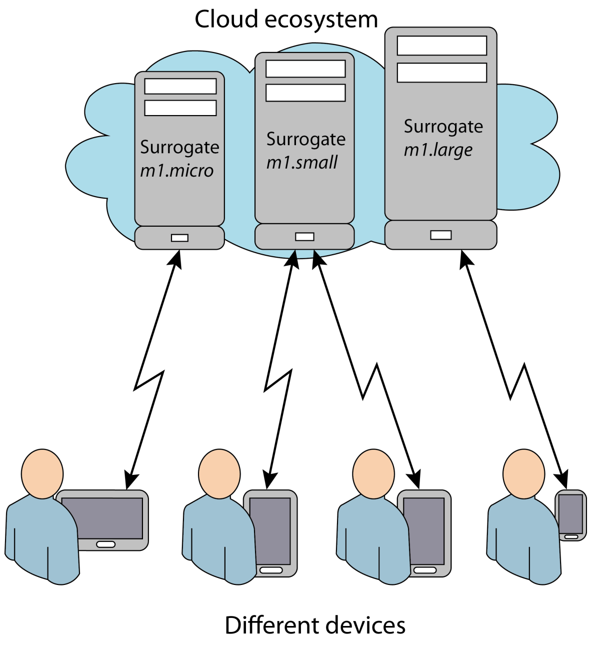


Figure 2.8: Characterization of the offloading process that considers the smartphones diversity and the vast cloud ecosystem

the requirements of user's satisfaction and experience, which is measured in terms of responsiveness of the app. Consequently, in the offloading decision, a smartphone has to consider not just potential savings in energy, but also it has to ensure that the acceleration in the response time of the request will not decrease. This is an evident issue as the computational capabilities of the latest smartphones are comparable with some servers running in the cloud, for instance, consider two devices, Samsung Galaxy S (i9000) and Samsung Galaxy S3 (i9300), and two Amazon instances, m1.xlarge and c3.2xlarge. In terms of mobile application performance, offloading intensive code from i9000 to m1.xlarge increases the responsiveness of a mobile application at comparable rates to an i9300. However, offloading from i9300 to m1.xlarge does not provide same benefit. Thus, to increase responsiveness is necessary to offload from i9300 to c3.2xlarge (refer to Chapter 4 to gain more understanding about it). It is important to note, however, that constantly increasing the capabilities of the back-end do not always speed up the execution of code exponentially, as in some cases, the execution of code depends on how the code is written, for instance, code is parallelizable for execution into multiple CPU cores (parallel offloading) or distribution into large scale GPUs (GPU offloading).

- ***Offloading scalability and offloading as a service*** —Typically, in a code offloading system, the code of a smartphone app must be located in both, the mobile and server as in a remote invocation, a mobile sends to the server not the intermediate code, but the data to reconstruct that intermediate representation so that it can be executed. As a result, an offloading system requires the surrogate to have similar execution environment as the mobile. To counter this problem, most of the offloading systems proposed to rely on the virtualization of the entire mobile platform in a server, e.g. Android-x86, .Net framework, etc., which tends to constrain the CPU resources and slows down performance. The reason is that a mobile platform is not developed for large-scale service provisioning. As a result, offloading architectures are designed to support one user at the time, in other words, one server for each mobile [19, 38]. This restrains the features of the cloud for multi-tenancy and utility computing. Moreover while a cloud vendor provides the mechanisms to scale Service-Oriented Architectures (SOA) [10] on demand, e.g. Amazon autoscale, it does not provide the means to adapt such strategies to a computational offloading system as the requirements to support code offloading are different. The requirements of a code offloading system are based on the perception that the user has towards the response time of the app. The main insight is that a request should increase or maintain certain quality of responsiveness when the system handles heavy loads of computational requests. Thus, a code offloading request cannot be treated indifferently. The remote invocation of a method has to be monitored under different system's throughput to determine the limits of the system to not exceed the maximum number of invocations that can be handled simultaneously without losing quality of service. Furthermore, from a cloud point of view, allocation of resources cannot occur indiscriminately based on processing capabilities of the server as the use of computational resources are associated with a cost. Consequently, the need of policies for code offloading systems are necessary considering both, the mobile and the cloud.

2.3 Achieving App QoE through Computational Offloading

In the previous subsection, we explained how computational offloading can be used by the mobile to increase battery life and accelerate the execution time of resource-intensive portions of code. However, beyond these basic benefits, computational offloading can also be utilized to enhance the perception that the user has towards the continuous usage of a mobile application. This perception is depicted as the fidelity of the mobile application during runtime [5]. Fidelity models the response time of an application as latency, which represents the time that takes to process a computational task and show its result to the mobile user. Fidelity depends on the computational resources that are assigned to run the application. This includes the amount of allocated resources, their processing capacity and runtime availability, among others. Since the cloud provides a vast ecosystem of servers with different computational settings, e.g. number of processors, type of processor, available memory, etc., a task that is offloaded to cloud can be accelerated based on the processing capabilities of the server. In other words, a server with low processing capabilities will execute a task slower than a server with high processing capabilities. Thus, based on the multiple types of servers, a mobile application can achieve multiple acceleration levels, which can be provided as a service based on the suitable and individual perception of each mobile user. Certainly, it is expected for a server in the cloud to have more computational power than a mobile device. This is the truth in most of the cases. However, some last generation devices are as powerful as some servers in the cloud. As a result, a mobile device has to be aware of the type of cloud server to be used in order to augment the processing resources of the mobile (discussed further in Chapter 5).

Figure 2.9 exemplifies this reasoning. We assumed that the processing resources of the device are lower than the servers in the cloud. The figure shows the response time that is expected when executing a mobile task, using the local mobile resources, a weak surrogate, e.g. m1.medium, and a powerful surrogate, e.g. m1.xlarge. By adapting the type of surrogate that executes the task, it is possible to tune the response time of the application in order to fit the expectations of the user regarding the performance of the app. Thus, improving the QoE of the user.

2.3.1 App QoE

Smartphone apps are developed following a stand alone or a client-server model. In terms of user functionality, the main difference between them is that a client-server app requires connectivity to a remote source to activate some of its features, e.g. web service. It is arguable that even though network connectivity will become ubiquitous [42], the latency of the communication will be always a issue for the device. Usually, a device requires considerable computational effort and energy to access remote services that are in low latency networks. Moreover, the perception of the user is degraded, which influences the reputation of the mobile application. As a result, stand alone apps are

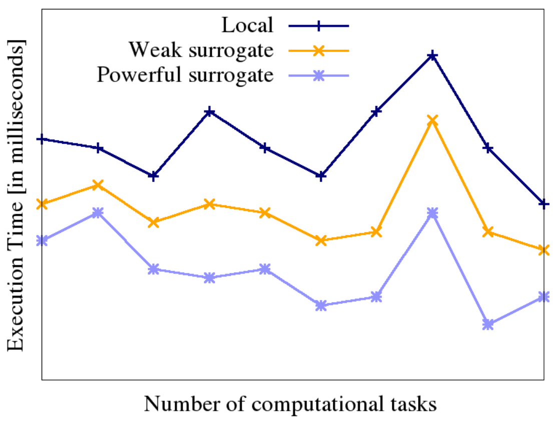


Figure 2.9: Fidelity of a mobile app that can be achieved by using different computational resources.

preferable to client-server apps (at low latency) because they are not tied to network connectivity, which means that the app functionality is always available, the perception of the user is not perturbed and the device requires less computational effort to execute the application. Hybrid apps that merge both models through disconnected operations also can be developed [3].

A user is able to perceive the response time of an app, e.g. instant reaction ($\approx$0.1 second), interaction is not disrupted, but delay may be noticed (1.0 second), interaction is disrupted ($>$10 seconds) [29]. Understanding and adapting an app based on user's QoE has been explored extensively [43, 44, 45]. App QoE through code acceleration is important to optimize the tradeoff between assigned resources and quality of the response time [5]. Moreover, app QoE defines the impact and survival of the app in the mobile market [46].

QoE can be measured following a subjective [43] or objective [47, 48] approach. With the subjective approach, users evaluate and score the app based on their experience. While the subjective method can produce accurate results in adapting app QoE, it depends on explicit user's evaluation, which is inconvenient in practice for user's interaction. On the other hand, the objective approach derives the subjective app QoE from technical and non-technical parameters of the app. The goal of the objective approach is to tune these parameters to improve app QoE.

Multiple approaches can be utilized to improve the QoE of mobile applications. We characterize these strategies into two types as follows:

2.3.1.1 Network Tuning

A telecommunication infrastructure is composed of multiple network devices, e.g. tower, beacon, access point, mobile, etc. The interaction of each network device influences the overall performance of the communication. Figure 2.10

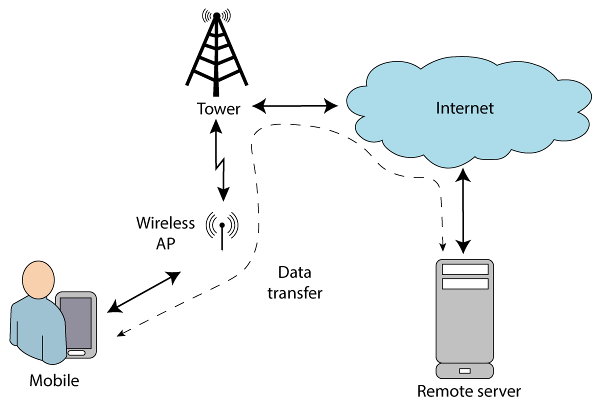


Figure 2.10: Telecommunication system overview

shows an overview of a telecommunication system. By measuring and monitoring the network, it is possible to extrapolate network metrics, e.g. bitrate, delay, etc. These metrics allow the telecommunication provider determine its level of coverage and quality of service provisioning. Network metrics are important for the provider as the metrics extract relevant information that can be utilized to optimally increase the size of the network. For instance, as the number of users increase in one location, the telecommunication provider can decide to place an extra tower nearby that location in order to provide its service for new potential users and to meet its service level agreement with customers, even in peak times.

Network tuning approaches focuses on improving the response time of client-server apps. By adapting network metrics dynamically, these approaches aim to improve the communication between the mobile application and the remote server. Network tuning can be achieved by software or hardware adaptation. Software adaption is oriented towards optimizing communication protocols [49], data transfer and fixing inefficiencies in the mobile apps [50], for instance, by analyzing the performance of different protocols, it can choose the most suitable based on the application requirements [51, 52], e.g. TCP (Transmission Control Protocol), HTTP (Hypertext Transfer Protocol). Hardware adaptation is oriented to modify the behavior of the network devices, for instance, signal strength of the network can be augmented by increasing the coverage of a tower [48].

Several past works study how cellular network operators can estimate network metrics to support better service provisioning for apps [47, 53]. QoE metrics for mobile applications can be extracted from various sources and analyzed with different approaches proposed in the literature, for instance, by measuring the impact of the bitrate, jitter and delay on VoIP calls, and analyzing that information with machine learning, it is possible to estimate the user's satisfaction regarding the quality of the audio [54]. By collecting history information from session length and abandonment rate of an application, it is possible to troubleshoot network devices, e.g. tower, in order to improve the web QoE in mobile browsers [48]. By analizing the streaming of a video in terms of response time, it is possible to built adaptive models that adjust the frames per second of a video based on the perception of the user. Moreover, by such perception, these models can also be used as a user engagement technique [55].

Since this work aims to adapt the QoE of a client-server apps (in terms of response time) when interacting with a user, we study how to estimate QoE metrics of these applications based on user's perception and satisfaction. Unlike existent works, our proposal is not oriented to network tuning. However, we relied on determining QoE metrics in order to adjust the response time of the application. Our solution transforms a stand alone app into a client-server app using computational offloading mechanisms. Certainly, stand alone apps that implement computational offloading can also benefit from network tuning. However, our proposal does not focus on improving communication, but accelerating the code of the mobile app when it is offloaded to remote powerful machines. In our system, code is accelerated to higher levels in order to counter the cost of communication latency. While this has been proposed by other works (Refer to section 2.2.1), our contribution aims to utilize computational offloading as a user engagement technique. The key insight of the work is to improve the response time of the application each time abandonment of the application is detected. We envisioned this approach for increasing the life time of a mobile application.

2.3.1.2 Resource Allocation

Generally, a mobile application is executed in the device using the resources that are assigned by the OS. One application with large execution space will execute better (smoothly) than one with short execution space, for instance, a mobile application running in normal mode differs from one running in energy saving mode. Determining the right amount of resources to be assigned is critical for the device in order to avoid over and under provisioning of resources. While this can easily be optimized by analyzing the history execution of the app, it arises a problem of perception for the user. Different users have different perceptions about the minimal response time in which a mobile application has to work. Thus, the resources assigned to execute a mobile application have to be adapted based on the perception of the user regarding the performance of the application. Naturally, it is expected that the resources assigned reduce the energy consumption of the device while keeping the smoothness of the app required by the user. Otherwise, the benefits of such optimization can turn into drawbacks, for instance, by assigning large space of resources to execute the app, the perception of the user is improved, but the satisfaction regarding the battery life declines.

Past works [5] modeled the response time of an app in terms of latency, which depicts the time that takes to process a computational task and display its result to the user. This is also known as fidelity of the application. QoE of a stand alone app is achieved by tuning its fidelity. It has been demonstrated that an app can be adapted to multiple levels of fidelity depending on the allocated mobile resources for app's execution [26], which is translated into multiple scales of responsiveness.

However, with the rapid adoption of cloud computing within the mobile architectures, more recent works [15] have studied how to allocate external resources in order to outsource a task to a more powerful infrastructure in order to extend battery life (Refer to section 2.2.1 about mobile computational offloading). These works proposed one server per each smartphone architecture [15], which is infeasible if we consider the amount of smartphones nowadays and the provisioning cost of constantly running a server for one particular user. High capabilities servers as surrogates are unrealistic for a single user in long term due to the high provisioning price. Moreover, these works do not consider that code can be accelerated based on utility computing. Thus, unlike previous proposals, our study combines fidelity adaptation, external resource allocation, and utility computing. Our goal is to achieve fidelity tuning using a pool of augmented cloud resources with multiple computational capabilities. To the best of our knowledge, this thesis presents the first system to adapt QoE for stand alone apps using code offloading.

2.3.2 Challenges and Technical Problems

In order to provide a cloud service that can be used by a smartphone app to enhance its fidelity, many issues have to be addressed. We envisioned that as part of an app released in a store, a counterpart computational service in a cloud also is released by the same party that developed the app to improve the QoE of the users of the app. Naturally, to achieve this, many challenges have to be overcome. First, implementing an objective strategy in the mobile that identifies the QoE of a particular user. Second, to grant the offloading architectures the ability to handle multi-tenancy, which has not been shown in other works. Finally, since the computational provisioning of cloud has a cost, capacity planning is required to optimize the amount of servers to handle a specific load of active users. Throughout this thesis, we study how to achieve app QoE using computational offloading and propose solutions for each of the described challenges.

A lot of frameworks have been proposed for computational offloading [25]. Besides a few works that focus on scaling up (vertical scaling) a server to parallelize the code of computational requests [37], we have not found architectures that can scale in a horizontal fashion. This clearly can be seen as current frameworks do not take into consideration the utility computing features of the cloud, which is translated into server selection based on provisioning cost. As a result, we extended our computational offloading framework presented in Chapter 4 with the ability to provision continuous computational offloading in multi-tenancy environments. Moreover, our system supports QoS policies to dynamically optimize the number of surrogates needed to handle a specific load of computational offloading requests without affecting the QoE of the active apps offloading to cloud. In this manner, QoS is introduced to the computational service provisioning of the cloud while considering QoE aspects of the mobile user. Finally, unlike a traditional architecture for code offloading that consists of a client and server, our system also includes a load balancer.

2.4 Summary

Mobile and cloud computing convergence is shifting the way in which telecommunication architectures are designed and implemented [3]. Mobile devices are looking towards cloud-aware techniques, driven by their growing interest to provide ubiquitous PC-like functionality to mobile users. These functionalities mainly target at increasing storage and computational capabilities. On one hand, storage limitations of the devices have been overcome by many cloud services provided in the Internet, which are built under different protocols. For example, Amazon S3 and Dropbox[9] provide REST-based access, while Picasa[10] and Funambol[11] provide SyncML-based access. Service-oriented integration enriches the mobile apps with a variety of services and facilitates the development of apps that requires access to cloud services via MBaaS. Moreover, mobile systems supported by SOA provide better high availability, fault tolerance, and scalability.

On the other hand, binding computational cloud services such as Amazon EC2 to low-power devices such as smartphones have been proven feasible with latest mobile technologies [17, 18], mostly due to virtualization technologies

[9]https://www.dropbox.com/
[10]http://picasa.google.com/
[11]http://www.funambol.com/

and their synchronization primitives, enabling transparent migration and execution of intermediate code. Multiple research works have proposed different code offloading strategies to empower the smartphone apps with cloud based resources [17, 18, 19, 20]. However, the effectiveness of its implementation in practice shows to be mostly unfavorable for the device outside controlled environments. In fact, the utilization of code offloading in real scenarios shows to be mostly negative [14], which means that the device spends more energy in the offloading process compared to the actual energy that is saved. Consequently, the technique is far away from being adopted in the design of future mobile architectures. In further chapters, we present our contributions, which overcome most of the explained issues. We also envisioned the utilization of computational offloading to achieve customized QoE for a particular user.

Part II

Contributions

Chapter 3

Mobile Cloud Middleware

Hybrid cloud and cloud interoperability are essential for mobile scenarios in order to foster the de-coupling of the handset to a specific cloud vendor, to enrich the mobile applications with the variety of cloud services provided on the Web, and to create new business opportunities and alliances [56, 57]. However, developing a mobile cloud application involves adapting different Web APIs from different cloud vendors within a native mobile platform. Vendors generally offer the Web API as an interface that allows programming the dynamic computational infrastructure that supports massively parallel computing [10]. Deploying a Web API on a handset is demanding for the mobile operating system due to many reasons like compiler limitations, additional dependencies, runtime environment incompatibility, etc., and thus, in most of the cases, the deployment just fails. Moreover, adapting a Web API requires specialized knowledge of low level programming techniques and most of the solutions are implemented ad hoc.

In terms of data storage facilitation of the cloud, existing mobile cloud approaches, such as data synchronization (via SyncML), allow the deployment of a Web API on the device for retrieving data from the cloud. For instance, Funambol [58] or gdata-calendar Web API can be integrated to an Android application to synchronize calendar information (e.g. alarms, tasks etc.). However, this approach focuses on replicating the data located in the cloud with the data located in the handset, which is not a real improvement to enrich app functionality. Alternatively, cloud services can be encapsulated as Web services that can be invoked directly using a simple REST mobile client [59]. However, due to the time consuming nature of a cloud request, this can cause an overhead in the mobile resources without a proper asynchronous communication mechanism. Moreover, by using a REST mobile client, the device is forced to perform multiple transactions and to handle the results of those transactions locally, which is costly from the energy point of view of the handset. The number of transactions is directly associated with the number of cloud services utilized in the mobile cloud application.

To counter the problems such as the interoperability across multiple clouds, invocation of resource intensive processing from the handset and to introduce the platform independence feature for the mobile cloud applications, we developed Mobile Cloud Middleware (MCM) [6]. The middleware abstracts the Web APIs of multiple cloud at different levels and provides a unique interface that responds (JSON-based) according to the cloud services requested (REST-based). MCM provides multiple internal components and adapters, which manage the connection and communication between different clouds. Since most of the cloud services require significant time to process the request, it is logical to have asynchronous invocation of the cloud service. Asynchronicity is added to the MCM by using push notification services provided by different mobile application platforms and by extending the capabilities of a XMPP-based IM infrastructure [33].

Furthermore, MCM supports tasks composition into a multi-cloud operation. A multi-cloud operation consists of delegating mobile tasks to a diversity of cloud services (e.g. from the infrastructure level, platform level, etc.) located on different clouds (e.g. public, private, etc.) and orchestrating (e.g. parallel, sequential, etc.) those transactions for achieving a common purpose. Developing this kind of mobile cloud apps requires, from the cloud perspective, the interoperability among cloud architectures. From the mobile perspective, a considerable effort to manage the complexity of working with distributed cloud services, a specialized knowledge to adapt each particular Web API to a specific mobile platform, and an efficient approach that avoids the unnecessary data transfer.

3.1 Design Goals and Architecture

MCM[12,13] is introduced as an intermediary between the mobile phones and the clouds for managing asynchronous delegation of mobile tasks to cloud resources. MCM hides the complexity of dealing with multiple cloud providers by abstracting the Web APIs from different clouds in a common operation level so that the service functionality of the middleware can be added based on combining different cloud services. Moreover, MCM enables the development of

[12]https://github.com/huberflores/MobileCloudMiddleware
[13]https://github.com/huberflores/InteroperableWebAPI

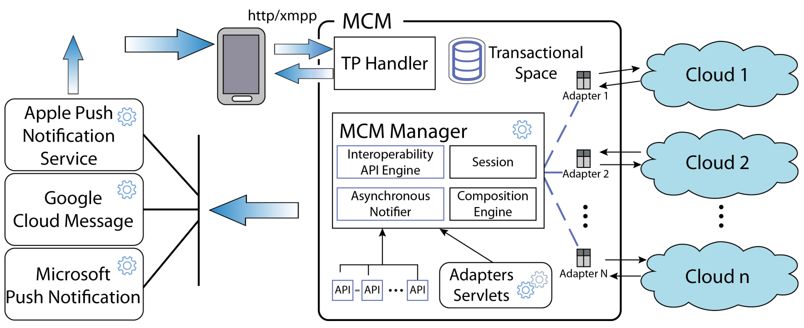


Figure 3.1: Architecture of the *Mobile Cloud Middleware*

customized services based on service composition in order to decrease the number of mobile-to-cloud transactions needed in a mobile cloud application. The architecture is shown in figure 3.1. When a mobile application tries to delegate a mobile task to a cloud, it sends a request to the TP-Handler component of the middleware, which can be based on several protocols like HTTP or XMPP. The request is immediately followed by an acknowledgement from MCM (freeing the mobile) and it consists of a URL with the name of the server, the service being requested, and the configuration parameters, which are applied on the cloud resources for executing the task. For instance, http://ec2-x-x-x-x.compute-1.amazonaws.com:8080/MCM/SensorAnalysis represents a processing task that triggers a MapReduce activity recognition algorithm over a set of sensor data (accelerometer and gyroscope) collected by the mobile in order to discover reading patterns. The request also includes information regarding the cloud vendor, type of instance, region, image identifier, and the rest of the parameters associated with that particular service. Notice that different services involve the utilization of different parameters within a request. Once at the MCM, the request is then processed by the MCM-Manager for creating the adapters that will be used in the transactional process with the clouds. Figure 3.2 shows the interaction logic of the components of MCM.

When the request is forwarded to the MCM-Manager, it first creates a session (in a transactional space) assigning a unique identifier for saving the system configuration of the handset (OS, clouds' credentials, etc.) and the service configuration requested. The identifier is used for handling different requests from multiple mobile devices and for sending the notification back when the process running in the cloud is finished. The transactional space is also used for exchanging data between the clouds (to avoid offloading the same information from the mobile again and again) and manipulating data acquired per each cloud transaction in a multi-cloud operation. Based on the request, the service transaction is managed by the Interoperability-API-Engine or the Composition-Engine (single or composite service invocation, respectively).

In case of a single service invocation, the request is handled by the Interoperability-API-Engine, which selects, at runtime, based on the request, the Web API to utilize in a cloud transaction. The engine extends the interoperability features to the Adapter-Servlets component, which contains the set of routines/functions that are used to invoke a specific cloud service. The MCM-Manager encapsulates the API and the routine in an adapter for performing the transaction and accessing the XaaS. Basically, an adapter is a runnable abstract class that provides a generic behavior for a mobile task. We utilized Gson[14] for loading and serializing mobile tasks.

In contrast, if the request consists of a composite service invocation, the Composition-Engine (explained in detail in section 3.2) interprets the service schema and acquires the adapters needed for executing the services from the Interoperability-API-Engine. The hybrid cloud property of an adapter is achieved by using the Clojure [15] component that encapsulates several Web APIs. Each adapter keeps the connection alive between MCM and the cloud and monitors the status of each task running in the cloud. An adapter can store data in the transactional space in order to be used by another adapter.

Once the single/composite service transaction is completed, the result is sent back to the handset in a JSON format. MCM-Manager uses the asynchronous notification feature to push the response back to the handset. Asynchronicity is added to the MCM by implementing the Google Cloud Messaging for Android (GCM), the Apple Push Notification Services (APNS), and the Microsoft Push Notification Service (MPNS) protocols for Android, iOS and Windows Phone 7 respectively. Alternatively, MCM also has support for sending messages using the Mobile Cloud Messaging Framework based on XMPP [33], which extends an ejabberd [16] infrastructure for delivering messages to any smartphone that implements an XMPP mobile client. Asynchronous notification support of MCM is explained in detail in subsection 3.1.1.

MCM is implemented in Java as a portable module based on Servlets 3.0 technology, which can easily be deployed on a Tomcat Server[17] or any other application server such as Jetty[18] or GlassFish[19]. Web APIs are encapsulated using Clojure, and thus are accessed by a common API. This encapsulation guarantees updating deprecated Web APIs with newer versions, which are released constantly by the cloud vendor. Moreover, Clojure is also considered as

[14]https://code.google.com/p/google-gson/
[15]http://clojure.org/
[16]https://www.ejabberd.im/
[17]http://tomcat.apache.org/
[18]http://eclipse.org/jetty/
[19]https://glassfish.java.net/

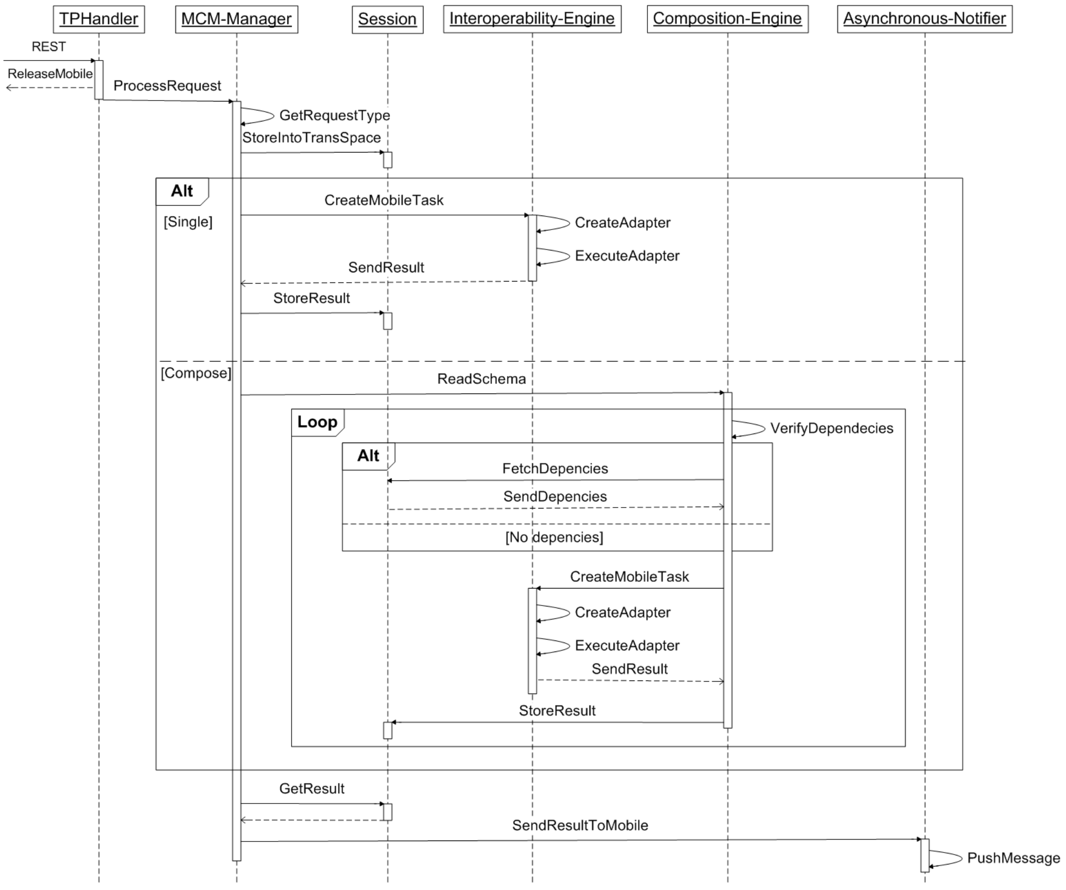


Figure 3.2: Interaction logic of the components of *Mobile Cloud Middleware*

its distributed nature introduces flexibility for scaling the applications horizontally and thus augmenting the faul-tolerant properties of the overall system. Moreover, Clojure also provides portability in the design. Thus, decreasing the effort of migrating the whole architecture to more suitable telecommunication programming languages.

Hybrid cloud services from Amazon EC2, S3, Google and Eucalyptus based on private cloud are considered. Jets3t API enables the access to the storage service of Amazon and Google from MCM. Jets3t is an open source API that handles the maintenance for buckets and objects (creation, deletion, modification). A modified version of the API was implemented for handling the storage service of Eucalyptus, Walrus. Latest version of jets3t also handles synchronization of objects and folders from the cloud. Typica API and the Amazon API are used to manage (turn on/off, attach volumes) the instances from Eucalyptus and EC2 respectively. MCM also has support for SaaS from Facebook, Google, AlchemyAPI.com[20] and face.com[21].

[20]http://www.alchemyapi.com/

[21]http://techcrunch.com/2012/06/18/facebook-scoops-up-face-com-for-100m-to-bolster-its-facial-recognition-tech/

MCM and the resource intensive tasks can easily be envisioned in several scenarios [60, 61]. For instance, we have developed several mobile applications that benefit from going cloud-aware. Zompopo [62] consists of the provisioning of context-aware services for processing data collected by the accelerometer with the purpose of creating an intelligent calendar. CroudSTag [63] consists of the formation of a social group by recognizing people in a set of pictures stored in the cloud. Finally, Bakabs [64] is an application that helps in managing the cloud resources themselves from the mobile, by applying linear programming techniques for determining optimized cluster configurations for the provisioning of Web-based applications.

3.1.1 Asynchronous Support for Resource-intensive Tasks

Some mobile applications, whose functionality depends on the cloud, can provide a tolerant response time to the user. For example, searching for a location in Google Maps or requesting for a preview of a picture in flickr. However, when a mobile application needs to perform a task which is expected to be time consuming, it cannot hang the mobile phone until the response arrives, e.g. image processing with MapReduce. Mobile devices tend to get stuck if the computation offloading requires long waiting, and in some cases the OS just kills the task when it senses that it is low on memory (Android case), or just does not allow performing another task at the same time. Delegating a mobile operation to cloud is a complex process when the request is time consuming and without a proper mechanism to handle the communication, this can cause an overhead in the mobile resources.

As a solution to address these issues, MCM implements an asynchronous notification feature for mobile devices that foster the de-coupling between the client and server. Mobile applications can rely on push technologies (aka notification services) for dealing with remote executions, and thus avoiding the effect of polling caused by protocols such as HTTP. Push mechanisms are well integrated within the mobile platform for low-energy consumption, and most of the times, messages can reach the mobile in a short period of time after they are pushed from the server. Messaging mechanisms are utilized by cloud vendors to trigger events from the mobiles to synchronize data with their cloud services, e.g. Gmail.

In the asynchronous process, when a mobile application sends a request to the middleware, the handset immediately gets a response that the transaction has been delegated to remote execution in the cloud, while the status of the mobile application is sent to local background. Now the mobile device can continue with other activities. Once the process is finished at the cloud, a notification about the result of the task is sent back to the mobile, so as to reactivate the application running in the background, and thus the user can continue the activity. The approach can also be used within a mobile device to concurrently execute several tasks in multiple clouds.

MCM implements asynchronous support for Android, iOS and Windows Phone platforms. However, these services are unreliable to be used in real-time applications (the details are addressed in section 3.1.3) as the services are public services and the mobile is competing with huge number of other customers. To address this issue, we have designed a mobile cloud messaging framework based on XMPP. The framework relies on Instant Messaging infrastructure to deliver notification to the smartphones. The main advantage of the mechanism is that it can scale on public cloud based on usage demand in order to keep high levels of quality of service. Moreover, the service is not tied to any mobile platform, and thus it is interoperable between different devices. The rest of the subsection describes each notification technology as follows.

1. ***Google Cloud Messaging*** —GCM is the enhanced notification service provided by Google for sending asynchronous messages to Android devices. It has been released as the replacement for the deprecated AC2DM (Android Cloud to Device Messaging Framework) service and includes new features such as unlimited message quota, decoupling of the mobile device from a Google account for receiving message data (devices running Android 4.0.4 or higher) and message throttling that enables to prevent sending a flood of messages to a single handset, among others.

   Basically, a mobile application that implements the GCM mechanism for receiving messages, first, has to register itself against the GCM server for acquiring a *registration ID*. Messages are sent to the mobile using this identifier from the application server, which lasts till the mobile explicitly unregisters itself, or until Google refreshes the GCM servers. An *API key* is required for the application server in order to pass message data to the GCM service. This key is generated by the developer using Google APIs console and it is used as the *sender ID* of the message.

   An application server sends a message to the mobile by sending the registration ID, the API key and the payload to the GCM servers, where the message is enqueued for delivery (with maximum of 5 attempts) or stored in case the mobile is offline. GCM allows up to 100 messages stored before discarding it. Once the device is active for receiving the message, the system executes an Intent Broadcast for passing the raw data to the specified Android application. GCM does not guarantee the delivery of a message and the sending order. Messages with payload can contain up to 4K of data and are encapsulated in a JSON format.

2. ***Apple Push Notification Service*** —Similar to above mechanism, Apple devices running iOS 3.0 or newer can also receive asynchronous messages provided through APNS, the messages of which are sent through binary interface (gateway.push.apple.com:2195, gateway.sandbox.push.apple.com:2195) that uses streaming TCP socket design. Forwarding messages to device happens through constantly open IP connection. TLS (or SSL) certificate (provisioned through iOS developer portal) is needed for creating secure communication channel and for establishing trusted provider identity. To avoid being considered a DoS (Denial of Service) attacker, using one connection for multiple notifications rather than creating new connection for each notification, is desired.

   APNS has a feedback service that records failed notification delivery attempts. This information should be checked regularly to avoid sending messages to devices that do not have the targeted application installed any more. For the same reason application should register itself at the messaging server for notifications at each start by providing at least its device token that it has received from APNS. Device token is a 32 byte hexadecimal number that is unique for an application on a device and does not change.

   Messages sent to iOS devices via APNS consist of JSON payload with maximum length of 256 bytes. Within message payload, values for keys alert, sound and badge can be used to customize the message alert being shown to user upon receiving it. Because the payload size is limited, it is used to provide enough information for the application to make request for additional details. When the device cannot receive notifications for some time (e.g. due to being offline or switched off) and multiple notifications have been sent, only the last one is guaranteed to be delivered.

3. ***Microsoft Push Notification Service*** —MPNS is the push technology provided by Microsoft for sending messages to the mobiles running with the Windows Phone 7 platform. MPNS maps each device into a set of URI channels that can be invoked via REST/POST with the possibility of creating up to 30 different channels for pushing data to the applications (one application corresponds to one channel). Prior to use of the MPNS mechanism, a phone has to request a push notification URI from the Push Client service (located in the phone), which handles the negotiation process with the MPNS server. A URI can be requested

without authentication, however, in this case, a limit of 500 messages per day is fixed. Once the mobile has obtained the URI, this is passed to the application server so that the URI can be integrated within any remote service that needs to notify data to a mobile application.

A message is sent from the application server to the mobile by performing a request to the URI, which works as an interface between the application server and the MPNS service. A request can be based on different HTTP headers (depending on the mobile version), MessageID (to identify the response), NotificationClass (to set the sending notification time interval), NotificationType (sets the type of the notification) and CallbackURI (for authenticated channels). When the message is received by the phone, a notification event is triggered. MPNS differentiates three types of notifications, *Toast Notifications*, *Tile Notifications* and *Raw Notifications*. A *Toast Notification* is used for raising alarms, passing parameters to the applications, etc. A *Tille Notification* is an interactive notification that enables to change the design properties of Tile elements such as color, image background, etc. Finally, a *Raw Notification* is utilized for passing information to a running application (if the application is not running, the notification is discared). Similar to APNS or GCM, MPNS does not guarantee the delivery of a message.

3.1.2 Generic Support for Asynchronous Delegation based on XMPP

XMPP is a near real-time communication protocol that relies on Extensible Markup Language (XML) and enables the exchange of structured data between network entities [65] (e.g. users, bots, etc.). XMPP uses decentralized client-server model, where each user connects to the server that controls its own domain. Thus, allowing the creation of an interoperable and federated architecture based on multiple authorities. Unlike other technologies (e.g. SIP) that can be implemented or extended to push data to cloud, XMPP is preferable as it enables to mantain a two-way asynchronous communication and fosters a flexible, scalable and manageable infrastructure which can be adapted for any mobile platform (interoperable).

3.1.2.1 Fundamentals and Extensions

XMPP uses globally unique addresses in order to route and to deliver messages. All entities are addressable by using unique identifiers called domainpart and JID for the server and the client, respectively. A XMPP session is established after creating a TCP connection, XMPP exchanges input/output XML streams for opening the channel that is going to be used during all the communication process. XML stream is a container for exchanging XML stanzas between entities. The stream is started by sending a stream header tag *<stream:stream/>* with appropriate attributes and namespace declarations.

After the stream header is accepted, the connection remains alive and the server sends a first level child element *<features/>* tag, which contains actions that the server is offering to the client for proceeding with the next step. Upon successful actions' execution, the XML stream must be restarted by sending the stream header to the server once again, same sequential process is repeated.

A server should enable an entity to maintain multiple connected resources simultaneously. Once the stream negotiation has finished, either party can send XML stanzas. A XML stanza is the base of the XMPP protocol and it can be defined as a first-level XML element whose name can be *<message/>*, *<presence/>* and *<iq/>* (info query) and whose qualifying namespace may vary from *'jabber:client'* or *'jabber:server'* depending on the entity.

A *message stanza* is used for pushing information to another entity using synchronous or asynchronous communication. *Presence stanza* is used for sending presence notifications. For example, when a user logs into server, the client sends a presence stanza to the server indicating that the user has changed status. The server sends this status notification to every entity that is online and is in the user's buddy list. *Info query stanza* is a request-response mechanism, (similar to the HTTP) that enables an entity to make a request to and receive a response from another entity.

When the presence for the entity or resource has been set, the client is able to exchange unbounded number of XML stanzas with other entities on the network until the stream is closed. The stream closing is rather simple. If the client wishes to log out from the server, it only needs to send the closing tag of the stream (*</stream:stream>*). The closing entity must not immediately close the TCP connection, but wait for the receiving entity to respond with the same. This indicates that the server has halted any data transmission to the entity and is in state to end the connection. Usually, the server closes the connection and the client is not involved in the rest of the process.

In order to extend the capabilities of the IM infrastructure for sending notification messages, while keeping intact the other IM features such as chat. An attribute *'notification'* is added by the messaging framework so that the notifications are routed to the XMPP notification manager of the mobile when this is received by the PacketListener of the XMPP client.

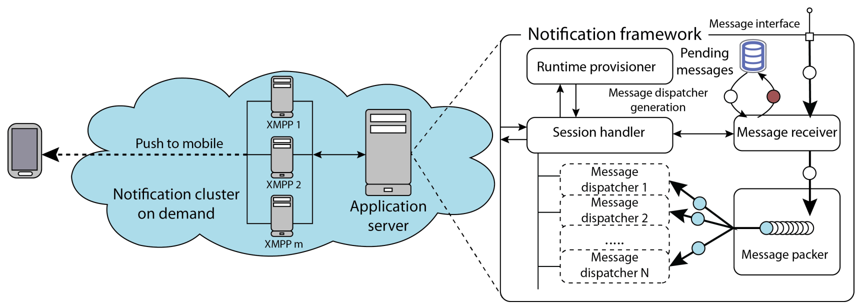


Figure 3.3: Mobile cloud messaging architecture

3.1.2.2 Implementation Details of the Messaging Framework

The messaging framework [22] is introduced as a component that follows a client/server architecture, which can easily be integrated with any application server for sending JSON-based notifications to any mobile platform that implements an XMPP mobile client functionality[23]. Thus, message distribution can be performed to a variety of mobile applications developed in multiple mobile technologies. The architecture is shown in Figure 3.3. The framework establishes an asynchronous communication between the mobile and the cloud, based on the JID of the user and it scales an instant messaging XMPP infrastructure for sending notifications to a large number of mobile devices.

From a server side perspective, the framework implements a *message interface* that receives two parameters, the JID of the mobile client and the text message in CSV format. When a notification message is created for being delivered, the message is passed to the *Message-receiver* for inspection. The *Message-receiver* analyzes whether the mobile client can receive the message by checking its status (e.g. online, away, etc.). Status data of the users is retrieved from the *Session-handler*, which manages the information of the complete roster. If the user's status is other than offline, the message will continue to the next step of the notification process. Otherwise, the message is put in a temporal space (pending messages) and the JID account will be monitored so that the process for that particular message is reactivated when the user comes online again. The aim of the inspection is to guarantee that the message is delivered to an active client and to reduce the queue of messages by filtering those that can not be processed by an inactive mobile. In some scenarios, the mobile client may go offline after the *Message-receiver* checks the status of the client. In that case, the message is sent by the framework and its delivery is delegated to a second mechanism, which is an inherent feature of the XMPP server for handling offline messaging.

[22]https://github.com/huberflores/XMPPNotificationServer
[23]https://github.com/huberflores/XMPPNotificationClient

Once the message has been verified for delivery, the message is passed to the *Message-packer* which encapsulates the text into a JSON payload (using json-simple library[24]. By default, the *Message-packer* tries to divide the text in attributes of the type name-value (similar to common notification services). In the case of a packing exception (unexpected format), the *Message-packer* creates a JSON with a unique attribute containing as value the overall text. Messages are distributed in a round-robin fashion by the *Message-packer*, among the available Message-dispatchers. A *Message-dispatcher* can be associated to an automated JID that contains the complete roster and it is used for adding the *'notification'* attribute to the XML streaming, which is sent as payload to the XMPP cluster. Each *Message-dispatcher* is added, monitored and removed dynamically by the *Session-handler* according to the data generated from the *Runtime-provisioner.*

Runtime-provisioner is in charge of determining the number of Message-dispatchers to be created by analyzing the queue length of the messages. This number is determined by the capacity of a single standalone framework for handling concurrent automated JIDs as shown in subsection 3.1.3. The framework is developed in Clojure, which is a distributed language suitable for cloud deployment and portability purposes. It uses the SmackAPI[25] for implementing the logic and behavior of the Message-dispatchers. Notification data is managed by MySQL and H2 databases, for storing pending messages and queueing messages, respectively. H2 uses an in-memory approach for handling the message information. This approach was preferred as the life span of the messages is short and to avoid unnecessary CPU load created by more sophisticated DBMS.

From a client's perspective, a mobile application is to be developed for receiving messages using the framework. It has to implement an XMPP client that extends the native notification features of the mobile platform. For example, in the case of Android, we have created an XMPP client that uses the SmackAPI and the *NotificationManager Class* of Android. It implements a *PacketListener* and *PacketFilter* that filters the received messages according to its type. The *NotificationManager Class* is used to create the notification event that informs the user about the message received. Once, the user selects the notification icon, an *Intent Broadcast* is created for passing the raw data to the application.

[24]https://code.google.com/p/json-simple/
[25]http://www.igniterealtime.org/projects/smack/

3.1.3 Asynchronous Delegation Performance

Several experiments based on the delivery rate of the message were conducted using the described mechanisms. The aim of the experiments is to determine the latency between a provider submitting a request and the target device receiving the notification (responsiveness). Mobile devices considered in the experiments are a Samsung galaxy S2 i9100 with Android 2.3.3, 32GB of storage, 1 GB of RAM and support for Wi-Fi 802.11 a/b/g/n; an iPhone 4 with 16GB of storage, 512 MB of RAM and support for 802.11b/g/n Wi-Fi (802.11n 2.4GHz only); and a Nokia Lumia 800 with Microsoft Windows Phone 7.5 Mango with 16 GB of storage, 512 MB of RAM and support for Wi-Fi 802.11 b/g/n. All the devices are connected via Wi-Fi to a network with an upload rate of ≈ 1409 kbps and download rate of ≈ 3692 kbps, respectively.

Notification servers are configured to run on Amazon EC2 infrastructure using small instances (a small instance has 1.8 GB of memory and up to 10 GB of storage). One EC2 computational instance is equivalent to a CPU capacity of 2.66 GHz Intel®Xeon™processor. Servers were running on 64 bit Linux platforms (Ubuntu). XMPP mechanism (server and client) is developed as described in subsection 3.1.2.2. GCM application server is written in Java using Servlets technology version 3.0. It is deployed as war project in a Tomcat server. It allows to register multiple devices simultaneously for receiving GCM notifications and to send messages to a specific device. Each GCM message is encapsulated in a JSON payload that includes an index number that identifies the creation of the message and a sending timestamp taken from the server. GCM client application is developed for Android 2.3.3. Once a notification arrives, the payload is extracted from the message and then it is stored in a SQLite database along with receiving client timestamp taken from the mobile.

In the case of APNS server, the gem library jpoz[26] written in Ruby on Rails is considered. Jpoz for APNS is a library that enables to send push notifications from Ruby scripts. Scripts are executed in the server for sending notifications to the mobile (with the same format as described for GCM) and then are processed at the phone, once the notification is triggered. Messages are stored in a SQLite database along with receiving client timestamp taken from the device. MPNS application server is deployed in a Tomcat server by implementing a Servlet that executes a POST request to the URI channel created from the Windows Phone 7 device. The request contains the complete headers + payload, where the payload consists of an index number that identifies the creation of the message and a sending timestamp taken from the server (similar to GCM and APNS). Once a notification arrives, the Windows

[26]https://github.com/jpoz/APNS

Phone client extracts the payload from the message and then it is stored in a database along with receiving client timestamp taken from the handset.

3.1.4 Evaluation and Analysis

For all the described mechanisms, the aim of the experiments is to determine the latency between a provider submitting a request and the target device receiving the notification (responsiveness). Messages are fixed to a size of 254 bytes, which is the lowest common denominator of the allowed message sizes of the considered approaches. Messages are formatted with similar characteristics so that they can ensure a fair comparison that is not affected by transportation factors such as data size, among others.

Messages are sent 1 per second for 15 seconds in sequence, then with a 30 minute sleep time, later followed by another set of 15 messages, repeating the procedure for 8 hours (240 messages in total). The frequency of the messages is set in this way in order to mitigate the possibility of being detected as a potential attacker to the cloud vendor, e.g. Denial of Service, and to refresh the notification service from a single requester and possible undelivered data. Moreover, the duration of the experiments guarantee having an overview of the service under different mobile loads, which may arise during different hours of the day.

Even though push architectures are conceptually based on Open Mobile Alliance (OMA) standard and follow a gateway server approach, notification mechanisms are tailored for a specific mobile platform as *black box* services. Thus, it is not possible to have a clear understanding about the architecture implemented by each mechanism. As a result, we can just hypothesize the causes of their performance, e.g. type of queue policy of the messages, number of active servers deployed, dynamic allocation of servers based on load, etc.

Results are shown in Figure 3.4 for each mechanism. According to the data, GCM is the most unreliable real-time mechanism that provides poor delivery rates for notifications, with an average of ≈ 0.75 sec, median of ≈ 0.66 and standard deviation (SD) of ≈ 0.69. From the GCM delivery rate diagram, it can be observed that the quality of service started to decrease as the number of messages increase across time. Consequently, messages tend to arrive without a specific order. Some of the reasons that can cause that behavior are: the utilization of multiple servers, where each server handles its own individual queue; the unequal distribution of messages among the active servers sending notification; high utilization of the notification system. Android is one of the most popular platforms for developers. Thus, the notification service is expected to handle heavy load of messages.

In contrast, APNS is a more reliable mechanism that allows to send messages under a shorter delivery rate, with an average of ≈ 0.57 sec, median of ≈ 0.58 and SD of ≈ 0.16. In most of the cases, messages tend to arrive in the same order as they were sent. This kind of behavior points out a possible APNS overprovisioning (more servers than required for managing the load) or optimal mechanisms to distribute the load of messages among the active servers. In the case of MPNS, sent messages tend to reach the device without a specific order (similar queues), but with a shorter inter-arrival time. Delivery time for MPNS has an average ≈ 1.07, median of ≈ 1.011 and SD of 0.57. Finally, our proposed XMPP mechanism shows to share a similar reliability as APNS for delivering messages, with an average delivery time of ≈ 0.6 sec, median of ≈ 0.75 and SD of ≈ 0.10.

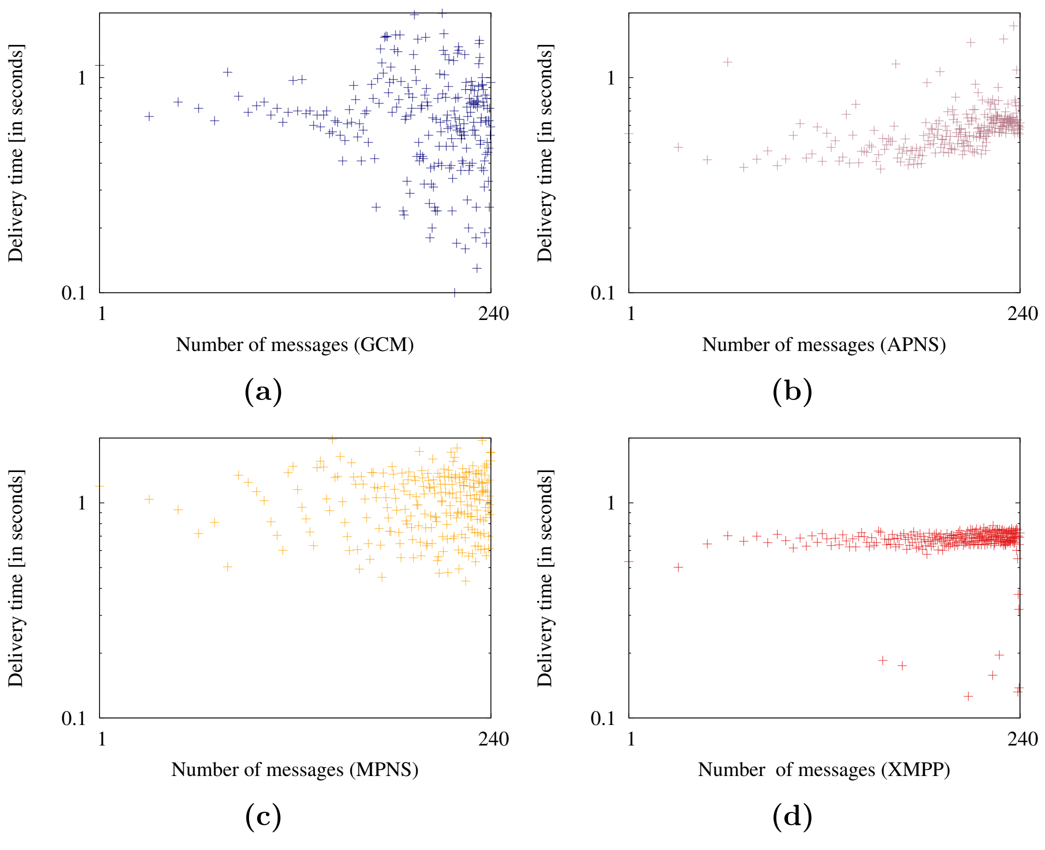


Figure 3.4: XMPP delivery rate againts different mechanisms

Naturally, in our experiments we used a single notification server to measure delivery rate of the messages. One single XMPP server is capable of handling a high amount of users. To verify the capacity that an XMPP server has to work as a gateway broker, which handle multiple notification requests, multiple tests were carried out. We deployed an XMPP cluster in Amazon using EC2 small instances running on 64 bit Linux platforms (Ubuntu). We increased the number of active servers in the cluster in order to determine the amount

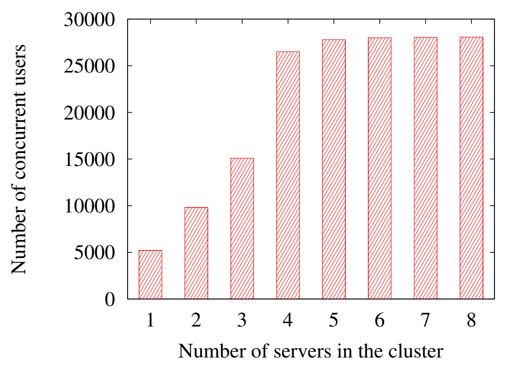


Figure 3.5: Number of active devices that can receive notifications through a XMPP cluster working as a notification service.

of devices that can receive notifications through the cluster. Different mobile loads were simulated using Tsung, which is a benchmarking tool.

The results are shown in Figure 3.5. Basically, a single XMPP node can handle ≈ 5200 users concurrently. Moreover, according to the results, it can be observed that the cluster with one load balancer is able to grow up to 6 nodes before the load balancer started to become a bottleneck that refuses more users to login. However, in order to increase the number of users, it may be possible to replace the load balancer for a more powerful cloud instance (e.g. m1.medium or m1.large).

3.2 Hybrid Cloud Service Composition of MCM

While delegating a mobile task to the cloud may enrich the mobile applications with sophisticated functionality and release the mobile resources of heavy processing, frequent delegation rates (mobile-to-cloud transactions) may introduce costly computational expenses for the handset. Therefore, approaches that enable to avoid unnecessary communication overhead such as those based on service composition must be encouraged. Service composition consists of the integration and re-utilization of existent distributed services for building more complex and thus more rich service structures. Most of these structures are developed graphically as control or data flow based models. For instance, YahooPipes is a composition tool based on the concept of Unix pipes. A pipe depicts a data resource on the Web (e.g. RSS feeds etc.) that can be filtered, sorted or translated. Several pipes can be joined together into a single result for extracting information according to the needs of the user.

Service composition enriches a single service invocation by adding, executing, orchestrating, and joining multiple service requests (treating each service as an individual task) in a common operation. In order to foster a flexible and scalable approach, to compose hybrid cloud services, and to bring the benefits of service composition to the mobiles, MCM implemented a service composition mechanism that enables to automate the execution of a workflow structure with a single mobile offloading. Moreover, tasks are composed using a declarative approach, where the workflow is modeled graphically and deployed in the middleware for execution by using an Eclipse plugin.

3.2.1 Hybrid Cloud Service Implementation

The MCM composition editor is developed as a plugin for the Indigo version of Eclipse. Basically, after configuring the plugin with the remote location of the MCM, it retrieves the list of services that are available at the middleware for mobile delegation (e.g. sensor analysis, text extraction, etc.) so that each service (cloud transaction) can be depicted graphically as a *MCM delegatiOn Component* (MOC). The plugin is developed by combining the capabilities of GEF[27] (Graphical Editing Framework) and EMF[28] (Eclipse Modeling Framework). GEF is used for creating the graphical editor that consists of a palette of tools (MOC, connector, and mouse pointer) and a blank frame in which the MOC is dragged and dropped multiple times for building the work flow. Each MOC provides standard inputs and standard outputs that represent the receiving/sending of messages in JSON format that are used for the intercommunication of components. Thus, the combination of MOCs is tied to the matching between inputs and outputs. Since an individual service is usually triggered for execution by a REST request that responds with a JSON payload, when passing parameters between MOCs, the JSON payload is analyzed and all the necessary parameters are extracted from it for creating a request that matches the input of the next MOC. This request is loaded into the MOC using Gson so that it can be executed.

A MOC is drawn by extending the draw2d[29] library (Node class) with a label object. When the component is active on the frame (focus on), its properties view pops up so that the component can be customized. The properties view consists of: 1) a description category that is used to specify the MCM service which is selected from a list (previously retrieved) contained in a ComboBoxPropertyDescriptor; once the service is selected, the service name is set in the label object with its respective URL value as attribute; 2) an inputs category that describes the list of inputs of the service selected in (1), which

[27]http://www.eclipse.org/gef/
[28]http://www.eclipse.org/modeling/emf/
[29]http://www.eclipse.org/gef/

basically represents the execution properties that depend on the Web API being utilized, along with the required data for executing the task at the cloud. Input parameters may be dynamic or static. A dynamic input value is one that is obtained from a connected MOC. In contrast, a static input value is defined by the user as plain value (e.g. file path); 3) an outputs category that describes the list of outputs of the service selected in (1), which represents the results of the cloud transaction. Both, (2) and (3) also provide information related to other MOC which may be connected to its inputs/outputs.

EMF is used for creating an XML-based representation of the work flow (serialization of the model). This serialization consists of describing each MOC and connection relations as data type and flow conditions, respectively, so that they can be deployed and published at the middleware for execution using the MCM-Composition engine. Notice that the XML description is utilized mainly for validating the execution of the workflow and for checking the dependecies prior to the execution of a MOC. In the XML description, each data type is described by mapping its graphical representation into object properties (e.g. height, weight, etc.) and input/output attributes (e.g. bucket name, security group identifier, etc.). For instance, the MOC *Start Instance* by default establishes a height of 50px and width of 150px, it is invoked by performing a request to the URL with value http://ec2-x-x-x-x.compute-1.amazonaws.com:8080/MCM-/StartInstance and it requires as input parameters, an image id (e.g. ami-xxxxxxxx), a provider name (e.g. amazon), an instance size (e.g. large, small, etc.), a region (e.g. us-east-1c) and a username, in order to generate an Instance object of typica library as output. This Instance object is passed to another MOC (e.g. RunScript) as serialized dynamic parameter along with the path of the file that is defined by the user as static input.

Once the XML description of the composed service is deployed at the middleware, the MCM-Composition engine is in charge of performing three tasks for executing the new composed service. These tasks consist of publishing, converting and executing. The publishing task is performed once the file is deployed. It consists of assigning a unique URL for invoking the service via the TP Handler. The service is named according to the name of the file which is deployed (if name already exists, service cannot be deployed). The converting task is performed once the service has been requested. It consists of parsing the XML file for getting the individual information of each MOC. This information is passed to the Interoperability API engine for creating the adapters (as described in subsection 3.1) and performing the cloud transaction. Notice that in runtime a MOC may have one or more associated adapters. Finally, the executing task is in charge of mapping the entire file (from top to bottom) and executing each service using the adapters created previously. The execution follows the structure of the workflow considering any parallel or sequential tasks. When a service is executed, the task is monitored by the MCM from start to end. Once the result of each task is obtained, a request is created with it and is redirected to the next MOC.

3.2.2 Hybrid Mobile Cloud Application - Demo Scenario

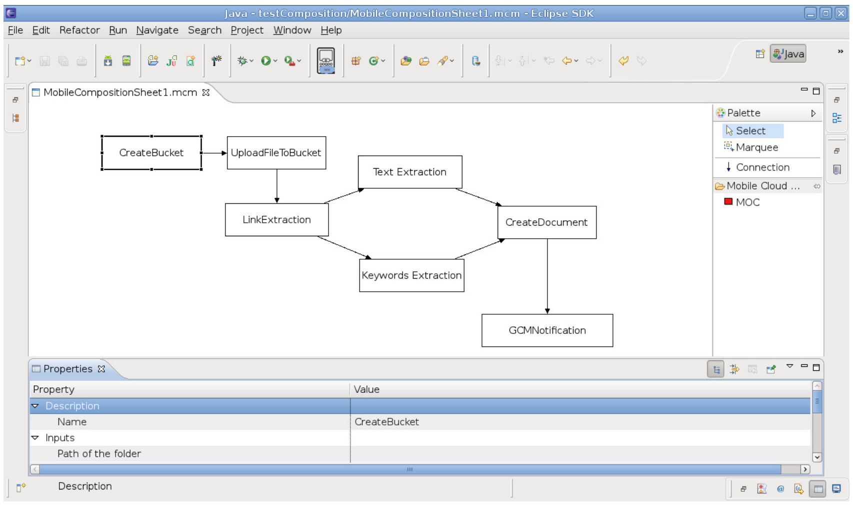
To demonstrate the composition feature of MCM, a hybrid mobile cloud application has been developed using MCM and its composition tools. We have developed the application in Android platform due to its popularity in the mobile market and unrestricted uses. However, the application could be developed for iOS or Windows Phone 7. The application benefits from its multi-cloud nature for performing a variety of cloud analyses, which are invoked by a single transaction. The aim of the application is to figure out whether the user likes the content of the Web pages that the user is reading on the mobile or not, so that the text of the most interesting articles can be extracted along with some keywords, which are obtained through machine learning analysis. Whether the user likes a particular page is calculated based on approximating the angle at which the phone is held to a fixed threshold which is set by obtaining stable accelerometer and gyroscope measurements. Later, from that point, it is sensed whether the handset experiments show intense movement or not. The information extracted from the analysis is stored in a Web document on the cloud for being accessed later through a URL using the Web browser of a mobile or a standalone computer.

Since most of the functionality of the mobile application is located on multiple clouds and is managed by the MCM, the client application running on the mobile is lighter and simple to build. The application consists of an EditText for typing an URL and a WebView for displaying its content. Once packed in an apk file, the application requires $\approx$ 512 Kb of storage on the mobile. When the application is launched, a concurrent process is triggered in the background. This process is used to store the information sensed by the accelerometer and the gyroscope sensors along with the active URL of the WebView, into a SQLite database. With each measurement written to the database, one tuple of the form, $<t_i$, $[x_i, y_i, z_i]$, $[g1_i, g2_i, g3_i]$, $URL >$is stored. Where t_i represents a timestamp measured in seconds, $[x_i, y_i, z_i]$ represents the data of the three axes of the accelerometer measured at time t_i and $[g1_i, g2_i, g3_i]$ is the data of the three directions of the gyroscope measured at time t_i.

When the application is closed (goes to the background), an *Async task* is executed on the *OnPause* method of the application. This task consists of a unique offloading to the URL of the compose service published at MCM. This offloading contains: 1) the data that will be used in the multi-cloud analysis (in this case, the database file); 2) the execution properties that allow to configure the cloud resources in runtime (bucket name, Amazon image id, instance size, region, Amazon username, eucalyptus image id, eucalyptus username, sensor analysis provider, text analysis provider, keyword analysis provider and document name); 3) a GCM request for registering the mobile at Google notification servers. The registration is mandatory for sending messages to the mobile using MCM and it is required only once. On this state, the application (activity) is also terminated so that the user can continue with other activities. However, the application will be re-activated via Broadcast Intent once a message from the notification service arrives with the result of the composition (URL of the document). The application uses MCM for invoking the services from Amazon, Eucalyptus, Google and AlchemyAPI.com. The cloud services are defined for composition in the Eclipse plugin as follows.

After adding a file with extension **.mcm* in the Java perspective of Eclipse, the composition tools of the plugin become enabled. The following services implemented at MCM are considered for the composition; *CreateBucket* represents a service that makes use of the Web Amazon API for creating a bucket in S3 and requires a bucket name as parameter. *UploadFileToBucket* uses jetS3t to locate objects in a specific Amazon bucket and requires a bucket name target. *StartInstance*, *EndInstance* and *RunScript* use typica for handling computational instances in Amazon and Eucalyptus. Here, the infrastructure parameters are used (instance size, region, amazon username, amazon image id, eucalyptus username, etc.). *KeywordExtraction* can be assumed as a black box service for the text analysis that is available by using the Web Alchemy API (text analysis provider). *CreateDocument* uses the gdata-docs API for creating the document, whose name is passed as parameter. Finally, *GCMNotification* represents the push notification service of Google.

Before creating the workflow of the application, some sub-composition is needed first. The subcomposition process consists of creating the *LinkExtraction* and *TextExtraction* services. The LinkExtraction is an Amazon computational service that applies a MapReduce activity recognition algorithm over the sensor data in order to understand how the user was holding the handset [66], and thus extracting the more interesting URLs (explained in Appendix A). LinkExtraction is created by connecting the MOCs StartInstance, RunScript



Property	Value
Description	
Name	CreateBucket
Inputs	
Path of the folder	

Figure 3.6: Workflow executed by MCM and triggered by a single service invocation

and EndInstance. Similarly, TextExtraction is a service running on Eucalyptus that implements the BulletParser[30] for extracting the text of a set of URLs (Web pages). TextExtraction shares the same logic as LinkExtraction and is created by connecting the MOCs StartInstance, RunScript and EndInstance. However, notice that RunScript differs on both services as the property *file path* varies.

The workflow is structured by connecting the mentioned MOCs. The logic of the workflow is shown in Figure 3.6 and it considers the following. After the database is offloaded to MCM, the middleware creates a bucket to locate the file (CreateBucket) and the location of the file (URL) is passed to the LinkExtraction service to obtain a list of URLs. Later, the list of URLs is passed in parallel to the TextExtraction and KeywordExtraction service. Once these services are finished and MCM obtains their results, MCM connects to Google docs to create a document with that information. Finally, MCM sends a message via GCM to the mobile device. The message contains the URL of the document which can be viewed in the browser of the mobile.

On the basis of the functional prototype of the mobile cloud application presented, we can derive that it is possible to handle process intensive hybrid cloud services from the smartphones via the MCM. Figure 3.7 shows the sequence of activities that are performed during the execution of the application. Here, the total application duration i.e. the total mobile cloud service delegation time for handling a multi-cloud operation asynchronously, is:

$$T_{mcs_a} \cong T_{tr} + T_m + \Delta T_m + \sum_{i=1}^{n} (T_{te_i} + T_{c_i}) + T_{pn} \tag{3.1}$$

[30]http://code.google.com/p/lightcrawler/

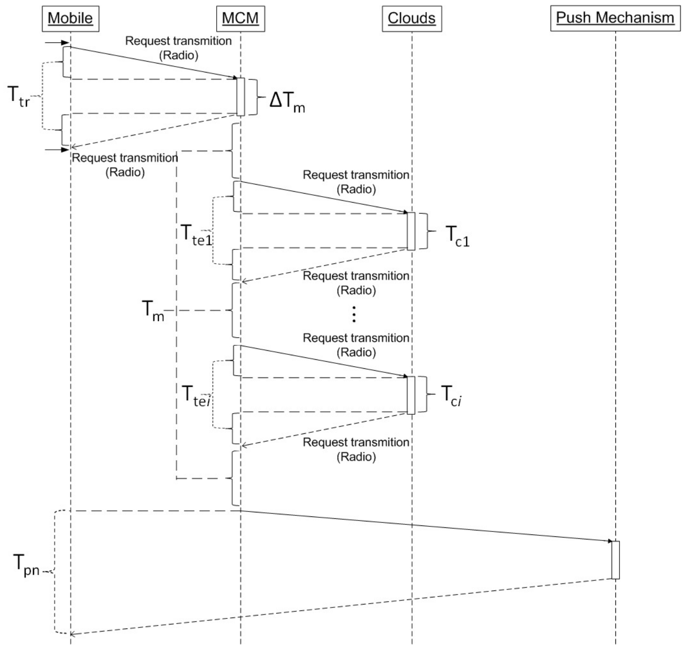

Mobile
MCM
Clouds
Push Mechanism
Request transmition (Radio)
T_{tr}
ΔT_m
Request transmition (Radio)
Request transmition (Radio)
T_{te1}
T_{c1}
Request transmition (Radio)
T_m
Request transmition (Radio)
T_{tei}
T_{ci}
Request transmition (Radio)
T_{pn}

Figure 3.7: Timestamps of the application scenario

Where, T_{tr} is the transmission time taken across the radio link for the invocation between the mobile phone and the MCM. The value includes the time taken to transmit the request to the cloud and the time taken to send the response back to the mobile. Apart from these values, several parameters also affect the transmission delays like the Transmission Control Protocol (TCP) packet loss, TCP acknowledgements, TCP congestion control etc. So a true estimate of the transmission delays is not always possible. Alternatively, one can take the values several times and can consider the mean values for the analysis. T_m represents the latency of receiving a request for delegation and sending a response to the mobile about its status. ΔT_m is the extra performance time added by the components of MCM for processing the request. T_{te} is the transmission time across the Internet/Ethernet for the invocation between the middleware and the cloud. T_c is the time taken to process the actual service at the cloud. $\cong$ is considered in the equation as there are also other timestamps involved, like the client processing at the mobile phone. However, these values will be quite small and cannot be calculated exactly. The sigma is considered for the composite service case, which involves several mobile cloud service invocations. However, in other cases, the access to multiple cloud services may actually happen in parallel. In such a scenario, the total time taken for handling the cloud services at MCM, T_{Cloud}, will be the maximum of the time taken by any of the cloud services ($Max_{i=1}^{n}(T_{te_i}+T_{c_i})$). Finally, T_{pn} represents the push notification time, which is the time taken to send the response of the mobile cloud service to the device. With the introduction of support for push notification services at the MCM, the mobile phone just sends the request and gets the acknowledgement back once the multi-cloud operation is performed. However, in this case, the delays completely depend on external sources like the latencies with GCM/APNS/MPNS frameworks and the respective clouds [33].

To analyze the performance of the application, a 5 MB of sensor data was stored in an Amazon bucket. Samsumg Galaxy S II (i9100) with Android 2.3.3, 32GB of storage, 1 GB of RAM, support for Wi-Fi 802.11 a/b/g/n was considered. Wifi connection was used to connect the mobile to the middleware. So, test cases were taken in a network with an upload rate of $\approx$ 1409 kbps and download rate of $\approx$ 3692 kbps, respectively. However, as mentioned already, estimating the true values of transmission capabilities achieved at a particular instance of time is not trivial. To counter the problem, we have taken the time stamps several times (5 times) across different parts of the day and the mean values are considered for the analysis.

The timestamps of the mobile cloud service invocation of the complete scenario is shown in Figure 3.8. The value of $T_{tr} + \Delta T_m$ is quite short ($<$ 870 msec), which is acceptable from the user perspective. So, the user has the capability to start more data intensive tasks right after the last one or go with other general tasks, while the cloud services are being processed by the MCM. The total time (workflow) taken for handling the cloud services at MCM, T_{Cloud} ($\sum_{i=1}^{n}(T_{te_i} + T_{c_i})$), is also logical and higher as expected. Cloud processing time also considers provisioning latency of computational resources. This latency represents the time of submitting a request for launching a resource and obtaining the resource in an active state. Figure 3.9 and Figure 3.10 show the execution time for each service that participates in the mashup. Cloud services created from scratch with IaaS were evaluated on different underlying hardware. Based on the results, we can observe that the allocation of cloud resources affects the execution time of a delegated mobile task, which is configurable dynamically by the middleware. Finally, T_{pn} varies depending on current traffic of the GCM service and has an average of $\approx$1.56 seconds. This notification average is obtained specifically to GCM through an 8 hours experiments. Refer to the analysis of performance of asynchronous delegation described in subsection 3.1.3.

3.2.3 Hybrid Cloud Service Composition Analysis

In order to analyze how MCM service composition adds value to the multi-cloud delegation process of a mobile cloud application, let us consider a similar analysis as the one presented by [36]. In their work, they claimed that a single offloading benefits the mobile resources if the mobile component, which

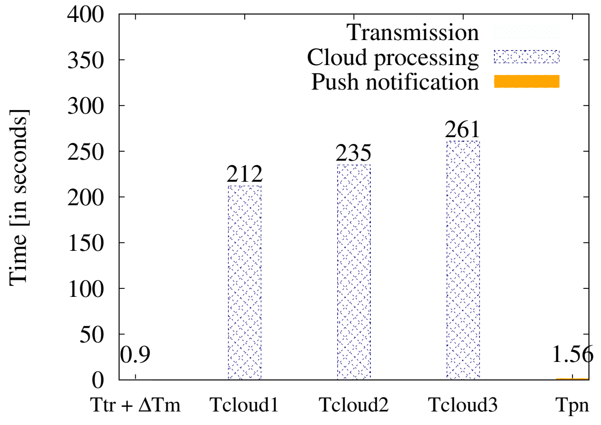


Figure 3.8: Mobile cloud service invocation timestamps. Different underlying hardware is considered for the execution of infrastructure services in *Tcloud.* Tcloud1, Tcloud2 and Tcloud3 use large, medium and small instances, respectively. Moreover, Tcloud also considers the provisioning time which has an average of ≈150 secs of any infrastructure type.

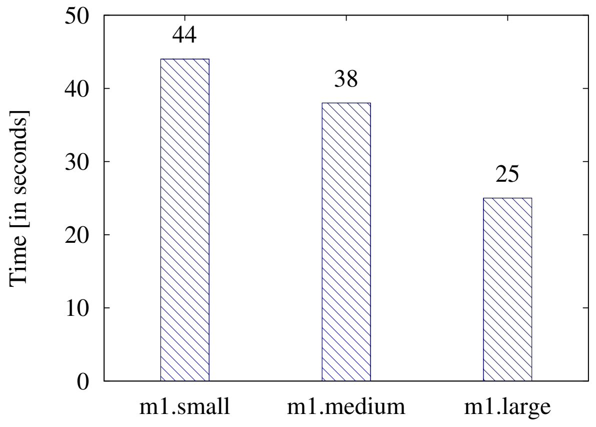


Figure 3.9: Execution time of sensor analysis service (*LinkExtraction*) on different instances: service at IaaS level

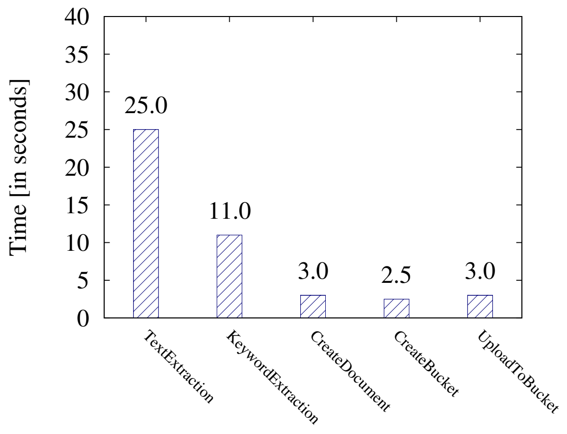


Figure 3.10: Execution time of the different cloud services that participate in the workflow: services at *SaaS* level

is offloaded to the cloud, requires huge amounts of computational resources to be processed and at the same time, the offloading process requires small amounts of data to be sent in the communication. Otherwise, it is preferable not to offload the mobile component and process it locally. We tried to apply the same principle in a delegation model as the mobile cloud communication is the one which introduces high overheads in the device [60]. In this context, a mobile cloud application that uses hybrid cloud services; it has to handle all the invocation logic locally, which is translated into multiple mobile cloud transactions. Moreover, the handset may also be forced to use extra processing power as each mobile task is delegated. This extra processing consists of data manipulation on the results acquired per each cloud transaction. Data manipulation may be needed for joining all the results collected from the cloud services or simply for re-converting the data in a suitable format for triggering the next service.

Due to the resource-intensive/time-consuming nature of the cloud services and the multiple frameworks that enable performing parallel processing on the cloud (e.g. Hadoop), for this analysis we do not consider that a cloud task can be performed on the mobile resources if the above condition is not met. We rather focus on how to decrease the number of mobile cloud communication required for delegating mobile tasks to hybrid cloud resources. With these assumptions in mind, the following example provides a simple analysis.

Suppose that E_w is the total amount of energy wasted by the mobile when executing a mobile cloud application. n is the number of hybrid cloud services in a mobile application (time to offload data). Let B be the bandwidth used in the communication between the mobile and the cloud and D is the size of the data in bytes that are exchanged. In each delegation, the mobile will consume (in watts), P_c for the processing performed by the mobile when handling the results of each cloud transaction, P_{tr} for transmitting and receiving data. For this analysis, we consider that transmission and receiving power are the same. However, depending on the approach for sending the result back to the mobile (e.g. notification services, real-time protocols such as XMPP etc), both will differ.

Conceptually, if the mobile cloud application is handled by the mobile resources using any approach discussed in subsection 3.1, the total time of energy consumed in the multi-cloud offloading process will be:

$$E_w \cong \sum_{j=1}^{n}((P_{tr} \times \frac{D_j}{B}) + P_{c_j}) \tag{3.2}$$

In contrast, when using MCM and its service composition mechanism, the hybrid cloud service integration occurs at the middleware, and thus n becomes equal to 1. Therefore, one data offloading is needed to trigger a bunch of different cloud services.

$$E_w \cong (P_{tr} \times \frac{\sum_{j=1}^{m}(D_j)}{B}) + P_c) \tag{3.3}$$

Where $\sum_{j=1}^{m}(D_j)$ represents the data sent per each cloud service requested (if any). Notice that in some cases, no data is sent as the output of one service may be the triggered input of the next service. So in the equation 3.3, m ($m \leq n$) is the number of services that participate in the composite service and require input from the mobile. The main purpose of the composition is to alleviate the mobile cloud service invocation T_{mcs_a} from unnecessary latency in the communication and to decrease the transfer of data.

3.3 Scalability of MCM

While MCM was successful in handling a multi-cloud operation from a mobile cloud application, the capabilities of MCM for handling heavy loads depend on its deployment aspects in the cloud and the dynamic configuration of those at runtime. Dynamic cloud reconfiguration mainly focuses on the distribution of application components that alleviate the entire system, when it is facing a certain condition (e.g. High CPU utilization, etc.), thus, increasing its capabilities for managing concurrency.

To verify the scalability of the middleware, MCM is located in a public cloud (Amazon EC2), in a cluster-based configuration that consists of a front-end node (Load Balancer- LB) and multiple end-nodes (MCM servers). Figure 3.11 shows the deployment scenario. Basically, the front-end node distributes the load between the back-end servers. Therefore, the LB requires a powerful CPU to handle the heavy demand. HAProxy[31] is considered as the LB as it allows dynamic behavior to the architecture and new MCM servers can be added while the system is running (hot reconfiguration). Back-end servers can be considered as commodity servers which can be replaced without affecting the overall performance of the cluster. Once, the scenario was set up, different mobile loads were simulated using benchmarking tools for testing the horizontal scalability properties of the middleware.

[31]http://haproxy.1wt.eu/

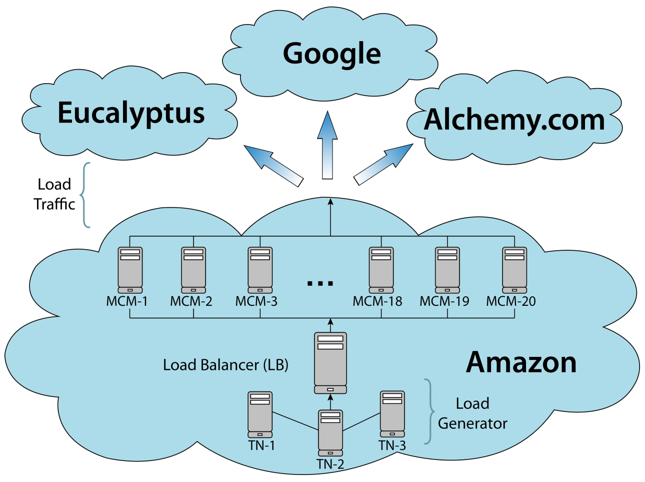


Figure 3.11: Load test setup for the MCM

3.3.1 Scalability Analysis of the MCM

Load testing of MCM was performed using Tsung[32] (open source multi-protocol load testing software). Tsung was deployed in a distributed cluster composed of three nodes running on separated instances (one primary and two secondary nodes). The primary node is in charge of executing the test plan and collecting all the results of each secondary node, so that information can be combined and analyzed into a single report using Tsung-plotter utility. The test plan is structured by blocks and consists of three parts, the server/client configuration part, in which the machines' information is defined. The load part that contains the information related with the mean inter-arrival time between new clients and the phase duration. Here, the number of concurrent users is defined. For instance, for generating a load of three hundred users in one second a mean of 0.0033 was used (Tsung primary node divided the load in equal parts among the available Tsung nodes). And finally, the session part, in which the testing scenario is configured and which consists of describing the clients' request (captured using Tsung-recorder).

A single client request consists of simulating the hybrid mobile cloud application described in subsection 3.2. The launch/terminate time of an Amazon instance is simulated by connecting to a large capacity running instance that performs the sensor processing. This time is simulated as launching a high number of instances that incurs in large utility costs (not feasible for a single user). The time is the average calculated from a set of individual sample times (launching/terminating) that were taken during the day for a small instance size. This simulation does not affect the overall results, since the communication with the service is established and the transaction is performed over the cloud resources. Same occurs with Alchemy.com service, which is constrained

[32]http://tsung.erlang-projects.org

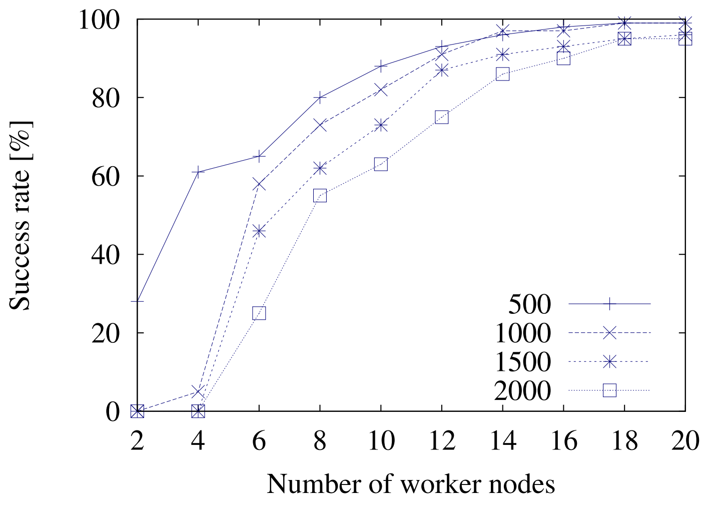


Figure 3.12: Success rate of concurrent requests over multiple server nodes

by a free request quota. This issue was solved by taking an estimation about the time that it takes for the service to process a request. GCM and other services did not present any problems. In the case of GCM, messages were routed to a single device.

Load traffic was simulated by n concurrent threads, where n varied between $\approx$ 150 and 650 per Tsung node (TN), making 500 to 2000 concurrent requests on the load balancer, thus, simulating a large number of concurrent users connecting to the MCM. On the cloud front, a load balancer and up to 20 MCM worker nodes were set up. To show the scale on demand of the solution, the number of server nodes was increased from 2 to 20. Initial amount of nodes was 2 and 2 nodes were added each time the setup needed to scale. Servers running on Amazon EC2 infrastructure were using EC2 m1.small instances. A small instance had 1.8 GB of memory and up to 10 GB of storage. One EC2 computational instance is equivalent to a CPU capacity of 2.66 GHz Intel®Xeon$^{\text{TM}}$processor (CPU capacity of an EC2 compute unit do change in time). Servers were running on 64 bit Linux platforms (Ubuntu). Finally, the load balancer was set up for using Round-robin scheduling, so the load could be divided into equal portions among the worker nodes.

In the load test of the MCM, the aim of the experiment is to measure 1) how the access policies of the framework are enhanced by scaling the infrastructure horizontally and 2) how the success rate of the requests depends on the number of framework nodes depending on the number of concurrent requests. A request is considered a success, if it gets a response back (i.e. transaction completed) before the connection or response timeout occurs. Similarly, the success rate indicates the number of requests from all performed requests that have succeeded. The results of the experiments are shown in Figure 3.12. From the diagram it can be observed that the success rate follows a logistic function, with the number of nodes. The performance of eight nodes drops to $\approx 75\%$ after receiving 1500 concurrent requests. However, 18 nodes can handle this load with almost 100% success rate. It can also be seen that with current test architecture adding more worker nodes does not show any visible improvement in the performance after 18 nodes in contrast when the setup was composed of 2, 4 and 6 nodes.

To sum it up, with current MCM implementation, pair nodes deployment may handle around 100-150 concurrent requests with almost 100% success rate. An addition of two nodes adds roughly the capacity of handling another 100 requests until the load grows up to ≈ 1800 concurrent requests, when the load balancer itself becomes a bottleneck. Hence, adding more nodes does not improve the performance as desired. The analysis also shows that the elasticity of the cloud helps to achieve this required setup easily.

3.4 Summary

Mobile Cloud Computing (MCC) is arising as a prominent research area that is seeking to bring the massive advantages of the cloud to the constrained smartphones and to enhance the telecommunication infrastructures with self-adaptive behavior for the provisioning of scalable mobile cloud services. Mobile cloud applications [60] are considered as the next generation of mobile applications, due to their promise of bonded cloud functionality that augment processing capabilities on demand, power-aware decision mechanisms that allow to efficiently utilize the resources of the device, and their dynamic resource allocation approaches that allow to program and utilize cloud services at different levels (SaaS, IaaS, PaaS). However, adapting the cloud paradigm for mobile devices is still in its infancy and several issues are yet to be answered. Some of the prominent questions are: how to decide from the smartphone, the deployment aspects (e.g. type of instance) of a mobile task delegated to the cloud? How to decrease the effort and complexity of developing a mobile application that requires accessing distributed hybrid cloud architectures? How to handle a multi-cloud operation without overloading the mobile resources? How to keep the properties (e.g. memory use, application size etc.) of a mobile cloud application similar to that of a native one?

To counter the problems of accessing multiple cloud services through Web APIs and invoking a time consuming operation from a mobile app [6], we develop MCM Framework. MCM fosters an MBaaS solution, where a middleware abstracts the Web APIs of multiple clouds at different levels and implements a unique interface that responds (JSON-based) according to the cloud services requested (REST-based). MCM provides multiple internal components and adapters, which manage the connection and communication between different clouds. Since most of the cloud services require significant time to process the request, it is logical to have asynchronous invocation of the cloud service. Asynchronicity is added to the MCM by using push notification services provided by different mobile application platforms and by extending the capabilities of a XMPP-based IM infrastructure [33]. Furthermore, MCM fosters a flexible and scalable approach to create hybrid cloud mobile applications based on declarative task composition. Task composition is considered for representing each mobile task to be delegated as a *MCM delegation component.* A composed task is developed graphically in an Eclipse plugin based on user driven specifications and it is modelled as a data-flow structure, where each task depicts a cloud service to be invoked. Once developed, a composed task is deployed within the middleware for execution that is triggered by a single invocation from the mobile. This means that data transmission is minimized in order to decrease computational effort in the device for going cloud-aware. Finally, a prototype of MCM is developed and several applications are demonstrated in different domains. To verify the scalability of MCM, load tests are also performed on the hybrid cloud resources. The detailed performance analysis of the middleware framework shows that MCM improves the quality of service for mobiles and helps in maintaining soft-real time responses for mobile cloud applications.

Chapter 4

Evidence-aware Mobile Computational Offloading

As explained in Chapter 2, the effectiveness of a computational offloading is determined by the ability of the system to infer where the execution of code (local or remote) requires less processing effort to the mobile, so that by correctly deciding *what, when, where and how to offload*, the device obtains a benefit [15]. Latest works offer partial solutions that ignore the majority of these considerations in the inference process. Most of the proposals demonstrate the utilization of code offloading in controlled environment by connecting to low-latency nearby servers, e.g. lab setups, and inducing the code to become resource intensive during runtime [25, 67]. As a result, in practice, in most of the cases, computational offloading is counterproductive for the device. Thus, the need to offload code or not is debatable [13, 36]. It is well known that computational offloading techniques are mainly needed to support applications that implement heavy routines oriented to computer graphics. Moreover, some researchers have concluded that most power-hungry applications are the most popular as their usage rate is higher in comparison with other applications. Unfortunately, most of the popular applications are based on data synchronization (e.g. Twitter, Facebook, Instagram, LINE, Whatsapp, etc.) [68, 69], which are unsuitable for code offloading strategies. However, code offloading will become critical as the mobile game revolution takes over the market and the social applications evolve into graphical interfaces that augment the context of the user in order to enrich his/her experience, e.g. augmented reality.

Code profiling is one of the most challenging problems in a computational offloading system, as the code has a non-deterministic behavior during runtime, which means that it is difficult to estimate the running cost of a piece of code considered for offloading [70]. Moreover, computational offloading is influenced by many other parameters of the system that come from multiple levels of granularity, e.g. type of device, communication latency, data size transferred, processing capabilities of the surrogate, etc. These parameters represent the context of the device. By evaluating these parameters, a decision process that takes place at the mobile determines whether offloading a task is productive or not. Thus, an offloaded task that is unfavorable for the device is the result of a wrong decision process, which tends to get imprecise based on the scope of observable parameters of the system that the process can consider into its decision [14].

The vast resource allocation choices in the cloud ecosystem and the large diversity of smartphones make the context very variable [14, 38], and thus it is a complex task determining or adapting the right conditions of the system to offload. However, we believe that computational offloading to cloud is possible without a negative impact on the device. We believe that a richer context can be expressed in terms of multiple dimensions, e.g. *what, when, where, how, etc.* A dimension defines the properties that a mobile application must meet in order to offload a task. For instance, a face recognition app installed in device (*which*) offloads a task at code level (*what*) under conditions (*when*) to a remote server of type (*where*) to be executed (*how*) in parallel. Moreover, dimensions can scale to consider other mobility parameters, e.g. *user's location*, which can facilitate the process of pre-caching apps functionality.

We claim that the instrumentation of apps alone is insufficient to adopt computational offloading in the design of mobile systems that rely on the cloud. Computational offloading in the wild can impose more computational effort on the mobile rather than reduce processing load [13]. In contrast to existing works, we overcome the limitations of computational offloading in practice by analyzing how a particular smartphone app behaves in a community of devices [14]. Computational crowdsourcing strategies are viable solutions to understand potential problems in software applications with high accuracy level, e.g. bugs, leaks, etc. The main advantage of relying on a community is to capture the diversity of cases in which an applications works. Our fine-grained framework at code level is inspired by the coarse-grained solution Carat [71], which attempts to find out energy bugs and hogs in the mobile apps. Carat analyzes big repositories of data that depict the runtime behavior of a mobile app in order to find anomalies that can turn into customized recommendations for preferable configurations, e.g. apps to kill, by which the mobile user can apply to make the battery life of his/her device last longer.

By following similar principles as Carat, it is possible to determine the conditions and configurations for offloading smartphone applications. For instance, by applying the Carat method [71] over a subset of data ($\approx$328,000 apps), we can get an idea about what is a resource-intensive app. Based on this data, which is collected in a real life deployment, we develop several case studies to motivate the viability and applicability of our approach. The subset contains for each app the expected % of energy drain. The Carat method consists of determining the energy drain distribution of a particular app, and then compares it with the average energy drain distribution of other apps running in the device. The key insight of the method is to determine the possible overlap between application energy distributions in order to detect anomalies in application energy usage.

By analyzing apps' category and specific purpose, we develop four case studies, which consist of face manipulation, games, puzzles and chess. Figure 4.1 shows the results of the case studies. For each case study, we extract a group from the subset of apps, where each app in the group is different from the rest. The four groups consist of about 550 face manipulation apps, about 7,805 game apps, about 717 puzzle apps, and about 166 chess apps. Each group is compared with the average energy drain of the subset, which excludes its energy drain. From the results, we can observe that around (a) 43.84%, (b) 44.56%, (c) 42.75% and (d) 33% of apps implement computational operations that require higher energy drain than normal, which is a significant number of apps. Consequently, computational offloading is required, e.g. customized alarm, to overcome the extra overhead introduced by the apps when showing resource-intensive behavior.

Generally speaking, higher granularity data is able to provide insights about the right conditions of these apps to offload to cloud. However, beyond equipping the apps with computational offloading, this requires to instrument the mechanisms with the ability to record its own local/remote execution, so that we can capture more specific details about what induces the code to become resource-intensive and what is the runtime behavior of the code in the device/surrogate. Thus, it is reasonable that the characterization of the computational offloading process can be modeled through a community of devices, so that by taking advantage of the huge amount of devices that connect to cloud, it can be possible to foster a more effective offloading strategy for the smartphones.

Implicit crowdsourcing that does not need incentives, but rather it's extrapolated from application usage can be used to collect history traces of the computational offloading process across the entire system. Traces can be analyzed using cloud analysis features to extract the characterization. The purpose of the characterization is to define the effect of a remote code execution in different conditions and configurations, where a condition depicts the interaction

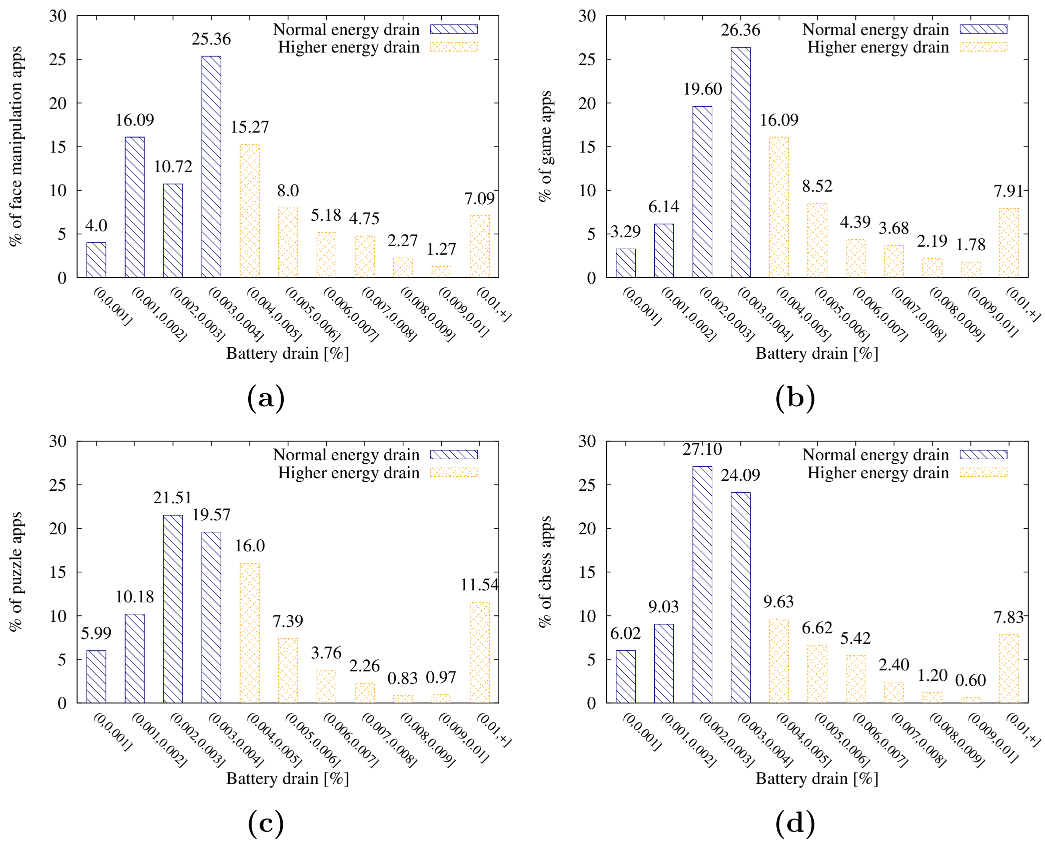


Figure 4.1: Smartphone apps that depict higher energy drain

aspects of the user with the mobile, e.g. available memory and CPU, input variability, etc., and a configuration represents the state of the components of the systems, e.g. bandwidth size, capacity of the cloud-surrogate, performance metrics of the mobile/back-end, etc. In this manner, we can find out the most accurate configurations (what to offload) for a specific application in a particular device based on multiple criteria, such as type of the surrogate (where to offload) and conditions of the system (when to offload). Additionally, it can be possible to determine offloading plans (how to offload) that enable the device to schedule code offloading operations, e.g. computational parallelization of code [72].

Furthermore, the characterization can also be utilized to identify reusable results. A reusable result is a portion of code that is commonly offloaded by multiple devices. These results can be cached in cloud to respond duplicated offloading requests from other devices. Logically, this accelerates the offloading process as the surrogate avoids the invocation time of the request. We envision that as a part of the characterization process, pre-cached functionality from the entire mobile application can be requested on demand as depicted in Figure 4.2. In this manner, our cloud assistance approach delivers a system, where the cloud is the expert and mobile devices ask the cloud for its expertise.

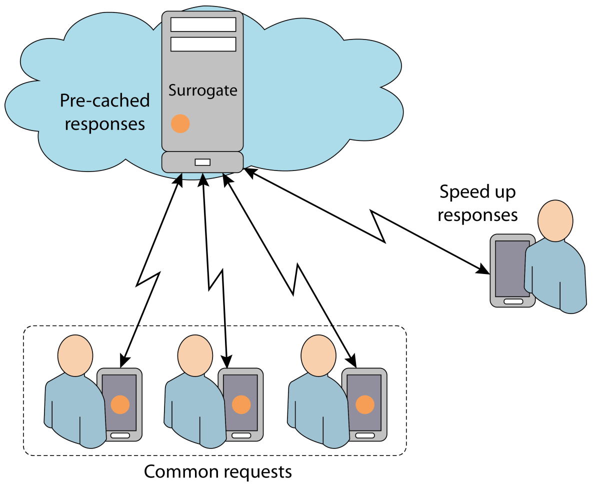


Figure 4.2: Characterization of the offloading process that considers the smartphones diversity and the vast cloud ecosystem

4.1 Design Goals and Architecture

EMCO handles two types of mobile communication with cloud, one synchronous for code offloading, and the other asynchronous for injection of data analytics via push notifications. Architecture is shown in Figure 4.3. EMCO follows a model, where the source code of the apps reside in both the mobile and server, but it releases the mobile from a fixed cloud counterpart as implemented by other frameworks. Thus, EMCO encourages a scalable provisioning style *as a service*, where EMCO is deployed in multiple interconnected servers and responds on demand to computational requests from any available server. EMCO provides a toolkit that facilitates the integration of the mobile and cloud mechanisms in the development life cycle. The rest of the section describes the toolkit and framework in detail.

4.1.1 Development Toolkit

Smartphone apps are instrumented with computational offloading mechanisms through different methods. However, some of the counterproductive effects of computational offloading in practice are caused by applying these methods, e.g. code annotations. To counter this problem, EMCO provides a toolkit, which consists in a GUI conversion tool that automates equipping the apps with EMCO and preparing them for client/server deployment. The conversion tool transforms all the methods that fulfil the requirements to offload to cloud [18]. Apps are also equipped with mechanisms that record local/remote execution of the methods into a SQLite database and upload that data to cloud. The tool receives as input an Android project and produces as output two projects, one for the mobile and the other one for the cloud. Each project

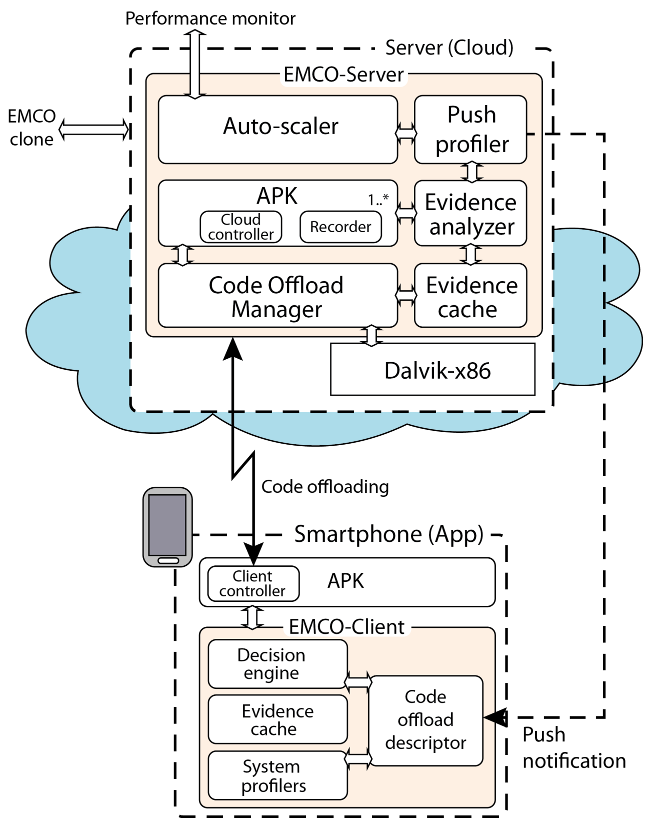


Figure 4.3: *Evidence-aware Mobile Code Offloading* architecture

is defined with maven-android[33], so that for each project an Android Application Package (APK) file is built automatically. The tool also provides the means for automatic deployment of the APK in the cloud.

4.1.2 Smartphone-side

The *EMCO-Client* is a lightweight mechanism running in the device that consists of a set of profilers, a decision engine and an asynchronous mechanism that receives data analytics from cloud. Client components are described as follows:

- ***System profilers*** are the local monitors that in real time estimate the value of the parameters, which influence the offloading process. According to Kumar et al. [36], code offloading is reasonable for the device, if in the presence of low *Round Trip Times* (RTT), the data sent in the communication channel is small, and requires huge amount of processing. Thus, profilers are in charge of monitoring network bit rate and data size to transmit. RTT is calculated using the channel of code offloading. Time samples are collected that depict the time that takes to connect via *Sockets*. We limit the number of samples to 5 in order to

[33]http://code.google.com/p/maven-android-plugin/

avoid a loop of no server found. Data size is calculated by determining the length of the object stream in bytes. In the case of the code profiler, it is powered by the cloud. We believe that the estimation about how much computational effort is required by the device to execute a portion of code in runtime is a complex task and the key reason that makes computational offloading to produce a counterproductive effect. Thus, in our system, computational effort is determined by the majority of the crowd, which implies that history traces from the community that describes the energy consumed to locally and remotely execute the code are analyzed (explained further in this section). This analysis occurs at the cloud considering the properties of each device. Notice that the remote case is more variable than the local one as it depends on the computational properties of the cloud server. The variations are caused by the time of code execution in the server. Estimation of the energy is captured using PowerTutor [73], which has been shown to provide lower estimation error by other works [71]. Code profiling information is pushed from the cloud into the *code offload descriptor*. Finally, the system profilers are implemented as Android services for convenience. An application can bind to the service in order to retrieve its information and multiple applications can use a service.

Snippet 1 Computational offloading descriptor in JSON

```
{
    "mobileApplication": "Chess",
    "deviceID": "i9300",
    "what-to-offload": [
        {
            "candidates-methods": "MiniMax"
        }
    ],
    "when-to-offload": [
        {
            "Energy-consumed": "180J",
            "Fuzzy-engine": [
                    {
                     "linguisticVariable" : "BANDWIDTH",
                         "linguisticTerms":[
                           {"id": "speed_low",
                           "functionType": "Trepezoidal",
                           "Min": "384kbps",
                           "Max": "512kbps",
                    },
                     [...]
                }
            ]
        }
    ],
    "where-to-offload": [
        {
            "best-surrogates": "m2.4xlarge, m2.4xlarge",
            "acceptable-surrogates": "m1.large, m1.xlarge",

        }
    ],
    "how-to-offload": [
        {
            "parallelization": 0,
        }
    ],
    "Other": [...]
}
```

- ***Code offload descriptor*** controls the data management —create, update, delete—of the data analytics sent from the cloud for a particular app. The data is stored into the app space and contains the characterization of the computational offloading process in JSON format. The characterization defines the offloading process based on multiple dimensions (Snippet 1). First, it specifies what to offload by indicating the candidate methods to the code profiler. Second, it defines when to offload the methods by establishing the informational thresholds that the device must detect. Next, the characterization defines a list of surrogates, which describes from the most convenient to the most acceptable server in which the device can offload. Later, it defines an execution plan for the code, e.g. parallelize the code in n processes. Finally, the characterization can define other dimensions that exploit code execution patterns, which are found from the analysis of the community history of the app. For instance, a dimension *reusable* can define that the result of a specific method is pre-cached to the cloud as the request is found to be the same in all the other cases from the community. A dimension *user-location* can define preferable surrogates to offload based on user's location, e.g. low latency hot spots. Besides to provide more fine-grained details to offload, the descriptor introduces additional advantages as it allows the decision engine to have a wide view of the system to evaluate the impact of a remote execution. The descriptor is loaded during runtime using GSON[34] when the application starts or an Android notification *Intent* occurs.

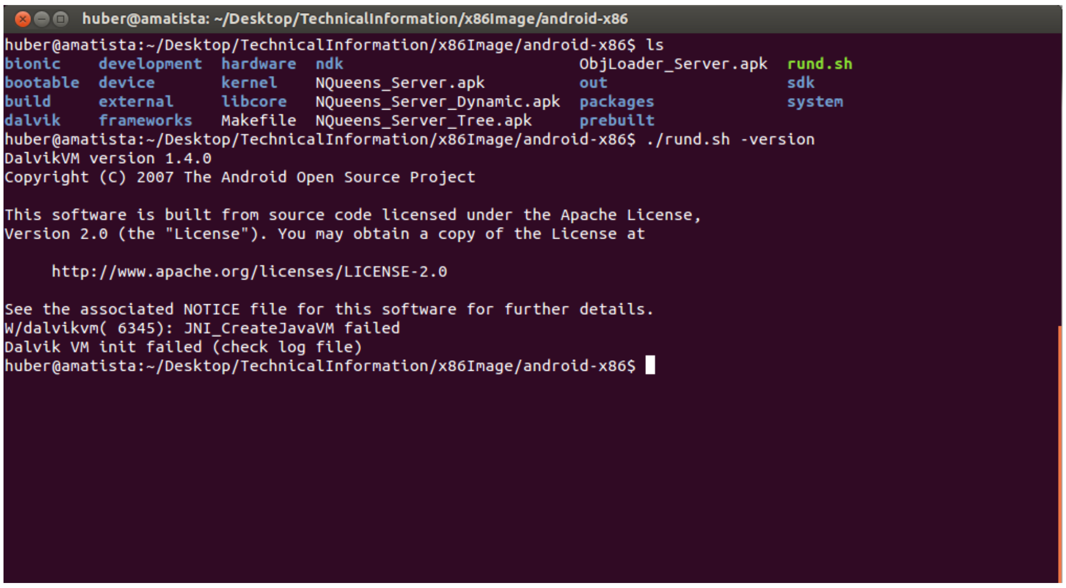
- ***Decision engine*** estimates whether it is productive or not to offload. The engine loads the information defined in the code offload descriptor to create a reasoner based on fuzzy logic. We implement a fuzzy logic described in [21] as we realize that the computational load to create the reasoner can be shared between the mobile and cloud. The cloud discovers the data values to create the fuzzy sets, rules, and membership functions, and the mobile just loads the information. Thus, the engine is lighter for the device. Once the reasoner is created, in order to decide whether to offload or not, the system profilers pass to the reasoner as input the parameters of the context along with the information of the code to execute.
- ***Evidence-cache*** in the client side, it is utilized to cache the result of methods which are calculated by the cloud. The results are stored into the cache in case a result is likely to be used again by the mobile app. In this manner, an app avoids to make the same remote method invocation twice or more.

4.1.3 Cloud-side

The *EMCO-Server* is deployed on top of our customized Dalvik-x86[35]. Dalvik is the virtual machine of Android to execute dalvik bytecode. Dalvik-x86 is built by downloading and compiling the source code of *A*ndroid *O*pen *S*ource *P*roject (AOSP) over the instance to target a x86 server architecture, and removing the *Applications* and *Application Framework* layers from the Android software architecture as shown in Figure 4.4.

Unlike other frameworks that rely on the virtualization of the entire mobile platform to execute the code remotely like in the case of Android-x86, we think that such virtualization is unnecessary (waste of CPU resources) and counterproductive (slows down performance). Proof of that relies on the fact that for equipping a server with that execution environment, it is required to have a lot of storage space available in the server. For instance, we needed storage space >100 GB to install the software, which includes AOSP code

[34]http://code.google.com/p/google-gson/
[35]Released as public in Ireland region of Amazon EC2 as *ami-c42fd9b3*



```
huber@amatista: ~/Desktop/TechnicalInformation/x86Image/android-x86
huber@amatista:~/Desktop/TechnicalInformation/x86Image/android-x86$ ls
bionic     development  hardware  ndk                          ObjLoader_Server.apk  rund.sh
bootable   device       kernel    NQueens_Server.apk           out                   sdk
build      external     libcore   NQueens_Server_Dynamic.apk   packages              system
dalvik     frameworks   Makefile  NQueens_Server_Tree.apk      prebuilt
huber@amatista:~/Desktop/TechnicalInformation/x86Image/android-x86$ ./rund.sh -version
DalvikVM version 1.4.0
Copyright (C) 2007 The Android Open Source Project

This software is built from source code licensed under the Apache License,
Version 2.0 (the "License"). You may obtain a copy of the License at

     http://www.apache.org/licenses/LICENSE-2.0

See the associated NOTICE file for this software for further details.
W/dalvikvm( 6345): JNI_CreateJavaVM failed
Dalvik VM init failed (check log file)
huber@amatista:~/Desktop/TechnicalInformation/x86Image/android-x86$ 
```

Figure 4.4: Low-level Dalvik-x86 compiler built from Android open source project

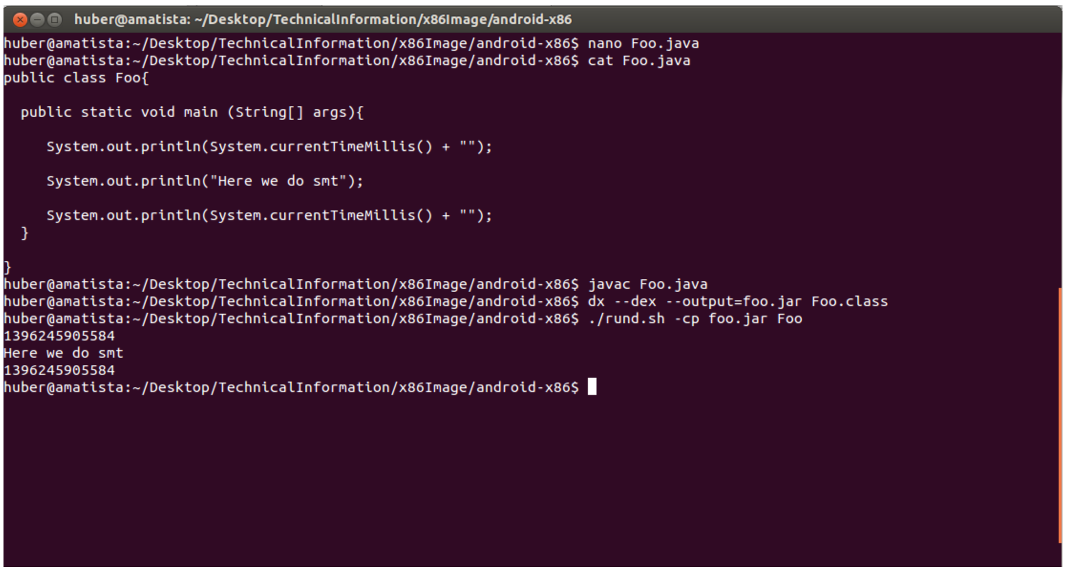
and Android SDK, among others. Moreover, once the OS is running in the server, the OS activates all its default features, e.g. Zygote, GUI Manager, etc., which are not necessary in the surrogate. As a result, the CPU utilization of the surrogate increases. Thus, in our system, we opted to extract the Dalvik compiler from the mobile platform, and deployed it straight over the underlying physical resources. Dalvik-x86 implements an executable script wrapper to the core of libraries that boot the compiler. The content of the script is shown in Snippet 2. In this manner, we reduce the storage size required by the system. Our Dalvik-x86 surrogate requires to have attached a volume size of 60 GB as minimum. Moreover, it does not active any default processes from the OS. The surrogate creates a *dalvikvm* process in the host machine per each offloading request that needs to be handled. The main advantage of this approach is that in the case an offloading request failed or gets stuck, it is possible to kill that particular process without restarting or stopping the complete offloading system.

Snippet 2 Wrapper that boots the Dalvik machine in a x86 machine

```
#!/bin/sh
#This wrapper works with our Dalvik x86 image running in Amazon
#Please contact at huber AT ut DOT ee for granting access to the image


# base directory, at top of source tree; replace with absolute path
base=`pwd`

root=$base/out/host/linux-x86
export ANDROID_ROOT=$root

# configure bootclasspath
# some extra jars are required besides services.jar, core.jar, ext.jar and framework.jar
bootpath=$base/out/target/product/generic_x86/system/framework
export BOOTCLASSPATH=$bootpath/core.jar:$bootpath/core-junit.jar:$bootpath/bouncycastle.jar:
$bootpath/ext.jar:$bootpath/framework.jar:$bootpath/android.policy.jar:$bootpath/services.jar

export LD_LIBRARY_PATH=$bootpath/lib:$LD_LIBRARY_PATH

# this is where we create the dalvik-cache directory; make sure it exists
export ANDROID_DATA=/tmp/dalvik_$USER
mkdir -p $ANDROID_DATA/dalvik-cache

exec $root/bin/dalvikvm -Xbootclasspath:$BOOTCLASSPATH $@$
```

The wrapper provides an interface to push bytecode for execution as system's process in the host operating system (Figure 4.5). When the server initiates, the available APK files are pushed into the Dalvik machine as a process waiting for request. Each APK file can be instantiated multiple times and listened in different ports. In this manner, when an offloading request occurs, code is invoked by forwarding the request to any process listening at the server. This means that the surrogate is released from the limitations imposed by the mobile operating system, such as to activate multiple instances from the same application, and to execute multiple applications concurrently, among others. The surrogate is stored as image in the cloud, and thus, the underlying resources can change on demand to increase the throughput of the system, which means that the server can increase the capabilities to handle multiple Dalvik processes simultaneously (multiple users) or to speed up the execution of code (different quality of service). The components of the server side run on top of the Dalvik-x86 and are described as follows:

- ***Code offload manager*** is responsible for handling the code offload requests. When a request arrives, the manager reads the request, and extracts an *object package*, which contains the information details about the app and all the data to invoke the method, e.g. type of parameters, value of parameters, etc. The manager uses the data to check in the *Evidence cache* component if the method was invoked before by other app of the community, so that there is an existent associated result. If the result is available, then the manager will send it back to the handset. This speeds up the offloading process as the remote execution time is


huber@amatista: ~/Desktop/TechnicalInformation/x86Image/android-x86

```
huber@amatista:~/Desktop/TechnicalInformation/x86Image/android-x86$ nano Foo.java
huber@amatista:~/Desktop/TechnicalInformation/x86Image/android-x86$ cat Foo.java
public class Foo{

  public static void main (String[] args){

     System.out.println(System.currentTimeMillis() + "");

     System.out.println("Here we do smt");

     System.out.println(System.currentTimeMillis() + "");
  }

}
huber@amatista:~/Desktop/TechnicalInformation/x86Image/android-x86$ javac Foo.java
huber@amatista:~/Desktop/TechnicalInformation/x86Image/android-x86$ dx --dex --output=foo.jar Foo.class
huber@amatista:~/Desktop/TechnicalInformation/x86Image/android-x86$ ./rund.sh -cp foo.jar Foo
1396245905584
Here we do smt
1396245905584
huber@amatista:~/Desktop/TechnicalInformation/x86Image/android-x86$ █
```

Figure 4.5: Execution of code using Dalvik-x86

avoided. If the result is not available, then the manager identifies the mobile app and its associated Dalvik processes in the server in which the request can be forwarded. Once a process is selected, the connection is forwarded to it, so that the *CloudController* of the APK file instance can invoke the method via Java reflection. The result obtained is packed and sent back to the mobile. Paralelly to this process, a trace is created by recording the execution of the method, e.g. execution time, type of server, CPU load, etc., and stored along with the data of the request and its result. Finally, the trace is passed to the *Evidence analyzer*, where the trace is analyzed with the rest of the traces from the community.

- ***Push profiler*** implements the cloud-based messaging component based on GCM service. The details of GCM are already addressed in Chapter 3, subsection 3.1.1. EMCO sends a notification as follows: EMCO server is registered to GCM service using an *API key* obtained from Google APIs console. Multiple servers can use the same key. A mobile application subscribes to EMCO to receive notifications and thus it obtains a *sender ID*. This ID is temporal and lasts until the application explicitly unsubscribes or until Google refreshes the GCM service. EMCO sends a message to the mobile by sending a push request to the GCM service. The request consists of the *API key*, the *sender ID*, and the message payload. Requests are enqueued for delivery (with maximum of 5 attempts) or stored in case the mobile is offline. Once the message reaches the device, the Android system executes a *brodcast intent* for passing the raw data to the *code offload descriptor*.

EMCO relies on push technologies to aggregate code offload descriptors into the smartphone apps. Since each message sent through GCM is limited to 1024 bytes, the number of messages to send in order to form a descriptor depends on the descriptor length. Messages are sent to the mobile based on different events. For instance, dynamic cloud allocation, periodical trace analysis, location detection of the mobile, etc. Since GCM is a public service used by huge amount of customers, it does not guarantee the delivery of a message and is unreliable to be used in real-time applications. To counter this problem, EMCO provides a generic interface, which can be used to easily integrate other push technologies, e.g. MQTT (Message Queue Telemetry Transport). We have also developed our own push server[36] based on XMPP, which can overcome the issues of proprietary push technologies [33].

- ***Auto-scale*** is the component that grants the framework the ability to scale for multi-tenancy. While a cloud vendor provides the mechanisms to scale SOA applications on demand, e.g. Amazon autoscale, it does not provide the means to adapt such strategies to a computational offloading system as the requirements to support code offloading are different. The requirements of a code offloading system are based on the perception that the user has towards the response time of the app. The main insight is that a request should increase or maintain certain quality of responsiveness when the system handles heavy offloading traffic, in other words, multiple devices offloading to a single server. Thus, a code offloading request cannot be treated indifferently. The remote invocation of a method has to be monitored under different system's throughput to determine the limits of the system not to exceed the maximum number of invocations that can be handled simultaneously without losing quality of service.

  EMCO can be deployed in a hierarchical control and supervision schema as shown in Figure 4.6, where an initial EMCO server acts as a parent (aka master), and a monitor server collects performance metrics from the available EMCO servers (based on collectD[37]). The parent uses the performance metrics to create EMCO children (aka clones), which are utilized to share the load of incoming mobile users. The clone can become a second level parent if it reaches its service utilization limits. Each EMCO server implements a performance-based policy that uses the traces to determine the minimun CPU free that has to maintain in

[36]https://github.com/huberflores/XMPPNotificationServer
[37]http://collectd.org/

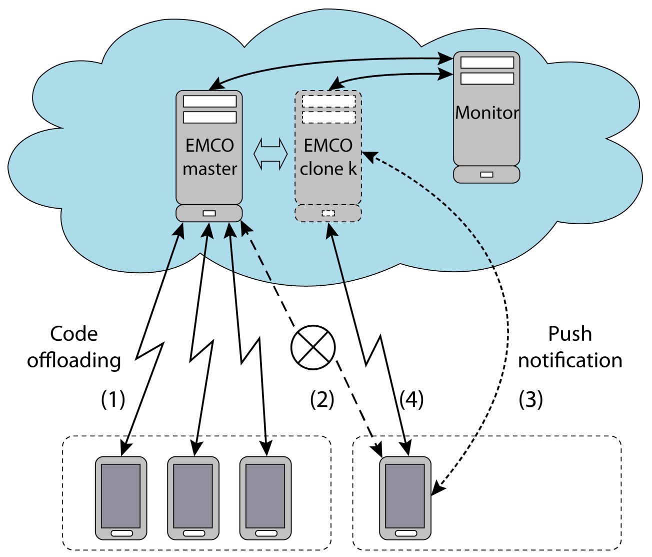


Figure 4.6: EMCO support for multi-tenancy and horizontal scaling

order to invoke a particular method. When a server is destroyed, the subscribed mobile devices are passed from the child to its parent. Each parent collects performance metrics from the collectD server, and when the parent reaches its service utilization limits (1) and cannot handle more requests (2), the parent creates another child. In this process, the list of subscribed devices is splitted between the parent and the child, and a push notification message is sent to each device to update the remote information of the surrogate (3). In this manner, code offloading requests are balanced among the servers (4).

- ***Evidence-cache*** in the server side contains the pre-cached results of method invocations that can be shared among the community. Since a computational task that is offloaded to cloud is serializable, the *Code offload manager* calculates per each request a MessageDigest key based on *SHA-1 checksum* to uniquely identify each request. By clustering all the traces using DBSCAN based on checksums, the *Evidence analyzer* can find those generic computational tasks in the set of traces. Once a computational task is detected as generic, the *Evidence analyzer* can push it along with its checksum into the *Evidence cache.*

- ***Evidence analyzer*** is in charge of analyzing the evidence, which is stored in terms of traces and extracting the dimensions to create the *code offload descriptor* for a particular app. Traces contain contextual information about the local and remote execution of the code at a particular point in time when the app is used. A trace consists of a list of features: device model, app identifier, name of method executed, local execution time, RTT, type of server, remote execution time, etc. A local trace differs from a remote trace as its remote features' values are assigned with *null*, e.g. "type-of-server = null". Additionally, in the case code is offloaded, a trace includes an additional parameter that shows if the remote invocation of code has failed or succeeded. Following the Carat principles [71], the goal of the analysis is to transform the evidence into a set of rate distributions, which can be utilized to compare local code execution with remote one. We compare distributions using as features, response time and energy consumed. Other features can also be included in order to refine accuracy. However, the main requirement for feature comparison is that the feature can be found in the mobile and the surrogate. For instance, the responsiveness of the app when executing the method in the device is comparable with the responsiveness of the app when executing the method in the surrogate.

Rate distributions are computed for each app, initially, traces are converted into a set of samples. Let $s_t = (c, p, \hat{f})$ denote a sample taken at time t of code execution with details c, e.g. app id, method name, which is processed at p, e.g. local or remote. The remaining features are denoted collectively as a set $\hat{f}$ of key-value pairs, e.g. "response-time=2s", "energy-consumed=180J", etc. We sort the samples based on t and splitted them into two groups, one for local and other remote. For each group a rate distribution is created. In this process, for each feature selected fs for comparison, consecutive pairs, e.g. (s_{t_1}, fs) and (s_{t_2}, fs), are converted into a rate distribution $u = \frac{(fs_2 - fs_1)}{(t2 - t1)}$, which later is associated with the rest of features to create a pair $R = (u, \hat{f})$.

Once the rate distributions are calculated for a feature, pairs found R_{local} and R_{remote} are compared. Comparison consists of finding an overlapping between the two distributions in order to remove those slices where overlapping is found (Figure 4.7). Usually, R_{local} values are higher than R_{remote} as local processing influences more effort to a feature. Thus, an overlapping depicts when remote is counterproductive for the device. Once the overlapping slices are removed from the samples, next rate distribution feature is computed as described previously over the new set of samples. This means that features are applied one after the other in a gradual refinement process, which aims to keep only those contextual properties that are shown to be favorable to offload.

Finally, after obtaining the last feature comparison with no overlappings, the dimensions of the *code offload descriptor* are created. We rely on the distance d between distributions to determine the level of improvement

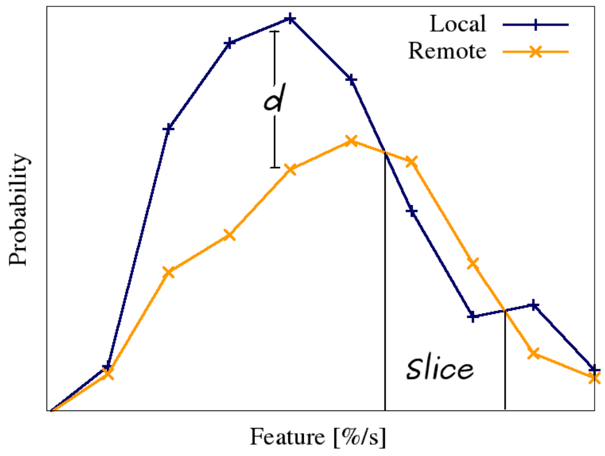


Figure 4.7: Comparison of local and remote distributions

desired as shown in Figure 4.7. The key insight is that the higher the value of $|d|$, the better the properties to achieve higher improvement. Dimensions are created by organizing all the features $\hat{f}$ of most frequent d into groups. Each group is organized based on a component granularity level, e.g. device (which), app (what), communication (when), server (where), etc. EMCO does not suggest the best configuration for an app, but the most common one, which means the one with higher frequency of d. Additionally, EMCO also extracts various alternatives for surrogates based on the levels of frequency of d.

The overall EMCO analysis is described in Algorithm 1 and consists of $\approx$ 1500 LOC written in Scala. In order to avoid overloading the surrogate with extra processing, the analysis is outsourced to the monitor server, which collects performance metrics.

4.2 Evaluation and Validation

To evaluate our EMCO framework, we used the smartphones, the technical specifications of which are detailed in Table 4.1. We located the server side of our system in Amazon cloud (Ireland region / eu-west-1). As cloud servers, we selected general purpose instances (m1.small, m1.medium, m1.large, m3.medium, m3.large, m3.xlarge, m3.2xlarge) and an optimized memory instance (m2.4xlarge). Additionally, we used a nearby-computer (Intel Core i3, 2.3GHz, 4 GB of memory) in a cloudlet fashion. To measure the energy consumed by the mobile in our offloading experiments, we relied on the Monsoon appliance, namely Mobile Device Power Monitor.

Two different kind of experiments were conducted, oriented to measure how our framework enhances mobile application performance, energy saving of the

Algorithm 1 Evidence analysis

```
Require: : path to the traces
 1: for each app in traces do
 2:     Create samples.
 3:     Specify features to compare
 4:         <$fs$>
 5:   for each feature in $fs$ do
 6:       Compute rate distribution using samples.
 7:           <$R_{local}$, $R_{remote}$>
 8:       Compare $R_{local}$ with $R_{remote}$
 9:           Remove slices that overlap
10:           Update samples
11:   end for
12:     Calculate $d$ using filtered samples.
13:     Create code offload descriptor
14:         EMCO policy creates the descriptor based on higher frequency of
          $d$
15:     Push descriptor to subscribed smartphone app
16: end for
```

device, and multi-tenancy provisioning. As use case in the experiments, we developed a chess game[38] with an artificial intelligence agent that challenges the mobile user based on multiple levels of difficulty. The game implements a minimax algorithm that enumerates all the possible moves to get the best moves that both the user and agent can perform in next n steps, wherein n can be configured. The algorithm determines the most effective move to be done by the agent in order to counter the users' move at playtime. We proceed to describe the experiments along with the findings as follows.

4.2.1 Mobile Performance and Energy Saving

Setup and methodology: The goal of this experiment is to demonstrate the gains in responsiveness that are obtained by using EMCO in comparison with other offloading architectures. We conducted the experiments in real scenarios and avoided the use of controlled configurations that lead to expected results. Since EMCO encourages an offloading strategy at method level, to have a fair comparison, we developed a code offloading framework based on Java annotations[39], which is similar to MAUI or ThinkAir. Firstly, the chess game was configured to offload with the annotation framework. The minimax algorithm was annotated as offloading candidate as it contains the most intensive processing task of the game. The server side used an Amazon cloud instance of

[38]https://github.com/huberflores/CodeOffloadingChess
[39]https://github.com/huberflores/CodeOffloadingAnnotations

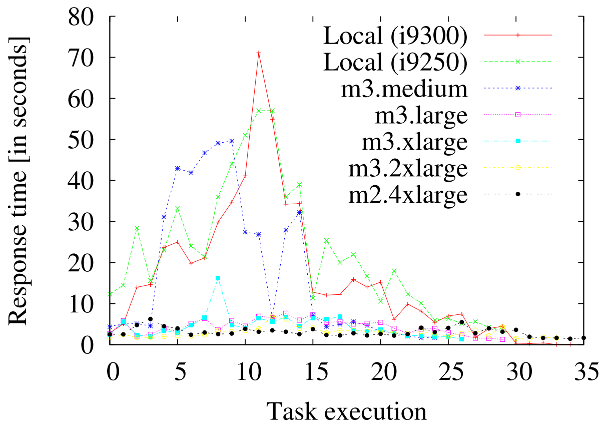


Figure 4.8: Average response time of the chess game application using different servers

type m1.medium. This particular kind of server was selected as it is extensively used in the evaluation of other offloading frameworks [13, 38].

Secondly, the chess game is configured to offload using EMCO. Since the framework is potentiated by crowdsourcing, it was necessary to collect offloading traces from the chess game. As a result, the server side consists of five Amazon instances running the EMCO server (m3.medium, m3.large, m3.xlarge, m3.2xlarge and m2.4xlarge). Cloud assistance is potentiated by traces collected from playing the chess game 240 times, which means that the game was played 48 times per server. The game was played with a configuration n=*4* as in that particular configuration the game becomes more of a challenge for the mobile user. Moreover, the need to offload to cloud is more evident.

Experimental results: We believe that increasing responsiveness is not an additional benefit of cloud offloading, but it is a mandatory requirement, which in the worst case should be fulfilled by maintaining the mobile application with similar response time as the one achieved by its local execution counterpart. Figure 4.8 shows the average response time during playtime (from 48 games per server) that our chess application obtains by offloading to cloud using different servers. The figure also shows the average response time of running the application in the mobile (i9300). From the results, two important observations can be done: first, offloading is associated with multiple levels of enhancement for the mobile; and second, the device nor the architecture is aware of the right matching to bind the mobile and cloud resources. Moreover, we can also observe that code is offloaded to cloud when it should not be offloaded.

To have a clear understanding about the different acceleration rates that can be achieved by using different servers, we used a static request of the minimax algorithm. Figure 4.9 shows the results. The input of the algorithm

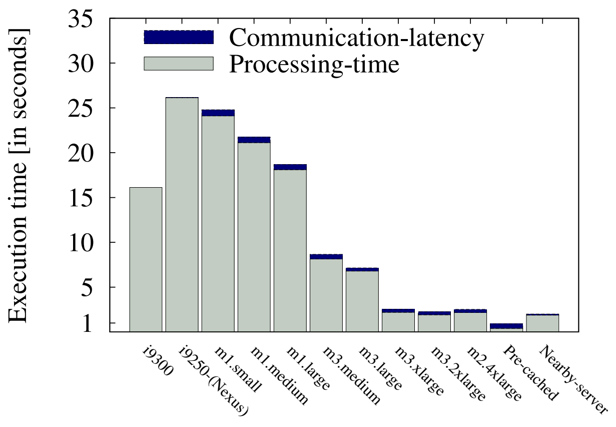


Figure 4.9: Acceleration rates of a minimax algorithm in multiple cases of execution (local, remote and pre-cached)

Units, device, and mobile platform	CPU	RAM (MB)
1, Samsung Galaxy S3, Android	Quad-core 1.4 GHz	1024
1, Google Nexus, Android	Dual-core 1.2 GHz	1024

Table 4.1: Technical specification of the mobiles to evaluate EMCO

was fixed to a specific state of the chessboard. In that particular state, the execution of the algorithm in the local resources of the mobile is $\approx$ 16 seconds. Despite saving energy, our experiments reveal that servers such as m1.medium and m1.large are not suitable as surrogates, as they slow down the performance of high capabilities smartphones. This is a critical issue that compromises user's satisfaction. We can also observe from the diagram that a response time higher than offloading to nearby servers can be achieved by relying on higher capabilities servers such as m3.2xlarge and m2.xlarge. This suggests that while the latency in communication cannot be controlled, the total time of the invocation can decrease by adjusting the tradeoff between utilization price and computational capabilities of the server.

Naturally, the utilization of powerful capability servers is associated with an expensive cost. Thus, we consider most of the offloading proposals in the field as infeasible in practice as they foster a highly coupled architecture (one mobile, one server), which represents to the mobile user, paying for active resources in the cloud that are utilized at minimum. Certainly, a server can be paused and resumed in the cloud on demand, but the addition of these features within the architecture has a direct impact on the mobile applications, for example additional time to resume the server can decrease responsiveness in the mobile, the entire architecture has to be reconfigured when a server is resumed, specialized Web APIs need to be implemented in the mobile, which in most of the cases cannot be deployed due to limitations of the mobile platform [10].

To address this problem our proposed framework provides two solutions. One the use of reusable results (pre-cached functionality) and the other the optimization of the surrogate to handle multiple users. This last point is explained in detail in the scalability subsection. Regarding the first solution, as explained before, a reusable result is identified in the analysis of the community traces, and it is associated to a request that is commonly requested by the crowd. The result of that request is stored at the cloud, and it is used to respond duplicated offloading requests. The percentage in which the response time is enhanced with reusable results is inversely proportional with the time that it takes to execute the code. In other words, the longer the processing, the less time required to answer the request. Figure 4.9 compares the response time of a request from the chess game, which is processed remotely and pre-cached in the cloud. Since the processing time of the code invocation is avoided, the offloading request is accelerated beyond normal execution. Arguably, the offloading result could be stored temporally in the cache of the device. However, the limited space of the cache in the device is unsuitable for a long term re-use. Clearly, the cache can be increased, but a cache that is too large can cause out of memory exceptions and leave the mobile application with little memory to work. In order to mitigate the cost of running continuously higher capability servers, after pre-cache results are stored, those can be transfered to lower capability servers to respond duplicated offloading requests, for instance, pre-cached results processed in m3.2xlarge can be transfered to m1.small, so that m1.small can respond at similar rates of m3.2xlarge. We conduct multiple experiments to determine the response time of a reusable result using different instances types. Figure 4.10 shows the results. We can observe that the response time of a reusable result remains almost the same when using different instances types from remote cloud. We can also observe that by using pre-cached functionality in cloudlets, the response time of the offloading request obtains an extra acceleration.

Regarding mobile application performance, cloud assistance is potentiated by traces collected from 240 executions of the chess game. Once it's available, we proceed to compared EMCO and the annotation framework. Figure 4.11 shows the results of offloading to m3.medium from the chess application during a single game using EMCO and the annotation framework. Clearly, we can see that EMCO uses the information from the characterization process to determine the best mobile cloud matching. Additionally, EMCO accelerates the offloading process by pre-caching mobile application functionality. In the diagram, requests 10, 15 and 19 used the pre-caching mechanism. Finally, we

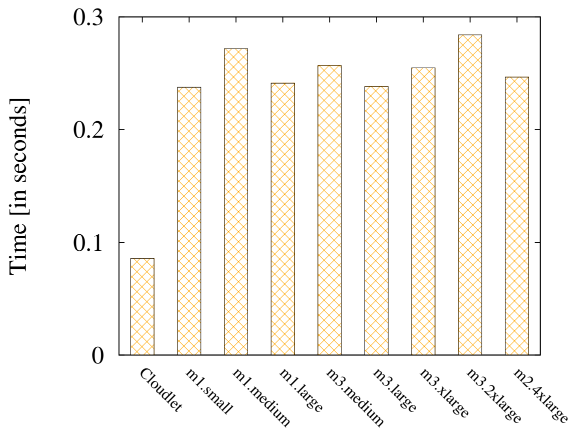


Figure 4.10: Response time of pre-cache results in different instances types

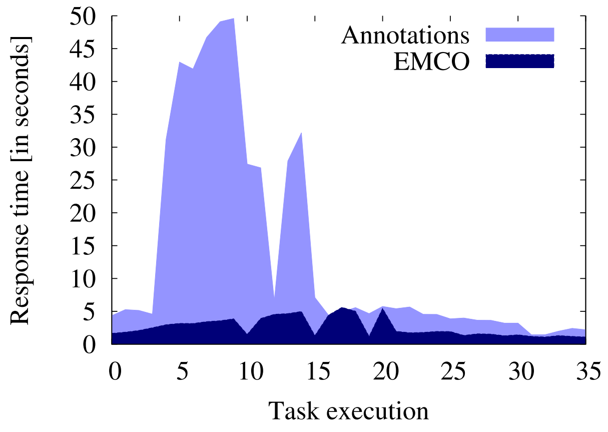


Figure 4.11: Comparison of EMCO with other frameworks, in terms of mobile application performance

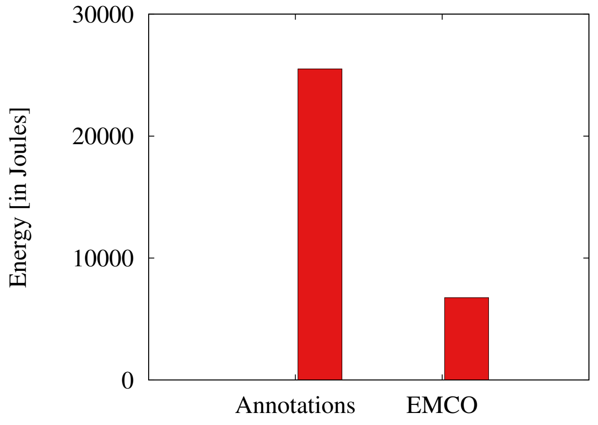


Figure 4.12: Comparison of EMCO with other frameworks, in terms of energy saving

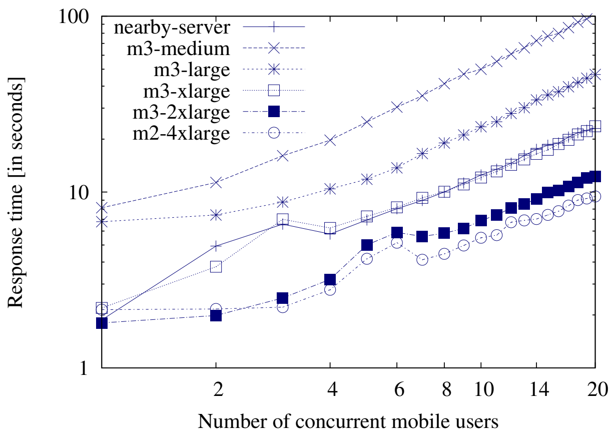


Figure 4.13: EMCO ability for multi-tenancy

demonstrate the gains in energy obtained by using EMCO. Figure 4.12 compares the energy wasted by the mobile when using EMCO and the annotation framework. EMCO shows to be more effective to save energy for the devices as it saves three times more energy in comparison with traditional offloading architectures. EMCO decision engine also shows to be lighter in comparison with other mechanisms like the one used by MAUI. We compare MAUI linear programming (LP) mechanism with our decision engine using the same amount of parameters, the results show that EMCO consumes $\approx 12\%$ less energy. The processing steps required to take a decision using LP depends on the number of parameters introduced in the engine.

4.2.2 Scalability of the Framework

Setup and methodology: In this experiment, we show how the cloud-based surrogate scales to face heavy loads of incoming requests. It is important to mention that most of the offloading frameworks are unable to scale for multi-tenancy, and those that claim to provide scalable features ignore key cloud aspects, such as horizontal/vertical scaling, load balancing, handling concurrent users, etc. The ability of EMCO to scale on demand provides insights about the inquiry concerned to *what will happen in a scenario that involves a large scale of devices offloading to cloud.*

To generate different loads of users, which are concurrently offloading to cloud, we developed an offloading simulator[40] that based on an input parameter n creates n concurrent threads (each thread depicts a user) that offload a minimax algorithm to cloud. We benchmark all the types of surrogates described at the beginning of the section using the same fixed input used in Figure 4.9. We left out the servers in which with a single request, the response time degraded in comparison with local execution (m1.small, m1.medium, m1.large).

Additionally, we analyzed our load balancing mechanism based on push notifications. Since our push mechanism is a key component in our framework, we conduct an experiment to measure its delivery rate. The aim of the experiments is to determine the latency between submitting a request for sending a message and the target device receiving the message. The message used in the notification was fixed to a size of 254 bytes, which is the minimum amount of data required to reconfigure the surrogate specified in the descriptor of the device. Messages are sent 1 per second for 15 seconds in sequence, then with a 30 minute sleep time, later followed by another set of 15 messages, repeating the procedure for 8 hours (240 messages in total). The frequency of the messages is set in this way, in order to mitigate the possibility of being detected as a potential attacker to the cloud vendor (e.g. Denial of Service) and to refresh the notification service from a single requester and possible undelivered data. Moreover, the duration of the experiments guarantee to have an overview of the service under different mobile loads, which may arise during different hours of the day.

Experimental results: The capabilities of a system for handling multi-tenancy are crucial in an environment that follows a utility model. EMCO is built to exploit the distributed and vast variability of the cloud ecosystem. EMCO handles an offloading request in an available dalvik process, which is started as listener when the server is activated. Dalvik processes are listening at the server to avoid the time that takes to push an apk file into the dalvik machine, which is in average ≈ 100 milliseconds. Figure 4.13 shows the number of concurrent users that a single instance from different types can handle. From the figure, we can observe that requests can be handled simultaneously, and the number of concurrent requests is related with the available resources

[40]https://github.com/huberflores/BenchmarkingAndroidx86

of the server. The response time of a single request augments as the load in the server increases. Four different surrogates (nearby-server, m3.xlarge, m3.2xlarge, m2.4xlarge) can handle up to 20 concurrently, below the average response time required by the mobile to execute the code once. The rest of the servers in the testbed (m3.medium, m3.large) can also handle the same amount of users, but just half of them can be handled under the requirements of acceleration and energy saving.

To achieve horizontal scaling, as any other cloud application, a load balancer is required. EMCO uses a notification server as load balancer. The notifications allow the mobile to reconfigure the surrogate, in which is subscribed. Notice that the response time depends on the protocol in which the mechanism works and the availability of the server at the cloud provider [33]. Refer to performance of notification services in Chapter 3. Interestingly, the load balancer shows to be more effective to distribute the incoming traffic among the available EMCO servers as the load balancer does not become a bottleneck when the servers are added dynamically. The reason is the hierarchical schema that does not rely on a unique server to distribute the load of incoming requests that stress out the entire system, instead, each EMCO relies on a collectD server to measure its own performance metrics. Based on the CPU load, each EMCO assumes the role of parent to create or destroy a child. When a child is created, a portion of the devices subscribed in the parent are transfered to the child, so that the devices are subscribed to the new server. In contrast, when a clone is destroyed, the subscribed devices are transfered back to the parent, and then the child is removed. In both cases, the devices are re-configured by sending a message via push notification. Removing a server is faster than adding a new one as configuration time to create a new server is higher. It takes in average one minute to have a working child available to offload. The time includes the allocation time of the server in the cloud, which is $\approx$ 40 seconds in Amazon, the transfer time of the data (subscribed devices), and the delivery time of the notification. Finally, it is important to mention that the reason of having an extra server with collectD (aka monitor) is due to the CPU load of server that measures the performance metrics increases in average 30% when the data is collected. Thus, measuring performance is delegated from the parent to the monitor, so that the parent just takes the decision based on the data gathered by the monitor.

4.3 Summary

The potential of computational offloading has contributed towards the flurry of recent research activity known as mobile cloud computing. While computational offloading has been widely considered for saving energy and increasing responsiveness of smartphones, the technique still faces many issues pertaining to real-life usage. This thesis overcomes the challenge to develop an architecture that potentiates the sustainability of the approach in practice. We design, build and validate a working implementation of a framework, namely Evidence-aware Mobile Computational Offloading (EMCO), which introduces the concept of cloud assistance in the architectures. EMCO relies on evidence collected from a community of devices to characterize the execution of code in all possible contexts. The aim of the characterization is to determine the conditions and configurations for offloading smartphone applications. Additionally, the characterization is also utilized to identify reusable results, which can accelerate the offloading process even further. The evaluation of our proposed framework shows to be more effective as it allows the mobile device to save three times more energy and accelerate the mobile applications up to 10x, and up to 30x using pre-cached functionality. Moreover, EMCO exploits better multi-tenancy in *as a service* style.

Chapter 5

Adaptive App Quality-of-Experience as a Service

The proliferation of smartphone apps is on the rise, in particular games. Yet, recent studies show that 80% of mobile apps in stores are installed, used for some time, and then uninstalled from the device[41]. This suggests that the competitive success of an app does not only depend on implementing interesting functionality, but also in engaging the mobile user with high app Quality-of-Experience (QoE) [55]. The QoE of an app is measured by combining user's satisfaction, expectation and perception about the app's functionality with technical and non-technical parameters that influence the runtime execution of the app [74]. Technical parameters include network communication, processing capabilities, allocated resources, etc., and non-technical parameters include Quality-of-Service (QoS) provisioning, service level agreement, etc.

However, the expectation of the user towards facilitating the mobile device with PC-like functionality is growing everyday. Mobile devices are constrained by its processing capabilities and energetic resources. This limits the acceptable perceptibility and interactivity of the mobile apps. As a consequence, quality of the results presented to the user —*fidelity* [5]— can vary abruptly, which compromises QoE.

QoE in client-server apps can be improved by tuning network parameters. Decreasing high communication latency between a mobile and remote server is one of the most challenging problems for a cellular operator or service provider as reducing latency requires to tune the network to prioritize specific data transfer, e.g. video streaming, voice over IP (VoIP), etc. Tuning involves to estimate QoE metrics that depict the objective behavior of a smartphone app, e.g. amount of video stalls in a streaming on demand app. Many parameters can affect data transfer in the network, for example physical environment,

[41]http://goo.gl/K2t0n4

complexity in the communication protocol, etc. Thus, measuring QoE metrics is a difficult task due to the poor understanding about the relationship between the network parameters and QoE metrics that influence the performance of an app [47, 74].

Unlike client-server apps, increasing QoE of stand alone apps is complex as it is limited by the mobile resources. However, as presented in Chapter 4, mobile applications can be instrumented with computational offloading mechanisms in order to augment the pool of computational resources [15]. Naturally, the offloading process of smartphone apps that are instrumented with computational offloading mechanisms can be improved when the network is tuned as any other client-server app that uses external service support. However, in addition to this, apps that implement computational offloading can take advantage of the vast utility features of the cloud to manage the acceleration of code execution. In other words, the total time of the code invocation in the cloud can be decreased by adjusting the tradeoff between utilization price and computational capabilities of the surrogate server, which enables the mobile application to enhance performance at multiple levels based on user's preference.

Besides the basic benefits provided from computational offloading, such as saving energy of the device and improving response time of the smartphone apps, in this chapter, we investigate how to use computational offloading to adapt dynamically the fidelity of smartphone apps to meet QoE requirements of the user. We envisioned that as part of an app released in a store, a counterpart computational service in a cloud also is released by the same party that developed the app to improve the QoE of the users that use the app.

5.1 Adapting App QoE: Design goals and Architecture

Despite the multiple variables that influence users QoE, previous works have identified *engagement* as one of its key factors [48]. The basic idea is that the more satisfied the user is with an app, the higher the frequency and the longer the sessions in which an app is used. A session is formed by the events of starting, using and closing an app. In our system, app QoE requirements are determined based on the well known metrics, session length and abandonment, which are extrapolated from Web QoE apps [75]. Figure 5.1 illustrates the metrics. Session length represents the time that the users spends using an app. The longer is the session, the more engaged the user is with an app and vice versa. Abandonment depicts the progressive loss of interest in the user

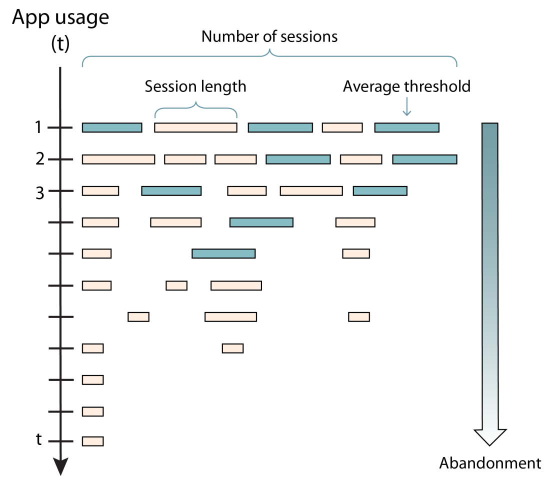


Figure 5.1: Session length and abandonment of an app

that is measured by the frequency of app usage, which is the number of sessions of an app in a specific period of time.

In this context, when user's disengagement with the app is detected, e.g. session length or frequency usage diminishes, the app responsiveness is increased gradually one step higher depending on its last execution. Since a QoE client stores detail information about code execution of the app, e.g. type of instance, time of execution, etc., it is possible to determine the processing capabilities that are needed to accelerate the execution of code further in comparison with previous cases. For instance, a request that was last time executed by a m3.medium of 2.5 Ghz, it can accelerate its execution of code if the request is processed by a higher capability instance like m3.large of 2.6 Ghz.

It is expected that accelerating the response time of the app will lead to better QoE for the user, which in return will cause to increase the productive life of the app in the market. Undoubtedly, the execution of an app cannot be accelerated all the time as there is a limit for accelerating the code. Moreover, code acceleration using cloud resources requires permanent network connectivity with low latency round trip times (RTT). Thus, once the user is engaged with the app again, the QoE is gradually decreased. In the best case, app QoE can be decreased as much as the one provided only by the processing capabilities of the mobile.

5.1.1 System Overview

The proposed system extends our computational offloading framework presented in Chapter 4. Figure 5.2 shows the overview of the system, which consists of a back-end, front-end and QoE client. Each part of the system is described as follows:

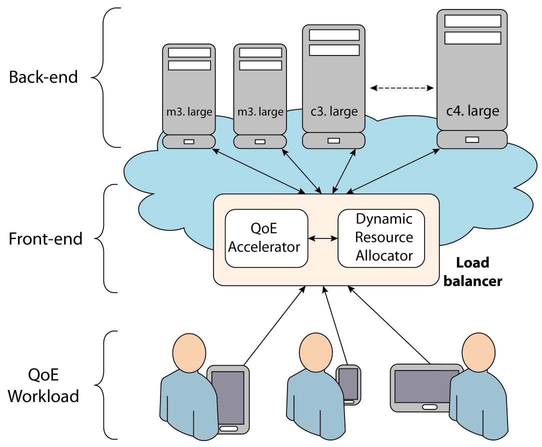


Figure 5.2: Overview of the system

1. ***Back-end*** — contains the surrogates, which are equipped with the same runtime environment as the mobile platform, so that a surrogate can execute the code offloaded from a smartphone app. Each surrogate is an instance, which in cloud terms provisions a computational service. The back-end delivers computational provisioning by a group of instances, where each instance is from a certain type. Each instance type provides different code acceleration that depends on the computational capabilities of the instance. The group can scale up or down, which means that instances can be added or dynamically removed on demand.

2. ***Front-end*** — is the entry point of the code offloading requests. A front-end provides a load balancer that receives the workload of computational offloading requests sent from the active users using an app. The load balancer contains two components, the QoE-Accelerator and the Dynamic-Resource-Allocator. The *QoE-Accelerator* is aware about the number of instances in the back-end and the type of each instance. It assigns to each type an acceleration class, which is used to differentiate types of code acceleration. The QoE-accelerator uses the processor speed of the instance as main variable to define code acceleration. Once a request reaches the load balancer, the *QoE-Accelerator* forwards each request to the right instance based on acceleration requirements in QoE of each user. These requirements are defined by the mobile app and they are sent along with the request. Finally, the *Dynamic-Resource-Allocator* determines the amount of instances that minimize provisioning costs in the cloud while fulfilling the QoE requirements of every user, which is explained in detail in subsection 5.2.

3. ***QoE client*** — extends the instrumentation of code offloading with a mechanism that enables the app to capture the time of code execution

at method level into a database. Along with the time of code execution, the client also stores where the code is executed, e.g. local or remote. Additionally, if the execution of code is remote, the client also stores the type of the instance that executed the method. The main idea is that when an app offloads to the cloud, it can attach to the request the stored information about the last execution of the method. An app may require or not to increase QoE. In general, if the app does not want to increase QoE, the information of last execution of the method is used to guarantee that the request will be executed once again by the same instance to achieve similar response time as last execution. If the app requires to increase QoE, the information of last execution turns into thresholds that need to be surpassed to achieve better app responsiveness. Inspired by [26, 76], our QoE client uses linear regression to estimate abandonment. Finally, many QoE clients offload to cloud as a QoE workload.

5.2 Dynamic Computational Provisioning

Since the allocation of computational resources in the cloud has a provisioning cost, it is important that the allocated resources are efficiently utilized to avoid over provisioning that causes higher costs. At the same time, it has to ensure that the allocated resources can cope with the service demand of the active users using the system. As a result, we develop an approach, the objective of which is to determine the effective amount of resources to be allocated depending on a QoE workload, while minimizing a provisioning budget.

The dynamic amount of allocated resources is estimated using *Linear Programming (LP)* [77]. LP is a technique the aim of which is to determine the optimal solution for a given problem. A LP model consists of a linear objective function that needs to be optimized and is subject to a set of constraints expressed as equalities or inequalities. The constraints define a finite space of solutions, known as feasible region, where the objective function can be evaluated. A LP algorithm finds a point inside the feasible region where the objective function is optimal, whether maximum or minimum, and still the constraints are satisfied. The objective function and constraints are composed by variables and parameters. The parameters are known values that are set before the execution of the LP algorithm. The algorithm's aim is to maximize or minimize the objective function by assigning values to the variables of the problem.

Let t represents the type of an instance. The back-end of the system consists of n instances. The parameters of the model also include:

- B, the maximum budget constraint.
- c_t, cost of an instance of type t.
- A_t, class that defines the speed up in code execution that can be achieved by using the instance of type t. Each instance type delivers an acceleration based on processor speed.
- NA_t, number of active instances of class A_t that are in the back-end. Notice that the total number of instances n is equal to $\sum_{i=1}^{k} NA_{ti}$, where k is the number of acceleration classes.
- CA_t, capacity of the instance of type t to handle code offloading requests per minute.
- CC, number of instances that the cloud is capable to launch for an account. Generally, private clouds are able to manage only a limited number of instances at a time. Similarly, public clouds such Amazon AWS can launch at most 20 CA_t instances on demand, if more than 20 instances needs to be launched the customer has to fill a form requesting the extra resources.
- W, the value of the current QoE workload in the system in requests per minute. W can be expressed in terms of sub workloads WA_t, which represent the workload for a particular class A_t. Thus, $W = \sum_{i=1}^{k} WA_{ti}$, where k is the number of acceleration classes.

The variables correspond to the number of instances to be allocated.

- x_t, the number of instances of type t to be allocated in the back-end. Generally, front-end is provided by the cloud vendor as an entry point for the cloud application and in some cases without a cost, e.g. Amazon Autoscale[42]. We assume the front-end is provided without a cost.

The model tries to minimize the cost of having x number of instances of type t running with a cost c_t. The object function is defined as a sum across all the instances types $t \in T$.

$$Min \sum_{i=1}^{n} x_{ti} * c_{ti} \tag{5.1}$$

Finally, the model comprises the following constraints:

[42]http://aws.amazon.com/autoscaling/

- The workload constraint $\forall$ the acceleration classes A_t.

$$\sum_{i=1}^{k} NA_{ti} * CA_{ti} > WA_{ti} \tag{5.2}$$

- Cloud's on demand constraint:

$$\sum_{i=1}^{n} x_{ti} < CC \tag{5.3}$$

The workload constraints states that the sum of all the capacity CA_t across all the instances from NA_t must be enough to satisfy the workload WA_t of a class A_t. We must remember that a mobile app offloads a task to cloud to increase QoE, in terms of responsiveness. As a result, each offloading request defines a goal in response time, which is the sum of the times for data transfer and code execution. Data transfer time is fixed by network latency and code execution depends on processing capabilities of the instance as explained before. However, the code execution time or service time of the instance also depends on the CPU utilization. If the CPU utilization is higher, the service time of the request will also be higher [30]. CPU utilization increases as the number of active users offloading to an instance also increase. Thus, the capacity of an instance depends on its service time.

Unlike other types of requests that can be characterized [29], the service time of a code offloading request can vary abruptly as the code has a non-deterministic behavior. Thus, the capacity CA_t is periodically estimated depending on the number of successful requests that are provisioned by the instance. A request is successful when the actual service time of the instance is lower than or equal to the goal service time required by the app. By analyzing the logs of the instance each previous hour, it is possible to determine the amount of requests that were successfully processed by the instance. Based on these observations, the capacity CA_t is determined for the next hour, in terms of requests per minute. Initially, when there is not log data to calculate capacity, a CPU utilization threshold of 80% is used instead. Many works have found this threshold to be the optimal for not over loading a cloud-based system [78].

The budget constraint states that the sum of all the cost c across all the instances types t must be under the specified budget B. The cloud's on demand constraint states that the number of running instances of the type t must be less than or equal to the allocation capacity CC of the cloud. In other words, the number of instances that need to be run must be under budget and not exceed the cloud's capacity, if any, and still satisfy the workload in the system.

5.3 Implementation of the System

This section describes the implementation details of each component of the system. We developed our system based on Android. The rest of the section describes the technical details of each component[43].

5.3.1 Back-end

Each surrogate of the back-end is a customized Dalvik-x86. Dalvik is the virtual machine of Android to execute dalvik bytecode. The implementation and deployment of this component is already described in detail in Chapter 4.

5.3.2 Front-end

The front-end is an instance that distributes the incoming QoE workload among the back-end surrogates. The front-end contains a back-end descriptor in JSON format (Snippet 3), which contains per each server, the information about where the *apk* files are located and the available ports for code offloading. The descriptor is updated when a server is added or removed.

Snippet 3 Back-end descriptor in JSON format

```
{
    "Chess_app": {
        "Surrogate: 54.73.45.xxx": {
            "ports": [
                "6001",
                "6002",
                ...
                "600N"
            ],
            "location": [
                "/home/ubuntu/android-x86/"
            ]
        }... others
    },
    "Nqueens_app": {
        "Surrogate: 172.16.32.xxx": {
            "ports": [
                "5001",
                ...
                "500N"
            ],
            "location": [
                "/home/ubuntu/dalvik-86/"
            ]
        }... others
    }
}
```

[43]https://github.com/huberflores/ScalingMobileCodeOffloading

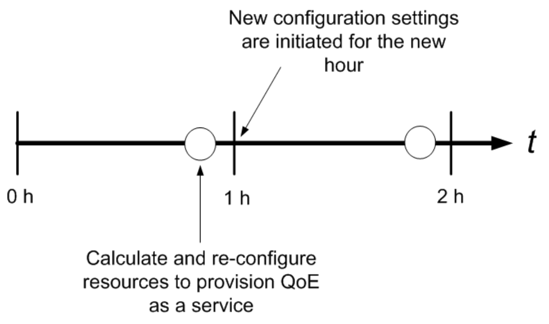

New configuration settings are initiated for the new hour
t
0 h
1 h
2 h
Calculate and re-configure resources to provision QoE as a service

Figure 5.3: Overview of the dynamic allocation resources for computational provisioning

Based on the QoE workload, the load balancer dynamically allocates the necessary instances to handle the workload. Since cloud vendors charge the cost of an instance per hour, e.g. Amazon, allocation of instances happen at the end of each hour as shown in Figure 5.3. The load balancer executes Algorithm 2 10 minutes before the end of each hour in order to determine next set of instances. This amount of time is enough to execute the algorithm and apply the new configuration settings. Once the new configuration is obtained for the next hour, the new configuration settings is compared with the current one. The goal is to find at least one instance in common, which won't be terminated during the transition and will continue provisioning the computational service for the next hour as well. The aim of this is to avoid termination, launch and configuration time in the system when in re-configured. Moreover, this also ensures the availability of the service during the transition. In the case that there are not instances in common between the new and current settings, then the 10 minutes is an average time, which is enough to terminate the overall configuration and start a new one. Our load balancer uses *jclouds*[44] library to manage the cloud infrastructure. The library has support for Amazon cloud and many other cloud vendors such as Azure, Rackspace, etc. Finally, LP model is develop using OptimJ[45], which is a Java based modelling language for solving optimization problems. OptimJ implements different LP solvers like GLPK and lpsolve, which are open source.

5.3.3 QoE Client

We develop our system based on Android. Each app that uses our system is instrumented with a code offloading mechanism that uses Java reflection [21], which allows to capture the details of code invocation during runtime, e.g.

[44]https://jclouds.apache.org/
[45]www.ateji.com/optimj/

Algorithm 2 Resource allocation of surrogates in the back-end

```
Require: : Provisioning time <$t$>, parameters of the LP model <parameters>
 1: if ((t + 10) = End of the provisioning cycle) then
 2:     Get log files from each surrogate in the back-end.
 3:     Calculate capacity of each instance type.
 4:     Calculate newConfiguration: new instances to be allocated.
 5:         newConfiguration = LPModel(parameters).
 6:     Get currentConfiguration: existent instances in the back-end.
 7:     Calculate reuseInstancesGroup = Compare new configuration settings
      with current one.
 8:   for each newInstance in newConfiguration do
 9:     for each currentInstance in currentConfiguration do
10:       if (newInstance equal to currentInstance) and newInstance is not
          marked for reuse then
11:           Add currentInstance to reuseInstancesGroup.
12:           Mark currentInstance for reuse.
13:           Continue.
14:       end if
15:     end for
16:   end for
17:   for each newInstance in newConfiguration do
18:     if newInstance is found in reuseInstancesGroup then
19:         Remove newInstance from reuseInstancesGroup
20:         Continue.
21:     else
22:         Launch newInstance.
23:     end if
24:   end for
25:   for each currentInstance in currentConfiguration do
26:     if currentInstance is not marked for reuse then
27:         Terminate currentInstance.
28:     end if
29:   end for
30: end if
```

Units, device, and mobile platform	CPU	RAM (MB)
1, Samsung Galaxy S3, Android	Quad-core 1.4 GHz	1024
1, Google Nexus, Android	Dual-core 1.2 GHz	1024

Table 5.1: Technical specification of the mobiles to evaluate QoE adaptation

name of the method, parameters, etc. Our code offloading mechanism transforms each method of the app into two new methods definitions, one local and one remote. The QoE client wraps these methods to capture their execution time, which is stored into a SQLite database. Execution details of the code are encapsulated into a code offloading request. The request also contains the requirements in QoE of the app, if any. At the front-end, the load balancer reads the request and forwards it to the correspondent instance. At the surrogate, the request is reconstructed based on its runtime details, so that code can be executed. Once executed, the response is sent back to the mobile app, which synchronizes the result with its execution flow.

To generate different loads of users that offload code to cloud, we developed a simulator that based on an input parameter n creates n threads that offload a specific portion of code defined by the developer, each thread depicts a user. The simulator also is implemented using Java reflection. The simulator receives as parameters, the inter arrival time that is the time between creating one request and the next request, and the experimental time that depicts the active time of the load of requests.

5.4 Evaluation and Analysis

To evaluate our system, we used the smartphones, the technical specifications of which are detailed in Table 5.1. We located the server side in Amazon cloud (Ireland region / eu-west-1). As cloud servers, we selected general purpose instances (m1.small, m1.medium, m1.large, m3.-medium, m3.large, m3.xlarge, m3.2xlarge) and an optimized memory instance (m2.4xlarge). We used these instances as they can be launched on demand indefinitely. Other higher capabilities instances, e.g. c3.8xlarge, require explicit request to the provider to be launched multiple times by a single account. In this section, we describe the experiments conducted, oriented to measure the performance, and scalability of our system.

5.4.1 Performance

Setup and methodology: The goal of the experiment is to demonstrate how app QoE can be adapted to multiple levels of response time based on cloud utility computing. We developed three different apps, which are instrumented with a code offloading mechanism [15] and our QoE client. We assume in the experiments that the smartphones have available network connectivity to offload. Apps are described as follow.

Chess app — implements an artificial intelligence (AI) agent that challenges the mobile user based on multiple levels of difficulty. The agent uses a *minimax* algorithm that analyzes how the chess pieces are located on the board, so that it can enumerate all the possible moves and determine the best move that the agent can take to counter the last move of the user. The portion of code that corresponds to the minimax logic is the one offloaded. This application is also studied in detail in Chapter 4.

Backtracking app — overlaps a 3D model on the surface of the mobile camera to augment the reality of the ambient. The app is equipped with ten models in format *3ds*, where the size *s* of each model is the range of *500KB<s<2000KB*. Since each model requires significant amount of processing to be loaded, each model is loaded individually as the user requires. The portion of code that corresponds to the model loader is the one offloaded.

Mission game app — is a mobile game developed using AndEngine[46]. The game consists of using a 2D character to defeat 2D enemies which appear randomly to attack it. The game implements five stages. During playtime, the game collects information about the number of enemies along with their positions and life time, so that at the beginning of each stage, this information can be used to improve the AI strategy of the game. The portion of code that corresponds to design the strategy is the one offloaded.

Results, experiences and lessons learned: Our system introduces an extra front-end component in the architecture. First, we explore how this new component influences the response time of a computational offloading request. As a result, timestamps are taken across the system as the request is processed in each of its components. Figure 5.4 models the response time $T_{response}$ of a computational request. The response time considers the time that takes to connect from the mobile to the front-end T_{m-f}, the time that takes to be routed from the front-end to a particular surrogate T_{f-b}, the execution time of the code in the surrogate T_{cloud}, the times that takes to send the result from the surrogate to the front-end T_{b-f}, and finally, the time that takes to send the result back from the front-end to the mobile T_{f-m}. We assume that $T_{m-f} = T_{f-m}$ and $T_{f-b} = T_{b-f}$ are equal as the same communication channel remains open both ways until the operation finishes. In this context, we

[46]http://www.andengine.org

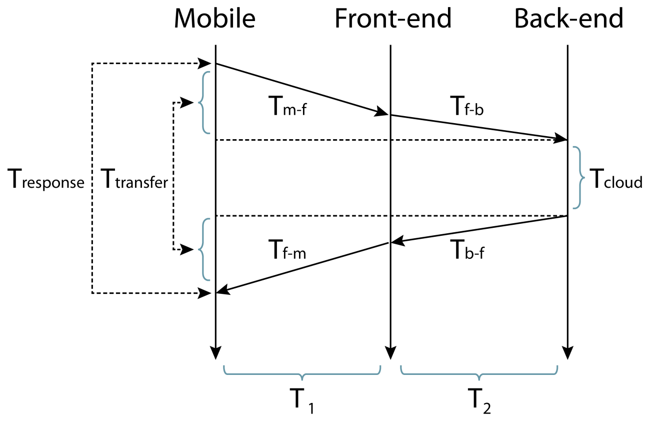


Figure 5.4: Timestamps taken across the system in each of its components

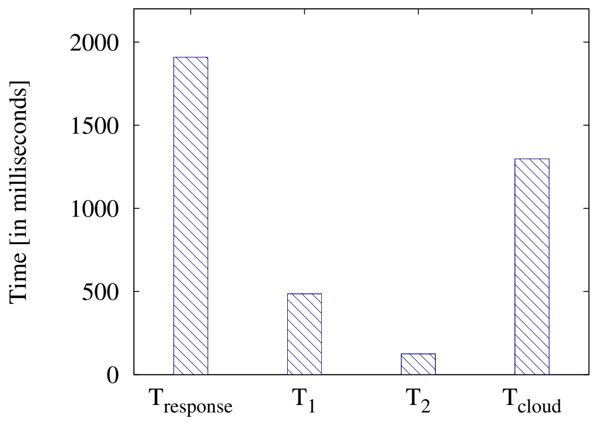


Figure 5.5: Actual times to handle the request in each component

define $T_1 = T_{m-f} + T_{f-m}$ and $T_2 = T_{f-b} + T_{b-f}$. Thus, the response time $T_{response} = T_1 + T_2 + T_{cloud}$.

Figure 5.5 shows the timestamps taken across the system. We can observe that the extra time introduced by the front-end is ≈ 150 milliseconds and the total communication time $T_1 + T_2$ is less than a second. Naturally, higher latency can influence the communication time to be higher and vice versa, which impacts T_1. T_2 is less likely to change drastically as the latency depicts the internal cloud communication, which is wired between servers in the same private network. Finally, the diagram shows that T_{cloud} is the most consuming operation that impacts the total response time. Fortunately, the total time of the code invocation in the cloud can be decreased by adjusting the tradeoff between utilization price and computational capabilities of the surrogate server as depicted in Figure 5.6, which shows the execution of a minimax algorithm in different instance types. The input of the algorithm is fixed to a specific state of the chessboard. In that particular state, the execution of the algorithm in the local resources of the mobile is ≈ 16 seconds for i9250 and ≈ 14 for i9300.

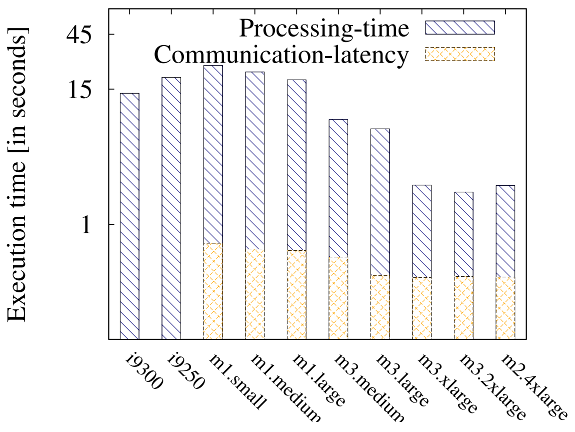


Figure 5.6: Acceleration of code based on instance types.

Next, we proceed to compare the local execution of an app in the mobile with all different cases that emerge from using different instance types as a surrogate. Instances m1.small, m1.medium and m1.large are excluded from this experiment because the response time of the app decreases in comparison with local execution. Increasing responsiveness is not an additional benefit of cloud offloading, but it is a mandatory requirement, which in the worst case should be fulfilled by maintaining the mobile application with similar response time as the one achieved by its local execution counterpart. Figures 4.8, 5.7 and 5.8 show the response time for chess, backtracking and mission game respectively. In general, we can observe that the response time of an app increases as the capabilities of the surrogate become higher. Naturally, the amount of acceleration induced by using different surrogates depends on the task itself. We can observe that while the chess app speeds up its execution up to 14X times, the backtracking app just accelerates up to 3.5X and the mission game app up to 2.5X.

However, these accelerations are enough to change the perception that the user has towards the app. For instance, in the case of the chess app, the response time is changed from *total interaction disruption* (>10 seconds) to *no disruption with noticeable delay* (>1 second). Same occurs with the backtracking app. In better cases, the perception is changed to almost *instant interaction*, like in the case of the mission game. More acceleration gains also can be achieved by using more specialized instances, e.g. c4.8xlarge. Definitively, there is a limit for code acceleration, which depends on how the code is written. Thus, to take the acceleration of code further, code parallelization can be exploited. Code parallelization is not within the scope of this thesis.

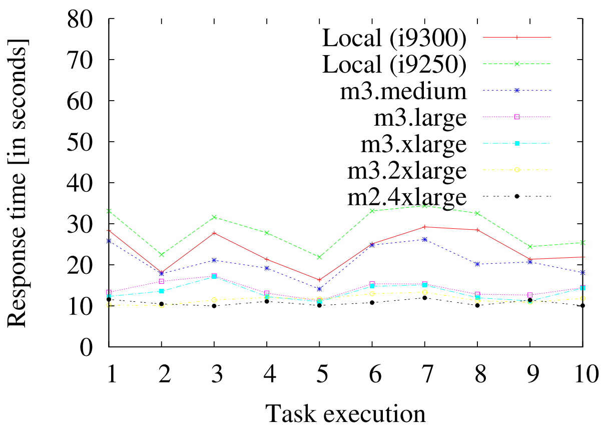


Figure 5.7: Response time of a mission game app.

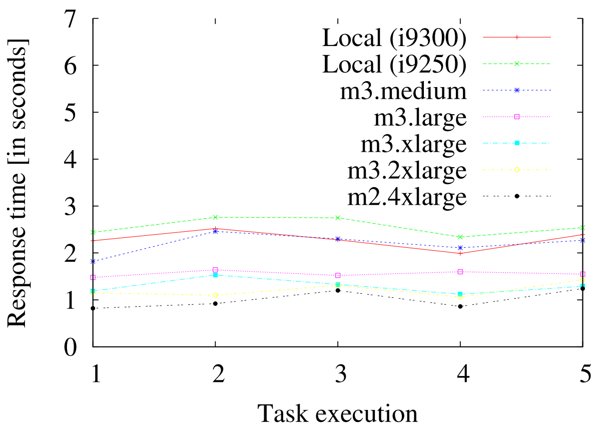


Figure 5.8: Response time of a backtracking app.

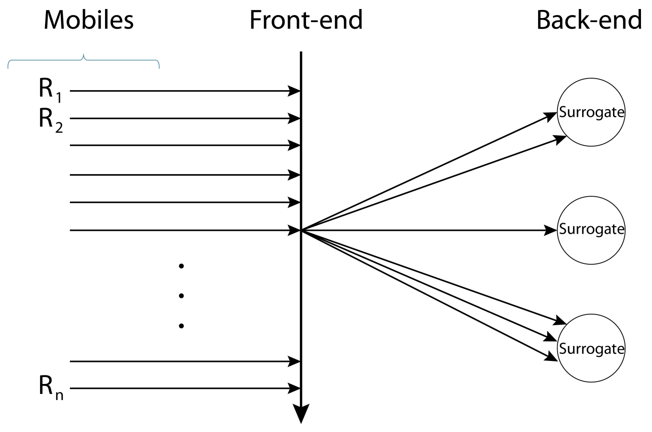


Figure 5.9: Load of incoming computational offloading requests.

5.4.2 Scalability

Setup and methodology: The capabilities of a system for handling multi-tenancy are crucial in an environment that follows a utility computing model. Previous works in computational offloading have proposed one instance per mobile architecture [25], which is unrealistic in practice. The goal of this experiment is to verify the capabilities of our system to handle multiple active users. As a result, we analyze the effect of handling multiple offloading requests in different instances types. We also measure the capacity of the front-end to distribute requests among the surrogates.

Figure 5.9 illustrates the experimental setup, where R_1 to R_n requests reach the front-end in order to be distributed to the back-end based on instance type. To benchmark all the surrogate types, we used our code offloading simulator described in section 5.3. The simulator uses the minimax algorithm to create the load of requests. The input of the algorithm is fixed with the same specific state of the chessboard that is used to create Figure 5.6.

Results, experiences and lessons learned: The capacity of the front-end is measured in terms of processing time of the request, which depicts the time taken by the front-end to decide the surrogate to route the request. Since the front-end receives and distributes the incoming load of requests, it is important to know the maximum capacity of the front-end before it turns into a bottleneck for the system. Usually, the front-end is provided free of charge by the cloud vendor, e.g. Amazon autoscale. In our experimental setup, we assume that the front-end is a m3.xlarge instance. Figure 5.10 shows the number of requests that the front-end can handle when facing a load of 250 requests with an interarrival rate of three seconds. The interarrival rate is the time between creating one and the next request. From the result, we can observe that the processing time to distribute requests remain in average $\approx$150 milliseconds even as the load increases.

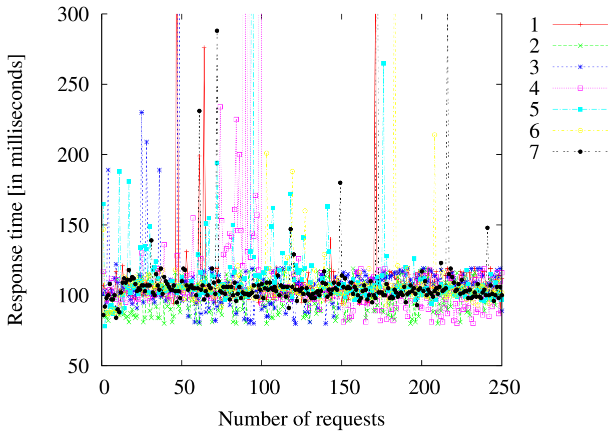


Figure 5.10: Capacity of the front-end to route the requests before turning into a bottleneck.

On the other hand, unlike the front-end, instances in the back-end face higher CPU utilization caused by the invocation of code. Computational offloading requests suffer from non-deterministic execution of code, which means that the processing time of the request can vary abruptly based on different parameters, e.g. input variability in the code. Thus, the requests cannot be easily characterized. We explored in Chapter 4, the capacity of a Dalvik-x86 to handle concurrent requests (Refer to Figure 4.13). By concurrent requests we mean that the interarrival rate is almost insignificant, so that the start time of all the requests is the same. To demonstrate horizontal scaling, we generate requests based on an interarrival rate of three seconds. We scale out gradually the servers from 1 to 7, which means that servers are added on demand dynamically to the back-end. We did not consider more servers as we did not see major improvements in response time as servers were added. We used as surrogates m3. 2xlarge instances, because they are the most powerful servers we can get without requesting explicitly to the cloud provider.

Results of the experiment are show in Figure 5.11. Interestingly, the response time of the requests drops up to 15 seconds and remains like that even when adding more servers. Initially, we thought that the front-end turned into a bottleneck. However, after a close inspection (Refer to Figure 5.10), we realized that the processing time of a request increases as a server receives the requests at different points in time. This means that unlike concurrent requests, the response time of those requests that reach the system in long interarrival times are processed slowly by the server. This suggests that to achieve short response time when handling multiple requests, all the requests that reach the system must be scheduled by the front-end to be processed at once. Computational requests that contain code that can be parallelized can

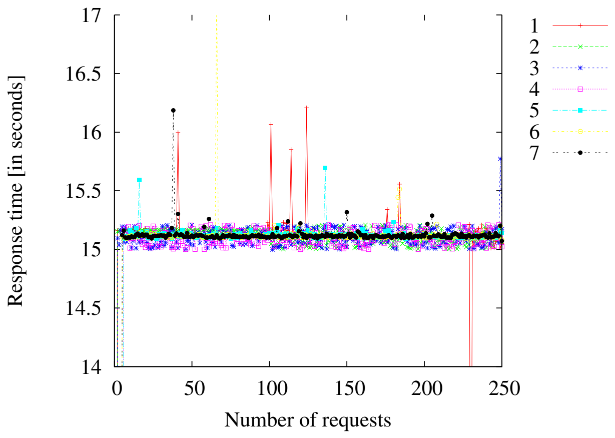


Figure 5.11: Response time of the requests when scaling horizontally, from 1 to 7 surrogates

benefit from this characteristic, so that a request is parallelized within the same server instead of splitting it into multiple servers.

5.5 Discussion

The main limitation of our allocation model is to assume that the provisioning cost of an instance is based on hours. Many other conditions or provisioning models could be adopted, for instance, charge cost based on the number of requests, type of the request, etc. In our system, as explained in Chapter 2, subsection 2.2.3, we included in the allocation model, the most relevant aspects which are critical for the provisioning of back-end surrogates for code offloading. Moreover, we chose this allocation model as it is generic enough to be deployed in any public or private cloud.

5.6 Summary

The proliferation of smartphone apps is on the rise, in particular stand alone apps like games. However, many of these apps do not survive for longer periods in the app stores and the ones that do survive are gradually forgotten by the user. To counter the problem of loss of interest in an app, we investigated how to improve the app QoE in order to foster app engagement. We modeled app usage in terms of session length and abandonment. By detecting when those metrics decreased, our system accelerates the code execution of the app to enhance its fidelity. Since the acceleration is constrained by the mobile resources, we used computational offloading to speed up the execution of code. By dynamically varying the computational capabilities of the surrogate in the cloud, it is possible to speed up the code at multiple scales. In this thesis, we present our experiences about adapting a computational offloading architecture into *as a service* pattern and propose a new system for adaptive app QoE. Moreover, we also introduce the quality-of-service policies to optimize allocation of computational resources. The results have shown that our system can change the perception of the user from *interaction with delay* to *instant interaction.* Naturally, this can only be done when the computational task can take advantage of more and higher processing resources in order to accelerate its execution. The execution of a task depends mainly on how the code is written. The restructuring of code is out of the scope of this work. In this context, our framework can take a task to its maximum limit of acceleration. However, this does not necessarily guarantee a drastic change in perception. Finally, we also found that a surrogate is able to provision code offloading in large scale scenarios.

Chapter 6

Conclusion

In this chapter, we present the conclusion of the thesis. The findings reported in the thesis are summarized by returning to the research questions and recapitulating the work that has been done to answer them.

The main research question (RQ) of this work is *how to bring the cloud infrastructure closer to the smartphone user?* This question was elaborated into six partial questions, where each question focuses on a specific mobile cloud issue, which is investigated in this thesis.

6.1 Research Questions Revisited

Mobile and cloud computing are converging as the promising technologies for the near future. Smartphones are looking towards cloud-aware techniques that allow the device to exploit the massive features of the cloud for its own benefit. Smartphones can benefit from the cloud by following multiple access schemas, e.g., REST, CORBA, RMI, etc. Our study focuses on the following issues.
RQ1. *How to outsource tasks from the mobile using SOA?* **RQ2. *How to outsource a task that requires long waiting without overloading the communication with constant polling?***

In this thesis, we investigated multiple approaches that can be utilized to enhance smartphone apps with cloud power. Since the design and deployment of cloud services mostly follow SOA principles, initially, we investigated how mobile cloud services can be integrated within the mobile applications. Smartphone apps may integrate those functionalities from different cloud levels, so that a mobile task can be outsourced by direct invocation of a service. This process is defined as *task delegation*. However, developing those kinds of mobile cloud applications requires integrating and considering multiple aspects of the clouds, such as resource-intensive processing, programmatically provisioning of resources (Web APIs) and cloud intercommunication. To overcome these issues, we have developed a *Mobile Cloud Middleware* (MCM) framework, which addresses the issues of interoperability across multiple clouds, asynchronous delegation of mobile tasks and dynamic allocation of cloud infrastructure.

Since most of the cloud services require significant time to process the request, which can cause energy drain for the device; it is logical to have asynchronous invocation of the cloud service. Asynchronicity is added to the MCM by adapting the push technologies to manage the synchronization of results between the mobile and cloud. These technologies are provided for each mobile platform, e.g., GCM, APNS, etc. Since we found that these technologies are unreliable in practice and constrained to a specific cloud vendor, we developed and integrated our own notification service, which is created by extending the capabilities of a XMPP-based IM infrastructure.MCM also fosters the integration and orchestration of mobile tasks delegated with minimal data transfer. We defined this process as a *multi-cloud operation based on task delegation*. A prototype of MCM is developed and several applications are demonstrated in different domains. To verify the scalability of MCM, load tests are also performed on the hybrid cloud resources. The detailed performance analysis of the middleware framework shows that MCM improves the quality of service for mobiles and helps in maintaining soft-real time responses for mobile cloud applications.

RQ3. ***Can code offloading be utilized in practice?*** **RQ4.** ***What are the issues that prevent code offloading to yield a positive result?***

While task delegation can enrich the mobile applications with sophisticated functionality, it requires the cloud to be reachable all the time by device. Thus, network connectivity is mandatory in order to activate the functionality of a mobile application. This means that the mobile applications are unable to work in offline mode. Fortunately, other techniques can be utilized to instrument the mobile applications at code level, which allow a mobile application to work in both, online and offline modes. These techniques are known as computational offloading strategies and may be also referred with other names such as cloud offloading and cyber-foraging, among others. Computational offloading is a key technique in augmenting the computational capabilities available for mobile applications with elastic cloud resources. It has been demonstrated that computational offloading can extend battery life of the device and improve the performance of the mobile applications. Lot of works have been proposed in this domain as explained in Chapter 2. However, the sustainability of the technique in practice is an open issue, which has not been explored in detail by most of the proposals in the field. In this thesis, we investigated the issues that prevent the adoption of code offloading, and proposed a framework, namely *Evidence-aware Mobile Computational Offloading* (EMCO), which uses a community of devices to capture all the possible context of code execution as evidence. By analyzing the evidence, EMCO aims to determine the suitable conditions to offload. EMCO models the evidence in terms of distributions rates for both local and remote cases. By comparing those distributions, EMCO infers the right properties, in which a mobile app has to offload. EMCO shows to be more effective in comparison with other computational offloading frameworks explored in the literature. EMCO reduces the counterproductive effect of computational offloading by providing more fine-grained properties encapsulated as dimensions, which provides a wider perspective of the entire system into the decision process. Moreover, our system exploits the surrogate to provide better computational provisioning and fosters the utilization of pre-cached functionality to accelerate even further the response time of code offloading. Since one key factor of the approach is collecting data, it is important to know how much evidence must be collected from a particular app. Enough data that captures all the possible execution cases of an app is gathered when the number of overlapping between the local and remote distributions approximates to zero. In short, there is no more overlapping between rate distributions. Since EMCO uses an incremental analysis approach, data duplication at any point in time is out of the question. This is because, once EMCO removes a slice from the distributions that particular slice is not considered in the offloading process again. Detecting when enough data is collected is important because means that there is not anymore contexts that influence wrong offloading. Naturally, the amount of data varies among the mobile apps. As a result, apps with longer input variability require more data than those that implement resource-intensive tasks with a static input, for instance, the *minimax* algorithm of a *chess app* collects more data than a routine from a *backtracking app* that loads static 3D models.

RQ5. ***How to adapt the app to different QoE levels using computational offloading in order to improve user's experience?*** **RQ6.** ***Is it possible to provide computational offloading as a service that scales based on incoming load of mobile users?***

Finally, beyond the basic benefits of computational offloading, such as shortening response time of the apps and making the battery of the device last longer, we investigated how computational offloading can be utilized to enhance the perception that the user has towards the continuous usage of a mobile application. This perception is modeled as the fidelity of the mobile application during runtime and it is explained in detail in Chapter 5. Since the cloud provides a vast ecosystem of surrogates with different computational settings, a task that is offloaded to cloud can be accelerated at multiples levels, which means that the response time of the app can be provided *as a service* based on the suitable and individual perception of each mobile user. Our main motivation behind providing fidelity at multiple acceleration levels is to provide adaptive QoE, which can be used as mean of engagement strategy that increases the lifetime of a smartphone app. We envisioned that as part of an app released in a store, a counterpart computational service in a cloud also is released by the same party that developed the app to improve the QoE of the users that utilize the app. To achieve this, we extend our computational offloading framework presented in Chapter 4 with the ability to provision continuous computational offloading in multi-tenancy environments. Moreover, our system supports QoS policies to optimize dynamically the number of surrogates needed to handle a specific load of computational offloading requests without affecting the QoE of the active apps offloading to cloud. We evaluated the performance and scalability of the system and the results have shown the feasibility of our system in practice.

As part of our work, we provide all the described frameworks, use cases and tools as open source in GitHub.

6.2 Discussion

There are many benefits and drawbacks that emerge from implementing offloading and delegation mechanisms within the development of mobile cloud applications. We highlight key differences in approaches, implementation effort, usability, and richness of the mobile applications.

- Offloading is preferable than delegation as mobile applications can be executed in standalone mode if there is not available connectivity to the cloud. Thus, task delegation is not suitable for contexts, where there is no presence of network communication.

- Delegation enriches the mobile applications with more sophisticated functionality than offloading. Even though, mobile components at Class-method level can be offloaded, the limitations of the compiler running in the mobile virtual machines (VMs) unable the developer to implement complex routines within the mobile applications. For instance, the Dalvik virtual machine of Android offers just a set of java functionality. Consequently, the richness of the language cannot be exploited and libraries such as *jclouds* or *typica* cannot be executed on mobile platforms.

- Offloaded mobile components require less execution time than delegated mobile tasks. Consequently, we can argue that delegated mobile tasks do not provide suitable interactivity to the mobile users. However, natively a mobile platform supports and implements for some processes this kind of behavior. Thus, it is not trivial that a mobile application may need long waiting times for completing an operation.

- There are multiple tradeoffs between offloading and resource augmentation. Thus, a mobile application is potentiated by cloud based on its goals (e.g. Energy-saving, responsiveness, etc.). However, we believe that through characterization of an offloading operation, a mobile application can be adapted based on the context, such that a specific tradeoff can be applied at specific context, in order to obtain the maximum benefit each time the device goes cloud-aware.

- Delegation fosters a model, in which, mobile applications are enriched with the variety of cloud services provided on the Web, and thus this allows creating new business opportunities and alliances.

- The effort required to develop a mobile application that follows a delegation model is greater than an application that uses offloading. By default, a mobile architecture for delegation is highly distributed and multi-functional. Thus, it is complex to maintain.

- Different offloading frameworks provide different granularity regarding the definition of mobile components. Currently, mobile components can be offloaded at Class, Class-method and Thread level as discussed in section 2.2.1. Each of these levels require a specialized back-end running in the cloud (e.g. Android x86). Moreover, each strategy enriches the mobile application at different performance rates. Refer to Table 2.1 for more detailed information.

- Asynchronous delegation suffers from reliability as notification services do not ensure quality of services for delivering messages. However, notification mechanisms are highly integrated with mobile platforms, and thus the mechanisms are optimized to work using low resource consumption.

- Code offloading may fail in some cases, as the current scope utilized by most of the proposed work to characterize an offloading operation is not enough to measure a real benefit for the handset. This can easily be realized as 1) mobile components share a non-deterministic behavior, which makes complex the process of evaluating their impact at runtime (e.g. input variability), and 2) cloud infrastructure play an important role in the overall system. Moreover, next generation technologies for mobiles are computational comparable with some instances running on the cloud. For example, Samsung Galaxy S3 computational power is similar to a micro instance running in Amazon. As a result, in this thesis, we propose new frameworks to counter the issues of offloading and delegation.

6.2.1 Limitations

Certainly, mobile architectures can be equipped with computational offloading mechanisms in order to improve the performance of the smartphone apps and increase battery life of the device. However, besides the problems, which are addressed in this work, many other issues mitigate the adoption of computational offloading. In this section, we discuss these issues.

Regarding the utilization of computational offloading to improve the experience of user, the main limitation is the continuous reachability to cloud resources. Since network connectivity cannot be ensured all the time, applications have to be executed in stand alone mode. This is a drawback from a point of view of user engagement. Once the response time of an application is accelerated with cloud in order to engage the user, it is expected that this acceleration will continue, and even increase further as the user gets more engaged with the application. Naturally, when network connectivity is not available, the mobile application is executed using the mobile resources, which degrades again the user experience. Certainly, fidelity techniques can be implemented to assign more mobile resources for executing the application smoothly; however, even by doing this, it may be that it is not possible to achieve the same acceleration that is achieved by using cloud. Moreover, by completely focusing on the mobile user, fidelity techniques can make the device waste rather than save energy.

Another limitation of using computational offloading to improve QoE is the price model of cloud resources. Currently, a server in the cloud, e.g., Amazon, is rented per hour. As a result, a device is forced to offload to that particular server during that provisioning time. If the mobile application needs to increase performance, then it has to wait until the server is terminated (end of the hour), so that it can rent another one with higher computational capabilities. Definitively, a user does not have to wait until the end of the hour to rent another server, however, since using a server in the cloud has a cost, it is not logical to allocate many servers in the cloud to augment the capabilities of a single device. Ideally, as explored in Chapter 5 many server (back-end) should be available in the cloud to augment the processing resources of all the incoming load of devices. The load of request has to be routed by a load balancer (front-end), and from the mobile, a request is sent to the load balancer, which decides the type of acceleration that has to be given to the request. By using this architecture design, a user should be charged based on the number of requests, where each request has a cost that depends on the type of server that was used to process the request. By the time this thesis was written, Amazon has started exploring with a model in which a request is charged based on code execution.

Regarding the security issues that arise from implementing computational offloading within the mobile architectures, the main risk is to suffer code injection attacks. However, for these types of attacks to work, the attacker has to have complete knowledge about the code of the application, e.g., name of method, type of parameters, etc. Based on this information, the attacker can create his/her own object with the same specifications. If this happens, the attacker can replace the object exchanged in the mobile cloud communication by its own object that includes the malicious routines. If the attacker succeeds in this process, the malicious code can be executed in the device without a notice. While the attack is complex to perform, the risk of the attack exists. As a result, the communication channel can be encrypted, e.g., SSH, but in this case, the computational effort to transfer one task from one place to another also increases. Thus, in order to use secure computational offloading, it is necessary to engage in the discovery of new encryption mechanisms or to use computational offloading just in low latency networks, e.g., nearby cloudlet servers.

In the case of the technologies chosen to developed the systems presented in this thesis, we have to mention that recently, Google has announced that the Dalvik VM of Android Operating System is replaced by a new runtime compiler, namely *Android runtime* (ART) [47]. ART is designed to be compatible with Dex format, which is the bytecode executed by the virtual machine. The major modification between ART and Dalvik is that ART implements *Ahead-of-Time* (AOT) compilation instead of *Just-in-Time* (JIT). This means that an application that uses ART is compiled once during the first execution and every subsequent execution, the static portions of the app will not be compiled again. The main goal of ART is to reuse the compiled code, in order to increase performance and decrease the overall power consumption of the device.

Our proposed systems presented in Chapter 4 and Chapter 5 implement a Dalvik machine deployed on a x86 architecture as surrogate (Dalvil-x86).

[47]https://source.android.com/devices/tech/dalvik/

While a device can obtain major benefits from ART, a surrogate is not influenced by this change, because in the offloading process, a portion of code sent by a particular mobile is reconstructed by the surrogate using a specific application state. This means that a code offloading request depends on the dynamic application state obtained during the execution of the smartphone app. Since this application state can differ based on the device features, e.g., Android version, type of device, etc., it is not possible to pre-cache app functionality based on a single device criteria.

Naturally, it is possible to reuse an application state in the response of multiple code offloading requests as presented in our contribution in Chapter 4, but in that case, our EMCO framework implements a generalized strategy to pre-cached app functionality that is based on analyzing a community of devices. This feature cannot be achieved by an ART-based surrogate by itself.

6.3 Future Directions

The contributions to MCC outlined in this thesis open up a number of research avenues including:

1. ***Energy-aware offloading as a service for IoT (Internet of Things):*** — It is well known that the main goal of MCC is to augment the processing capabilities and energetic resources of low-power devices, e.g., smartphones. To achieve this, applications installed in the devices are instrumented with offloading mechanisms, e.g., code offloading. However, despite of this instrumentation, applications are not aware about the productive or counterproductive effect that can be influenced in the mobile resources by outsourcing a task. For instance, how much the code should be accelerated?, how much energy can be saved? etc.

   In this thesis, we overcome the problem of determining the context required to offload a task by analysis in the cloud the runtime history of code execution from a community of devices. By relying on the massive computational resources of the cloud to process *big data*, we aim to exploit the knowledge of the crowd. However, many other sources of information collected from a community of devices can provide insight about how to configure the offloading process, e.g., sensor information, user's interaction, etc. To illustrate this, let's consider the following cases:

   Case 1: a smartphone that calculates and transmits its GPS coordinates every time the user uses an application. If the frequency of app usage is high, then the device will run out of energy quickly, e.g., Tinder. If we assume that the end service in the cloud stores the data received, the data can be analyzed to build a prediction model in the cloud that suggests when the user changes his/her location. In this manner, the cloud service can be aware about the user's location and can configure the mobile app to recalculate and transmit GPS data when drastic changes of user's location are detected by the model. By implementing this approach, the device can save significant amounts of energy as the computational tasks of calculating and transmitting GPS data are not tied to app usage, but user's movement that is monitored by the cloud.

Case 2: a low-power device, e.g., Arduino microcontroller, that monitors an environment via sensors, e.g., temperature. Since a client that connects to the microcontroller expects to obtain real time information, the microcontroller senses the environment regularly. Moreover, in order to provide scalability for multiple users, the environmental information is sent to the cloud, such that any user can access it from there. Naturally, this process requires considerable amount of energy of the device. However, by analyzing the collected data, it is able to equip the cloud service with the awareness to schedule the sensing process of the microcontroller based on opportunistic contexts, for instance, sensing data is likely to be replaced by other sensing data from a nearby device, sensing data can be predicted based on history data stored in the cloud, etc., in any situation, the main goal is to schedule from the cloud, the behaviour of the device, so that the device can be alleviated from unnecessary computational effort. Undoubtedly, it is expected that the change of behavior won't change the quality of service or experience of the user.

2. ***Context-aware hybrid computational offloading:*** — Computational offloading is a promising technique to augment the computational capabilities of mobile devices. By connecting to remote servers, a mobile application can rely on code offloading to release the device from executing portions of code that requires heavy computational processing. Yet, computational offloading is far away to be adopted as a mechanism within the mobile architectures, mainly due to drastic changes in communication latency to remote cloud can cause energy draining rather than energy saving for the device [13, 21]. Moreover, in the presence of high communication latency, the responsiveness of the mobile applications is degraded, which suggests that in order to avoid collateral effects,

the benefits of computational offloading can just be exploited in low latency proximity using rich nearby servers [79], which are also known as cloudlets.

Fortunately, 5G is arising as a promising solution to overcome the problem of high latency communication in cellular networks. 5G fosters the utilization of Device to Device (D2D) communication [80, 81] to release the network from data traffic, and accelerate the transmission of data in end-to-end scenarios. By relying on D2D, and extrapolating features from remote cloud and cloutlets models, we envisioned a context-aware hybrid architecture for computational offloading. Our hybrid architecture introduces the concepts of network and cloud assistance, which can be utilized to coordinate the proximal devices in order to create a D2D infrastructure. Since the computational capabilities of next generation smarphones are comparable with some servers running in the cloud, we believe that multiple mobile devices can be merged together via D2D in order to create dynamic infrastructure in proximity that can be utilized by the devices themselves to share the load of processing heavy computational tasks.

Network assistance can be provided by cellular towers. The towers besides routing the communication between end-to-end points can be equipped with the logic to determine which devices are connected geographically close. When devices in proximity are detected, the tower can induce the devices to transmit data via D2D instead of using the cellular tower. The cellular towers can also be utilized to determine closer infrastructure (e.g., base stations), in which the device should be connected to reduce the communication latency, like in the cloudlet model. Similarly, cloud assistance can be utilized to group devices in a D2D cluster. Since devices are offloading to cloud-based servers (e.g., Amazon), the cloud can be equipped with the logic to determine which devices shared a common location. Cloud assistance introduces an extra level of complexity in the system than network assistance, due to a device is forced to send as part of the offloading process, the information about its location (e.g., GPS). However, cloud assistance alleviates completely the cellular network from computational offloading traffic, as all the process is managed entirely by the cloud [80].

Additionally, our hybrid framework introduces an infrastructural profiler running in the mobile that enables the device to determine the computational capacity of the available D2D infrastructure, which is in proximity. Our hybrid system enriches the offloading process by introducing more alternatives to offload, for instance, a mobile application can decide to offload to cloud or split the task among multiple nearby devices. Another possibility is that a mobile application can offload a task to cloud, store the result of that offloading request in mobile storage, and then propagate that result to other devices. Thus, avoiding other devices to offload to cloud.

Finally, a mobile application offloads to a cloud motivated by saving energy or improving the performance. In contrast, a mobile offloads to nearby devices motivated by the idea to have low latency infrastructure in proximity, similar to cloudlets, in which, the performance of a mobile application can be augmented without affecting the perception and interaction of the mobile user. Moreover, by offloading in proximity, the network is released from computational offloading traffic. We envisioned a computational offloading system in which a mobile application can offload code based on its context, where context is determined by evaluating the impact of code execution in a remote infrastructure and a D2D infrastructure in proximity.

3. ***The effect of computational offloading in large scale provisioning scenarios:*** — While the technique has been prove to be feasible with latest mobile technologies [21], still there are a lot of open issues regarding cloud deployment and provisioning in real scenarios. Previous works proposed a one server per each smartphone architecture [25], which is unrealistic in practice if we consider the amount of smartphones nowadays and the provisioning cost of constantly running a server for a particular user.

   Besides a few works that focus on scaling up (vertical scaling) a server to parallelize the code of computational requests [37], we have not found architectures that can scale in an horizontal fashion. This clearly can be seen as current frameworks do not take into consideration the utility computing features of the cloud, which is translated into server selection based on provisioning cost. Moreover, unlike a traditional architecture for code offloading that consists of a client and server, our system also includes a load balancer similar to Amazon autoscale, which differs from our balacing mechanism based on push notifications that is presented in Chapter 4.

   We are interested on analysis whether *it is possible to support large scale provisioning for computational offloading?* As a result, we want to study the capacity that cloud servers have to process multiple requests at once while maintaining requirements in code acceleration, which influences directly the response of a smartphone app. Moreover, we also want to analyze the effect of code acceleration in different cloud servers in order to foster surrogate selection based on utility computing, which can highlight new directions for the design of future mobile architectures supported by cloud computing, e.g., GPU offloading.

4. ***Tuning the fidelity of smartphone apps with mobile crowdsourcing:*** — In this thesis, by offloading to cloud, we present a solution to accelerate the response time of the mobile apps. The ultimate goal of the approach is to enhance the QoE of the mobile apps in terms of *fidelity*. By improving the QoE, we aim to engage the user in order to increase application usage.

   While our solution focuses on accelerating the execution of resource-intensive tasks, which are candidates to offload, it has been demonstrated that there are portions of code that cannot be offloaded [18]. An application that contains code that cannot be outsourced, it can also tune its fidelity by analyzing locally the tradeoff between resource consumption and quality of the response time [82].

   Changing fidelity of mobile apps has been prove to be feasible by collecting data locally in the device [26]. However, this process is slow, because history data is required, and sensitive to changes, because the device is constantly upgrading and installing new apps. Thus, in order to overcome these problems, we envisioned fidelity tuning via data analytics from a community of devices.

   Our idea is that apps are instrumented with mechanisms that capture their local execution at high level, e.g., method name, etc. This data is uploaded to the cloud for analysis. Based on the analysis, the cloud can perform individual diagnosis to each device and suggest optimal fidelity execution of each app installed in the device.

Chapter 7

Teenustele orienteeritud ja tõendite-teadlik mobiilne pilvearvutus

Arvutiteaduses on kaks kõige suuremat jõudu: mobiili- ja pilvearvutus. Tänapäeval kasutatakse mobiili- ja pilvetehnoloogiat igapäevaste tegevuste jaoks nagu suhtlemine, videote vaatamine, mängimine jne. Kui pilvetehnoloogia pakub kasutajale keerukate ülesannete lahendamiseks salvestus- ning arvutusplatvormi, siis nutitelefon võimaldab lihtsamate ülesannete lahendamist mistahes asukohas ja mistahes ajal. Seega on mõistlik liidestada pilvetehnoloogia mobiiliga, et saavutada arvutisarnane funktsionaalsus liikvel olles. Nende kahe ala koondumisega on esile kerkinud mobiilne pilvearvutus (Mobile Cloud Computing, MCC).

Täpsemalt on mobiilseadmetel võimalik pilve võimalusi ära kasutades energiat säästa ning jagu saada kasvavast jõudluse ja ruumi vajadusest. Sellest tulenevalt on käesoleva töö peamiseks küsimuseks *kuidas tuua pilveinfrastruktuur mobiilikasutajale lähemale?* Kuna mobiili- ja pilve ressursside ühendamiseks leidub erinevaid arhitektuurseid lahendusi, siis antud küsimusele vastuse leidmiseks vaatlesimegi mitmeid võimalusi. Mobiilseade võib töö delegeerida pilvele, et vabastada enda arvutuslikke ressursse. Töö tellitakse teenusetasemel (töö delegeerimine) ja teisaldatakse kooditasemel (arvutuslik teisaldamine). Töö tellimine eeldab, et pilv on seadmele kättesaadav kogu aeg. Seetõttu on võrguühendus mobiilirakenduse funktsionaalsuse aktiveerimiseks vajalik. Seevastu töö teisaldamine võimaldab mobiilirakendusel oma funktsionaalsust aktiveerida võrguühenduse olemasolust sõ ltumata.

Olemasolevad lahendused on näidanud, et töö edastamine pilvele rikastab mobiilirakendust keerukama funktsionaalsusega, kuid see ei taga energiasäästu ega paremat jõudlust rakendusele. Kuna pilveteenuste disain ja juurutamine jälgib peamiselt SOA põhimõtteid, siis algselt uurisime me käesolevas väitekirjas, kuidas mobiili pilveteenuseid integreerida mobiilirakendustes. Saime teada, et töö delegeerimine pilve eeldab mitmete pilve aspektide kaalumist ja integreerimist, nagu näiteks ressursimahukas töötlemine, asünkroonne suhtlus kliendiga, programmaatiline ressursside varustamine (Web APIs) ja pilvedevaheline kommunikatsioo. Nende puuduste ületamiseks lõime me Mobiilse pilve vahevara Mobile Cloud Middleware (Mobile Cloud Middleware - MCM) raamistiku, mis kasutab deklaratiivset teenuste komponeerimist, et delegeerida töid mobiililt mitmetele pilvedele kasutades minimaalset andmeedastust.

Teisest küljest on näidatud, et koodi teisaldamine on peamisi strateegiaid seadme energiatarbimise vähendamiseks ning jõudluse suurendamiseks. Sellegipoolest on koodi teisaldamisel miinuseid, mis takistavad selle laialdast kasutuselevõttu. Koodi teisaldamine on kasulik, kui suudetakse säästa energiat, muutmata seejuures rakenduse reaktsiooniaega ning vastupidi kahjulik, kui kulutatakse töö teisaldamiseks rohkem energiat võrreldes töö lokaalse jooksutamisega. Selles töös uurime lisaks, mis takistab koodi mahalaadimise kasutuselevõttu ja pakume lahendusena välja raamistiku EMCO, mis kogub seadmetelt infot koodi jooksutamise kohta erinevates kontekstides. Neid andmeid analüüsides teeb EMCO kindlaks, mis on sobivad tingimused koodi maha laadimiseks. Võrreldes kogutud andmeid, suudab EMCO järeldada, millal tuleks mahalaadimine teostada. EMCO modelleerib kogutud andmeid jaotuse määra järgi lokaalsete- ning pilvejuhtude korral. Neid jaotusi võrreldes tuletab EMCO täpsed atribuudid, mille korral mobiilirakendus peaks koodi maha laadima. Võrreldes EMCO-t teiste nüüdisaegsete mahalaadimisraamistikega, tõuseb EMCO efektiivsuse poolest esile.

Lõpuks, lisaks peamistele koodi maha laadimise eelistele, uurisime kuidas arvutuste maha laadimist ära kasutada, et täiustada kasutaja kogemust pideval mobiilirakenduse kasutamisel. Seda kogemust suhestatakse ajaga, mis kulub ülesande arvutuslikuks tööks ja kasutajale tulemuste näitamiseks. Kuna pilve arhitektuur võimaldab seadistada ülesannete täitmist mitmel moel, siis on võimalik pilve laetud ülesannet kiirendada mitmel tasemel. See tähendab, et rakenduse reaktsiooniaja saab ette näha igale mobiiltelefoni kasutajale vastavalt neile sobivale ja individuaalsele tajule. Meie peamiseks motivatsiooniks, et sellist adaptiivset tööde täitmise kiirendamist pakkuda, on tagada kasutuskvaliteet (QoE), mis muutub vastavalt kasutajale, aidates seeläbi suurendada mobiilirakenduse eluiga. Meie nägemuse kohaselt on ühe osana rakenduste poes avaldatud rakendusest pilvel asuv arvutuslik teenus, mis parandab jooksvalt rakenduse tarbijate kasutuskvaliteeti.

Chapter 8

Appendix

8.1 Appendix A: Sensor Classification Algorithm using MapReduce

The integration of micromechanical sensor technologies within the smartphones makes it possible to enrich the usability experience of interacting with a mobile application by sensing the user's context and to understand certain human activities based on the tracking of the user's intention. Several prototypes and signal processing algorithms have been developed for human motion classification and recognition allowing reliable (more than 90% accurate) detection of basic movements [83].

The accelerometer simultaneously outputting tilt, is the most common sensor that is included within a modern mobile device as it allows tracking information that can be used for infering multiple human movements. Depending on the number of axes, it can gather the acceleration information from multiple directions. Generally a triaxial accelerometer is the most common in mobiles from vendors such as HTC, Samsung, Nokia etc. Therefore, acceleration can be sensed on three axes, forward/backward, left/right and up/down. For example: in the case of a runner, up/down is measured as the crouching when he/she is warming up before starting to run, forward/backward is related with speeding up and slowing down, and left/right involves making turns while he/she is running.

Similarly, the gyroscope sensor is an actuator based on the principles of angular momentum conservation that is used for establishing position, navigation and orientation of the device, among others. It consists of three axes or freedom degrees (spinning, perpendicular and tilting) mounted in a rotor which are composed by two concentrically pivoted rings (inner and outer). The gyroscope is used within the mobile for enhancing techniques such as gesture recognition and face detection. Furthermore, the combination of accelerometer

8.1 Appendix A: Sensor Classification Algorithm using MapReduce

Algorithm 3 Sensor Processing with MapReduce

- Map
  Require: CSV file
  - map function parameter is a <*key, value* >pair, where:
    - *key* - line number
    - *value* - line content
  - *value* is split into several variables (gyroscope's and accelerometer's values, timestamp, url)
  - gyroscope's values are examined, they have to be smaller than 0.05 (which means minimal rotation)
  - accelerometer's values are compared to the fixed threshold (to indicate that the user is holding the phone in his hand)
  - if the sensors' values are in range, the map will emit a <*key, value* >pair, where:
    - *key* - timestamp in seconds
    - *value* - url
- Reduce
  Require: *timestamp, list <url >*
  - timestamp consists of the relative time in which the measurement was taken
  - count the elements in the list and emit a <*key,value* >pair, where:
    - *key* - time range
    - *value* - url
  - sort the list in descending order.

and gyroscope sensor data allows to increase the motion accuracy, and thus approaches such as video stabilization are implemented on the mobile.

We collected and synchronized in this thesis, the measurements of the gyroscope and accelerometer along with the URL of the mobile web browser in order to identify repetitive patterns that can be classified as a specific human activity (e.g. walking, reading, etc.). Algorithm 3 shows the parallelizable process of sensor analysis. We used MapReduce as the sensor information can be collected daily from the mobile and uploaded to the cloud, thus, creating a big repository of analyzable data that requires resource-intensive processing. Basically, the algorithm aims to classify all the sensor readings into an interval threshold, which is defined prior the experiments in a training phase. This interval was determined by identifying what are the actual sensors readings of the accelerometer and gyroscope when the device is hold by the user in a reading position. Naturally, the meaning of a reading position can vary. As a result, in this work, we define a reading position as all the space in front of the user in a radius of $\approx$ 50 cm from his/her eyes as starting.

References

[1] Mahadev Satyanarayanan, James J Kistler, Lily B Mummert, Maria R Ebling, Puneet Kumar, and Qi Lu. *Experience with disconnected operation in a mobile computing environment.* Springer, 1996. 15

[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, et al. **Above the clouds: a berkeley view of cloud computing**. *EECS Department, University of California, Berkeley, technical report.*, 2009. 15

[3] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. **The case for vm-based cloudlets in mobile computing**. *IEEE Pervasive Computing Magazine*, **8**(4):14–23, 2009. 16, 17, 33, 45, 49

[4] U. Hansmann, R.M. Mettala, A. Purakayastha, and P. Thompson. *SyncML: synchronizing and managing your mobile data.* Prentice Hall, 2003. 16

[5] Dushyanth Narayanan, Jason Flinn, and Mahadev Satyanarayanan. **Using history to improve mobile application adaptation**. In *Proceedings of 3rd IEEE workshop on mobile computing systems and applications*, pages 31–40. IEEE, 2000. 16, 19, 21, 44, 45, 48, 107

[6] H. Flores, S. N. Srirama, and C. Paniagua. **A generic middleware framework for handling process intensive hybrid cloud services from mobiles**. In *Proceedings of the 9th international conference on advances in mobile computing & multimedia (MoMM)*, pages 87–95. ACM, 2011. 16, 24, 28, 53, 80

[7] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. **Mobile cloud computing: a survey**. *Future generation computer systems*, **29**(1):84–106, 2013. 16, 17, 19, 38, 40

[8] Huber Flores, Satish Narayana Srirama, and Rajkumar Buyya. **Computational offloading or data binding? bridging the cloud infrastructure to the proximity of the mobile user**. In *Proceedings of IEEE 2nd international conference on mobile cloud computing, services, and engineering (MobileCloud)*, pages 10–18. IEEE, 2014. 16, 17

[9] Qian Wang and Ralph Deters. **SOA's last mile connecting smartphones to the service cloud**. In *IEEE international conference on cloud computing (CLOUD)*, pages 80–87, 2009. 16, 29

[10] Huber Flores and Satish Narayana Srirama. **Mobile cloud middleware**. *Journal of systems and software*, **92**:82–94, 2014. 16, 17, 20, 43, 52, 100

[11] Rajesh Balan, Jason Flinn, Mahadev Satyanarayanan, Shafeeq Sinnamohideen, and Hen-I Yang. **The case for cyber foraging**. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 87–92. ACM, 2002. 17

[12] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. **Cloudlets: bringing the cloud to the mobile user**. In *Proceedings of the third Mobisys workshop on mobile cloud computing and services (MCS)*, pages 29–36. ACM, 2012. 17, 34

[13] Marco V Barbera, Sokol Kosta, Alessandro Mei, Vasile C Perta, and Julinda Stefa. **Mobile offloading in the wild: findings and lessons learned through a real-life experiment with a new cloud-aware system**. *Proceedings of IEEE INFOCOM*. 17, 18, 40, 82, 83, 99, 134

[14] Huber Flores and Satish Srirama. **Mobile code offloading: should it be a local decision or global inference?** In *Proceeding of the 11th annual international conference on mobile systems, applications, and services (MobiSys)*, pages 539–540. ACM. 17, 18, 21, 36, 41, 50, 83

[15] Huber Flores, Pan Hui, Sasu Tarkoma, Yong Li, Satish Srirama, and Rajkumar Buyya. **Mobile code offloading: from concept to practice and beyond**. *IEEE Communications Magazine*, **53**(3):80–88, 2015. 17, 21, 33, 48, 82, 108, 118

[16] Xiaohui Gu, Klara Nahrstedt, Alan Messer, Ira Greenberg, and Dejan Milojicic. **Adaptive offloading for pervasive computing**. *IEEE pervasive computing magazine*, **3**(3):66–73, 2004. 17

[17] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. **MAUI: making smartphones last longer with code offload**. In *Proceedings of the 8th international conference on mobile systems, applications, and services (Mobisys)*, pages 49–62. ACM, 2010. 17, 18, 35, 37, 38, 49, 50

[18] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. **Clonecloud: elastic execution between mobile device and cloud**. In *Proceedings of the 6th conference on computer systems (EuroSys)*, pages 301–314, 2011. 17, 18, 35, 38, 39, 49, 50, 86, 137

[19] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. **Thinkair: dynamic resource allocation and parallel execution in the cloud for mobile code offloading**. In *Proceedings of IEEE INFOCOM*, pages 945–953. IEEE, 2012. 17, 18, 38, 39, 43, 50

[20] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. **COMET: code offload by migrating execution transparently**. In *Proceedings of the 10th USENIX conference on operating systems design and implementation*, pages 93–106. USENIX, 2012. 17, 18, 38, 39, 50

[21] Huber Flores and Satish Srirama. **Adaptive code offloading for mobile cloud applications: exploiting fuzzy sets and evidence-based learning**. In *Proceeding of the 4th ACM MobiSys workshop on mobile cloud computing and services*, pages 9–16, 2013. 17, 38, 40, 90, 115, 134, 136

[22] J. Dean and S. Ghemawat. **MapReduce: simplified data processing on large clusters**. *Communications of the ACM*, **51**(1):107–113, 2008. 18

[23] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. **Hive: a warehousing solution over a map-reduce framework**. *Proceedings of the VLDB Endowment*, **2**(2):1626–1629, 2009. 18

[24] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alexandru Nicolau. **SPARK: a high-level synthesis framework for applying parallelizing compiler transformations**. In *Proceedings of international conference on VLSI Design*, pages 461–466. IEEE, 2003. 18

[25] Paramvir Bahl, Richard Y Han, Li Erran Li, and Mahadev Satyanarayanan. **Advancing the state of mobile cloud computing**. In *Proceedings of the 3rd Mobisys workshop on mobile cloud computing and services (MCS)*, pages 21–28. ACM, 2012. 18, 40, 49, 82, 122, 136

[26] Mahadev Satyanarayanan and Dushyanth Narayanan. **Multi-fidelity algorithms for interactive mobile applications**. *Wireless Networks*, **7**(6):601–607, 2001. 19, 48, 111, 137

[27] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*, **1**. Prentice Hall Englewood Cliffs, 1996. 19

[28] Jan Bosch. **Software architecture: The next step**. In *Software architecture*, pages 194–199. Springer, 2004. 19

[29] Daniel A Menasce, Virgilio AF Almeida, Lawrence W Dowdy, and Larry Dowdy. *Performance by design: computer capacity planning by example.* Prentice Hall Professional, 2004. 19, 32, 45, 113

[30] Henry H Liu. *Software performance and scalability: a quantitative approach*, **7**. John Wiley & Sons, 2011. 19, 113

[31] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. 19

[32] Andreas Klein, Christian Mannweiler, Joerg Schneider, and Hans D Schotten. **Access schemes for mobile cloud computing**. In *Proceedings of 11th international conference on mobile data management (MDM)*, pages 387–392. IEEE, 2010. 20

[33] Huber Flores and Satish Narayana Srirama. **Mobile cloud messaging supported by XMPP primitives**. In *Proceedings of the 4th Mobisys workshop on mobile cloud computing and services (MCS)*. ACM, 2013. 20, 53, 55, 74, 81, 94, 105

[34] Verdi March, Yan Gu, Erwin Leonardi, George Goh, Markus Kirchberg, and Bu Sung Lee. **μcloud: towards a new paradigm of rich mobile applications**. *Procedia Computer Science*, **5**:618–624, 2011. 29

[35] R. Aversa, B. Di Martino, M. Rak, and S. Venticinque. **Cloud agency: a mobile agent based cloud system**. In *Proceedings of international conference on complex, intelligent and software intensive systems*, pages 132–137. Ieee, 2010. 29

[36] K. Kumar and Y.H. Lu. **Cloud computing for mobile users: can offloading computation save energy?** *Computer*, **43**(4):51–56, 2010. 35, 36, 74, 82, 87

[37] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. **Odessa: enabling interactive perception applications on mobile devices**. In *Proceedings of the 9th international conference on mobile systems, applications, and services (Mobisys)*, pages 43–56. ACM, 2011. 38, 39, 49, 136

[38] Cong Shi, Karim Habak, Pranesh Pandurangan, Mostafa Ammar, Mayur Naik, and Ellen Zegura. **COSMOS: computation offloading as a service for mobile devices**. *Proceedings of MobiHoc*, 2014. 38, 40, 43, 83, 99

[39] Aki Saarinen, Matti Siekkinen, Yu Xiao, Jukka K Nurminen, Matti Kemppainen, and Pan Hui. **SmartDiet: offloading popular apps to save energy**. In *Proceedings of the ACM SIGCOMM 2012 conference on applications, technologies, architectures, and protocols for computer communication*, pages 297–298. ACM, 2012. 40

[40] Guangyu Chen, B-T Kang, Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Rajarathnam Chandramouli. **Studying energy trade offs in offloading computation/compilation in java-enabled mobile devices**. *IEEE transactions on parallel and distributed systems*, **15**(9):795–809, 2004. 41

[41] Antti P Miettinen and Jukka K Nurminen. **Energy efficiency of mobile clients in cloud computing**. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 4–4. USENIX Association, 2010. 41

[42] Mark Weiser. **Ubiquitous computing**. *Computer*, (10):71–72, 1993. 44

[43] Kuan-Ta Chen, Cheng-Chun Tu, and Wei-Cheng Xiao. **OneClick: a framework for measuring network quality of experience**. In *Proceedings of IEEE INFOCOM*, pages 702–710. IEEE, 2009. 45

[44] Peter Brooks and Bjørn Hestnes. **User measures of quality of experience: why being objective and quantitative is important**. *IEEE network magazine*, **24**(2):8–13, 2010. 45

[45] Xuan Bao, Songchun Fan, Alexander Varshavsky, Kevin Li, and Romit Roy Choudhury. **Your reactions suggest you liked the movie: automatic content rating via reaction sensing**. In *Proceedings of the ACM international joint conference on pervasive and ubiquitous computing*, pages 197–206. ACM, 2013. 45

[46] J. Froehlich, M.Y. Chen, S. Consolvo, B. Harrison, and J.A. Landay. **MyExperience: a system for in situ tracing and capturing of user feedback on mobile phones**. In *Proceedings of the 5th international conference on mobile systems, applications and services (Mobisys)*, pages 57–70. ACM, 2007. 45

[47] Vaneet Aggarwal, Emir Halepovic, Jeffrey Pang, Shobha Venkataraman, and He Yan. **Prometheus: toward quality-of-experience estimation for mobile apps from passive network measurements**. In *Proceedings of the 15th workshop on mobile computing systems and applications (HotMobile)*, page 18. ACM, 2014. 45, 46, 108

[48] Athula Balachandran, Vaneet Aggarwal, Emir Halepovic, Jeffrey Pang, Srinivasan Seshan, Shobha Venkataraman, and He Yan. **Modeling web quality-of-experience on cellular networks**. In *Proceedings of the 20th annual international conference on mobile computing and networking*, pages 213–224. ACM, 2014. 45, 46, 47, 108

[49] Robert Hsieh and Aruna Seneviratne. **A comparison of mechanisms for improving mobile IP handoff latency for end-to-end TCP**. In *Pro-*

ceedings of the 9th annual international conference on mobile computing and networking, pages 29–41. ACM, 2003. 46

[50] Leo A Meyerovich and Rastislav Bodik. **Fast and parallel webpage layout**. In *Proceedings of the 19th international conference on world wide web (WWW)*, pages 711–720. ACM, 2010. 46

[51] Venkata N Padmanabhan and Jeffrey C Mogul. **Improving HTTP latency**. *Computer networks and ISDN systems*, **28**(1):25–35, 1995. 46

[52] Hari Balakrishnan, Venkata N Padmanabhan, Srinivasan Seshan, and Randy H Katz. **A comparison of mechanisms for improving TCP performance over wireless links**. *IEEE/ACM transactions on networking*, **5**(6):756–769, 1997. 46

[53] Mojca Volk, Janez Sterle, Urban Sedlar, and Andrej Kos. **An approach to modeling and control of QoE in next generation networks [Next Generation Telco IT Architectures]**. *IEEE Communications Magazine*, **48**(8):126–135, 2010. 46

[54] Kuan-Ta Chen, Chun-Ying Huang, Polly Huang, and Chin-Laung Lei. **Quantifying Skype user satisfaction**. In *ACM SIGCOMM computer communication review*, **36**, pages 399–410. ACM, 2006. 47

[55] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. **Developing a predictive model of quality of experience for internet video**. In *Proceedings of the ACM SIGCOMM*, pages 339–350. ACM, 2013. 47, 107

[56] Quan Z Sheng, Xiaoqiang Qiao, Athanasios V Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. **Web services composition: a decade‘s overview**. *Information Sciences*, **280**:218–238, 2014. 52

[57] Liangzhao Zeng, Boualem Benatallah, Anne HH Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. **QoS-aware middleware for web services composition**. *IEEE Transactions on software engineering,*, **30**(5):311–327, 2004. 52

[58] A. Onetti and F. Capobianco. **Open source and business model innovation. The Funambol case**. In *Proceedings of International Conference on operating systems*, pages 224–227, 2005. 52

[59] Jason H Christensen. **Using RESTful web-services and cloud computing to create next generation mobile applications**. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 627–634. ACM, 2009. 52

[60] H. Flores, S. Srirama, and C. Paniagua. **Towards mobile cloud applications: offloading resource-intensive tasks to hybrid clouds**. *International journal of pervasive computing and communications*, **8**(4):344–367, 2012. 57, 76, 80

[61] S.N. Srirama, C. Paniagua, and H. Flores. **Social group formation with mobile cloud services**. *Service Oriented Computing and Applications*, **6**(4):1–12, 2012. 57

[62] Satish Narayana Srirama, Huber Flores, and Carlos Paniagua. **Zompopo: mobile calendar prediction based on human activities recognition using the accelerometer and cloud services**. In *Proceedings of 5th international conference on next generation mobile applications, services and technologies (NGMAST)*, pages 63–69. IEEE CS, 2011. 57

[63] Satish Narayana Srirama, Carlos Paniagua, and Huber Flores. **CroudSTag: social group formation with facial recognition and mobile cloud services**. *Procedia computer science*, **5**:633–640, 2011. 57

[64] Carlos Paniagua, Satish Narayana Srirama, and Huber Flores. **Bakabs: managing load of cloud-based web applications from mobiles**. In *Proceedings of the 13th international conference on information integration and web-based applications and services (iiWas)*, pages 485–490. ACM, 2011. 57

[65] Peter Saint-André, Kevin Smith, and Remko Troncon. *XMPP: the definitive guide : building real-time applications with Jabber*. O'Reilly Media, 2009. 60

[66] Carlos Paniagua, Huber Flores, and Satish Narayana Srirama. **Mobile sensor data classification for human activity recognition using MapReduce on cloud**. *Procedia computer science*, **10**:585–592, 2012. 71

[67] Wenzhong Li, Yanchao Zhao, Sanglu Lu, and Daoxu Chen. **Mechanisms and challenges on mobility-augmented service provisioning for mobile cloud computing**. *IEEE Communications Magazine*, **53**(3):89–97, 2015. 82

[68] Aki Saarinen, Matti Siekkinen, Yu Xiao, Jukka K Nurminen, Matti Kemppainen, and Pan Hui. **Can offloading save energy for popular apps?** In *Proceedings of the 17th ACM international workshop on mobility in the evolving internet architecture*, pages 3–10. ACM, 2012. 82

[69] Eemil Lagerspetz and Sasu Tarkoma. **Mobile search and the cloud: the benefits of offloading**. In *Proceedings of IEEE international conference on pervasive computing and communications workshops (PERCOM Workshops)*, pages 117–122. IEEE, 2011. 82

[70] Byung-Gon Chun and Petros Maniatis. **Dynamically partitioning applications between weak devices and clouds**. In *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*, page 7. ACM, 2010. 82

[71] Adam J Oliner, Anand P Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. **Carat: collaborative energy diagnosis for mobile devices**. In *Proceedings of the 11th ACM conference on embedded networked sensor systems (Sensys)*, page 10. ACM, 2013. 83, 88, 96

[72] Liyao Xiang, Shiwen Ye, Yuan Feng, Baochun Li, and Bo Li. **Ready, set, go: coalesced offloading from mobile devices to the cloud**. *Proceedings of IEEE INFOCOM.* 85

[73] Mark Gordon, Lide Zhang, Birjodh Tiwana, R Dick, ZM Mao, and L Yang. **Power tutor, a power monitor for Android-based mobile platforms**, 2009. 88

[74] Markus Fiedler, Tobias Hossfeld, and Phuoc Tran-Gia. **A generic quantitative relationship between quality of experience and quality of service**. *IEEE network magazine*, **24**(2):36–41, 2010. 107, 108

[75] Muhammad Zubair Shafiq, Lusheng Ji, Alex X Liu, Jeffrey Pang, and Jia Wang. **Characterizing geospatial dynamics of application usage in a 3G cellular data network**. In *Proceedings of IEEE INFOCOM*, pages 1341–1349. IEEE, 2012. 108

[76] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. **Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage**. In *Proceedings of the 13th international conference on human computer interaction with mobile devices and services*, pages 47–56. ACM, 2011. 111

[77] Hamdy A Taha. *Operations research: an introduction*, **557**. Pearson/Prentice Hall, 2007. 111

[78] Yagiz Onat Yazir, Chris Matthews, Roozbeh Farahbod, Stephen Neville, Adel Guitouni, Sudhakar Ganti, and Yvonne Coady. **Dynamic resource allocation in computing clouds using distributed multiple criteria decision analysis**. In *IEEE 3rd international conference on cloud computing (CLOUD)*, pages 91–98. Ieee, 2010. 113

[79] Zhang Yang, Dusit Niyato, and Ping Wang. **Offloading in mobile cloudlet systems with intermittent connectivity**. 135

[80] Jakob Mass, Satish Narayana Srirama, Huber Flores, and Chii Chang. **Proximal and social-aware device-to-device communication**

via audio detection on cloud. In *Proceedings of the 13th International conference on mobile and ubiquitous multimedia (MUM)*, pages 143–150. ACM, 2014. 135

[81] Cong Shi, Vasileios Lakafosis, Mostafa H Ammar, and Ellen W Zegura. **Serendipity: enabling remote computing among intermittently connected mobile devices**. In *Proceedings of the thirteenth ACM international symposium on mobile ad hoc networking and computing*, pages 145–154. ACM, 2012. 135

[82] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, B-g Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. **Mantis: efficient predictions of execution time, energy usage, memory usage and network usage on smart mobile devices**. 137

[83] S.J. Preece, J.Y. Goulermas, L.P.J. Kenney, D. Howard, K. Meijer, and R. Crompton. **Activity identification using body-mounted sensorsŮa review of classification techniques**. *Physiological measurement*, **30**(4):R1, 2009. 140