ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DISI

INGEGNERIA INFORMATICA

TESI DI LAUREA

in
Reti di Calcolatori M

Smart execution of distributed application by balancing resources in mobile devices and cloud-based avatars

CANDIDATO:
Giacomo Gezzi

RELATORE:
Chiar.mo Prof. Ing. Antonio Corradi

CORRELATORE:
Chiar.mo Prof. Cristian Borcea

Anno Accademico 2014/2015
Sessione III

Abstract

L'obiettivo del progetto di tesi svolto è quello di realizzare un servizio di livello middleware dedicato ai dispositivi mobili che sia in grado di fornire il supporto per l'offloading di codice verso una infrastruttura cloud. In particolare il progetto si concentra sulla migrazione di codice verso macchine virtuali dedicate al singolo utente. Il sistema operativo delle VMs è lo stesso utilizzato dal device mobile. Come i precedenti lavori sul computation offloading, il progetto di tesi deve garantire migliori performance in termini di tempo di esecuzione e utilizzo della batteria del dispositivo.

In particolare l'obiettivo più ampio è quello di adattare il principio di computation offloading a un contesto di sistemi distribuiti mobili, migliorando non solo le performance del singolo device, ma l'esecuzione stessa dell'applicazione distribuita. Questo viene fatto tramite una gestione dinamica delle decisioni di offloading basata, non solo, sullo stato del device, ma anche sulla volontà e/o sullo stato degli altri utenti appartenenti allo stesso gruppo. Per esempio, un primo utente potrebbe influenzare le decisioni degli altri membri del gruppo specificando una determinata richiesta, come alta qualità delle informazioni, risposta rapida o basata su altre informazioni di alto livello.

Il sistema fornisce ai programmatori un semplice strumento di definizione per poter creare nuove policy personalizzate e, quindi, specificare nuove regole di offloading. Per rendere il progetto accessibile ad un più ampio numero di sviluppatori gli strumenti forniti sono semplici e non richiedono specifiche conoscenze sulla tecnologia.

Il sistema è stato poi testato per verificare le sue performance in ter-

mini di mecchanismi di offloading semplici. Successivamente, esso è stato anche sottoposto a dei test per verificare che la selezione di differenti policy, definite dal programmatore, portasse realmente a una ottimizzazione del parametro designato. Questi ultimi test sono stati eseguiti utilizzando un numero variabile di dispositivi mobili.

Contents

List of Figures

List of Tables

Introduction

The mobile devices are becoming the center of the information technology. Since the cost of smartphones, tablets and wearable devices is decreasing, this technology has become affordable to a larger and larger number of users. While this factor has driven many companies to invest their efforts and capabilities in it, at the same time the dedicated software for this kind of device has evolved and so its features and expertise capacity. Therefore, mobile development tools has been improved to achieve new goals and new functions supporting also distributed systems.

Nowadays, it is possible to use a large number of distributed mobile application on a standard mobile device with powerful system and architecture to manage the complexity of the environment. The research and industry community has adapted many tools from the traditional distributed systems scenario to the new mobile configurations. Some approaches allow to use directly a backbone structure to manage a mobile distributed system as a traditional one. The development in the *virtualization* technology and the resulting success of the Cloud Computing have supplied new tools for the mobile computing apps. The possibility to exploit the computation and resources of powerful servers could enable the execution of computing and communication intensive applications. The mobile devices could indeed rely on the offloading of computation, communication or data storage management through the Cloud Computing.

A suitable solution for this environment would be a system able to provide a simple access to the Mobile Cloud Computing functions. The programmer should be able to easily partition the application discerning between functions to offload and to execute locally. The system should also automatically analyse various parameters of the device state in order to decide whether the *offloadable* code could be offloaded or not. However, the user may want to define new policies in order to modify the automatic offloading decisions and to endorse additional parameters not originally considered by the automatic decision engine.

With the definition of custom policies, the system could also support mobile distributed applications providing a powerful tool able to consider more complex scenarios. Therefore, this system should improve the average performance of a distributed application taking decision based on the global resources or on the users requests. Furthermore, the selection of multiple policies could be also available to the final user at run-time providing to the final user a innovative tool to modify the behaviour of the entire distributed application execution. For instance, a user wants a function completed as soon as possible from the group, this constrain would be translated executing pieces of code remotely accelerating the time of execution. In a different scenario a user may want to not share information with the cloud infrastructure, and so the offloading system should execute locally the methods dependent on this data.

The thesis is composed of five chapters.

The first chapter aims to introduce the distributed systems and the middleware level software. After an overview on the traditional systems, the chapter will focus on the mobile environment and its middleware level software. The final task of the chapter is to introduce the idea of cloud computing and show its application on the Mobile Computing, also called Mobile Cloud Computing.

The second chapter gives an overview on the computation offloading applied to the Mobile Cloud Computing presenting the heuristics use to take decisions and some crucial challenges. It discusses some previous projects on the mobile computing offloading and it highlights the main differences between the projects.

The third chapter initially presents the Avatar system, the thesis project is indeed integrated with this Mobile Cloud Computing system. The second part of the chapter deepens the thesis project challenges and scenarios. Finally, the chapter explains software design and the desired programming model to provide.

The forth chapter gives a full overview about the enabling technologies necessary to implement the design shown in the previous chapters.

Finally, the last chapter focuses on the implementation architecture and components of the thesis project. The results and the evaluations about the final implementation of the code are discussed in the last part of the thesis.

Chapter 1

Mobile Distributed Computing Assisted by the Cloud

The purpose of this chapter is to explore the target scenarios of the thesis project. The first section focuses on the distributed systems definition and what are their main characteristics and peculiarities. After that, the main functions of middleware level software are enlisted and it is explained how they can help development and deployment of distributed applications. Since the previous sections are meant to give an introduction about the world of distributed systems, the following section applies and expands the previous concepts to the mobile devices environment. At this point, the chapter focuses on the mobile middleware challenges and differences. In order to support the previous concepts, a section is dedicated to the related works of mobile middleware. The rest of the chapter is dedicated to the middleware for mobile computing software and what are the motivations which drive the research on this matter. Finally, in the last section of the chapter introduces some application examples which benefit of the mobile cloud computing software.

1.1 Distributed Systems

Before deepening into middleware software level and mobile systems, it is necessary to introduce the base theory behind traditional middleware and distributed systems. These concepts are expandable and applicable to the mobile distributed systems and their middleware level with some additional precautions. The **Section 1.2** and **Section 1.2.2** show how the general concept are applicable to the mobile environments are what are their benefits.

The improvement of networks and computation performances has supported the evolution of the computer systems and their architectures. In order to provide more interesting and powerful services, industries have adopted distributed systems. A distributed system is one in which components located at networked computers communicate and coordinate their actions by only passing messages [1].

1.1.1 Features of Distributed Systems

A group of computers connected by a network could be separated by any kind of physical distance [2]. The conformation of a distributed system is more complex of single computer architecture, so it is important to take care of new complexity factors introduced by the communication and synchronisation:

- *Concurrency*: In a distributed environment it is possible that every user is executing separately his task. In addition, users could share resources, such as files or databases. In order to manage the concurrency of shared resources is possible to add further servers to the network;

- *Time Synchronisation*: As mentioned earlier, multiple computers need to synchronise their tasks in order to provide a common service or achieve a shared goal. The idea of synchronisation is strongly related to the idea of a shared time, it is therefore essential for a distributed system to have a global clock. However, there are limits for the accuracy of it. The lack of precision is a consequence of the messages communication and there are many solutions based on the desired quality of service;

- *Independent failures*: In a distributed systems the idea of failures is more complex than in a single tier scenario. In fact, there are many new ways in which the system could fail for network issues. The network could fail denying the synchronisation and the message passing between the system members. This problem could open many new scenarios to manage. For instance, one member fails and cannot run its task any more or the network in itself fails and the computers cannot interact with each other. An advanced distributed system should handle this kind of failures and try to isolate them. In this way every member of the system could fail independently without affecting the rest of the group execution;

The need to manage this complex scenarios and to support distributed systems leads the research to introduce a dedicated class of software technologies. The **Section 1.1.2** aims to deepen this software level.

1.1.2 Middleware for Distributed Systems

The idea of middleware was introduced to manage the complexity and heterogeneity inherent in distributed system . As shown in the **Figure 1.1**, middleware is a layer of software located between the operating system and the application programs. It provides a common programming abstraction across the distributed system relieving application programmers of this kind of complex and time-consuming programming. In addition, middleware services often save the developer from implementation errors and improves application performances.

In few words, the middleware software masks heterogeneity of hardware and networks. Further, some advanced middleware is also able to manage differences between operating systems and programming languages. A famous example of middleware is CORBA, it is a standard definition not really an implemented software. The idea of the CORBA specification is to hide from the developer the implementation choices and design providing well-defined services and behaviours.
Finally, another important task for this kind of software is to provide transparency for one or more of the following dimensions: location, concur-

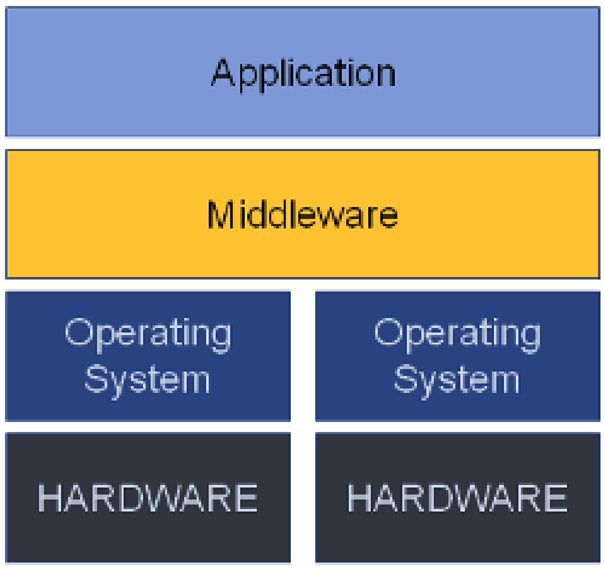


Figure 1.1: Middleware Layer in a distributed system.

rency, replication, failures and mobility. Therefore, it is possible to classify the middleware software based on their features and on the level of transparency which they provide [3].

- **Remote Procedure Call**: The Remote Procedure Call (RPC) is considered an *ancestor* of middleware software. The RPC provides an abstraction capable to invoke a procedure whose body is across a network. The RPC has a not really flexible model and it does not provide a good level of scalability. This software class does not take care of resources optimization and it is strongly synchronous, for this reason it is not possible to manage parallelism without using multiple threads.

- **Database Middleware**: The Database middleware are designed to mask the heterogeneity of databases proving a common interface to the application level. It is indeed possible to have multiple different databases in the network nodes but to use them as a single shared resource.

- **Message-Oriented Middleware**: In the Message-Oriented Middleware (MOM) the distribution of code and data is managed with the message passing paradigm. This kind of software takes care particularly of the communication and synchronisation between network members, and they can provide support for persistent messages, replication and real-time performances. This class of software is able to provide spatial and time decoupling and manage various *Quality of Service* (QoS) specifications. JMS, Microsoft MSMQ and IBM MQSeries are example of MOM systems.

- **Distributed Object Computing Middleware**: Distributed Object Computing middleware (DOC) is designed to manage the distribution of code and data using the object abstraction. In other words, it is possible to invoke methods of remote objects just like they are of local object. From a software engineering point of view, the development of distributed applications over a DOC middleware can benefit of the same features and positive aspects of the object-oriented programming techniques. The central part of this kind of software is the *Object Request Broker* (ORB) and it is the component capable to resolve remote object invocation and find proper services for a certain request. In the **Figure 1.2** is given a simplified representation of the CORBA architecture and interactions between a client and a distributed object. The ORB core is the essential player in the CORBA environment and there are many different implementations with different tasks and objectives. CORBA is an example of DOC middleware as COM, .NET and EJB.

Most of the middleware software does not require the user to learn a new programming language to use its functionalities. They are generally designed to work with common languages, such as Java, C++ and C#. Currently there are three main ways to program a middleware: a dedicated library of functions, an external interface definition language (IDL) or the language supports natively the distribution, for instance Java Remote Method Invocation (RMI). The first method allows the user to call middleware functionalities and services through a library software invok-

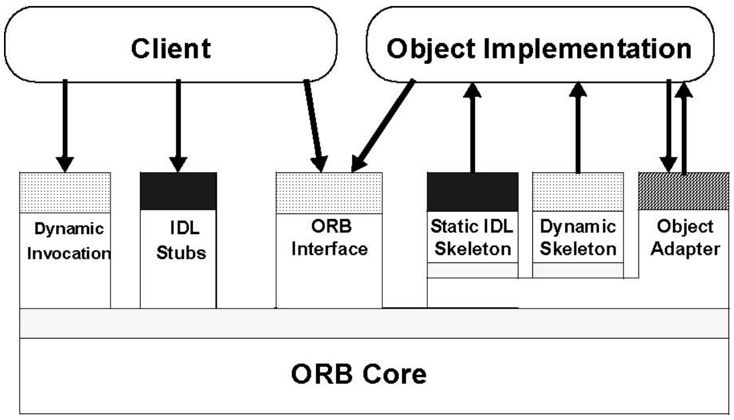


Figure 1.2: Simple representation of the CORBA architecture and its basic components.

ing certain pieces of code. The second way is more flexible. The developer has to define IDL interfaces, that means to add a medium level between the middleware and the application level and so it is possible to use multiple languages inside the same distributed system. CORBA uses IDL and supports a large number of programming languages. The last one is not very flexible but from a performance point of view is clearly faster then the IDL mechanism.

Another fundamental task of the middleware software is to exploit the abstraction level provided to manage resources of the distributed system. So, the middleware could abstract the three kinds of low-level resources of an operating system: communication, computation and data storage. In addition to a common operating system, a middleware provides a complete view and management of these resources within the distributed system. As mentioned earlier, the middleware software is classified depending on the services that they provide and how they manage synchronisation and data sharing [3]. It is possible to revise the classification considering the resources abstraction which they provide as it shown in the **Table 1.1**.

In addition to the resources management, there is another fundamental concept to mention about distributed systems: the Quality of Service (QoS). The QoS is a new concept and it was introduced to describe the per-

Middleware Category	Communication	Processing	Storage
Remote Procedure Call	Yes	Yes	No
Database Middleware	Yes	Limitated	Yes
Message Oriented Middleware	Yes	No	Limited
Distributed Object Computing Middleware	Yes	Yes	Yes

Table 1.1: Middleware categories support for low-level resources.

formances provided to the final user of distributed services. This new concept helps to manage more properly the dynamic nature of a distributed system. In few words, the goal of this idea is to capture the high-level requirements and to translate it in low-level resources management. So, it is not a static decision but it is something to modify and to adapt at run-time. In this way the distributed system can adapt its resources and services but it could also adapt itself to new requirements. For this reason, the QoS is considered a novel concept for software engineering and maintenance of distributed systems.

The middleware software is extremely suitable for the QoS concept. It is indeed possible to exploit its programming abstraction in order to introduce the QoS in the system and to also provide a simple way to manage and configure it to the developer. Generally, the DOC middleware excels at QoS management and CORBA is particularly suitable for this task. Therefore, a distributed application supported by a QoS management is more stable because the available resources are variable based on the current system state. On the one hand, the range of variation of the resources is limited and well-known, so the application works in a safe context. On the other hand, the application is more flexible because the resources configuration is not *hard-coded* in the source code. Further, the decoupling provided by the *QoS-enabled* middleware allows the developer to not focus on the low-level protocols or *APIs* used to provide the actual QoS to the final user.

1.2 Middleware for Mobile Distributed Systems

The thesis project aims to provide a middleware offloading service for distributed applications but, in particular, it is interested in mobile settings. For this reason this section focuses on mobile distributed systems and their novel challenges.

1.2.1 Mobile Distributed Systems

Smartphones and tablets have become sources of huge amount of data and they are the first choice as personal device for most people. There are a large number of applications for mobile devices which allow to share information between users in order to provide new and rich experiences, not only referring to social network and behaviour.

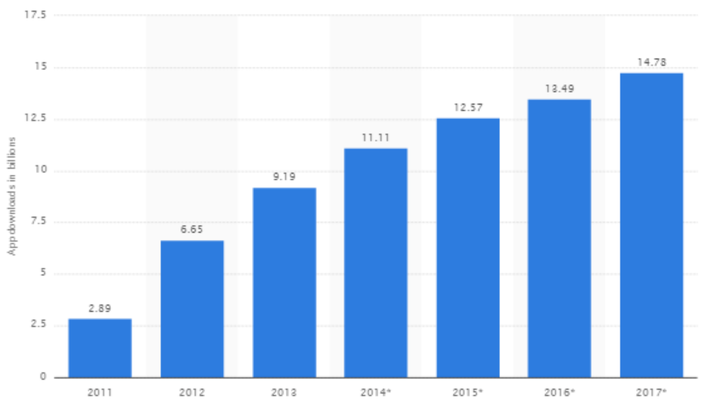


Figure 1.3: The graph show the amount of downloads of mobile applications in the world.

Most of the enterprises and companies have already invested their efforts on mobile applications or they are planning to extend their business in this direction. The **Figure 1.3** shows the number of mobile apps downloaded on devices across the globe from the year 2009 to the projection in the year 2017. The market of mobile apps will keep on increasing in the next years and with it also the revenues. The global mobile applications market is strongly influenced by the mobile systems technologies and by their manufacturers, such as Apple, Google, Samsung and LG. Initially, the mobile applications were designed as thin clients for web services, such as e-mail or messenger apps. Nowadays, the devices are always connected to the Internet and they are equipped with a good amount of hardware resources. In the last years the industries interest has been moving to more complex mobile services requiring advanced tools for the app development. Applications of this kind introduce new challenging problems:

- *High mobility*: The mobile devices could temporary and asynchronously lose the network connectivity for many reasons. It means slow connection sessions and so the need to discover other hosts dynamically.

- *Limited resources*: As mentioned earlier, the hardware available in the current mobile systems is progressively increasing. However, they have scarce resources compared to other computing systems, such as slow CPUs and little memories. Further, they have to deal with the battery consumption which is an unavoidable constraint for mobile application;

- *Dynamic changes*: The environment can change faster than in common systems, that could be related to the location or the context conditions of the user. In addition, the network connectivity could strongly vary without ceasing and it might introduce delays or desynchronisations.

1.2.2 Mobile Middleware

As mentioned in the **Section 1.1**, the developer should not focus on the distribution problems, such as heterogeneity and resource sharing. The traditional middleware software indeed takes care of this issues and provides high-level tools. In the same way, the mobile middleware services also have to support the high mobility for mobile distributed application

[4]. An existing solution is to use a mixture between totally fixed and totally mobile systems. This kind of distributed systems is named *Nomadic Distributed System*. In few words, the mobile devices move from a location to another maintaining an active connection to a fixed infrastructure as it shown in the *Figure 1.4* [1]. The nomadic distributed systems are particularly interesting for the purpose of the thesis. This project will indeed integrated in a nomadic distributed system, Avatar [5]. Avatar will be presented better in the **Section 3.1**.

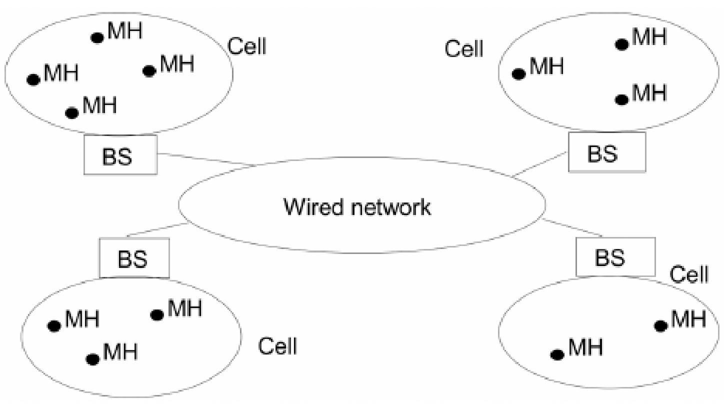


Figure 1.4: Topology of a Nomadic Distributed System network.

The backbone network is the core of distributed nomadic system, it is able to provide services to the mobile devices and to carry the computation and connectivity load. In this scenario it is possible to manage transparently the network disconnections or variations. In addition, dedicated services for re-synchronisation are provided. Some of the requirements shown in the **Section 1.1** for fixed systems are still valid for the nomadic distributed system. However, the heterogeneity in this scenario is complicated by the presence of node with different nature, mobile and fixed. It is also possible that different mobile connections coexist in the same network. Finally, the scalability is similar to the fixed system, in other words, it is the capability to manage a large number of mobile devices at the same time. The fault tolerance depends on the application purpose, in some cases it might be possible that a network disconnection is considered not a fault but just a regular behaviour of the program. The scenario could vary additionally if the mobile devices could assume the role of service providers as the backbone servers. In this case the result would be strongly different in terms of discovery, quality and failure management.

The alternative to the nomadic distributed systems is the Ad-hoc mobile distributed system. Since one requirement of the thesis is to be integrated with the Avatar project it is not really fundamental to explain this topology of systems. However, the main difference of a Ad-hoc system respect to the nomadic systems is the absence of a fixed infrastructure. So, the devices could join different clusters and communicate with the other members or in some case use the intra-cluster communication. The system is highly dynamic and generally more complex to manage.

The Ad-hoc and nomadic systems are similar for many aspects, for this reason some mobile middleware are feasible for both models [1]. Generally when the choices which influence a mobile middleware design are related to three main aspects:

- **Computational Load**: Unlike the fixed systems the hardware and so forth the computational resources available in mobile devices are relatively scarce and limited. Therefore, the middleware services designed should be lightweight and not impact negatively on the performance. It is indeed worth to sacrifice some functionalities like replication or intensive synchronisation operations.

- **Asynchronous Communication**: It is ordinary that a mobile device access to the network only when it needs some data or service. As mentioned previously, during the interaction with the service or data provider the connectivity could vary by orders of magnitude in the time. In addition, the connection could cease totally. It is important to provide a communication base on asynchronous requests and responses. In this way even if the device lose the connection it can reconnect later and collect the result.

- **Adaptability**: This characteristic is similar to the previous one but it is correlated with the availability of services. The dynamic and unpredictable nature of this kind of system causes new scenario respect to fixed ones. A mobile device indeed could lose the connectivity with a service provider. For that reason the middlware behaviour should manage this situations and it should use *context-aware* techniques to manage better dynamic scenarios.

1.3 Related Works for Mobile Middleware

This section aims to show some related works on the mobile middleware scenario. Many common middleware solutions has been adapted to the mobile environment. This traditional middleware software has been modified in order to be lighter-weight and to be suitable for the mobile nature of these networks.

As mentioned in the **Section 1.2.2**, a main challenge in the mobile distributed systems is the reliability and the network variations. Some of the traditional middleware level software could be modified in order to fit this issues. In this section it will be shown how the different categories of middleware could handle the problem with various approaches.

Object-oriented middleware is the most spread category of this software level. For this reason the community has adapted them to the mobile distributed systems in order to exploit their advanced features. This task has been not simple because this software is generally heavy and it does not support mobile protocol by default. As mentioned previously, CORBA [6] is an important example of DOC middleware and so the developers has worked on it trying to make it suitable for the mobile environment. As shown in the **Figure 1.2**, the core component of the CORBA architecture is the ORB, in particular the Internet Inter-ORB protocol (IIOP) [7] allows the communication among devices. This protocol has successfully been ported to mobile settings and it could be used as a simplified version of the standard ORB. An example is the DOL-MEN project [8] which manage medium wireless unreliability using the Light-weight Inter-Orb protocol (LW-IOP). In addition, in order to keep track of hosts location the names of the machines are translated using a dynamic naming server. This mechanism partially solves the mobile middleware challenges stated in the **Sec-**

tion 1.2.2. CORBA and IIOP are combined with the Wireless Access Protocol (WAP) [9] stack in order to use the standard CORBA services on the fixed network with mobile devices connect through WAP and a gateway. In this configuration IIOP manages the message exchange functionalities [1].

For what concerns the Message-oriented middleware there are many successful attempts to port them in the mobile environment. Java Message Service (JMS) [10] has been successfully ported to mobile distributed systems. As the standard edition it supports both point-to-point and publish/subscribe models. The publish/subscribe paradigm is based on topics, a device could register on a topic and publish or be notified every time a message is delivered to its queue. For the point-to-point communication a mobile message can directly send a message in another device queue. Summarizing, JMS is strongly based on queues, the topics could be considered as multiple listener queues. In the **Figure 1.5** is given a really simple but representative representation of the JMS architecture. The MOM is a good solution for asynchronous communication. In particular the publish/subscribe model would be a good support for disconnected operations and distributed services.

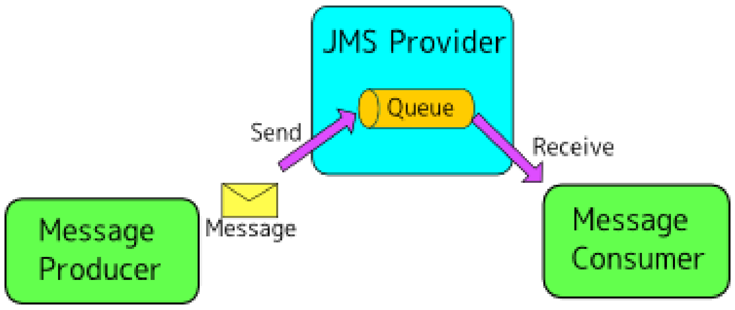


Figure 1.5: A general representation of the high-level architecture of Java Message Service.

The communication and message passing paradigm are the main feature for mobile middleware in nomadic distributed systems. However, as mentioned earlier the adaptability and the dynamic variation of the context are really important for what concerns the QoS management. It is fundamental that a middleware level services is adaptable to different scenario in order to guarantee an established Quality of Service. A very relevant example in QoS-oriented middleware is Mobiware [11], it is based on CORBA and IIOP and developed in Java. It is capable to adapt the system in order to maintain a certain level of QoS. A mobile device is seen as a terminal node of a network, the main operations and services are offered by the Asynchronous Transport Network (ATM). Mobiware is based on the hypothesis that every device connectivity is fluctuating but almost continuous. So, a every device is always directly connected to an access point and it could switch to a new access point in order to keep the connectivity with the backbone network.

1.4 Middleware for Mobile Computing

The fixed and mobile distributed systems have different issues and peculiarities. In the **Section 1.2.2** is shown how there is possible to merge the two architectures in order to benefit of their features. The *nomadic distributed systems* are the implementation of this concept. The combination of cloud computing, mobile computing and wireless networks is commonly defined as *Mobile Cloud Computing (MCC)* [12]. Before deepening into the specific model of the Mobile Cloud Computing, it is necessary to give an overview about the concept of *Cloud Computing (CC)*. It is a model which enables the execution of heavy computation applications using a remote resources provided on-demand through the Internet. The main goal of the Cloud Computing is to supply quickly the necessary resources and to decrease their management costs [12]. The Cloud Computing could be divided in three different layers of services:

- *Software as a Service (SaaS)*: The SaaS level is the higher of the layers. It provides applications, such as a web application or a mobile application, to manage the software which is running on the cloud architecture. The user of this services is totally unaware of the complexity of the distributed systems and he does not need to concern about system configurations.

- *Platform as a Service (PaaS)*: The user may need more control on the cloud architecture, for this reason the PaaS level allows the user to customise some configuration. The user has the freedom to select the operating system to deploy on the servers and then he can proceed in the software development.

- *Infrastructure as a Service (IaaS)*: This level is dedicated to more expert users. In addition to the operating system he can set up the computation environment. The user has at his disposal a powerful processing core and a considerable amount of data storage in order to execute heavy computational software.

The Cloud Computing model is designed to achieve the following features:

- *on-demand*: the architecture provides only the necessary resources to the user software;

- *elastic*: the architecture is highly scalable and it can adapt the provided resources to the dynamic need of the user application.

- *quality of service guarantee*: it is able to supply the QoS necessary to the application.

- *pay-per-use*: the user is charged only for the cost of the actual used resources.

As mentioned earlier, the availability of wireless network enables mobile devices to be almost always connected to the cloud. The Cloud Computing has evolved to support mobile distributed application and it generates a the Mobile Cloud Computing model. An overview of the Cloud Computing topology is shown in the **Figure 1.6**.

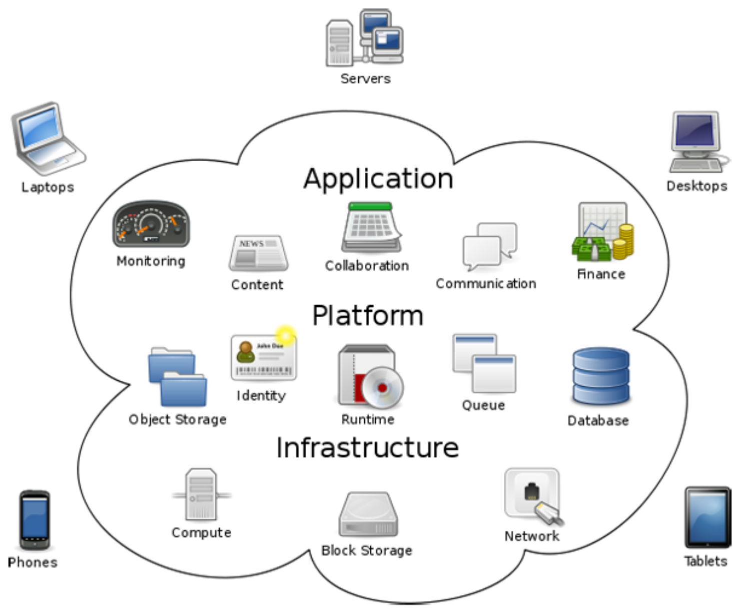


Figure 1.6: A simple representation of the Cloud Computing topology with some example services.

1.4.1 Mobile Cloud Computing

The Cloud Computing can enable a mobile device the execution of complex and heavy applications supplying the needed resources. For instance, an employee could bring his personal device to his workplace, and use it to access privileged enterprise content and applications stored in the cloud [12]. The Cloud Computing can provide scalable mobile computation and support the big data mobile applications. Furthermore, the appearance of the handsets benefits of the cloud support, the manufacturers can indeed rely on the cloud resources limiting the sizes for their products.

The Mobile Cloud Computing could be define as a computational model combining mobile computing and the cloud, where the cloud can handle large storage and processing for mobile devices remotely [12].

The architecture of the Mobile Cloud Computing is based on the *Mobile Computation Augmentation* model. The *augmentation* is the process to improve, to enhance or to optimising computation capability of a mobile device by leveraging varied feasible approaches, hardware or software [13].

The hardware approaches involves the use of high-end physical components, such as CPU, memory, storage, and battery.
The software approach could be the computation offloading, remote data storage, wireless communication, resource-aware computing or remote service request. The applications which can benefit of the Mobile Computation Augmentation are generally computing intensive, data intensive or communication intensive. An example application for the first kind is a speech or image recognition software, for the second kind it could be an enterprise software and for the last kind it could a video streaming software. A particular kind of Mobile Computation Augmentation is the Mobile Cloud-Based Augmentation [13]. It is a synonymous of Mobile Cloud Computing. It is important to explore what are the advantages of such approach:

- *Empowered Processing*: The mobile device can perform virtually a task using the support to the computation. In few words, a device could be not able to complete a task because the CPU or the memory could be not enough or the battery could be too low. So, the mobile device can offload the entire application or part of it to the cloud. In this way the device could perform the task and deliver the results of applications beyond its native capabilities. The Cloud can guarantee a higher-level reliability and availability than the hardware solution or another architecture.

- *Battery Conservation*: The hardware *augmentation* can be achieve using very powerful CPU or memory. The side effect of this solution is the power consumption. Improving the performance often means to manufacture new components which need an higher amount of energy to be empowered. The Mobile Cloud Computing can save battery offloading tasks to the Cloud. Migrating the code would help the device to perform computation intensive application but also to save energy. Therefore, a fundamental task of the MCC is to optimise the energy consumption. The execution migration could potentially decrease the energy consumption but it is important to consider that in some scenarios the mobility support and the resource elasticity can neutralise the offloading effects.

- *Expanded Storage and Data Safety*: Accessing data in the Cloud could seriously improve the storage capability of a mobile device. The requirement is to be connect to the Internet in order to upload or download data. In addition, it is necessary to synchronise manually or automatically the device with its relative cloud storage space. Furthermore, storing data on a mobile device could be not totally safe, it is possible to lose the device or that a malicious person could access to personal data. Device malfunctions or physical damages are other causes of data loss or corruption. It is also not possible to apply encryption on mobile device storage because it could cause too much overhead and it could negatively affect the responsiveness. Therefore, using a reliable cloud storage, the user can avoid the issues related to physical damages of the hardware or robberies. In addition, the data stored in the Cloud are ubiquitously accessible. The user can change the device or the location and be sure that it can always rely on the cloud storage.

- *Protected Offloaded Content*: The MCC is based on *virtualization* technology to isolate the user execution environment from other users and also from the infrastructure stack software. The Cloud infrastructure could provide powerful encryption technologies to protect the user data from other users of the platform.

- *Heterogeneity*: the application development for MCC applications could be simplified in terms of heterogeneity management. A cloud component indeed could be built once and it can be used for different mobile target platform, such as Android, iOS or Windows Phone. A company could save money and resources avoiding the multi-platform development for a single application.

The Mobile Cloud Computing model could be implemented in different ways. It is indeed possible to identify three main categories related on the position of the cloud resources.

- *Distant Immobile Cloud*: This infrastructure is composed of private or public cloud services based on a large number of stationary servers located in vendors or enterprises properties. The advantages of this typology of infrastructure is the high availability, reliability and elasticity of the cloud resources. On the other hand, public cloud resources could be subjected to security glitch. So, it is possible that malicious users exploit some weakness of the infrastructure to access to private data of other users. Furthermore, the WAN latency related to the distance between the two endpoints of the communication generates sometimes considerable delays on the interaction.

- *Proximate Immobile Computing Entities*: This infrastructure is composed of stationary computers located in public areas nearby the mobile devices. Nowadays, the amount of computers in public place is huge, and this machines are often executing light task wasting most of their resources. The concept of this infrastructure is to exploit such computational power to augment the mobile devices capabilities. So, on the one hand they can assist the computation of mobile devices taking charge of their task. On the other hand, since the distance between this computers and the mobile devices is small, they can reduce latency and wireless network traffic. This infrastructure also suffer of security issues.

- *Proximate Mobile Computing Entities*: In this infrastructure every single mobile device can play the role of cloud-based server for other devices. So, devices can cooperate to obtain heavy task migrating execution to other members of the group. The limitation of this infrastructure is also related to the device hardware specification.

1.4.2 Existing Work on Mobile Cloud Computing

This section aims to give some examples of existing works about the Mobile Cloud Computing. In the **Section 1.4.1** it is shown that is possible to categorise the Mobile Cloud-Based Augmentation in three macro groups: *Distant Immobile Cloud*, *Proximate Immobile Computing Entities* and *Proximate Mobile Computing Entities*. In order to distinguish better this three categories, this section lists some example per each of them.

Avatar [5] is a Distant Immobile Cloud project. In particular, the thesis project will be integrated with the Avatar system. For that reason more details are given in the **Section 3.1**. However, briefly, Avatar is a system designed to leverage cloud resources to support fast, scalable, reliable, and energy efficient distributed computing over mobile devices. The system assigns resources to a single user as a dedicated virtual machine which supports the user mobile device computation.

CloneCloud [14] is another example of Mobile Cloud-Based Augmentation. The CloneCloud system is a system capable to migrate entire mobile platforms into the cloud infrastructure and runs mobile applications without requiring any kind of modification in their original source code. It is designed to be fine-grained and thread-level, it can indeed execute locally the remaining application when the cloud infrastructure is running the intensive computation tasks. However, if an application needs to access to a shared memory the system is able to migrate it to the cloud. The CloneCloud infrastructure executes the mobile applications as distributed computing code, however the developer is totally unaware of this behaviour. As mentioned earlier, this system is capable to offload local threads to the servers, the local execution will proceed normally but the intensive computation tasks are execute remotely. When CloneCloud migrates a thread it is also able to block other local threads if they have a shared state with the first one. An overview of the CloneCloud architecture is shown in the **Figure 1.7**.

One the one hand, the overall execution time could be sensibly reduced by the threads migration.
On the other hand, the communication overhead required to move the entire mobile platform, the application code, and the memory state can mitigate the advantages of the *augmentation*. Furthermore, in order to keep the state of the execution synchronised the CloneCloud systems has to deal

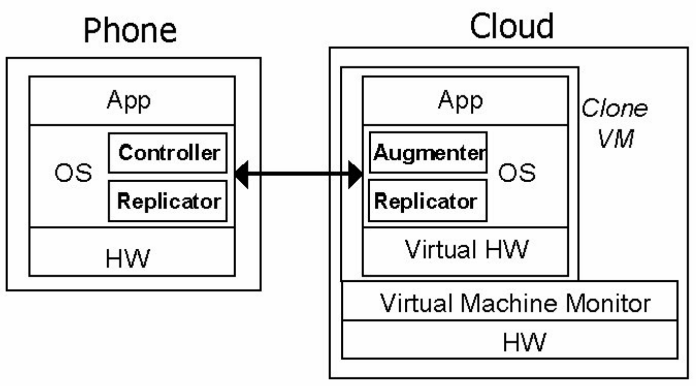


Figure 1.7: CloneCloud architecture with the system components used to support the execution on mobile devices.

with multiple communication which are a further overhead.
Summing up, the CloneCloud model has the most relevant benefits when the computation is substantially heavy and the overhead generated by the communication is lighter. This is a common consideration for computation offloading and general mobile computation augmentation, and the **Chapter 2** gives further details about this trade off.

Cloudlet [15] is an example project of the Proximate Immobile Computing Entities model. A variable number of computers collaborate and communicate in order to augment the devices performances, this entities takes the name of Cloudlet. So, a device using a WiFi network can offload directly to the Cloudlet its heavy computation tasks. The proximate infrastructure could even be supported by a distant cloud layer in order to manage backups and help the Cloudlet in case of lack of resources. The advantages of this approach is that it is capable to reduce the security risks, to provide a one-hope offloading mechanism and in addition to minimise the time of communication. The devices are thin clients of the Cloudlet infrastructure, it is possible to see this architecture similar to the client/server model. The resources are managed by an virtual machine abstraction. The Cloudlet could also exploit the hardware virtualization technology in order to guarantee more security and privacy of the mobile device user. Furthermore, this technology could simplify the management of the heterogeneous resources at the same time. The motivation to use VM is the possibility to isolate the offloading mobile environment from the permanent software of the Cloudlet physical computers. The application scenario of the Cloudlet project is a public place as a restaurant or a shop which uses the underexploited computing resources to augment the nearest mobile devices, for instance costumer handsets.

The main difference with CloneCloud is that Cloudlet does not need to migrate the entire OS but the mobile environment should be already running on the VM abstraction. Unlike CloneCloud which transfer the entire mobile device memory stack in the beck-end cloud infrastructure, Cloudlet use lightweight software interface for the heavy computation components, the authors decided to call it VM overlay. In the **Figure 1.8** is possible to see that Cloudlet has three levels. The first level is dedicated to the device which need to be supported by the cloud. In the second there are all the proximate machines described earlier. However, there is also a third level of fixed cloud services capable to support the heaviest tasks.

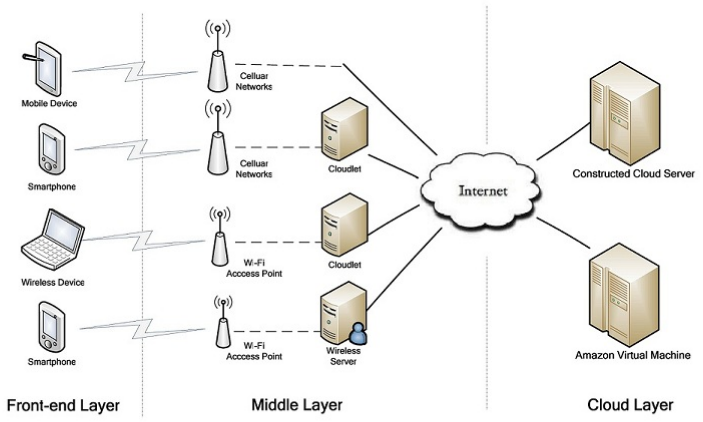


Figure 1.8: The tree layers of the Cloudlet architecture.

On the one hand, the success of this project has been made by the flexibility of the model. It indeed allows to private owners of unused resources to deploy the Cloudlet abstraction on their own computers. That is possible because the cost of the maintenance, the security and privacy level, and also the energy consumption.

On the other hand, from a mobile user prospective the delay generated by the offloading mechanism is sometimes unacceptable.

MOMCC [16], or Market-Oriented Mobile Cloud Computing, is an example of Proximate Mobile Computing Entities model. MOMCC is based on the Service Oriented Architecture (SOA) [17]. In few words, MOMCC creates a cluster of mobile devices in order to run resource-intensive tasks. A mobile cloud computing application for MOMCC is a composition of prefabricated blocks, services, developed by expert programmers. It is possible to develop independent services and publish it on the Universal Description Discovery and Integration (UDDI) [18]. UDDI is a discovery service use by the SOA architecture, in this way the services published are visible to the mobile devices. For instance, a provider of an UDDI service could be a mobile network operator.

The actual execution of services is made by a large number of nearby devices, which can be reward for their resources sharing with money. The model provides a distant cloud support in case the available resources around the service request are not enough for the task.
Summing up, the mobile devices which want to provide some resource sharing have to register with the UDDI and negotiate the services to host. Mobile devices which want to receive support for resources-intensive tasks could query the UDDI for a service at run-time. The service applicants have to pay for the provided support.

MOMCC is a not complete project, the Proximate Mobile Computing Entities model is an open challenge.
On the one hand, the business model proposed is not still considered a applicable to the real market but potentially it could be a good source of incomes for mobile device owner.
On the other hand the security and privacy issues are still a sensible problem to solve for the MOMCC authors.

1.5 Application Examples for Mobile Cloud Computing

As mentioned in the **Section 1.4.1**, many applications can benefit of the Mobile Cloud Computing technologies. Developers of mobile resources-intensive applications can design thin clients and move all the heavy tasks on the cloud-based background infrastructure. The aim of this section is to show some application examples suitable for the MCC model.

Lost Child [5] is an application developed by the Avatar team. The application is based on image recognition algorithm and so it is computation intensive. Lost Child needs a considerable amount of communications which makes the Mobile Cloud Computing more suitable to support the execution. The application aims to find lost child in crowded city areas. In more details, a parent lost his child in Time Square, Lost Child searches the child face in the recent pictures taken in the surrounding locations by other users. The application reconstructs a real-time trajectory of the child movements according to the positive matches. Naturally, in order to be analysed a picture should belong to a user which wants to contribute to the application tasks.

The face recognition software is based on heavy computation and the cloud infrastructure could help to execute it faster without affecting the users battery duration. Furthermore, the lost child parent needs to broadcast the research request in order to ask help. Without a back-end infrastructure the broadcasting will consume a lot of bandwidth and energy, the Mobile Cloud Computing can take care of this task and make the request on the device behalf. In this way the parent device could last more allowing the user to make more request and eventually have a bigger time window for the participants replies.

FaceDate is another application developed by the Avatar team. It is based on the face recognition but the application scenario is totally different. Essentially, FaceDate helps the users to find a date based on their aesthetic preferences. A user can specify some sample pictures of faces which he likes. The Face Date application will search if there is some possible match for the user tastes. The match is done in both direction, for instance the first user search for a date, Face Date finds a suitable match in the second user profile picture. Before presenting the result to the first match he analyse the second user preferences and the first user profile image, if there is a match it will show a positive result to the first user.

In this case the computation is also heavy because the face recognition algorithm requires many resources. The broadcasting required to start a research would be bandwidth and energy consuming. For this reason also in this case the Mobile Cloud Computing is a suitable support for the application. The Face Date app would consume less energy, run faster and save bandwidth. In this way the user experience would be severely improved.

Chapter 2

Overview of Offloading in Mobile Cloud Computing

This chapter aims to introduce the concept of offloading, main topic of the thesis. First of all, a general idea and a definition of the offloading is given in the first section. More in details, this section gives information about the offloading decision evaluation and the main challenges to overcome in order to design a complete and efficient software. The second section enlists and compares projects and works on this topic.

2.1 Offloading

As mentioned in the **Chapter 1**, despite the fast and constant growth of the mobile hardware specifications this kind of devices are still limited compare to standard computing machines. This restrictions are produced by various traits of these devices: they are powered by battery, equipped with relatively slow processors and poor memories. In addition they are dependent from wireless networks, their bandwidths are order-of-magnitude lower than wired standard networks. The available mobile applications are ceaselessly requiring more computing resources and network bandwidths [19]. In the same time the energy required by current apps is increasing and the current batteries are not capable to keep the device operative for a long period.

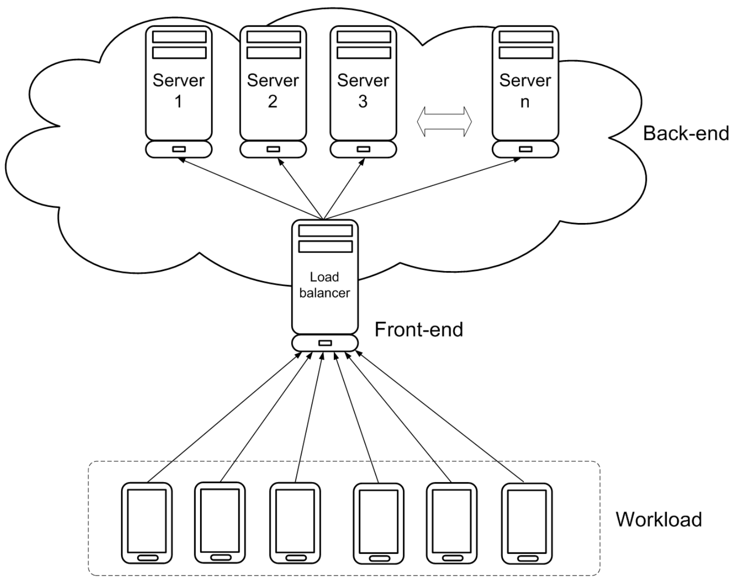


Figure 2.1: Offloading base architecture, limited mobile devices offloads their computation to a back-end cloud service.

Offloading is a feasible solution for the presented issues. Offloading means to migrate computation from a device with limited resources to a more resourceful computer, for instance a server as it shown in the **Figure 2.1**. It is essentially different from the standard client/server model, where a simple and light client always migrate computation to the back-end server. The offloading could be seen as a similar model of the processes migration used to load balance multiprocessor complex systems. However, they are strongly different. The offloading transfer computation outside the user resources to another external resource, the migration of processes in multiprocessor is done in a single real or virtual machine. Another adopted synonymous for the computation offloading is *cyber foraging*, they both express the same concept.

The offloading has been and it is still an important matter of research. Initially it was studied and deepened by research teams and industries in order to enlarge the capabilities of laptops and limited computers. However, in the last period the research has been developed to focus on mobile systems. As stated previously, mobile devices are a perfect target for computation offloading research. In addition virtual computation is currently more efficient and effective than in the previous years. The progress on this technology commits strongly to the offloading technique.

Therefore, the goals of the cyber foraging is to augment mobile devices capabilities using powerful machine to offload intensive computation tasks. However, it is not always true that offload computation on a back-end machine would improve the performances, it depends to a large number of parameters and conditions. Further, in some environment the bandwidth is more precious than performances or battery duration. In this direction has been proposed many algorithms to analyse and take efficient offloading decision. As will be shown later, the offloading could be designed in many different theoretical ways. Naturally, these choices are led by the specific application of the software.

Since the offloading technology is a tool to obtained the already mentioned goals in terms of performances and battery optimisation, it is important to schedule the remote execution using smart and effective rules. *What is exactly an offloading decision?* It is an answer to the question *when* and *what* to run on the remote endpoint. This challenge has driven many researches about algorithms and heuristics to establish a global or a more specific rule to reply to this question. As a consequence of this academic studies, it is possible to abstract two general rules capable to give a final decision for the remote execution: performance gain and battery consumption optimization

2.1.1 Performances Heuristic

As stated previously, the performances are a core goal for the offloading mechanism. Mobile devices are certainly slower of a server machines, for that reason may be simple to decide to always offload the code. Actually, this is not a wise answer for every situation. It is important to understand which are the conditions to migrate computation to the back-end server. The first step is to state that every program is divisible in two macro parts: code which has to be executed locally and code that might be offloaded. In the one hand, there are many reason because a code cannot be remotely migrated:

- *User Interfaces*: The user interaction code is practically impossible to migrate on another machine. The user device should provide and manage the interfaces and the user related events. It is really an overkill task and it is not reasonable to avoidable introduce overhead and delay in order to migrate these tasks.
- *I/O Logic*: Like for the user interfaces, the I/O accesses should be managed by the user device. In other words, sensors, file system or external hardware should be managed locally.
- *Not Repeatable Logics*: It is very risky to offload code that manage an unrepeatable interaction with external resources. A clear example is a request to a bank web service, the result of a double invocation would considerably damage users wallet. The offloading might fail for many reason, loss of connection, failure of the beck-end architecture or state inconsistencies.
- *Privacy Policy*: It is often forgotten that the user may have some preferences in terms of privacy. In this case the offloading decision could change based on privacy policies and trust on the beck-end architecture.

In the other hand, any type of code which does not fit the previous categories could be included in the second category. Let define the local time of execution of the *offloadable* code as

$$time_{local} = \frac{w}{speed_{mobile}}, \tag{2.1}$$

where $speed_{mobile}$ is the mobile device speed and w is the amount of computation for the second part of code [19]. At this point is necessary to compute the time of execution for the same code for the beck-end architecture. In this case it is necessary to also consider the time needed to transfer data to the server in order to resume the execution state.

$$time_{comm} = \frac{data_{input}}{B}, \tag{2.2}$$

where B is the available bandwidth and $data_{input}$ is the amount of data input to transfer. The time of the remote execution could be defined similarly to the **Equation 2.1**.

$$time_{remote} = \frac{w}{speed_{server}}, \tag{2.3}$$

where $speed_{server}$ is the back-end server speed. Finally, the offloading technique improves the program performance if the following relationship is true:

$$time_{local} > time_{remote} + time_{comm};$$

$$\frac{w}{speed_{mobile}} > \frac{w}{speed_{server}} + \frac{data_{input}}{B}.$$

After some simple algebra it is possible to rewrite the previous Inequality in the following form:

$$w \times (\frac{1}{speed_{mobile}} - \frac{1}{speed_{server}}) > \frac{data_{input}}{B}. \tag{2.4}$$

The **Inequality 2.4** leads some considerations:

- If the ratio $\frac{data_{input}}{B}$ is considerably large the server speed not affect the performances gain. In this case the decision should be to avoid the migration of the code;
- The perfect task for the offloading gain in terms of performances has heavy computation and limited data to exchange;
- The network bandwidth is an important parameter of the offloading decision;

2.1.2 Battery Optimisation Heuristic

In the **Section 2.1.1** it is given a general rules to evaluate the gain in terms of performances of the offloading technique. The battery consumption is often a big problem for mobile devices and despite the advanced technologies, the battery capacity is still not enough for the user requirements. Many surveys shows that the battery duration of a device is the most important features for users, look in the **Figure 2.2** for further information.

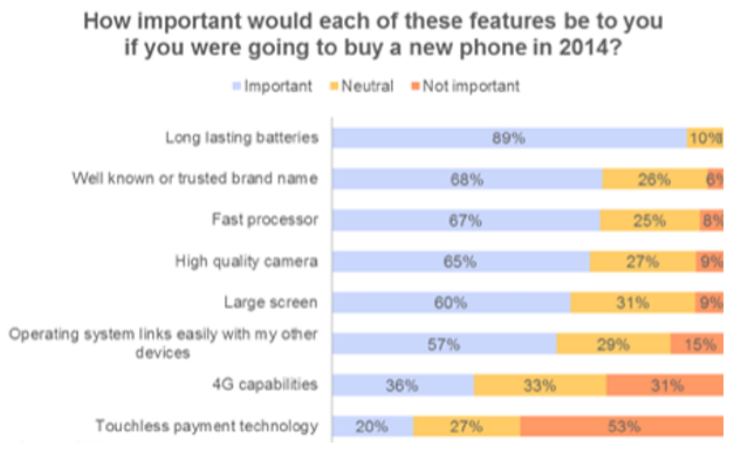


Figure 2.2: Handsets features voted important, neutral and not important by British users in GMI's survey, April 2014.

The cyber foraging is potentially a good solution to this problem, migrating the intensive computation on more powerful machine might indeed improve the battery duration. As the performances evaluation, also in this case is necessary to understand when the offloading saves really energy. The two variables $speed_{mobile}$ and w are already define in the **Equation 2.1**, let $power_{mobile}$ be the power available in the mobile device. A modified version of the **Equation 2.1** could be defined in order to obtain the energy needed to execute the second part of the code, defined in the **Section 2.1.1**.

$$energy_{local} = power_{mobile} \times \frac{w}{speed_{mobile}}. \tag{2.5}$$

As mentioned earlier, the communication is the necessary price to run the code on the back-end server. So, it is essential to compare the cost and the gain before proceed with the migration of the task [19]. The energy needed to transfer the code and poll the result is:

$$energy_{comm} = power_{sending} \times \frac{data_{input}}{B} + power_{polling} \times \frac{w}{speed_{server}}, \quad (2.6)$$

where $power_{sending}$ is the power to transfer the information, $power_{polling}$ is the power to wait the reply keeping to poll the network for a result and the other variables are already defined in the **Section 2.1.1**. In order to decide if the offloading will save energy or not the relationship between the **Equation 2.5** and **Equation 2.6** is:

$$energy_{local} > energy_{comm};$$

$$power_{mobile} \times \frac{w}{speed_{mobile}} > power_{sending} \times \frac{data_{input}}{B} + power_{polling} \times \frac{w}{speed_{server}}.$$

After some algebra the resulting equation is:

$$w \times (\frac{power_{mobile}}{speed_{mobile}} - \frac{power_{polling}}{speed_{server}}) > power_{sending} \times \frac{data_{input}}{B}. \quad (2.7)$$

The **Equation 2.7** is similar to the **Equation 2.4** and it leads the same considerations:

- If the $data_{input}$ are large the speed of the server does not influence the result and even with the most powerful machine available the offloading does not save energy.
- The offloading save energy when the computation of the task is heavy and the data to synchronise are not many.
- The bandwidth is an important parameter also in terms of energy optimisation.

The final boolean decision, whether to run the code on the back-end or not, depends strongly to the bandwidth available on the mobile device.

The WiFi networks are generally a good scenario for the offloading because they often provide high bandwidth. However, when the connection is based on a cellular network the result may be different, this kind of network not always are capable to supply fast communication and they could be a financial cost for the user. The evaluation of the previous rules should be made always before the migration to avoid waste of performances, energy and also money.

2.1.3 Offloading Challenges

In the **Sections 2.1.1** and **2.1.2** a global model for offloading decisions was given. However, a decision could be taken in a static or dynamic way. On the one hand, a static decision is taken at development time, in few words, the program is partitioned during the implementation. This kind of decision is based on the strong hypothesis that the parameters are predictable accurately. In particular, the parameters $speed_{mobile}$, $power_{sending}$, $power_{polling}$ and $power_{mobile}$ of the **Equations 2.4** and **2.7** are generally estimable with decent accuracy. It is also possible to know the $speed_{server}$ if the server is own by the developer or the provider guarantees a precise level of performance. The real variable of the previous equations are the $data_{input}$, the w and the B. It is also possible to predict them using some statistic algorithms for data prediction or history-based machine learning. On the other hand, dynamic decisions are certainly more adaptable to different run-time and context configurations. As the static decision, also this kind of evaluation could be based on prediction technique, such as Bayesian scheme, neural networks and more advanced machine learning mechanisms. Unlike the static decision the dynamic ones are generally cause of higher overhead on the machine. The decision making model is indeed based on parameters monitoring that is clearly an active job for the device. It is common also for the dynamic decisions to partitioning the application at development time deciding what are the portion of code to be considered for the offloading. Moreover, it is generally not worth to pay the cost of the monitoring and it is not always possible to justify the usage of dynamic decisions over static ones. So, it is generally necessary to evaluate the trade off endorsing performances or adaptability.

A first point of choice for an offloading software is to use dynamic or static decisions. In addition, the various cyber foraging solutions can be classified based different solutions that they are targeting:

- Inter-operability: the software should manage devices and servers with different resources. For instance, the current devices are all equipped with WiFi and cellular networks support. Despite their higher performance and lighter power consumption the WiFi networks work only in limited areas. For this reason an offloading software could manage the issue and it can switch from a network type to the other. The decision could change because the cost of the cellular networks and also the available bandwidths are generally more limited compare to WiFi. This behaviour should be transparent for the developer.

- Mobility and Fault Tolerance: The offloading technique is directly based on communication, that introduces some unwanted scenarios. For example, the network could fail or the server could be not available. For this reason the software could provide alternative solutions, such as running the code locally after a time-out expiration.

- Privacy: as mentioned earlier, in some cases the user might not have confidence in remote servers. The offloading software can provide some forms of cryptography or security measure to hidden information from potential abuses. The overhead of the information security mechanisms is well-known and not always acceptable for the application purpose. Thus, it is another important trade-off to analyse before the implementation.

- Context Adaptability and Monitoring: the current devices are equipped with many kinds of sensors, such as accelerometers, gyroscopes, multiple microphones and light sensors. In this direction, the offloading behaviour could vary based on the user environment. For instance, a different location could drive the software to run the code remotely because the user is using his private WiFi network. On the contrary, if the user is in a public WiFi network maybe the code must be executed locally to avoid sniffing or other kind of hacking.

- Multi-Server support: in order to improve the performances the offloading software could use multiple back-end machines at the same time or in different moments. So, different pieces of code could be remotely run in different machines. The decision on which machine the computation task should run can be based on load balancing or parallel processing.

The design choices for the offloading software are often dependent on the application purpose. Indeed, the cyber foraging is suitable for many kind of mobile apps, such as image recognition and computer vision, gaming and multimedia.
The parameters presented in the **Section 2.1.1** and **Section 2.1.2** have different relevance based on the goals of the app. For example in a computer vision, multimedia or image recognition application, if the data are stored on the device, the amount of data, $data_{input}$, to move is big and only if the bandwidth, B, is high the offloading would give good results. In the **Figure 2.3** is shown an example of face detention output of a mobile application.

2.2 Existing Work on Offloading

This section is dedicated to the related works in the mobile offloading topic. The goal of the thesis is different from the previous projects. It focus indeed on distributed policies for the offloading and in particular it provides a simple programming model to partitioning the application and to define custom policies. The design and the goals of this project will be explained in the **Section 3.4**. However, the main differences between the thesis project and the related works are shown in this section in order to clarify our decisions and novelties.

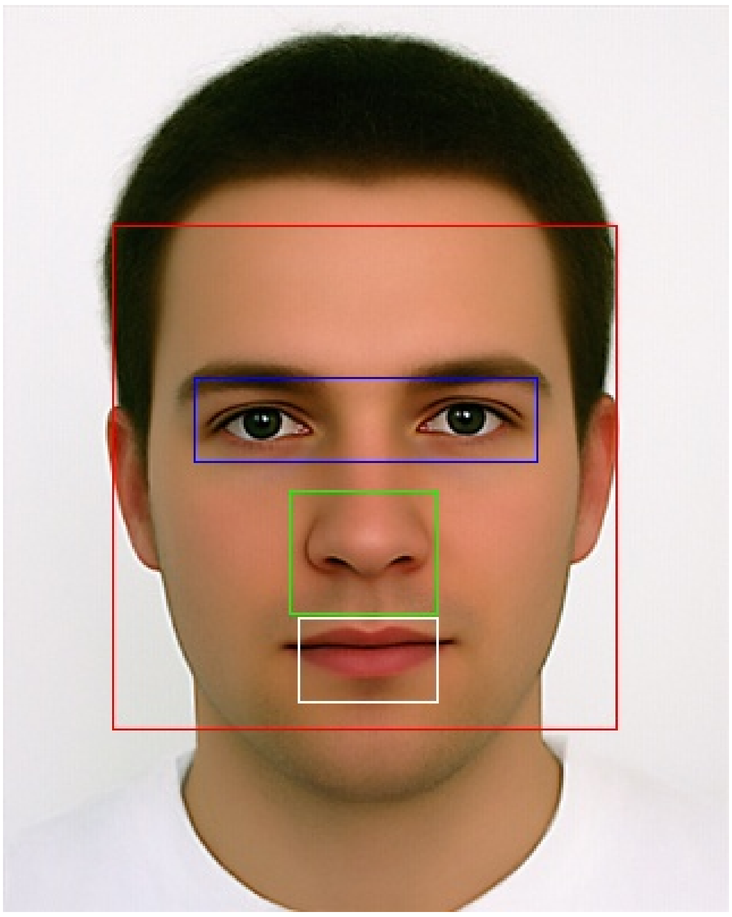


Figure 2.3: In the figure it is shown an output example of an OpenCV recognizer for an Android app. The face is an average face, not a real one, computed with computer vision algorithms.

ThinkAir [20] is an important mobile offloading and resource allocation project based on Android devices . The research group aimed to provide a simple to use framework able to migrate application to the cloud. Unlike the previous works, ThinkAir focuses on elasticity and scalability of the cloud and it emphasises the capability to execute remotely parallel methods using multiple VM images. Like many offloading projects, ThinkAir shows an improvement of the performances when the cloud is supporting the mobile execution. ThinkAir provides a method-level computation offloading as the thesis project. So, they provide a programming language to allow the developer to define which are the methods suitable for the offloading mechanism. However, the most relevant innovation of this research is the possibility to perform on-demand resource allocation, and exploits the parallelism by dynamically creating, resuming, and destroying VMs in the cloud [19]. The idea to use parallelism has been driven by the high demand of resources necessary to execute current apps. This advanced computing feature is strongly impactive in terms of performances and also energy consumption.

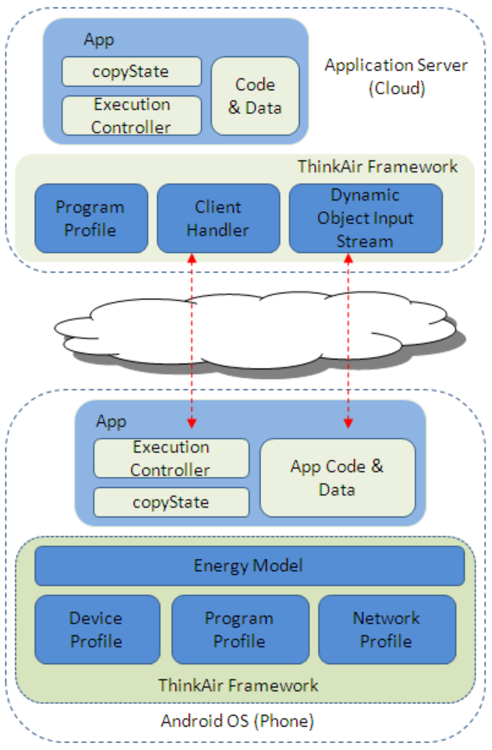


Figure 2.4: An overview of the ThinkAir framework components.

In the **Figure 2.4** is possible to notice the presence of many components, in particular the main parts of the systems are: a custom compiler, the Execution Controller and the Profilers. ThinkAir provides a simple to use library, in particular, the annotation *@Remote* allows the developer to specify the suitable offloading methods. As mentioned earlier, one fundamental component of the project is a custom compiler which performs the task of taking the source files and generates automatically the *remoteable* method wrappers and related utility functions. The method invocation is managed through the Execution Controller, which can detect if a certain method is suitable for the offloading and it handles all the tasks related to the code migration. This includes software-level profiling, decision making and communication with the server. All this operations are hidden from the developer, which has control exclusively on the *@Remote* annotation. The project thesis aims to provided more control on the offloading operations giving the tools to the developer to influence the decisions. ThinkAir uses four level of profiling in order to take offloading decisions:

- Hardware Profiler: it has the task to provide energy information to the energy estimation model. It monitors CPU, screen brightness, 3G and WiFi interfaces.

- Software Profiler: it takes care of analysing code execution. It use a large number of parameters in order to profile the program. The Software Profiler is based on the Android Debug API.

- Network Profiler: It takes track of the network state and it focuses on parameter such as RTT, number of packets transmitted and received per second, uplink channel rate and uplink data rate for WiFi interfaces.

- Energy Estimation Model: this profiler is based on the PowerTutor [21] model which accounts for power consumption of CPU, LCD screen, GPS, WiFi, 3G, and audio interfaces.

The back-end cloud architecture is composed of virtual machines which run a port of the Android x86. ThinkAir has six different kind of predefined virtual machine images with a different amount of available resources, such as CPU, hard drive space and RAM memory. The component which manages the life cycle of virtual machines is the VM Manager. It dynamically allocates new VMs based on the current request.

MAUI [22] is another milestone project for fine-grained offloading of mobile code. Unlike the thesis project and ThinkAir, MAUI was developed for .NET environment. Its main goal is to optimise the battery consumption of heavy computation software, such as face recognition, arcade

games or voice-based language translation. In order to not impact to much on the original application code, MAUI uses the .NET Common Language Runtime (CLR) features [23]. This tool allows the framework to manage the code environment during the execution and eventually to modify the behaviour of it. In particular, MAUI uses programming *reflection* and *type safety* to identify the *remoteable* methods and to extract the related objects states. MAUI generates two version of the target application, one for mobile devices and one for the remote execution.
MAUI is capable to offload the code not only in a cloud-based architecture

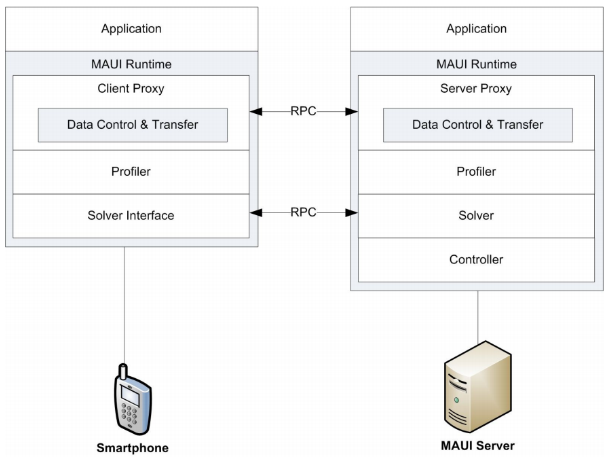


Figure 2.5: An overview of the MAUI architecture.

but also in machines such as desktop or laptop computer. As mentioned earlier, the MAUI profiling focuses on the battery usage, it monitors the energy consumption of local and remote execution. It also uses an history-based model to estimate the time of the execution, local or remote. So, the MAUI framework decides to offload the code if the migration saves battery.
On the mobile device side, MAUI provides three main components: a *solver interface*, a profiler and a client proxy. The solver interface helps the client to interact with the solver, the latter is the component which takes the offloading decisions. As stated previously, the profiler collects data

about energy consumption and network usage. Finally, the client proxy is the component which practically migrate the code and takes care of transferring the needed information. The MAUI server also contains the client proxy and the profilers. They have perform similar task to their related components on the client side. In addition, the server architecture contains the *solver* and the *controller*. The first component is the decision engine of the MAUI system, it has access to a call graph of applications and it schedules the method invocations and execution. The controller performs the authentication and resource allocation tasks.

Cuckoo [24] aims to provide a simple to use programming model using development tools already known by the mobile application developer. Cuckoo migrate partially mobile applications on cloud or nearby computing infrastructure as MAUI and ThinkAir. Cuckoo was developed for the Android platform. The Cuckoo's programming model is different from the previous two works. Cuckoo indeed uses already defined Android construct: activities and services. An overview on the Android platform is given in the **Section 4.1**.

The developer should use activities to define the interactive parts of the code, and, on the other hand, he should use services to define computation intensive functions. In addition to the mentioned constructs, Cuckoo can the separation of the code using the Android Interface Definition Language (AIDL) [25]. Using the interface the build system creates a remote service which contains a simple code done by Cuckoo Remote Service Deriver (CRSD). Like the RPC model, the Cuckoo Service Rewriter (CSR) framework generates stubs per each defined AIDL interface. The stubs allows transparently local and remote invocation based on the information provided by the Cuckoo Resource Manager (CRM). The final step generates the Android *APK* which makes the application installable on Android devices. In the **Figure 2.6** it is shown the building flow and Cuckoo can migrate code execution on every JVM machines, cloud or nearby infrastructures. The smartphone is the responsible to install the services on the server machine, after that the address of the server is stored by the CRM.

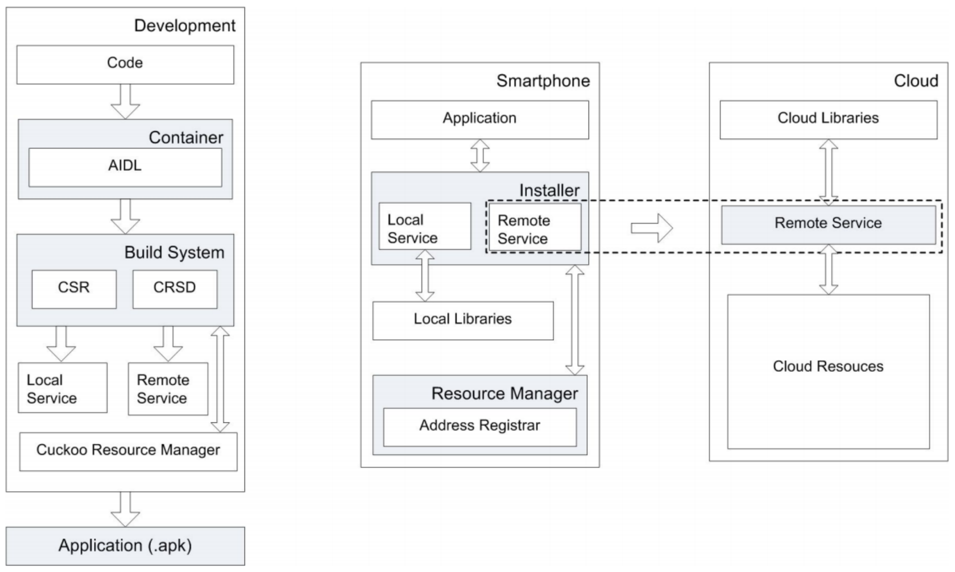


Figure 2.6: In the left part of the picture is possible to see the development flow of a Cuckoo app. In the the right side it is shown an overview of the Cuckoo architecture.

As mentioned earlier, Cuckoo can partially offload applications and to make it possible the framework uses exclusively Android language constructs. The framework does not support asynchronous interactions or state synchronisation with the remote endpoint. In addition, Cuckoo does not consider privacy policy or security issues.

2.2.1 Comparison

The **Section 2.2** presents three of the most related works of the project thesis. The projects are different in terms of implementation, generality, target mobile architecture and offloading decisions. This sections focuses on resuming and comparing the different approaches provided by the related works.

ThinkAir [20] and Cuckoo [24] are both developed for the Android platform, in addition, ThinkAir claims to provide support for Android native code (Android NDK).

Project	Platform	Privacy	Decisions	Back-end Infrastracture	Scalability Level	App. Partitioning
ThinkAir [20]	Android	No	Dynamic	Cloud	Medium	Annotation + Custom Compiler
MAUI [22]	Microsoft .NET	Authentication	Dynamic	Cloud/Nearby Infr.	Low	Annotation + Reflection
Cuckoo [24]	Android	No	Static	Cloud/Nearby Infr.	High	Android IDL + automatic. stubs generation

Table 2.1: The table shows the different approaches taken by the related works of the **Section 2.2**.

A really important point of comparison between the projects is the kind of offloading decision made, ThinkAir uses multiple level of profilers and an energy estimation model which makes the decision dynamic and more adaptive. MAUI [22] uses history-based decision, so it collects information of previous execution and thanks to them it tries to predict the battery consumption and the time of execution. The ThinkAir approach is more accurate but at the same time it is not generally reusable. The battery estimation function and the CPU monitoring are strongly based on specific models. *Sokol Kosta et al.* [20] monitored CPU, LCD screen, GPS, WiFi, 3G, and audio interfaces on two specific handsets: HTC Dream and HTC Magic. For the battery estimation they used the PowerTutor [21] model which is very accurate exclusively for this two device models. The MAUI model of profiling is more general because is based on previous execution, however if the model fails even the migration decision would be wrong. Another disadvantage of the MAUI profiling is the direct necessity to run the code in development environment, in order to provide to the MAUI solver an initial history. Cuckoo project does not focus on profiling they provide a static model of decisions, that is based on simple information. From an overhead point of view, Cuckoo is lighter-weight compare to ThinkAir and MAUI that need to execute profiler logic in order to make their decisions model dynamic.

Another fundamental feature of the offloading software is the application partitioning. The ThinkAir and MAUI approaches are similar in this specific matter. The partitioning is made from the developer who could define and divide interactive logic from intensive computation one. A difference between these two projects is that while ThinkAir uses a custom compiler to modify the marked classes and methods, MAUI prefers a more dynamic approach using the .NET reflection tool. The disadvantage of the reflection is the related overhead. Cuckoo performs the partitioning task using Android constructs and AIDL. The stubs necessary to mask the remote invocation are automatically generated by the CRSD. The Cuckoo's approach is more scalable compared to the others but it does not provide support to asynchronous call and state synchronisation. If the developer would need to transfer the state he should use a Representation State Transfer-like (REST) mechanism.

When cloud storage and computing is involved is often needed to wonder if the provided privacy security is enough. ThinkAir and Cuckoo do not provide any kind of security level. This decision could be dangerous because it does not avoid to malicious user to install software to illegally access resources. Unlike the other project, MAUI provides an authentication procedure. However, even in this case the privacy level is not considered high because there is not control on the resource access.

Finally, MAUI and Cuckoo are able to offload code in every kind of infrastructure, cloud-based or simple computers. ThinkAir is designed to always use the cloud infrastructure in order to avoid the parallel computing on different virtual machines. This feature is really novel because it provides more flexibility to the beck-end infrastructure allowing on-demand resources allocation and parallel execution of the migrated code. However, the Cuckoo solution is the more scalable because it is possible to invoke different services from different sources. As mentioned previously, Cuckoo exploits a dynamic services instantiations made by the mobile device, this concept improves substantially the scalability of the entire systems allowing parallel execution and dynamic allocation totally managed by the user device. Clearly, the same service could be provided by multiple servers in order to maintain a good level of fault-tolerance.

Chapter 3

Project Analysis

This chapter focuses on the thesis project analysis and design. The project will be integrated with the Avatar Project developed by the New Jersey Institute of Technology. In the first part of the chapter it is presented the Avatar architecture and its programming model. In the second part the writing deepens on the offloading framework design and programming model. Finally, the chapter focuses on the achievements that the thesis project should obtain in order to supply a stable and lightweight software. This goal should be achieved without affecting negatively the application performances.

3.1 Avatar Project

The thesis project will be integrated with the Avatar project [5] of the New Jersey Institute of Technology. Avatar is a system which supplies cloud resources to support fast, scalable, reliable, and energy efficient distributed computing over mobile devices [5]. So, every user is associated to a set of dedicated cloud resources. An *avatar* is an entity which can support the execution and the communication of the user mobile applications in the cloud. In few words, it is possible to say that an avatar is a single instance of a surrogate in the cloud for a mobile device [26]. The per-user dedicated resources are implemented as a virtual machine in the cloud that runs the same operating system of the device. Therefore, the decision to use the same operating system on the VMs was made in order to maintain the application code unmodified, to provide isolation of the resources and to make the adoption of this new technology simpler for developers.

The Avatar project aims to provide: *high availability* to user applications and devices; a *high level* programming model, which must be simple and flexible; *isolation* and effective resources management for mobile user apps in the cloud; *high efficiency* for mobile devices; and mobile data *privacy* management. Avatar is designed not only to support single-user application but especially it aims to provide an effective and efficient support platform for distributed applications. Avatar is able to create and to manage group of users involved in a common distributed application.

3.1.1 Architecture

In order to understand how Avatar could achieve its goals, the architecture design of the system is introduced in the following section. In the **Figure 3.1** is shown a complete scheme of the Avatar architecture. The most atomic parts of the architecture are: the user abstraction which is composed by the mobile device plus the related avatar, the Storage Service, the Discovery Service and the Group Management Service.

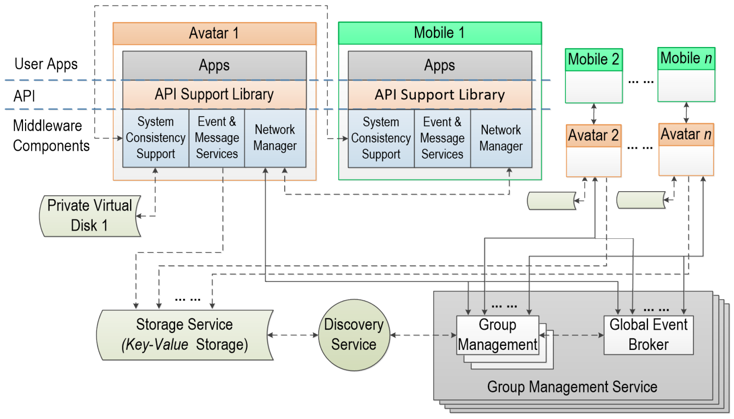


Figure 3.1: The Avatar Architecture.

A user is represent in the Avatar group as a combination of two parts: one running on the mobile device and the other one on the Avatar virtual machine(AVM). As shown in the top part of **Figure 3.1**, both the parts run the same application code, an *APIs* library layer and a set of middleware components. Since the APIs level will be deeply explained later, this section mainly focuses on the Avatar middleware, Moitree. Moitree is composed by a set of middleware services:

- **System Consistency Support**: each avatar needs to be synchronised in order to correctly and efficiently schedule computation tasks and to maintain the data consistency with the mobile device. This task is managed by the System Consistency Support (SCS) component. To manage this mechanism the SCS is implemented as a *daemon* process which runs on the mobile device and on the Avatar virtual machine. Every application supported by Avatar could specify data to be synchronise and how frequently execute this function;

- **Event and Message Services**: the Avatar communication part is strongly based on events and messages. This component takes care to send and to receive events and messages from and to other users. The Event and Message Service (EMS) has the task to forward events and messages to the destination app. More in details, in order to handle events and messages, the EMS implements two different queues: an event queue (EQ) and a message queue (MQ). The first of those is used to post events generated by the application, such as a group creation event. The second one, MQ, is used to distributed data and eventually to get results among the group members. It is important to emphasise that events and messages are not dependent on network low-level addresses or names. They are associated to channels established within each group. Each message or event generated by a mobile device is forwarded to the relative AVM and then it is delivered to each recipient of a group. This decision was made to limit the mobile device data traffic, the Avatar virtual machine indeed takes care of the communication on the behalf of the handset. When the event/message arrives at the avatar, the EMS forwards it to a GMS server, where the recipient list can be determined. Then, the event/message is dispatched by the GMS server to the recipient avatars, and these avatars finally forward it to the corresponding apps.

- **Network Manager**: The Network Manager shields the developer from the complexity of the communication between the mobile device and the Avatar, in addition it takes care of system events fired by the group management.

- **Storage Service**: The middleware services need to take track of Avatar applications installed on the device. So, the Storage Service maintains a registry of the available apps on a device and on the associated avatar. A registry entry is composed of a unique ID and an application name. The Storage Service could also maintain information about the user location and time. The virtual disk owned by a VM is not part of the Storage Service but it is a primary private disk for the avatars and its apps local storage.

As mentioned, a distributed application is composed of a group of users who cooperate in order to achieve a common goal. So, the Avatar architecture is designed to support intergroup communication. As it shown in the **Figure 3.2** a central entity for the Avatar group is the Group Management System (GMS).

The GMS is the fundamental entity which creates, modifies and deletes groups. As a group manager it also takes care of the intergroup interactions and those are generally defined by four channels:

- *Broadcast*: this channel allows to send messages to all members of the group. It is the only unidirectional channel available, no response is included in this model;

- *Anycast*: this channel allows to send messages to a random member of the group;

- *Point2Point*: this channel allows to directly send messages to a specific member of the group;

- *Scatter-gather*: this channel allows to send messages to all members of a group and then to collect results from some of them.

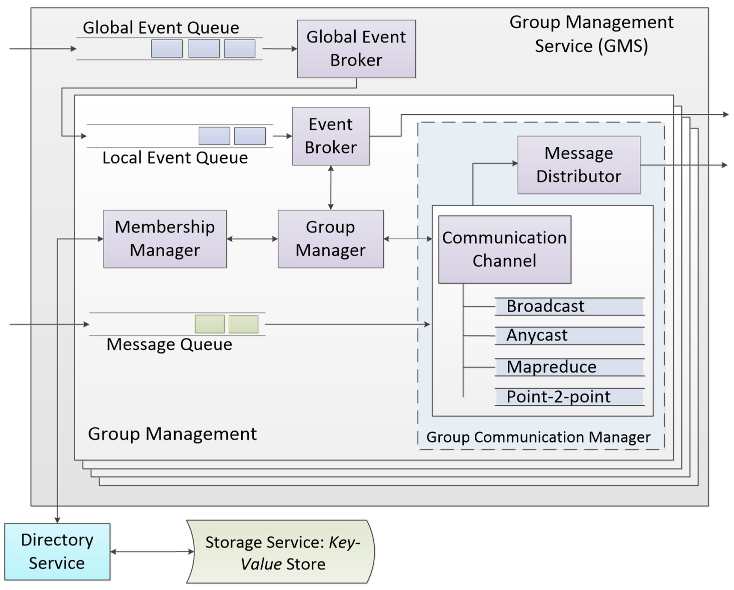


Figure 3.2: A detailed representation of the Group Management Service.

The EMS is always connect to the GMS. As mentioned previously, the developer does not need to know the location of other users but he just needs to select one of the available channels to obtain his goal. The Event and Message Service located in the device and in the avatar sends the message to the GMS Message Queue and then the GMS will take care of the delivery. The group manager does not know directly the location of all group members but it relies on a specific entity for this task, the Avatar Discovery Service. This component stores all location information of members and it can be queried in order to retrieve them.

In addition the GMS has an internal subcomponent that acts as a broker for global events. This kind of events are sent by user applications to notify the group. From **Figure 3.2** it is possible to notice the presence of two queues types for the event management:

- **The global queue** stores all events from the groups. It is the Global Event Broker that takes care of forwarding messages to the right destination group.
- **The local queue** is local to a single group. The Event Broker is another local component which manages specific events for the group.

3.1.2 Moitree's Programming model

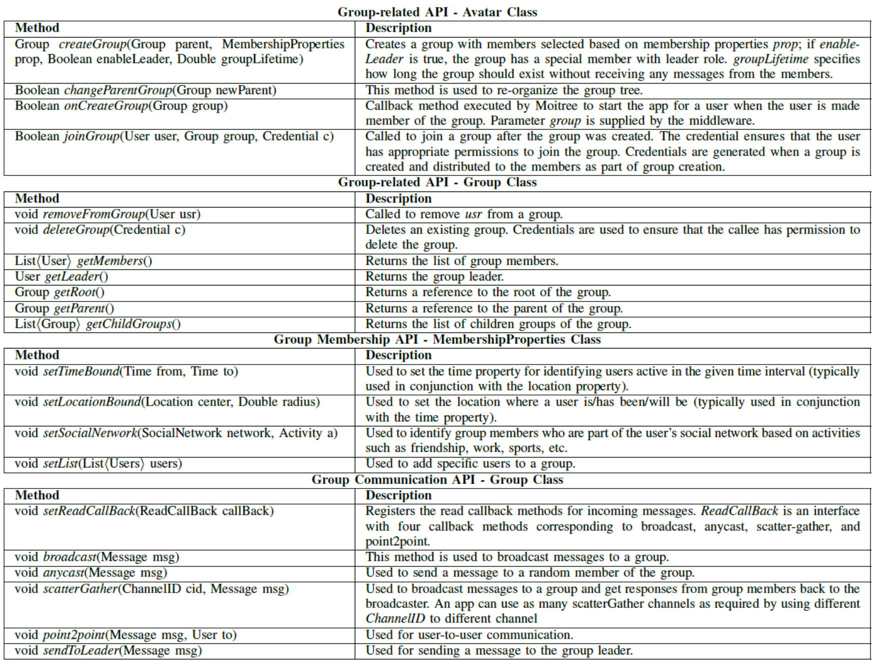
The Moitree middleware [26] is the tool through which the user applications are able to communicate and to use the Avatar system model. Between the applications and middleware services there is an interface intermediate level, the Application Framework Support library (Moitree APIs). This level provides to the application level a set of interfaces in Java programming language in order to interact with the middleware and to use its services. The APIs provided are divided in three main classes:

- **Group Management APIs**: They are methods to create, modify and delete groups. For instance, to establish a new group, the developer should use the *newGroup* method. Avatar supports a groups hierarchy model, in the creation phase of a group it is possible to define its parents. Another important parameter is the lifetime of the group, it defines the time before the group will be deleted in case of absence of any intergroup communication. Finally, if the developer wants to assign a leader to the new group he can use the *enableLeader* flag and the GMS will automatically elect the initiator as the leader. A user could join a group simply calling the *joinGroup* method or *removeFromGroup* to it;

- **Group Hierarchy APIs**: Every user belonging to a certain groups could obtain informations about the group hierarchy, for example there are two methods: *getParent* and *getRoot* to obtain the references to a parent or to the root group respectively. In other words every user included in a group are automatically member of its parent. In addition, it is possible to obtain the list of children of a group with the *getChildsGroup* API;

- **Group Communication APIs**: As mentioned in the **section 3.1.1**, there are four different type of channels available and they are all asynchronous. There is a dedicated scatter-gather channel per each application and it is generated at the first invocation. A channel can be always referred using its *ChannelID*, a unique identifier. The *setReadCallBack* method allows the developer to specify a callback function per each type of channel available.

In the previous list are shown only some fundamental API available in the Moitree set. However, in the **Figure 3.3** it is available a more detailed summary of the Avatar middleware API.



Group-related API - Avatar Class

Method	**Description**
Group *createGroup*(Group parent, MembershipProperties prop, Boolean enableLeader, Double groupLifetime)	Creates a group with members selected based on membership properties *prop*; if *enableLeader* is true, the group has a special member with leader role. *groupLifetime* specifies how long the group should exist without receiving any messages from the members.
Boolean *changeParentGroup*(Group newParent)	This method is used to re-organize the group tree.
Boolean *onCreateGroup*(Group group)	Callback method executed by Moitree to start the app for a user when the user is made member of the group. Parameter *group* is supplied by the middleware.
Boolean *joinGroup*(User user, Group group, Credential c)	Called to join a group after the group was created. The credential ensures that the user has appropriate permissions to join the group. Credentials are generated when a group is created and distributed to the members as part of group creation.

Group-related API - Group Class

Method	**Description**
void *removeFromGroup*(User usr)	Called to remove *usr* from a group.
void *deleteGroup*(Credential c)	Deletes an existing group. Credentials are used to ensure that the callee has permission to delete the group.
List⟨User⟩ *getMembers*()	Returns the list of group members.
User *getLeader*()	Returns the group leader.
Group *getRoot*()	Returns a reference to the root of the group.
Group *getParent*()	Returns a reference to the parent of the group.
List⟨Group⟩ *getChildGroups*()	Returns the list of children groups of the group.

Group Membership API - MembershipProperties Class

Method	**Description**
void *setTimeBound*(Time from, Time to)	Used to set the time property for identifying users active in the given time interval (typically used in conjunction with the location property).
void *setLocationBound*(Location center, Double radius)	Used to set the location where a user is/has been/will be (typically used in conjunction with the time property).
void *setSocialNetwork*(SocialNetwork network, Activity a)	Used to identify group members who are part of the user's social network based on activities such as friendship, work, sports, etc.
void *setList*(List⟨Users⟩ users)	Used to add specific users to a group.

Group Communication API - Group Class

Method	**Description**
void *setReadCallBack*(ReadCallBack callBack)	Registers the read callback methods for incoming messages. *ReadCallBack* is an interface with four callback methods corresponding to broadcast, anycast, scatter-gather, and point2point.
void *broadcast*(Message msg)	This method is used to broadcast messages to a group.
void *anycast*(Message msg)	Used to send a message to a random member of the group.
void *scatterGather*(ChannelID cid, Message msg)	Used to broadcast messages to a group and get responses from group members back to the broadcaster. An app can use as many scatterGather channels as required by using different *ChannelID* to different channel
void *point2point*(Message msg, User to)	Used for user-to-user communication.
void *sendToLeader*(Message msg)	Used for sending a message to the group leader.

Figure 3.3: A summary table with the main Avatar APIs.

3.2 Problem Statement

The thesis project aims to provide an offloading software capable to migrate the execution of mobile distributed application to the cloud infrastructure.

As stated in the **Chapter 2**, to design an offloading software is necessary to manage:

- *Application Partitioning*: It is essential to partition the application dividing code which could be not migrate from the heavy computation code. As mentioned previously, some I/O or user interface interaction cannot be offloaded to the beck-end server.

- *Offloading Decisions Management*: to offload code is not always a wise decision in terms of resources utilisation. For that reason, the thesis should use a solution which should decide when the code needs to be migrate on the other side and when it is better to avoid it.
- *Distributed Decisions Approach*: the thesis does not aim exclusively to design a end-to-end offloading mechanism. It should provide a simple to use and powerful tool to define new policies capable to describe distributed conditions.

Firstly, it is essential that the programming model provided to the developer is as simplest as possible. Some previous offloading works is based on automatic partitioning of the application. This approach sometimes is not enough flexible for the application scenario. Only the application developer knows exactly what should be considerable for the migration of the execution and what should be executed locally. The project should also consider privacy issues which is not possible to automatically decide without the developer intervention.

The offloading decisions could be based on many information and conditions. The goal of the project is to provide a programming tool to define new policies as simple language constructs. In addition, the policies should be fine-grained. In simpler words, a policy could influence offloading decisions on single method of classes. The project could also provide some default decisions based on the heuristics of *Section 2.1.1* and **Section 2.1.2**. These expressions are general and valid for almost every kind of application scenario. It is important to reiterate that the thesis project goal is not to design new offloading heuristics or rules but to provide a tool to define rules suitable for the specific case.

The distributed offloading decisions are a subgroup of offloading general decisions. However, they should take in account information further than device status. An example, could be the QoS or the privacy policies required by an user member of the distributed system. In the **Section 3.3** some example applications for distributed offloading decision is given in order to clarify the concept with practical application.

The offloading framework will be an Avatar middleware service, for that reason it should be designed considering the system peculiarities and characteristics. As mentioned earlier, unlike CloneCloud, Avatar is based on the hypothesis that the mobile devices and the cloud infrastructure can run the same mobile platform. It is important to keep in mind that an Avatar application runs in both the endpoints, so the programming language classes and libraries are loaded at run-time. This aspect is fundamental for the offloading software design because it saves time and network bandwidth avoiding the transferring of the mobile platform and of the application.

3.3 Scenario

In order to support the statements of the **Section 3.2** this section provides some application scenarios.

An example could be a disaster rescue application. A rescue squad could be made up of officers and drones. Some of the disaster areas are not easily accessible to rescuers but , for instance, they could be reachable to flying drones. The drones can examine the region trying to identify some life signs or dangers. If something is found they can send a notification to the squad in order to allow them to help or to mange the problem. An effective way to scan a area could be to reconstruct a 3D model of the disaster zone taking pictures or videos from different angles. The nature of this process is strongly distributed, since every single drone in a specific area should contribute to the 3D model. The disaster rescue operations are strongly characterised by fast reaction times and team cooperation. In this scenario it could be really relevant to react to dangerous or urgent circumstances.

The human rescuers could be equipped with mobile devices, in addition the drones could be driven by a Artificial Intelligence (AI) software.

The cooperation between human members and robots could be made possible though a distributed application. So, the overall disaster zone could be divided in subregions monitored by different members of the rescue squad. For instance, there could be an human rescuer and multiple drones in a region. If the officer notice some suspicious objects or shapes it could take a picture of the object and broadcast it to the drones. The AI software could evaluate the direction of the picture related to the position of the human rescuer or other information in order to infer the real position of the target. After that they could start to scour the surrounding in order to build a 3D model of the area allowing the rescuer to identify the nature of the target object.
As mentioned earlier, the nature of this application is distributed, multiple devices cooperate in order to obtain a common goal. A drone is considerable as a mobile device with movement capabilities. The offloading technology could seriously improves the application execution, the drones and the mobile devices are indeed equipped with limited resources. The battery duration in this scenario is really essential, since the drones durability is related to it. The same consideration is even valid for the mobile devices of the rescuers. The computer vision functions could be offloaded to the cloud infrastructure to decrease the execution time and save resources.

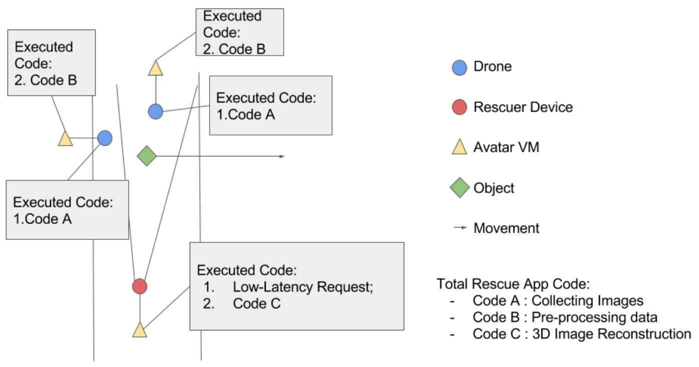


Figure 3.4: Rescue app sample scenario. The rescuer needs a low-latency response from the drones, since the suspicious object is moving.

In this scenario it is possible to apply multiple offloading policies based on the current situation. The sample code is composed of two main pieces: a data collecting task and preprocessing operations. The offloading decisions could be applied on the second part of the code according to the QoS specified by the requester user. For instance, the human rescuers could specify different condition in order to receive a different result in terms of latency or quality of the 3D images. In certain situation, as the scenario shown in the **Figure 3.4**, the officer could need low latency responses in order to promptly react to a specific situation. In this case the drones have to endorse the time of execution instead of the other parameters, such as battery consumption, bandwidth and quality of the 3D model. So, the offloading decisions on the single mobile devices of the team will be influenced by another group member in order to grant a desired QoS. In the example the preprocessing will be executed on the avatars in order to accelerate the delivery to the requester.

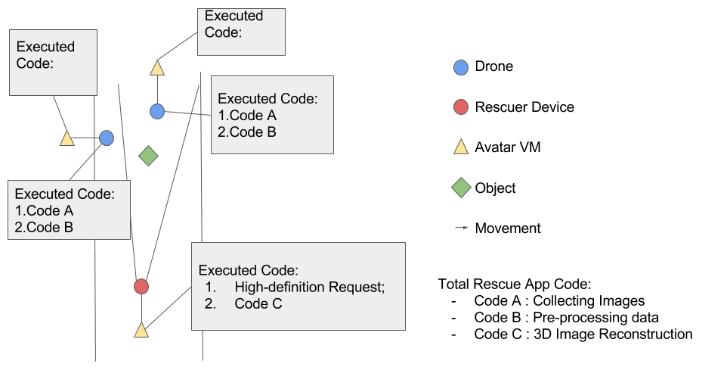

Executed Code:
Executed Code:
Executed Code: 1.Code A 2.Code B
Executed Code: 1.Code A 2.Code B
Executed Code: 1. High-definition Request; 2. Code C
Drone
Rescuer Device
Avatar VM
Object
Movement
Total Rescue App Code:
- Code A : Collecting Images
- Code B : Pre-processing data
- Code C : 3D Image Reconstruction

Figure 3.5: Rescue app sample scenario. The rescuer needs high-definition data.

In a different scenario the human rescuers could need high quality 3D reconstruction in order to check further information on the specified area, for instance because there are too many obstacles to understand well what it is happening. A schematic representation of the described conditions is given in the **Figure 3.5**. In this case the offloading decision could be based on the new policy. For instance the preprocessing of the images could be done in the device in order to acquire more detailed data and not excessively consume the network bandwidth. An alternative could be to endorse the battery consumption in order to make the drones working for a longer time. The bandwidth could also be a main factor for other scenarios with low connectivity.

Summing up, the goal of the thesis project is to design a programming model and a software implementation to enable to modify offloading decision of the single devices based on the other member demand. The examples given earlier are mostly related to QoS of the distributed application, however as stated previously the privacy could another essential parameter of the migration decisions.

3.4 Offloading Middleware Service

The problem statement of this thesis work is given in the **Section 3.2**. This section focuses on how to design the project in order to solve the requirements of the problem.

The offloading framework should take fully charge of the application execution flow. It means that the developer could write the application as a plain as a plain Avatar distributed application. Another fundamental functionality that the thesis project should provide is the support for Android native code offloading. More details about the Android platform are shown in the **Section 4.1**. Some of the offloading related works provided classes and methods for the offloading tasks, instead of that the thesis project should automatically intercept the code and then automatically decides where to run it However, the programmer will not lose the control over the offloading decision and, so it is fundamental to provide a simple way to define custom offloading policies. Since Avatar project aims to help the distributed application development the thesis project should take care not only of the local status of the mobile devices as the related works, but the offloading decisions should be also influenced by the other users involved in the entire distributed application. Unlike the common computation offloading software, the project should not only focus on battery consumption, network bandwidth and latency for a single device but it aims to improve the average of these for the entire group. It means that some users could be penalise by a single decision but the average usage of the resources would be improved by the next decisions.

The computation offloading in certain scenarios is really a significant improvement for the time of execution and the battery consumption. For that reason, the decision could be generally to execute the code on the AVM according to the heuristics shown in the **Section 2.1.1** and in the **Section 2.1.2**. That is always a wise decision if the focus of our research would be only on the resources and communication optimization. However, it is important to consider further concepts. For instance, Avatar has been designed to provide privacy policy. The privacy could be a blocking parameter for the offloading decision, so it is important to take this matter in account. The Avatar model is designed to entirely offload the communication on the AVM. Therefore, a device cannot straightly reach another handset without passing through the virtual machines. The GMS indeed only knows the network location of the virtual machines. So, the thesis project cannot really influence the intergroup communication. However, the offloading decisions influence the network bandwidth anyway. The data necessary to run the code remotely could be too big to be transferred, in this scenario it is probably better to run the code locally saving bandwidth and in some cases even battery. Another similar case is a distributed application which needs to collect data from devices sensors and use them for some kind of computation. In some case it could be better to not offload the sensor streaming in order to keep low the delay and the bandwidth usage. For instance, it could be better to use some preprocessing operations on the data and then execute the main functions on the avatar. Summing up, the good approach to offloading decision is to find a trade off based on data location and size, privacy policies and estimated performances. Further, it is the thesis goal to take the global resources of the group in account and not only the local state of the device.

3.4.1 Design

This section focus on the offloading service design and on its components. Some of the offloading related works partition the application in threads, they are able to run an whole thread locally or remotely based on different level of decisions. The nature of the Avatar system suggest to use method offloading. We want to use a mechanism similar to the Java RMI model. The application partitioning is based on the developer preferences, so we will provide a set of Java annotations to classify and mark classes. We will deepen on the programming model provided in the **Section 3.4.2**.

In order to achieve the goal presented earlier we identified the following macro components:

- **Code interceptors**: The offloading framework needs some tools capable to intercept the invocation of certain methods or classes in order to insert the offloading logic around them. The developer can mark classes or methods and the interceptors should catch their execution and notify the framework. This behaviour is realisable through the Aspect Oriented Programming model. As mentioned in the **Section 4.2.1**, there are many ways to obtain this behaviour but the project should provide a lightweight interception avoiding to slow down the application execution.

- **A Decision Maker**: This is an intermediate component, after the interceptors catch the code execution, the framework should evaluate some information, for example profilers data and/or distributed policies. The decision maker component is practically the brain of the thesis project and it is the part which influences the offloading task and its output is a boolean response: migrate the code or not on the avatar.

- **An Offloading Execution Manager**: The interceptors are just the beginning of the offloading flow. After a join point is picked up by a pointcut the decision maker component commands the offloading execution manager to run the code locally or remotely. If the decision is to run locally the execution manager let the code flow proceed regularly, otherwise it should take care of migrating the code and keeping the application blocked. Further, it should collect the result from the avatar and resume the natural execution of the code locally. This component is also available in the avatar and it takes care to execute the migrated code and push back the result to the mobile device. It is really important to maintain transparency during this operations, the user experience should be not affected by the execution manager functions and the developer should not manually invoke them.

- **Profilers**: The decision maker component needs information in order to take a smart decision. This data are collected partially by profilers, which observe the state of the device. They can be split in multiple level with different monitoring target.
- **Avatar Communication Interface**: As stated previously, the offloading service needs to be integrated with the Avatar system in order to transfer and synchronised the code execution. Therefore, it is needed a specific interface in the Avatar middleware services to plug in the offloading framework. The thesis project needs a point-to-point communication between the user device and its avatar.

The components introduced above and shown in the **Figure 3.6** are macro parts of the offloading project. In the implementation process, it will be necessary to split these components in finer grained parts in order to modularise and to maintain the code.

It is now necessary to deepen some of the macro components presented above in order to give more detailed information and to identify better their functionalities

The Decision Maker engine is a fundamental component of our architecture and it uses different level of information in order to select where to execute the code. As mentioned earlier, it should bases its decisions on the profiler information and distributed policies but it should especially provide a customisation tool for the developer. So, the developer defined

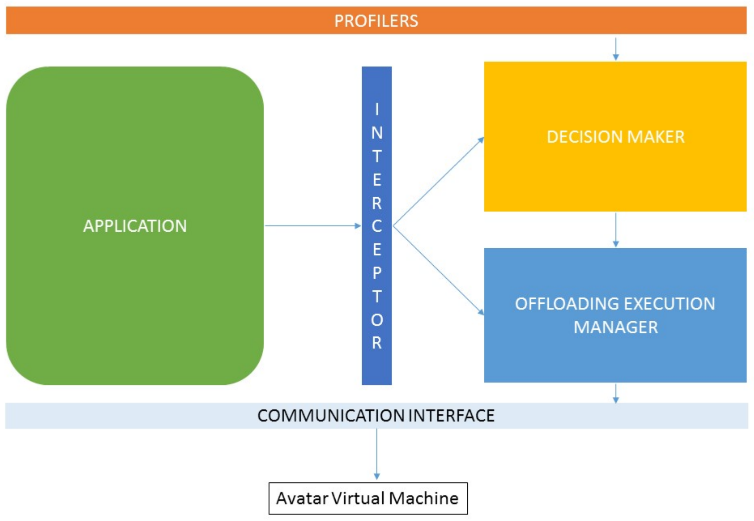


Figure 3.6: A representation of the offloading framework components.

policies are really a core instrument of the thesis project, because the decisions could be based on specific parameters impossible to predict with generic heuristics. For example, a certain applications require to maintain a high level of privacy, for others could be better to optimise the network usage and/or the battery energy consumption. Therefore, the developer could control the decision engine in order to adapt the behaviour on the specific application case and in addition to assign offloading policies to single part of the code.

The communication wrapper is a simple component which connects the offloading functions with the Avatar event-based interaction. The offloading framework needs a different kind of communication compared to the common intergroup communication of the Avatar project. It needs an end-to-end synchronous interaction in order to send bundle with executable code and to wait until Avatar VM response. Naturally, it is possible to use the common Avatar Network Manager but it is important to define a certain format of the bundle in order to synchronise the two different machines: mobile device and virtual machine. In particular, in the **Section 3.1.2** it is stated that the communication is always asynchronous, the offloading needs a synchronous interaction between the two endpoints. The device which offloaded the code should wait the result before executing the next operation to not break the sequential nature of the application code. In order to implement asynchronous call the developer might use different threads.

Finally, the most important component is the execution manager, it practically implements all the logic to offload the code. If the framework decides to offload the code it has to collect information from the Android Virtual Machine (ART or Dalvik). The needed information are all related to the run-time state of the Java code, such as parameters of the methods, state of the objects and type of the results. Thus, the implementation has to carefully manage this situation in order to make it possible to resume the execution from one endpoint to the other. Further, unlike a client-server scenario in the Avatar environment both the mobile device and the AVM have the same code installed. This would simplify the implementation of the computation offloading and its performances.

3.4.2 Programming Model

The offloading framework aims is to provides a programming model to the developer in order to use its functionalities in a simple and high level manner. As mentioned earlier, the target programming language is based on the Plain Old Java Object (POJO). The POJO is a software engineering term to describe a Java object not bound by any special restriction or external class path. In few words, a POJO described in a simple Java class that not dependent to other classes. Any kind of target logic specified in the POJO is exclusively related to the final goal, it should not include utilities or library invocations. A common way to design a POJO paradigm is to use the Java annotations tools. The developer could indeed define simple Java classes and he can marks them with some kind of annotation. The framework should locate only the marked class and it should react with a specified behaviour. In the **Section 4.3.1** is shown how to use Java annotations and how they are designed and their main features.

This section focuses on the programming model and how it is designed. The programming model is the access point for programmers, so it is really important to organise it in a clear and simple way. The implementation will start from this language definition and it should provide specific behaviours. Firstly, the developer should specified what part of the code is subject to the offloading decisions. This annotation is *@Remoteable*. A class with the *Remoteable* annotation specified that all the defined methods of the class could be offloaded.

```java
@Remoteable
public class MyClass {

   public static int zero() {
      return 0;
   }

   public int one() {
      return 1;
   }

   public int two() {
      return 2;
   }
}
```

Listing 3.1: The sample code shows a class annotated with @Remoteable.

In the code above the class is marked with the *Remoteable* annotation, so its *static* and *virtual* methods could be offloaded. The *Remoteable* annotation could be applied also to methods. It is indeed possible to not mark the class and mark just a single method, for example if the developer wants to offload only this method. In this case the Offloading Execution Manager will not transfer the state of the object but it will consider exclusively the method scope.

```
public class MyClass {

   public int one() {
      return one;
   }

   @Remoteable
   public int two() {
      return 2;
   }

}
```

Listing 3.2: The sample code shows a method marked with the Remoteable annotation.

In order to allow the programmer to specify certain behaviour for specific methods, there are other two annotations: *@onMobileDevice* and *@onAvatar*. The first one states that the code should be executed always on the device and the second one on the Avatar VM.

```
@Remoteable
public class MyClass {

   @onMobileDevice
   public static int zero() {
      return 0;
   }

   @onAvatar
   public static int one() {
        return 1;
    }

    public int two() {
        return 2;
    }

}
```

Listing 3.3: The sample code shows a class marked with Remoteable, in addition two of the defined method are marked with onMobileDevice and onAvatar, relatively.

In the previous example only the second method is subject to the offloading decision, the methods *zero* and *one* are marked with *onMobileDevice* and *onAvatar* annotations respectively.

As mentioned before, the Execution Manager should be capable to synchronise the state of the offloading object between both the user endpoints. However, the state transferring operations could cause a large number of different inconsistencies. As explained previously, the offloading scope could be a class or a method, the latter does not need the entire object state synchronised but only the specified parameters during its invocation. It is essential to specify that if the target of the annotation is a class, by default the Execution Manager does not send the instance status but it will only recreate a new instance of the same class. Naturally, the programming language so defined could be limited, for that reason it is possible to mark with a specific annotation the field of the class to be synchronised. This annotation is *SynchronizedField*.

```
@Remoteable
public class MyClass {

    @SynchronizedField
```

```
        private int value;

    }
```

Listing 3.4: The sample code applies the SynchronizedField annotation to a defined field.

In the code shown above, the *value* will be synchronised when the offloading will take place. The developer would like to select also different synchronisation policies, such as *lazy* and *eager*. The lazy policy simply sends the specified fields when the offloading mechanism starts. The eager policy transfers the state to the other endpoint every times it changes. For this kind of synch probably it will be needed a memory cache in both endpoints to keep the information until the real execution.

```
    @Remoteable(SynchType.EAGER) // or SynchType.LAZY
    public class MyClass {
        @SynchronizedField(SynchType.LAZY)
        private int value;
    }
```

Listing 3.5: Lazy and Eager synch policies applied to a Remoteable class.

The synchronisation policy type could be specified as a *Remoteable* or *SynchronizedField* annotation argument as shown in the previous example.

As discussed earlier, the developer should be provided of a tool to define new policy for the offloading decision. The *AvatarPolicy* interface is defined for that goal. Every new custom policy defined by the programmer should implement this interface and with it its methods. The method *testPrecondition* is the function evaluated by the Decision Maker component, it returns a boolean value: true to execute remotely and false for locally. In order to be automatically resolved by the offloading framework, the new policy should be marked with the *DefPolicy* annotation in addition to the AvatarPolicy implementation. In the following example it is defined a simple policy to show how to define a new policy.

```
@DefPolicy
public class MyPolicy implements AvatarPolicy {
   private int batteryLevel;

   public MyPolicy(int batteryLevel) {
      this.batteryLevel = batteryLevel;
   }

   @Override
   public boolean testPrecondition() {
      if(this.batteryLevel <= 20) {
         return true;
      } else {
         return false;
      }
   }
}
```

Listing 3.6: The code defines a new policy though the DefPolicy annotation.

The example always denied the offloading except if the percentage level of the mobile device battery is lesser than a fixed threshold. The developer defined policies could be applied to methods and classes. For that purpose it is needed to use another annotation, *CustomPolicy*.

```
@CustomPolicy(policyClass = MyPolicy.class)
public int finacci(int number) {
   ...
}
```

Listing 3.7: This code applies the custom policy to the specific method.

The argument of the *CustomPolicy* annotation is the Java class object of the defined policy. It is important to keep in mind that the method annotation are more priority than the class ones.

3.4.3 Offloading Distributed Decisions

The concept of one to one offloading is implemented in many solutions, the offloading framework of Avatar has different goals. On the one hand, it is designed to be simple and not intrusive for the applications development process. On the other hand, the framework introduces a novel idea more feasible for the distributed applications environment. As mentioned previously, the Avatar targets software are distributed mobile applications, so we decided to extend the concept of offloading to a distributed scenario. In a more practical way, the offloading framework should decide where to execute the code base on global resources. It means that the evaluation should be done not just on the local status of the device but considering also the other members status. Each user node of the distributed application could be exposed to the local offloading decision but also to the distributed decision based on the whole application state. Giving a simple example: The device of the first user has to run an *offloadable* code. The local offloading decision could be to run the code locally, for instance to save bandwidth. However, a second user desires the result in the fastest way as possible, so the first user could change its local behaviour to be quicker in terms of execution time. This decision naturally will override the first decision but it will be coherent with the first user request. As mentioned earlier, the developer could specify per each class or method an offloading behaviour. When the execution reaches the method marked with *Remoteable* the Decision Maker component decides where to run the code. Thus, a first layer of the decision is the local state evaluation, in addition in the second layer the Decision Maker analyses the QoS required by the other group users and it takes a definitive decision. Summarising, the local decision should be influenced by the other user QoS demands. For that reason we provide the tools to define new custom policies to the developer. Further, it is possible to define some global policies to give the programmer a entry level set of policies.

The offloading concept is generally correlated to battery consumption of the mobile device, network bandwidth usage and time of execution. This parameters could be also applied to the distributed offloading idea. In fact, the offloading framework should optimise the average battery consumption of the group, the network bandwidth to communicate and the total delay caused by the time needed to execute the code. The optimisation should be driven by the QoS demand of the users or other kind of specification, like privacy or social policies. For example in some cases the offloading decision could be always to save bandwidth in order to make the application execution quicker or vice versa.

As mentioned before, it is possible to provide standard offloading policies based on the Quality of Service demand of the user. In particular a standard model which focuses on embrace most of the possible scenarios. This model could be modified and adjusted to the specific application. Further, it is possible to extend the model including new parameters and new rules to make it considering more information and alternative cases. The hypothesis of this model is to have some information from external profilers or previous executions of the full application. It is indeed not the goal of this thesis project to implement profilers or algorithms to estimate execution time or battery consumption. Some previous research work as ThinkAir [20] proposes an accurate estimation model for execution time, battery consumption and network bandwidth. The limitation of the ThinkAir profiler model is the not re-usability, it is indeed based on electrical measurements on the device battery and it is currently available for only two hardware configurations. For that reason, this thesis project should not focus on the profiler research but it should provide a tool to define new distributed policies allowing the developer to define his own rules. Returning to the general model, another necessary hypothesis is to have the following inputs for the evaluation:

- The battery level of the mobile device;

- The battery average percentage needed to run the computation;
- The battery average communication percentage needs to transfer the information;
- The execution time needed to run locally the code;
- The execution time needed to run remotely the same code;
- The average amount of local data to transfer on the Avatar in order to run the code and eventually the size of the result;

With this set of inputs is possible to design a simple algorithm which could applied different decisions based on the QoS specified by the group. The decision could be based on the simple rules shown in the following pseudo-code.

```
Time Execution Decision:
if(TimeCommunication > TimeExecution(Remote)-
   TimeExecution(Local)
    return Local;
else
    return Remote;

Network Decision:
if(LocalData(Input) + LocalData(Output) >
   RemoteData)
    return Local;
else
    return Remote;

Battery Decision:
if(Battery%(LocalExecution)> Battery%(Communication
   ))
    return Remote;
    else
        return Local;
```

Listing 3.8: Pseudo-code of a policy for offloading decisions.

This three rules should be combined to reflect the specified dynamic QoS required by the distributed application. For example, a user wants to obtain a result quickly. The weight assigned to the execution time decision of the other group member should be higher than the battery and network ones. So, it is possible to compose a complex rule to model a large number of feasible scenarios in order to achieve better result in terms of performance and user experience.

Another possible decision could be related to user privacy policy. For example a user of a distributed application does not want to transfer some data on the cloud. For that reason the offloading decision should be influenced by this will and eventually denies the remote execution. In addition the offloading of a specific user could be influenced by the privacy policy of one or more other users.

Chapter 4

Enabling Technologies

This chapter focus on describing the technical tools and frameworks used to achieve the goals previously described. Google's Android has been selected as mobile platform and in the following chapter it is explained what are its main features and provided tools for the mobile development. In order to minimise the impact on the application development and to manage code interception and injection it is needed to introduce the Aspect Oriented Programming paradigm and, in particular, the AspectJ framework. In the final part of the chapter there is an overview on two essential Java tools used to provide a simple programming model to the final developer.

4.1 Android

Android [27] is a complete and open source platform for mobile devices, supported by Google and owned by Open Handset Alliance. Android was borned to revolutionise the mobile systems, the main goal is to isolate the hardware layer from the software one. This design has the positive edge to facilitate the distribution of mobile applications on a massive scale and it helps to create a big user and developer community. The approach undertaken by the Android platform has been successful, it is indeed the most spread worldwide spread platform. In the **Figure 4.1** it is shown that the market has been led by Android in the last three years, in addition the gap between it and its strongest competitor, Apple's iOS [28], is incredibly large.

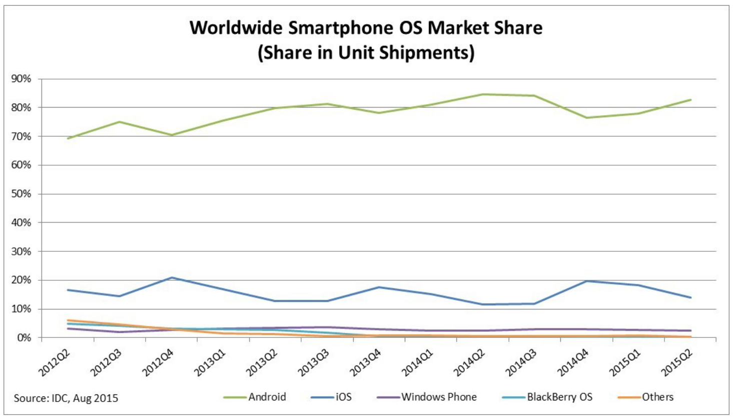


Figure 4.1: In the graph is shown the percentage of adoption for the most relevant mobile operating systems in the last three years.

Android could be considered a complete platform because it provides a great variety of tools, software and drivers for a large number of smartphones, tablets and many kind of mobile systems. It supplies a set of tools and a stable framework to develop quickly and simply mobile applications. The Android SDK is the only mandatory software to develop apps for smartphones, in addition the Android Platform Tools contains an emulator, so, it is possible to debug and to test applications without a real handset. By the way, for our research project it will be necessary to test on a actual device in order to measure and test the power consumption and cellular network usage in a real environment. For what concerns the user support, Android supplies simple interfaces without forcing the costumer to know specific configurations and to set up complex parameters. The platform also is an excellent solution to manage the hardware and to implement drivers for the manufacturers of this industry.

As mentioned earlier, Android comes with an open source license, in few words the available tools are totally free from ownership, from the hardware management to the application level. So, it is possible to look into and to eventually modify the implementation of every level of the Android stack, such as native libraries, application framework and user level applications. More in details, the Android's license belongs to businessfriendly (Apache/MIT) family, so everyone could freely extend, modify or use the platform for every kind of goal. A lot of third part libraries has been inserted in the Android's stack under new terms of license. This phenomenon has been pushed by the will to integrate and constantly update the Android platform features. The developer has free access to every level of the open-source platform, he could exploit this Android peculiarity to deeply understand and eventually to modify the behaviour of the system. The produced modifications and extensions are not strictly forced to be published or presented to the community. Moreover, as mentioned previously, for a producer is simple to import the Android stack on his handset.

Android is exclusively a mobile platform, this decision was made because a mobile device has totally different limits compared to desktop and enterprise systems. The mobile handsets are in fact smaller in terms of size, this peculiarity causes lower memory capability and computation performances. In addition, they have to deal with battery consumption which represent a further constraint on applications development. Since the platform has been designed for every kind of mobile device, Android does not establish or specify any preconditions on the screen size or resolution, the chipset or any other technical specification.

Google decided to invest resources on the Android project, that because Google is a media company and its revenue are mostly due to the advertisement market. The big company aims to be the primary service provider and to fairly play against other competitors. This philosophy is totally against the tide of some other industries which mostly base their business on the license revenues. The Open Handset Alliance is a non-profit cooperative composed of big hardware manufacturers in the mobile systems market, such as HTC, Asus, Intel, Motorola, NVIDIA, and some network provider as T-Mobile. The main goal of this cooperative is to innovate and to provide a good experience to the final costumer.

4.1.1 Architecture

The Android architecture, as a common desktop systems, is composed of multiple layers. It is possible to see in the **Figure 4.2** a concise and simplified representation of the Android structure.

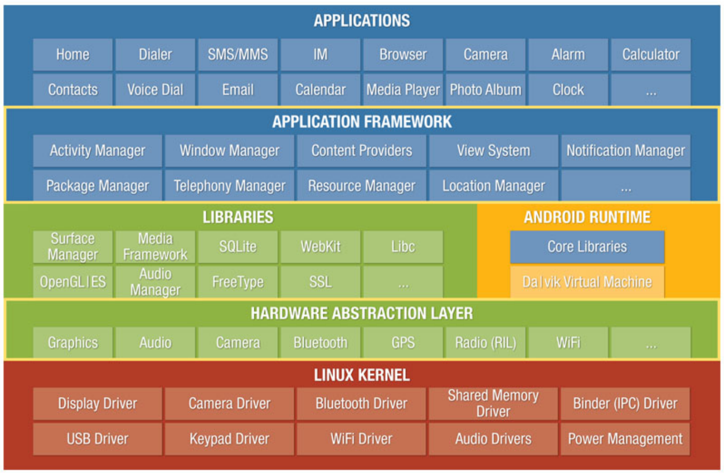


Figure 4.2: Android Architecture.

The core of the system is based on the Linux Kernel (3.10) and it helps to manage the abstraction gap between the device hardware and the application framework. The handsets manufacturers could directly work on this level in order to set up and to install drivers for hardware communication. The abstraction layer allows to isolate the upper levels from the hardware specifications and details. Thus, it is possible to develop high level applications and to grant a user experience without being worried about the hardware. The Android team has selected the Linux kernel in order to ensure a good reliability of the system lightly penalising performances. It is indeed a fact that users generally prefer better stability for the telephone dealer application even if the device would be partially slower. In addition to basic apps, the team wants to provide more refined and evolved services, in this direction Linux proved to be worth supporting all requirements. Linux is also know to be the best operating system under a open source license.

In the upper level, **Libraries**, a set of fundamental libraries and software tools. For instance, there are specific libraries to manage the 2D and 3D rendering, the browser engine WebKit and to support database access, SQLite.

The run-time environment is composed of a core library and a virtual machine. The Android virtual machine is a modified version of a common Java machine, in the past it was a Dalvik VM but in the latest version it is an Android Runtime virtual machine, ART. Dalvik VM, and then ART, is designed to work and to manage low performances hardware, such as current mobile handsets. Mobile specifications and performances are improving quickly, they are equipped with multi-cores processors and up to 4 Gb of RAM. Linux is a multi-user operating system, so every application corresponds to one single user. In fact, it is assigned a unique user ID per each application, in few words there is only one user allowed to access the application files. There is also the possibility to assign the same user ID to multiple applications but the Android developers strongly discourage this strategy except if there is a specific need as an heavy low level communication between two applications. In order to guarantee a stable system, every single application is executed in a unique Linux process. This architecture is known as *sandbox* and it protects the system from potential dangerous third party applications. In the upper level there are managers and base system apps, they allow to manage user applications installation, file system support and telephone dealer services. This level is also called **Application Framework**.

In the top level, *Application* in **Figure 4.2**, there are user applications. They are all written in Java programming language and executed in a virtual machine. A simple example could be the dealer application, another one could be the calendar app. It is important to specify that the Android application source code is different from the Java ME one. In fact, it is previously compiled and translated to byte code and then optimized for the Android virtual machine using a *dex* format.

4.1.2 Application Framework

In order to deeply understand the behaviour of the Android platform it is needed to deepen the Application Framework. In particular, it provides useful tools for the thesis project implementation. It indeed supplies a rich environment composed of a large number of services, they are significant for the software development. Inside this level it is possible to find a great variety of Java libraries, many services which are called *managers* and which allow the communication between applications and system resources, such as sensors, WiFi and cellular network hardware.

It is also important to briefly introduce the main resources provided by the Application Framework in order to help the explanation of the offloading framework later. These entities are: activities, intents, services, broadcast receivers and AsyncTasks:

- **Activities**: The activity is a fundamental component of every Android application, it takes care of the user interactions. Every activity represent a single interface shown to the user, it is similar to a web page. Generally a common application has multiple activities. The main activity is always present in every Android application as a homepage for a website and it could show contents or also launch new activities. When a new activity is launched, the Android Framework pauses the previous activity, it imports the UI objects from the XML layout file and it allocates a certain amount of RAM memory. By default every activity of the same application is executed in the same Linux process but it is even possible to specify a different behaviour modifying the Android Manifest file. The activities management is an expensive job, so the Application Framework provides a specialized service: the Activity Manager. This component has the task to create, to destroy and to manage every single application of the system. At this point it is important to describe a bit the life cycle of an Android activity:
  - **Starting State**: the manager starts the initialization functions after checking the application is not runnig. In few words, it

provides memory and resources to the new activity.

– **Running State**: the activity is currently visible on the device screen and it owns the focus, it is managing the user interaction. During this state the running activity has the maximum priority and it can count on a proper amount of resources in order to provide the best user interaction. Only one activity per each application could be in the running state.

– **Paused State**: the activity has not the focus any more but he has still the maximum priority because it is visible to the user. A simple example could be a *dialog box* that pauses the activity taking the focus for a short time.

– **Stopped State**: the activity is now not visible. There is no user interaction, so the framework deallocates the memory assigned to it. This state could be followed by a new running state or by the destroyed state. In few words, it is possible to destroy or to restart a stopped activity.

– **Destroyed State**: the manager automatically decides that an activity is not necessary any more and it deletes it from the memory. The activity before being totally destroyed could run functions to save data or its state;

A more detailed representation of the life-cycle flow is shown in *Figure 4.3*.

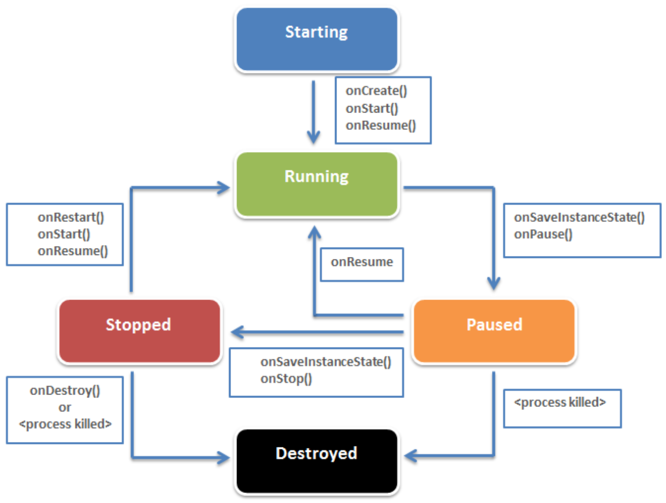


Figure 4.3: Detailed flow diagram of an Android activity life-cycle.

- **Intents**: An intent is simply a message which is sent to Android components. It could specify to launch a new activity, to start or to destroy a service or it could be used as a broadcast message for the Broadcast Receivers. The Intent is an asynchronous event, so the sending code could keep on executing without waiting for a reply. There are two different categories of intents: explicit or implicit; for the first kind it is needed to specify the addressee of the message, instead for the second kind it is possible to specify only the type of the recipient.

- **Services**: the services are designed to execute the same functions of an activity but without the user interaction. They run in background and they were introduced to be independent from the UI. In the one hand, a Service has a simpler life cycle than an activity, it can only be launched or destroyed. In the other hand, it is true there is less control on them for the developer and for the system in itself. Thus, it is really important to manage well the concurrent access of services to shared resources, such as RAM and CPUs.

- **Broadcast Receivers**: The Broadcast Receiver is an Android implementation of the common *Observer* pattern. The receiver could be registered on the Application Framework for a certain kind of intents. It will be awaken by the system only after receiving the desired type of intent, it means that it will be mostly in the standby state. Since the Broadcast Receiver does not have a UI or a reserved memory space it is only able to execute the specified code for a proper event. For this reason it is possible to compare it as a callback object.

4.2 AspectJ

In the **Section 4.1** Android was introduced as a mobile platform for application development. This section focus on AspectJ [29], a Java implementation of the Aspect Oriented Programming paradigm. A fundamental tool to manage interception and injection of code maintaining unchanged the original application. In order to understand how AspectJ works and how it is designed, it is necessary to introduce the software engineering Aspect Oriented Programming pattern. Finally, this section will focus on two important tools of the Java language: Annotations and Reflection. These two Java features help to provide a easy way to develop applications without mixing the offloading framework code with the business logic.

The AspectJ project is a seamless aspect oriented extension to Java [29]. The previous sentence states that the AspectJ's goal is to support the Aspect Oriented Programming, in addition that it is developed in Java. The company which released AspectJ is Xerox PARC (Palo Alto Research Center Incorporated). PARC is also well-known for other various contribution to information technology and hardware systems. The AspectJ software is totally open-source, it is indeed available in Eclipse Foundation projects. It is fully supported by Eclipse IDE and thanks to that, AspectJ is de facto a standard for AOP in the Java environment. Its success is mainly related to his simplicity and usability for the developers. Moreover, AspectJ is not only a Java framework but it is a programming language extension to Java. It uses a similar syntax to Java but it has different goals and naturally it is more focused on the AOP paradigm.

AspectJ is widely used from several well-known frameworks, such as

JBoss AOP [30], Nanning, Spring AOP [31] and AspectWerkz [32], the latter has been included in AspectJ 5. From a technical point of view AspectWerkz is significant for the thesis work because it is quite different from the AspectJ idea. In fact, while AspectJ defined a new language to specify constructs and models, AspectWerkz uses exclusively Java Annotations to achieve that goal. This feature of AspectWerks, also available in AspectJ 5, is really useful for the offloading framework implementation, in particular because it is strongly compatible with the Android development, this aspect will be presented later.

Before proceeding to explain in details the AspectJ custom language and its features, we need to briefly introduce some notion about the theory behind Aspect Oriented Programming.

4.2.1 The Aspect Oriented Programming

The Aspect oriented programming (AOP) has been proposed as a technique for improving separation of concerns in software development. AOP is the result of a combination of procedural programming and object oriented programming (OOP) and it is considered a significant improvement in software modularity [33].

The Aspect oriented programming core idea is that while the inheritance mechanism of the object-oriented programming is extremely important, it is often unable to modularise all aspects of a complex system. For that reason, AOP focuses more on system modularity in order to naturally spread out business logic of the software from support or utilities modules.

In order to achieve the goal previously discussed, AOP provides language mechanisms and tools which are able to represent the crosscutting nature of a complex system, such as file system access, database operations, remote communication support and etc. Thanks to AOP it is finally possible to program crosscutting concerns in a modular way obtaining simpler code, easier to develop and maintain, and potential reuse of the modules.

Before delving deeply into AOP, some standard terminology should be introduced to help understanding the concepts [34]:

- **Cross-cutting concerns**: In the OOP model, objects are designed to solve a single specific task or function but it is also true that they often share common secondary requirements. As mentioned earlier, a simple example could be the data-access layer or the interactions with the user interface. Thus, shared utilities and standard functionalities tasks are properly called Cross-cutting concerns;

- **Advice**: An Advice is an additional portion of code that the developer would like to add to the existing code. For instance, a classical timer behaviour, so it should be started before a certain piece of the software and then it should be stopped at the right moment. In particular there are three kind of advices: **before** the code, **after** the code or **around** the code (before and after);

- **Join point**: This term refers to the specific point of code execution in which the developer would like to apply the cross-cutting concern. Referring to the previous example, a point-cut is reached when the timer should start, and another point-cut is reached when the timer should stop, and for example it prints a log message.

- **Pointcut**: A Pointcut is a predicate that matches join points. More specifically, every advice is related to a certain expression, the pointcut, so every time join point is reached, the related rules are evaluated and only if the result is positive the advice is executed.

- **Aspect**: The combination of the point-cut and the advice defines an aspect. For instance the timer functionality added to the initial code.

- **Target object**: A target object is an object *advised* by one or more aspects. In literature it is also termed *advised object*.

- **Weaving**: Weaving is the fundamental operation of the AOP, it is defined as the action to assemble modules in their final form. The weaving operation is regulated by the rules defined in the aspects. An important distinction between weaving types is when this operation is executed: compile time or run-time.

The aspect oriented programming has a great potential for development uses and it is indeed possible to add fields, methods or interfaces to existing classes at compile time or even at run-time [35].

The AOP could be implemented following several different strategies and the result would be fundamentally various in terms of performances and overhead. I am going to list different solutions in order to highlight the differences between them:

- **Dynamic Proxies**: This solution is widely used, for example the famous lightweight middle-ware Spring has adopted it. It is based on the software engineering Proxy pattern and it exploits the Reflection mechanism of OOP languages, such as Java and C# to implement the AOP paradigm. More in details, the proxy is a surrogate or delegate object for an underlying object. A proxy has the capability to force method calls to execute in a certain execution moment without modifying the target object. So, it is possible to dynamically define proxies and to encapsulate target objects in them. As it is shown in **Figure 4.4**, to properly exploit this pattern is needed to define interfaces for target objects and to implement the proxy classes. In particular, the proxy class should implement the same interface of the target object in order to keep unchanged the way external objects collaborate with him. The last point is actually the key to allow interaction with other objects without affecting the original code.

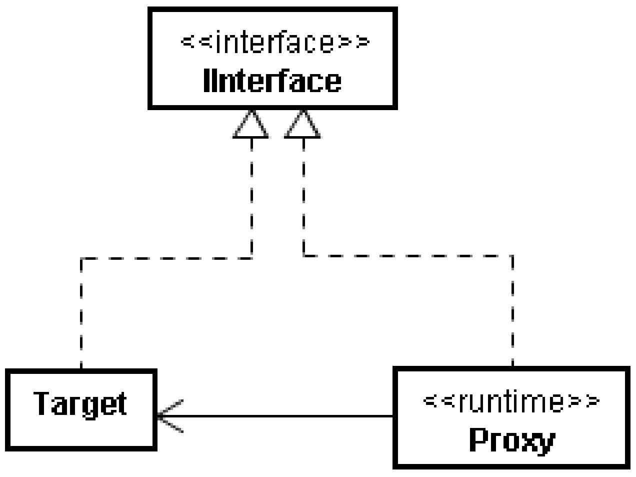


Figure 4.4: Proxy pattern representation.

- **Dynamic Byte Code Generation**: It is a low level implementation of the AOP. It generates dynamic subclasses and the target methods have hooks to invoke advices. This solution is also used by Spring framework. In other words, the AOP compiler inserts inside target classes references to advice objects, so when the code is called it automatically calls the right advice [36].

- **Java Source Code Generation**: It is a Java implementation of AOP. It is also a low level strategy, it practically creates new code sources including the crosscutting code. It is used in some EJB implementations and it is an extended implementation of Dynamic Byte Code Generation.

- **Custom Class Loader**: It is provided a custom class loader able to inject advices at loading time, in other words when the target classes are loaded. Adopting this solution it is important to keep in mind that he might cause issues with class hierarchy management. It is used by AspectWerkz.

- **Language Extension**: The most famous example of this strategy is indeed AspectJ project. It defines a new language or an extension of a programming language in which pointcuts and aspects are define as language classes or constructs. In this way this solution is able to provide an important feature: the inheritance between aspects.

4.2.2 Custom Language

After introducing the theory and the basic concept about Aspect-oriented programming in the **Section 4.2.1**, it is possible to proceed explaining more in details how AspectJ project is designed and what are its major features.

AspectJ adds to Java just one new concept, a join point [37]. A generic AOP framework behaviour is shown in **Figure 4.5**, it is also suitable to describe AspectJ architecture. In the **Figure 4.5** there are two source files: the Java source and the aspect definition file. The AOP compiler inject new AOP advices , with the cross-cutting concern code, inside the Java source files. It also resolves pointcuts in order to manage the weaving function. So, every time the execution reaches a join point which verify the pointcut expression the compiler inject the advices code in the Java class. The result of this process is a new modified binary file which include business logic of the application and crosscutting concerns.

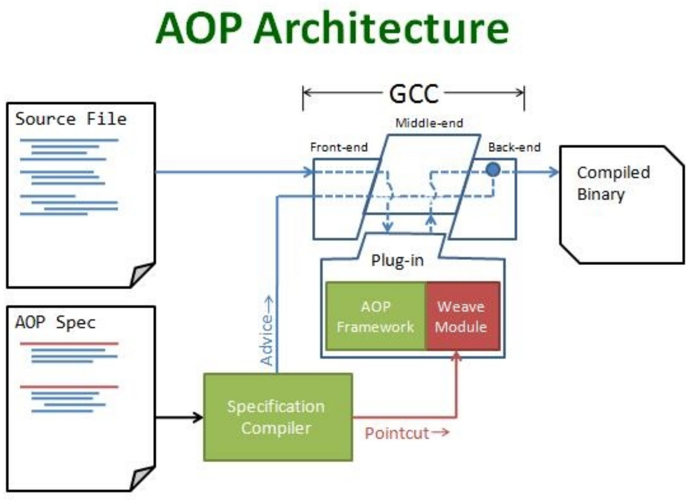


Figure 4.5: A general Aspect-oriented programming framework architecture.

The AspectJ project add new constructs to the standard Java language in order to represent the basic concepts of the AOP. Thus, it defines *point-*

cuts, *advice*, *inter-type declarations* and *aspects*. An important point is that pointcuts and advices are resolved at run-time, inter-type declarations are statically injected in the program class-hierarchy and aspects are the structure to define these new constructs. In order to deeply exploit the potentiality of AspectJ, we are going to show all the principal AspectJ language constructs and some related example to support the explanation.

As in AOP also in AspectJ a *Pointcut* picks up certain join points in the program execution flow.

```
call(void Point.setX(int))
```

Listing 4.1: Call API of AspectJ.

The previous code defines a new pointcut which picks up every join point with a signature corresponding to *void Point.setX(int)*. It is possible to combine more pointcuts and to obtain complex and specific behaviours. AspectJ also allows developers to define their own named pointcut, for example:

```
poitcut move(): call(void Point.setX(int)) ||
    call(void Point.setY(int));
```

Listing 4.2: Definition of a new AspectJ pointcut.

In few words, the code defines a pointcut which picks every join point with one of the specified signatures. The pointcuts shown earlier are generally explicit because the developer has to fully specify the signature of the method. It is also possible to define a poincut without specifying the whole method name but just using *wildcards*, properties of the related method. The AspectJ's team named this feature *property-based crosscutting*. A simple example is to pick up join points referred to every method which starts with *get* and which is defined by the class *Point* regardless the parameters signature.

```
call(void Point.get*(..))
```

Listing 4.3: Definition of a generic pointcut.

In some case the developer would like to pick up only methods owned, or better defined, by a certain class. Hereunder, there is a simple example to achieve this goal.

```
call(* Figure.*(..))
```

Listing 4.4: Pointcut which picks up every method of the Figure class.

In this case we are also using the property-based crosscutting avoiding indeed to specify the whole signature. Finally, for what concerns pointcuts definition, it is really important to introduce another AspectJ core functionality: *cflow*. It is not another wildcard as the previous examples, but it is a different tool able to identify join points contained in other join points context. Trying to clarify the concept, pointcuts defined by the *cflow* construct are able to pick up join point on whether they occur in the dynamic context of other join points. So, we are establishing a new pointcut which depends on a join point context, that could be defined as a more dynamic approach for pointcuts definitions.

```
cflow(start())
```

Listing 4.5: Sample of the AspectJ cflow API.

This pointcut picks up each join points which occurs after that the method *start* is called and before it returns a result.

Another fundamental construct is the advice. As mentioned earlier, pointcuts are able to pick up join points but they do not execute any type of code. So, in order to define the actual behaviour of a crosscutting concern it is needed to exploit the AspectJ advice construct. An advice is the clue that can bring together a pointcut and a certain defined code. This code is executed at each join points discerned by a pointcut. AspectJ provides a large number of advice types. So, it is relevant to define when the advice should be executed. In a simpler way, the advice execution depends on how it is defined by the developer: before, after or around the join point. For instance:

```
before(): start() {
    System.out.println("Starting...");
}
```

Listing 4.6: Definition of a advice: before.

From the code above, it is clear that the standard Java code included in the body of the construct represents the crosscutting concern function. In the example it is specified that before the execution of the method *start* the developer wants to print a message on the console. It is also possible to define the advice with *after* or *around* and to obtain an equivalent behaviour.

The **Inter-type declarations** are a powerful tool, they are able to modifying classes and their hierarchy acting outside the standard code. It is indeed possible to merge multiple classes or also to modify the inheritance relationship between them. The inter-type declarations are statically managed, it means that they are resolved at compile time and not at runtime. They could help to define a crosscutting concern at once and then the developer could insert methods and fields necessary to implement a new capability inside the target classes using inheritances or interfaces. An example could help to clarify this powerful feature.

```
aspect PointObserving {
    private Vector Point.observers = new Vector();

    public void notifyObservers(Point p, Screen s) {
        s.display(p);
    }

    public static void addObserver(Point p, Screen s
        ) {
        p.observers.add(s);
    }

}
```

Listing 4.7: Sample of the Inter-type declaration provided by AspectJ.

The example shows how to realise an observer object through an aspect and inter-type declarations. In practical inside the class *Point* it is created a new private field *observers*, visible to the encapsulating aspect *PointObserving*. In order to fully implement the observer pattern it should be defined a specific advice that notifies the observers registered at a certain join point.

```
    after(Point p): changes(p) {
        for(Screen screenObserver : p.observers) {
            notifyObserver(p,screenObserver);
        }
    }
```

Listing 4.8: Observer pattern implemented using the Inter-type declarations.

In this case, the advise notify the observer *Screen* after the creation of a new object *Point*.

The **Aspect** construct is implicitly shown in the previous example. It wraps advices, pointcuts and inter-type declarations inside a language module. This module is the complete representation of a cross-cutting concern in the AspectJ custom language. It is similar to a Java class and it could define methods, fields and eventually initialisers. By default AspectJ instantiates aspects as a singleton object and its internal advices could access to a non-static state as in Java classes. In addition it is possible to change the standard behaviour of aspects initialisation to achieve more complicated and specialised instantiations.

4.2.3 Annotation-based Development

The **Section 4.2.2** is focused on an overview of the basic constructs of the AspectJ custom language [38]. For the thesis project is particularly relevant to apply the AspectJ framework to a mobile environment, more precisely on the Android platform. As already mentioned, AspectJ 5 includes the AspectWerkz project. Therefore, AspectJ inherited the annotation-based definition for aspects implemented by the AspectWerkz developers. The adoption of annotations helps to achieve the thesis goal because it is possible to compile the final application using a regular Java 5 compiler, or a newer version. The weaver function is executed only after the standard Java compilation, for instance introducing an additional build state or a custom class loader. For the thesis project it is not needed to deepen the whole AspectJ annotation-based language and all its features but it is better to focus on aspect definition. For this goal, the AspectWerks annotation-based definition is the best solution.

As it is done in the **Section 4.2.3**, I am going to explain some feature of the annotation-based language in order to show the tools that are going to be used in the thesis project implementation. A new aspect could be defined using the **org.aspectj.lang.annotation.Aspect** annotation.
An example:

```java
@Aspect
public class MyAspect {}
```

Listing 4.9: Sample usage of the Aspect annotation.

Therefore, to be consistent with the **Section 4.2.2** it is shown its equivalent for the AspectJ custom language:

```
public aspect MyAspect {}
```

Listing 4.10: Definition of an aspect using the AspectJ common language.

For the sake of simplicity, in the following examples it will be only shown the annotation-based definition. In order to define a new pointcut it is needed the **org.aspectj.lang.annotation.Pointcut** annotation.

```
@Pointcut("call(* *.*(..))")
void anyMethodCalled() {}
```

Listing 4.11: Definition of a sample pointcut using the Pointcut annotation.

In the code above is defined a pointcut using the related annotation. The argument of the annotation is the body of the specific poincut. This pointcut simply picks up every method of the application. Generally, in a real environment the developer will defined meaningful rules. For what concerns the advice definition, AspectJ provides three different annotation based on the execution time:

- **org.aspectj.lang.annotation.Before**;
- **org.aspectj.lang.annotation.After**;
- **org.aspectj.lang.annotation.Around**;

An example for the advice *before* is:

```
@Before("anyMethodCalled()")
public void () {
  System.out.println("Method called!");
}
```

Listing 4.12: Advice definition using the Before annotation.

This advice is really simple but it highlights that the annotation argument is a pointcut defined previously. In other words, if AspectJ associates this advice to every methods of the application, and the final result is a console message every time a method is called. It is also possible to define inter-type declarations with the annotation-based language. The AspectJ documentation clearly states that inter-type declarations in the annotation-based language are limited. This decision has been made because the developer team wants to support compilation of AspectJ applications through a common Java 5 compiler. Thus, it is possible to define only inter-type declaration backed by interfaces, which means it is not possible to introduce new constructors or fields. In addition is not possible to call modified classes if they are not already woven and available in a binary form. By the way, this limitation does not affect the thesis project because it is possible to entirely manage the implementation using the limited inter-type declarations. It could be sufficient to show a complete example of the inter-type declaration construct.

```
@Aspect
public class MoodIndicator {

   // this interface can be outside of the aspect
   public interface Moody {
      Mood getMood();
   };

   // this implementation can be outside of the
      aspect
   public static class MoodyImpl implements Moody {
      private Mood mood = Mood.HAPPY;

      public Mood getMood() {
         return mood;
      }
```

```
    }

    // the field type must be the introduced
       interface. It can't be a class.
    @DeclareParents(value="org.xzy..*",defaultImpl=
       MoodyImpl.class)
    private Moody implementedInterface;

    @Before("execution(* *.*(..)) && this(m)")
    void feelingMoody(Moody m) {
       System.out.println("I'm feeling " + m.getMood
          ());
    }
}
```

Listing 4.13: A sample of code which shows the usage of Inter-type declaration in the AspectJ annotation-based language.

First of all it is needed to define an interface and a specific class which implements it. The **org.aspectj.lang.annotation.DeclareParents** is a inter-type declaration, the *value* argument specifies the target of the annotation, in our case every class member of the *org.xzy* package. The second argument *defaultImpl* represents the real class implementation of the interface. The target of the annotation is the field of the *Moody* interface. In the last part of the code we are defining a new advice. This advice called *feelingMoody* is picked up by the pointcut defined above it. The specified pointcut picks up every methods of the application, when it is executed and not called as the previous examples. In addition, we have the **this** operator which imports inside the advice the aspect object in itself. So as sown, it is possible to use the field *implementedInterface* as a standard Java field in the advice scope. Therefore, with this mechanism it is possible to define several scenarios and it helps to maintain unchanged the standard Java classes and methods.

4.3 Java Language Tools

AspectJ and some of its features are presented in the **Section 4.2**. In addition to AspectJ it is also needed some support from the Java language in order to provide a simple programming model for the offloading framework. So, the developer should not manually call the framework classes in order to execute the offloading functions and utilities but he could define plain Java Objects and leave all the complexity to the framework.

On one hand, the offloading software need a tool able to mark classes or methods and then to catch join points through AspectJ. Finally, the framework could invoke some advice to implement the crosscutting concerns, that is the offloading tasks.

On the other hand, the thesis project also needs a run-time tool to modify and to obtain information about living objects and their states. The first requirement could be fulfilled by the **Java Annotations**, whereas the second could be feasible using the **Java Reflection**.

4.3.1 Java Annotations

The Java annotations are a form of meta-data which provide additional information about the program. The annotations are used similarly to standard comments, in few words they are not a part of the source code. For this reason they cannot directly influence the behaviour of the code. The annotations could be affixed to the following Java constructs: classes, methods, variables, parameters and even packages. They could be considered as an alternative way to define instructions for the Java Virtual Machine(JVM) and compiler [39]. Thus, there are three main categories to distinguish annotations:

- **Compiler instructions**: they are able to influence the Java compiler behaviour, for this specific category of annotations the standard Java Development Kit provides three built-in instances: *@Deprecated*, *@Override* and *@SuppressWarnings;*
- **Build-time instructions**: A build-time annotation could ask a build

software, such as Ant, to generate and/or to compile a specific source code, to make a new formatted file or to manage packaging into JAR files. This kind of annotation is used by middlewares, such as JBoss, to generate deployment files.

- **Run-time instructions**: The annotation types introduced earlier are not able to provide information at run-time. In order to supply this functionality, Java allows to define new custom annotation and to keep them available during the program execution. The only way to access to these annotations is the *Reflection*;

As mentioned earlier, is not possible to affect directly the behaviour of a Java program using the annotation at run-time. This assertion is not totally correct, it is indeed possible to use run-time annotations to mark language construct and then decide to change the program execution flow. This functionality is possible through the Java Reflection which is deepened in the **Section 4.3.2**.

At this juncture, it is possible to introduce some annotation using some examples.

```
@Override
public void run() {}
```

Listing 4.14: Override annotation sample code.

This certainly is the commonest built-in annotation and as mentioned previously, this annotation is a compiler instruction. In few words, it specifies that the marked method has to replace the respective superclass one.

There are many built-in annotations and it is not fundamental for the thesis project to present them all. So, I will present only part of them needed for my thesis project. It is possible to define custom annotation. In fact, Java provides custom annotation definition in order to mark classes, fields and methods. In order to define a new annotation it is provided a built-in annotation **@interface**.

```java
@interface Author {
   String name();
   String date();
   String team();
   int group();
   int version();
   String[] collaborators;
}
```

Listing 4.15: Definition of a new annotation.

In this way we are defining an annotation, *Author* which contains several information about the marked code.

```java
@Author(
   name = "John Paul",
   date = "11/7/2015",
   team = "UI",
   group = 3,
   version = 2,
   collaborators = {"Jack Doe", "Frank James"}
)
public class Clazz implements SuperClazz {}
```

Listing 4.16: The sample code applies the Author annotation to the target class.

The previous code applies the defined annotation to a class, all the properties are filled with related data. As annotation properties is possible to use Java primitive types and strings. There is no support for inheritance between annotations. On the other hand, an annotation could be a property of another one. In addition is possible to mark an annotation with another one, this feature allows to add information and/or rules to a custom annotation definition. For example, it is relevant to present the **@Retention** annotation. It is fundamental to specify what category of annotation the developer is going to define.

```
@Rentention(RetentionPolicy.SOURCE)
@interface Author {
   String name();
   String date();
   String team();
   int group();
   int version();
   String[] collaborators;
}
```

Listing 4.17: Retention annotation sample code.

The example annotation is marked with the enumeration *RetentionPolicy.SOURCE*, it means that the related information is only visible in the source level, so generally they meant to be used only by IDEs. The RententionPolicy has three different value:

- *RetentionPolicy.SOURCE*: as I said, this part is available only in the source level;
- *RetentionPolicy.CLASS*: The marked annotation is maintained at compile-time but not at run-time;
- *RetentionPolicy.RUNTIME*: The information is kept at run-time and also at compile-time;

With the *@Target* annotation it is possible to specify what are the constructs on which the developer wants to apply the defined annotation.

```
@Target({ElementType.TYPE})
@Rentention(RetentionPolicy.SOURCE)
@interface Author {
    String name();
    String date();
    String team();
    int group();
    int version();
    String[] collaborators;
}
```

Listing 4.18: Target annotation sample code.

The above example shows the @Target annotation and its argument is an array of enumeration *ElementType* which represent the targets type of the custom annotation. In particular ElementType values are:

- *ElementType.ANNOTATION_TYPE*: can be applied to an annotation type;
- *ElementType.CONSTRUCTOR*: can be applied to a constructor;
- *ElementType.FIELD*: can be applied to a field or property;
- *ElementType.LOCAL_VARIABLE*: can be applied to a local variable;
- *ElementType.METHOD*: can be applied to a method-level annotation;
- *ElementType.PACKAGE*: can be applied to a package declaration;
- *ElementType.PARAMETER*: can be applied to the parameters of a method;
- *ElementType.TYPE*: can be applied to any element of a class;

There are more features available to extend the annotation mechanism, for the sake of my project it is not needed to deepen further Java annotations.

4.3.2 Java Reflection

An overview about the Java annotations was given in the **Section 4.3.1**. As mentioned earlier, the run-time annotation type needs to use the Java

Reflection ability. Therefore, the reflection core features will be presented in this section.

The Reflection is a special ability of a software to examine itself during its execution and to modify eventually its structure or behaviour. Several languages are equipped with the reflection, such as Java, C#, PHP, Perl and Python. For the thesis project the reference environment is the object oriented programming, in particular Java. In this scenario the reflection is able to inspect classes, methods and fields at run-time without any previous information, such as their names and types [39]. In addition it also allows to instantiate new objects during the execution. It is important to keep in mind that the reflection partially modifies the common approach to OOP, a program supported by this type of mechanism is indeed more dynamic than a common one. Using the reflection the developer could set up the program ignoring names and details about classes, methods and field. In fact, the goal is to discover this information at run-time gaining in terms of adaptability. For that reason, the reflection is really a useful tool for software testing and meta-programming.

```
Object myObj = MyClass.class.newInstance();
Method myMethod = myObj.getClass().
   getDeclaredMethod("myMethod", args);
myMethod.invoke(myObj);
```

Listing 4.19: Java Reflection APIs sample code. It defines a new instance of MyClass and then extract a defined method in order to invoke it later.

In this example there are shown two important features of the Java reflection: a new run-time instantiation and a run-time access to a declared method of the related class.

Another important ability of the Java language is the **introspection**, it is similar to reflection at first glance, however they cannot be considered equivalent. The introspection is the ability to examine types and properties of living objects at run-time. Therefore, the introspection is an additional feature which combined with the reflection one is a powerful toolbox to improve the software adaptability.

```
...
if(myObj instanceof MyClass) {
   MyClass myClassInstance = (MyClass) myObj;
   myClassInstance.myMethod();
}
```

Listing 4.20: The code checks the implemented class of a run-time object.

In the example above the program checks if the object is an instance of the class *MyClass* and, in the positive scenario, it invokes the desired method. Finally, it is useful to give an example about how to modify the value of a field at run-time.

```
Field field = MyClass.class.getDeclaredField("name"
   );
\\ if the field is private it is possible to
\\ force it to public.
field.setAccessible(true);
Object value = field.get(myObj);
System.out.println(value);
field.setAccessible(false);
```

Listing 4.21: The sample code modifies a field of a run-time object changing the its accessibility.

The Java language allows the developer to modify the accessibility of a certain field in order to manipulate its value using the reflection at run-time.

In conclusion, the reflection and the introspection are two interesting abilities of Java and some other OOP language. In particular,these two programming language abilities will seriously help the implementation of the offloading framework proving tools to add or to adapt dynamically the behaviour of the Android application.

Chapter 5

Project Architecture

*The **Chapter 3** focuses on the design of the thesis project. This chapter describes the implementation choices and details of the project. In the first section of the chapter it is introduced the architecture implementations and its related components. The second section starts with a presentation of test applications used to verified the designed software of the thesis and its performances. The adopted metrics and results are included in the second section of this chapter.*

5.1 Architecture

In the **Chapter 3** the thesis software is described from a high-level point of view. The macro components of its architecture are listed and their features are described. It is now possible to deepen the components and to explore some development choice to achieve the desired goals. The first part of the section focuses on the *Interceptors*, as mentioned previously, they are necessary to pick up a specific method execution and pre-execute or post-execute the offloading logic.
Following the application flow the intercepted code is forwarded to the Decision Engine component, which performs the task of taking the offloading decisions.
The Decision Engine forwards decisions to the Execution Manager. While the Decision Engine could be seen as the brain of the offloading service, the Execution Manager could be compare with the system hearth.

5.1.1 Interceptors Implementation

In the **Section 4.2** the Aspect-oriented programming, and in particular, the AspectJ framework are presented. As stated earlier, the AOP is a software engineering pattern designed to decouple in an effective way the business logic of the application from the corss-cutting concerns. The **Section 4.2.1** presented the Dynamic Proxy pattern, it makes possible the interception logic and it is natively support by the Java programming language. Since the target of the project are Android apps, the referenced native language is Java. So, the suitable solutions for the interception are two, and are both valid: Java Dynamic Proxies or AspectJ framework.

The Java Dynamic Proxies are extremely simple to configure and test. In the **Figure 4.4** is shown the model of the software engineering Proxy pattern. The target object and its related proxy should implement a common interface, this is necessary to keep transparent the proxy invocation on the behalf of the target. The proxy concept is implemented by the *java.lang.reflect.Proxy* class. In order to intercept the code it is also needed to implement the interface *java.lang.reflect.InvocationHandler*. This interface contains a single method, *invoke*, which has three defined arguments: the proxy object instance, the related *java.lang.reflect.Method* object, which represents the method, and the arguments list. In this method the developer can define its own interception logic.

```
...
@Override
public Object invoke(Object proxy, Method method,
   Object[] args) throws Throwable
{
   Object result;
   Log.d("MyInvocationHandler", "pre-execution");
   result = method.invoke(this.target, args);
   Log.d("MyInvocationHandler", "post-execution");
   return result;
```

```
    }
    ...
```

Listing 5.1: Code sample for the InvocationHandler invoke method. The code prints Android logs before and after the method execution.

Tthe code above gives a simple example of an implementation of the method *invoke* of *InvocationHandler* interface. The piece of code prints a string before and after the execution of the intercepted method.
The *java.lang.reflect.Proxy* is a factory class and in order to obtain a proxy object which implements a defined interface, it is necessary to call the method *newProxyInstance*. This method requires: the target interface class loader object; a class object array with all the interfaces that the proxy should implements; and a class which implements the *InvocationHandler* interface.

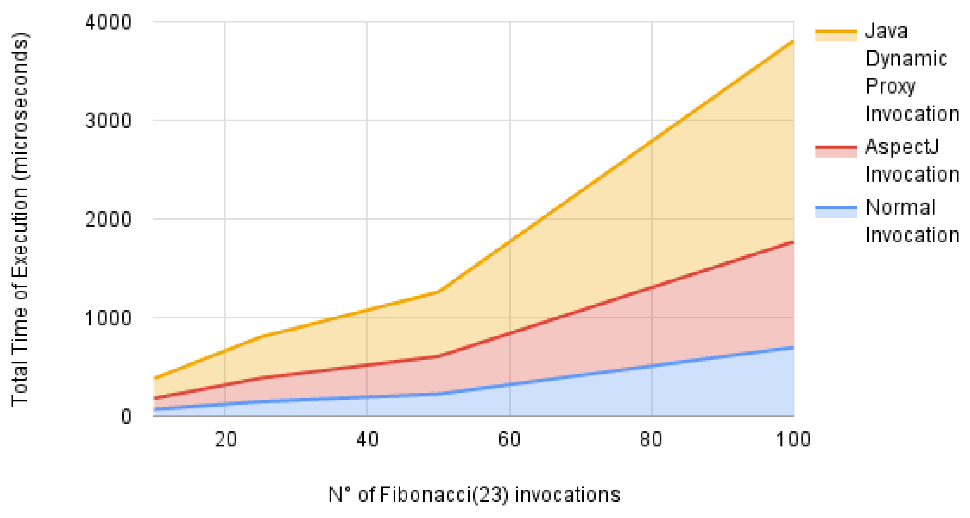


Figure 5.1: Comparison between the AspectJ and the Dynamic Proxy Proxy interception overhead on a Fibonacci's function implementation.

As explained in the **Section 4.2**, the same behaviour of the Java Dynamic Proxy could be obtain using the AspectJ framework. AspectJ should be faster than the other solution because it injects some parts of the code at compile-time and not entirely at run-time. In order to verify this idea we measured the overhead generated by the interception using the two different approaches.

The test method was a Java implementation of the Fibonacci's function executed for many times and the results are shown in the **Figure 5.1**. The AspecJ's overhead is more limited than the Java Dynamic Proxy approach. In addition, the AspectJ framework allows us to intercept annotated classes and methods without any kind of intervention of the developer. For the mentioned reasons, we decided to use AspectJ for the goals introduced in the **Section 3.4.2**.

```
@Aspect
public class AspectRemoteable {
   private static final String
      POINTCUT_METHOD_REMOTABLE =
      "execution(@Remoteable * *(..))";

   @Pointcut(POINTCUT_METHOD_REMOTABLE)
   public void methodAnnotatedWithRemoteable() {}

   @Around("(methodAnnotatedWithRemoteable())")
   public Object weaveJoinPointRemotable(
      ProceedingJoinPoint joinPoint) throws
         Throwable {
      ...
   }
   ...
}
```

Listing 5.2: Code extract from the AspectRemote definition. This aspect intercepts the invocation of methods marked with the @Remoteable annotation.

The extracted code is from the *AspectRemoteable aspect* class of the Offloading Middleware Service. The defined pointcut intercepts every single method marked with *@Remoteable* and it executes the cross-cutting concern define in the method *weaveJoinPointRemotable*. AspectJ provides an object *ProceedingJoinPoint* which representes the run-time status of the intercepted object. The *ProceedingJoinPoint* gives us the possibility to collect and modify the state of the target object, and to decide where and when to run the intercepted methods. We have implemented the same behaviour for the other annotations explained in **Section 3.4.2**.

5.1.2 Decision Maker Implementation

The Decision Maker, or Decision Engine, is the component designed to evaluate information from custom developer offloading policies and some additional data from the device status. So, it has principally two tasks: it should take track of the offloading policies defined by the user and evaluate the device state. In the **Section 3.4.2** it is shown how to define a new policy and how to assign it to a method or a class. From an implementation point of view the interceptors catch the definition and the instantiation of the new policy classes, they can also store a reference to its run-time object in a designed data structure. The Offloading Middleware Service contains a specific component for this task, *PolicyContainer*. It is a Java singleton object populated at run-time with the policies initialised in the developer code. Since the policies are meant to be dynamic and not static, the container should link to the related live object. When the Decision Maker evaluate the policies it queries the *PolicyContainer* in order to get the policy associated with the method or with the class of the method. In this way, when a *remoteable* method is going to be executed. the Decision Engine can evaluate the developer rules and decide if migrate the code on the remote machine.

However, there is still another condition to evaluate, the state of the device. The device could be disconnected or the data traffic could be not suitable for the communication. So, the offloading decision needs to consider this factors in order to be accurate. For the connection monitoring we use the Android monitoring tools. As mentioned in the **Section 4.1**, Android provides a complete stack to support the mobile applications development. In particular, Android provides a notification service to propagate information about the state of the device. In order to listen to this notification, it is necessary to use the *BroadcastReceiver* and register it to a specific intent type. For the network monitoring purpose the correspondent type is *Intent.ACTION_MANAGE_NETWORK_USAGE*. The Android system notifies the *BroadcastReceiver* when the state of the network is changed. The *BroadcastReceiver* can obtain the network information from the *ConnectivityManager*, for example the type of the network, WiFi or cellular networks, or the link speed of the connection. The same behaviour could be applied to the battery monitoring, Android indeed propagates an intent every time the battery percentage changes, its specific name is *Intent.ACTION_BATTERY_CHANGED*.

5.1.3 Execution Manager Implementation

The Execution Manager is the engine of the offloading mechanism. This component includes the logic: to encapsulate the run-time object in a container with all the necessary information to resume the execution remotely; to call the communication interface in order to send the message; to unwrap the received package from the remote machine; manage the execution; to send the object back with the updated states and its result; open the response package and resume the regular execution on the original endpoint. So, the internal architecture of the Execution Manager is divisible in two entities: a client and a server. The client part is in the device which runs the code and wants to offload it. The server part is in the machine which can support remotely the original execution of the device.

As mentioned earlier, it is thus possible to modularise the Execution Manager component in two main parts, client and server. The Execution Manager part running on the client is called Execution Controller and it is represented by the following interface.

```
public interface IExecutionController {

      Object executeMethodRemote(ProceedingJoinPoint
         execObject) throws Throwable;

      Object executeMethodLocal(ProceedingJoinPoint
         execObject) throws Throwable;
}
```

Listing 5.3: Interface definition of the IExecutionController. This interface provides two different methods for the local and the remote execution.

The two methods are dedicated are called by the Decision Maker or by the interceptors directly if the developer has specified a static partitioning for the related classes or methods. For instance if the method is marked with *@onMobileDevice* the Interceptors directly call the *executeMethodLocal* without querying the Decision Maker component. On the other hand, if the static decision specifies to run the code remotely the method called will be *executeMethodRemote*. The Execution Controller in this case will anyway query the Decision Maker in order to verify if the connection is established or if the available bandwidth is enough to migrate the code. The argument of the interface methods is a *ProceedingJoinPoint* object, which is the AspectJ representation of the intercepted target object. The object contains similar information to the *Reflection* representation of a run-time object.

On the one hand, the logic of the local execution operations is simpler than the other one. Because it is actually an invocation of the *proceed* method of the AspectJ *ProceedingJoinPoint* class. The AspectJ object will simply continue the regular local execution.

On the other hand, the remote execution is a more complex scenario to manage. After verifying that the offloading is viable, the Execution Controller has to wrap the run-time object with some related information in order to provide an executable bundle to the other side. The *ProceedingJoin-*

Point is not a *Serializable* object, that is a Java object not suitable for the network communication. For this reason, the Execution Controller needs to reconstruct a transferable object with some information from the *ProceedingJoinPoint* object and additional data. The described object will implements the interface *IExecutableBundle*. In the **Figure 5.2** it is shown the IExecutableBundle content in the UML format.

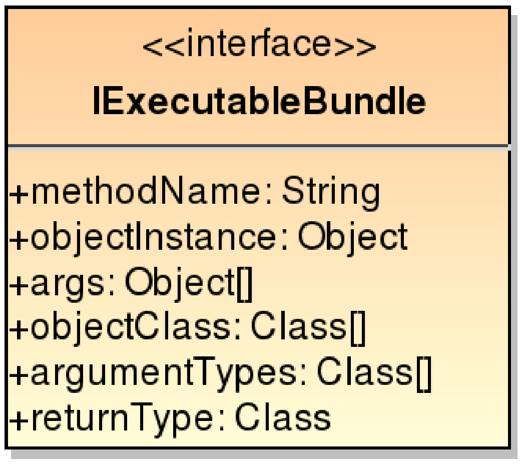


Figure 5.2: UML representation of the IExecutableBundle interface.

The IExecutableBundle implementation is the representation of the execution state. The communication layer transfers the bundle from an endpoint to the other.
The Execution Controller has to keep unchanged the sequential nature of the *offloadable* code. So, after the invocation of the sending method the Execution Controller blocks the related thread until the remote execution is not accomplished. Since The Android platform supports the multi-threading programming, the ExecutionController has to block multiple threads and to unlock the correct one when it receives the related response from the remote endpoint. The IExecutableBundle implementation indeed contains also the id of the related thread. This field is not shown in the UML above because it is a low-level implementation choice.

On the server side it is located the other part of the ExecutionManager, the *Code Offloading Execution Service (COES)*. The execution object is received through the communication interface. It performs the task of unwrapping the IExecutableBundle implementation and resume the execution from the other endpoint. The Code Offloading Execution Service is implemented using an Android service. As mentioned in the **Section 4.1.2** the Android services are designed to run code in the background without any user interactions. This decision is driven by the necessity to run the code even if the application is not shown on the screen. An Android activity would thus not be feasible for the described task. The server part of the Execution Manager needs to manage multi threads, different threads from the mobile devices may want to offload tasks. In more details, the COES needs to run every offloading request in a different thread in order to not affect the multi-threading design of the application.

5.1.4 Synch Controller Service Implementation

As mentioned previously, the developer could vary the synchronisation behaviour. The programmer could select two types of synchronisation: *lazy* and *eager*.
Firstly, the lazy synchronisation is simpler, in the moment of the offloading request the entire object state will be transferred on the other endpoint. This solution is always suitable but in some case the offloading could be seriously delayed by this approach.

Therefore, the eager policy of synchronisation could be suitable for big state transferring. The concept is indeed to keep consistent the mobile devices data with their related avatars. The developer can mark fields with *@SyncrhonizedField* in order to ask the offloading software to transfer the field state during the offloading functions. Thus, if the selected synchronisation policy is eager the Offloading Middleware Service updates a remote cache every time a field marked with *@SynchronizedField* is modified. As mentioned before, this approach could improve the offloading execution time, since the state of the run-time object is already available on the other machine.

The responsible for the eager state synchronisation is the *Synch Con-*

troller Service. Like the Code Offloading Execution Service, the Synch Controller Service is implemented as an Android service in order to be executed in a background thread. The state changes are stored in a data structure, it is defined by the StatusCache class.

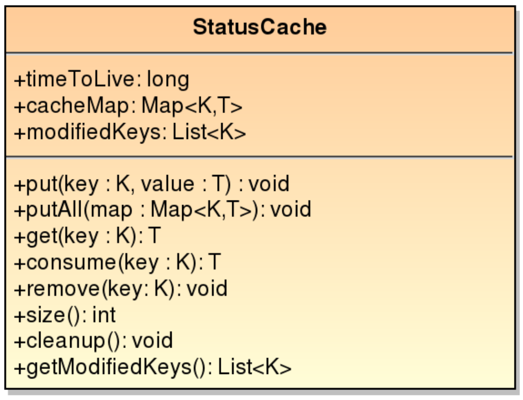


Figure 5.3: UML representation of the StatusCache data structure.

As it is shown in the **Figure 5.3** the data updated values are stored inside a *Map*, which is a Java native data structure. The StatusCache provides various methods to access or modify the content of the internal map. The map is implemented by a *java.util.concurrent.ConcurrentHashMap* to manage concurrency on the contents. In more details, the map uses the class *CacheKey* as keys and the class Object for the values. The CacheKey class is needed to reference live objects and to distinguish also different instances of the same class. For this purpose we use a couple of fields to identify each instance: an unique ID and the field name of the owner class. The Synch Controller Service is present in both the offloading endpoints, it manages the synchronisation based on a time-out in order to optimise the communication. The *modifiedKeys* field helps to keep track of the modified objects which are due to be synchronised at the next time-out expiration. The StatusCache has a time to live in order to be cleared and to avoid memory overloading.

When the Code Offloading Execution Service receives a package and if the assigned synchronisation policy is eager, it will instantiate a new object and put as internal state the updated one owned by the Status Cache.

The Code Offloading Execution Service could access the Status Cache only through the Synch Controller which manages the life cycle and the updates of the temporary memory.

5.1.5 Communication Interface Implementation

The communication support is an essential tool in order to make the offloading functions work. As mentioned in the **Section 3.1**, an important feature of the Moitree middleware is the communication support. Since the offloading middleware service should be used as a middleware service of Moitree, it is possible to use its communication layer. In particular, it is possible to define a Moitree's dedicated channel for the offloading mechanism.

The communication required for the code migration is peer to peer with a blocking behaviour. In simpler words, the execution on the mobile device should send directly the Execution Bundle to the related avatar and wait for a result. The nature of the Moitree communication is event-based, this means that it is not possible to use blocking native call. The Moitree APIs indeed allows the developer to specify callbacks related to a specific event. The Code Offloading Execution Service registers on the avatar a callback to unwrap the execution bundle received from the mobile devices and run the code.

```
...
AvatarApplicationInfo avatarApplicationInfo =
   AvatarApplicationInfo.getInstance();
      Avatar avatar = avatarApplicationInfo.getAvatar();
      avatar.setOffLoadingDataListener(this.channel, new
          AvatarOffloadingDataListener() {
         @Override
         public void onOffloadingDataAvailable(
            Serializable serializable) {
            IExecutableBundle executableBundle = (
               IExecutableBundle) serializable;
```

```
                try {
                    BlockingQueue<IExecutableBundle> queue =
                        null;
                    int Tid = executableBundle.getTid();
                    if(queueMap.containsKey(Tid)) {
                        queue = queueMap.get(Tid);
                    } else {
                        throw new NullPointerException();
                    }
                    queue.put(executableBundle);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (NullPointerException e) {
                    e.printStackTrace();
                }
            }
        });
    ...
```

Listing 5.4: The sample code registers a callback to the Avatar communication layer and it notifies the specific blocked thread when it receives the offloading result from the AVM.

In the mobile device we also need to register a callback to manage the offloading execution result messages. In addition, as stated previously, it is necessary to support the multi-threading behaviour. For this reason, we introduce a specific component which performs the task of multiplexing the communication channel and manage the unblocking of multiple threads. This component is named Communication Manager. It runs on a dedicated thread and it listen to the communication channel. The code above show the callback registered by the Communication Manager to manage the unlocking of multiple threads. Every thread which is waiting for the offloading response is blocked on a *java.util.concurrent.BlockingQueue* referenced by its own thread ID in a data structure. Thus, when an IExecutableBundle with a specific thread ID is received the Communication Manager put it in the related queue awaking its thread.

5.2 Evaluation

After introducing the design and the implementation choices of the offloading middleware service, we show some evaluation results by splitting the evaluation parts in: micro-benchmarks and macro-benchmarks. We have taken this decision in order to be sure that the system is working properly in terms of offloading performances without introducing delays and overheads. Since the project is integrated with the Avatar communication part, we need to separate the delay introduced by the communication and the actual performances of the offloading functions. Furthermore, as mentioned earlier, the thesis project aims to support not only single user applications but also distributed scenarios. The goal of the evaluation is to verify that the offloading software behaves correctly in both the environments, and we need to check it separately.

The first level focuses on the performances related to the offloading mechanism applied to a simple scenario, such as an end to end code migration. So, in this level we do not consider the distributed environment or the other group members as we stated in the **Section 3.4.3**. We want to measure how the software performs simply referring to the code offloading behaviour.

The macro-benchmarks level aims to show the evaluation of a distributed scenario application considering different policies specified by the developer. Therefore, we want to show how the Offloading Middleware Service could optimises specific parameters of a distributed app execution based on different developer policies.

5.2.1 Test Applications

We use two applications in order to test the offloading middleware service and to collect results.

The first application is an image manipulation app. The application allows the user to select an image and to select different kind of graphic filters, such as blurring, gray-scale rendering or inverting the colours. This computation could be done locally on the mobile device or rather using the remote virtual machines support. This application is particularly useful to measure the micro-benchmarks of the thesis project software. The image manipulation filters suitable for the task are characterised by heavy computation and the size of the target pictures is large enough to check the communication overhead. The user can select an image and apply different filters, such as grey scale filter and blur filter.

The blur filter is implemented by a Gaussian function with a user defined radius. We implemented two different versions of the same Gaussian implementation, one in plain Java code and one in C++. One of the offloading middleware service goal is to support NDK code offloading, for this reason we want to test how it behaves in this scenario. The code used for the Java and for the C++ implementation is relatively the same, considering the languages syntax and characteristics. In the **Figure 5.4** it shown the result of the Gaussian blur filter applied to an input picture.

The second application is FaceDate, an introduction of this application is in **Section 1.5**. However, the flow could be partially modified from the original version in order to make it more user-friendly and to test appositely the offloading middleware service. The original application trains the face recogniser engine at start time using the pictures included in a default directory. This requires the user to wait some seconds before start using the app. Furthermore, if the user specifies new preference images the app restarts the training to consider the new data. This decision was taken because the OpenCV [40] Java APIs did not support the face recogniser update. In the latest version the OpenCV developers have introduced an update method, so, it is possible to modify the recogniser state with one or more new pictures.

The new FaceDate version uses the update functionality of the OpenCV libraries. In the new scenario if the user adds some new pictures the application does not instantly retrain the recogniser. The application updates

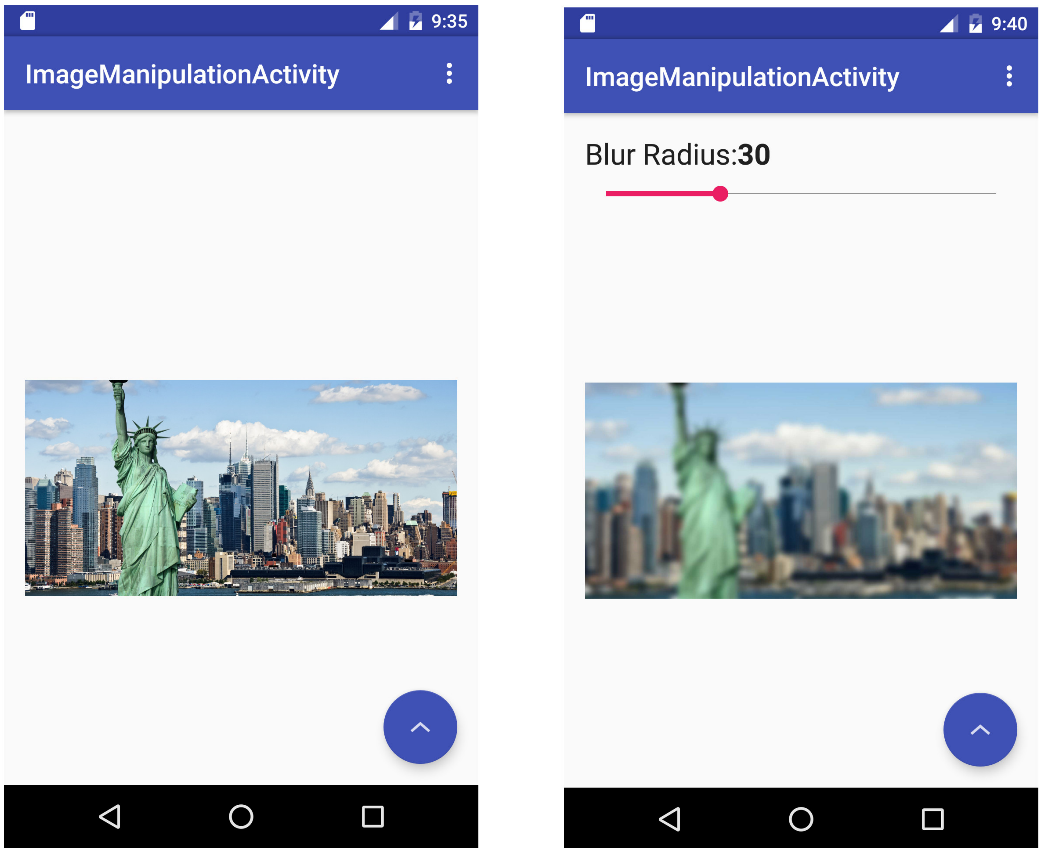


Figure 5.4: Input e Output of the Image Manipulation App used to test the Offloading Middleware Service.

the recogniser state only when a new request of match come from another user. This decision is made to avoid that the user waits the end of the training every single time he modifies the preference dataset. The update is a part of the find match functionality.

The offloading middleware service could be applied to the FaceDate new version, it can actually reduce the time needed for the training and to find a match. Furthermore, we can define different custom policies in order to modify the offloading behaviour of the FaceDate application. For test purpose we selected two extreme policies, a policy which mostly decides to offload the code in order to improve the performances and an opposite case in which is necessary to keep the computation local in order to guarantee the user privacy. We have found two samples that are applicable to real commercial application and they are simply to understand.

- *Privacy Policy*: This policy focuses on provide a high level of privacy. As mentioned earlier, the app uses human faces pictures, so users could use different faces to specify his preferences. The user may want to not transfer this information on the avatar, for instance because he does not trust to store sensitive pictures on the cloud. So, this policy executes all the functionalities related to the pictures on the mobile devices. For instance, the training and the update functions should be keep on the device because they need the pictures in order to accomplish their tasks. On the other hand, the match functions uses only extracted features to work, it is thus possible to offload this functionality to the Avatar VM;
- *Low-latency Policy*: The user who requests a search could care for a quick response, for instance the user is accessing a certain area and he wants to find a date before moving to a new place. The offloading should modify its behaviour based on the time of execution. For example, according to the time of execution estimation the on-line update could be done on the local device or on the avatar. If the time of execution of the mobile devices is less than the communication overhead the offloading software would decide to run locally. Conversely, if the offloading of the update method helps to decrease the time of execution, considering also the communication delay, the decision will migrate the code to the Avatar VM;

In the following code we assign the policy *FaceDatePolicy* to a the method *testUpdateAndMatch*.

```
@Remoteable(SynchType.LAZY)
public class FaceDetection implements Serializable
   {

   @CustomPolicy(policyClass = FaceDatePolicy.class
      )
```

```
    public int testUpdateAndMatch(byte[] imageBytes,
         PreferenceUpdateObject prefs)    {
        testTrain(prefs);
        return faceMatch(imageBytes);
    }

}
```

Listing 5.5: This code applies a policy to a defined method using the annotation CustomPolicy

The *FaceDatePolicy* is an implementation of the low-latency policy described earlier.

It is even possible to define more policies, such as battery consumption optimisation, network bandwidth preservation or location based policy. The latter indeed could influence the offloading decisions analysing the position, or it is even possible that the user may want to use only certain networks for the offloading because he is afraid of malicious activities in public places.

5.2.2 Metrics

In order to evaluate the results obtained in the test procedure it is necessary to introduce some metrics. We want to split the evaluation results in two level: micro-benchmarks and macro-benchmarks.

The micro-benchmarks focus on the offloading mechanism performances and they evaluate how well the offloading behaves with different computation level and/or with different data to transfer. As mentioned in the **Chapter 2**, the execution time of a piece of code is an important parameter to weight an offloading software. The battery preservation is another important goal of the offloading. So, we would like to collect the battery consumption of the test applications in the offloading scenario and in the plain mobile execution one, relatively.

The time of execution could be measure running the code multiple times and collecting the overall time needed to complete the task. After that, it is possible to estimate the average value in order to isolate the delays issued by the Android platform operations and by the Java language management. The overall execution time of the offloading task contains also the communication delay.

The battery consumption could be measure evaluating the energy needed to finalise the execution of a specific method. Unfortunately, the Android platform does not provide a fine-grained functionality like this. However, it is possible to measure the required battery needed to run an application. So, a workaround could be to run repeatedly a single method for a long time. The measurement would be done for both the scenarios, local and remote. The screen could be a big source of power draining but the evaluation would not be affected by it because the screen is used in both cases. This approach includes also the required energy for the communication.

The macro-benchmarks aim to evaluate the performances of the Offloading Middleware Service in a distributed scenario taking in account the custom policies defined by the developer. In this case we need to verify if the offloading software can really satisfy the developer policies and guarantee better performance for the specified parameters.

5.2.3 Micro-Benchmarks Results

For the first benchmarks level we have used two devices a LG Nexus 5 and a Motorola Nexus 6. While the Nexus 5 is equipped with a Quad-core Krait 400 @ 2.3 GHz and 2 GB of RAM, the Nexus 6 has a Quad-core Krait 450 @ 2.7 GHz and 3 GB of RAM. On the other side, we have used a server with a Hexa-Core Intel Xeon Processor E5-2620 @ 2.4 GHz and 82 GB of RAM. The server hosts a virtual machine with 6 virtual cores and 4 GB of RAM, our Avatar VM. The virtual machine and the two handsets are equipped with Android, the server instead has Linux Mint 64-bit. All the tests have been conducted on a WiFi network in the labs of the New Jersey Institute of Technology.

We have used the image manipulation app introduced in the **Section**

5.2.1 with different images. The goal of this evaluation is to verify the improvement in terms of time of execution introduced by the offloading. We want to verify that the increment of the computation complexity would also improve the contribution of the offloading technology. In addition, another parameter to check is the communication overhead introduced to move the picture from the local device to the avatar machine and to move back the blurred image.

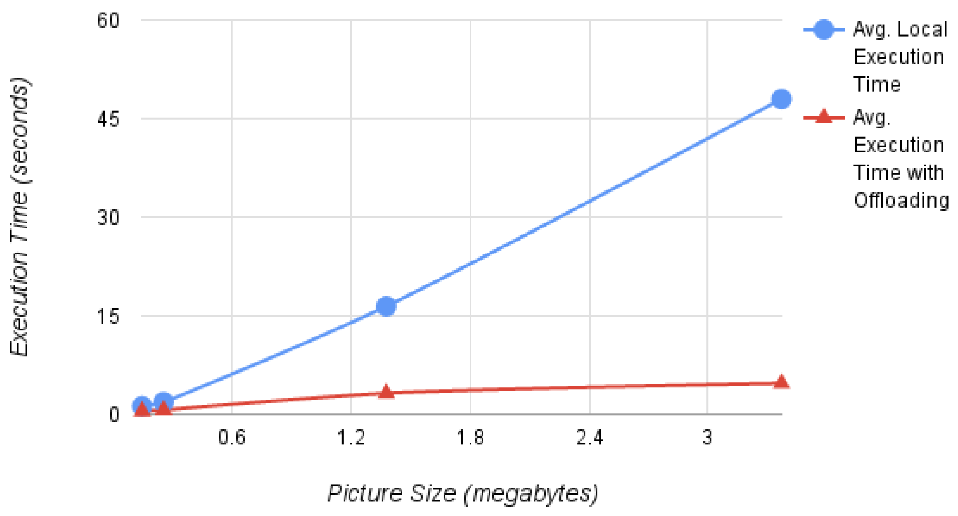


Figure 5.5: Results collected executing a Gaussian Blur filter with the offloading support and without it. In this case the code filter is implemented with Java.

The **Figure 5.5** shows the result collected during the experiments. The Gaussian filter computation varies based on the image size. If the picture is relatively small the offloading contribution is null, or better its overhead introduced delays. In this case a smart decision could be to execute the blur functions on the mobile device. However, when the size of the pictures increase and consequently the computation, the offloading contribution becomes stronger. The benefits of the offloading mechanism are particularly visible in the last measurement. We aim to provide the support for the Android NDK because it is particularly suitable for the development of computing intensive application, such as signal processing, computer vision, data mining and real-time apps. For this reason we also implemented the filters in C++ code and the **Figure 5.6** shows the collected results.

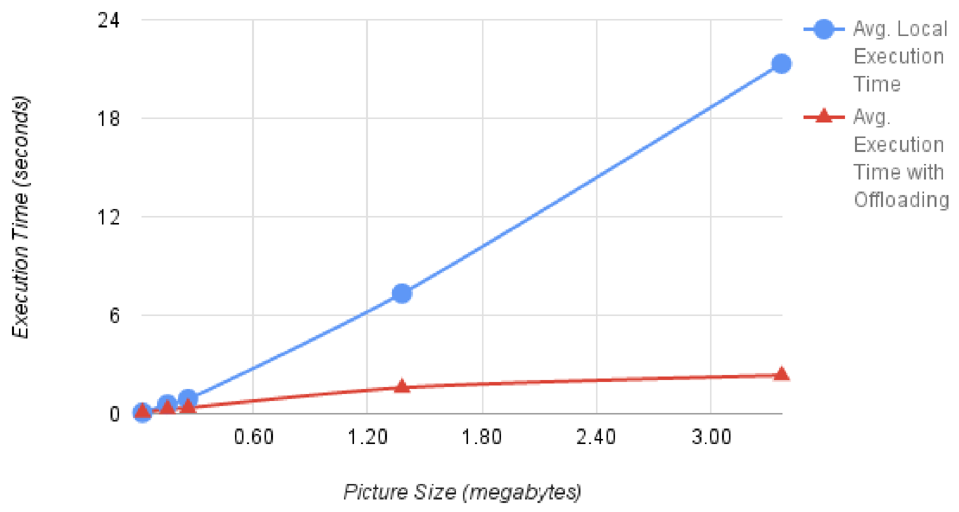


Figure 5.6: Results collected using a C++ implementation of the Gaussian Filter function.

As expected, the native code version is faster than the Java implementation, however, the offloading has positive results even in this case. If we compare the local execution in the Java version with the offloading execution in the C++ scenario, the needed time is many times lower and the user experience would be positively influenced.

The other micro-benchmarks parameter defined in the **Section 5.2.2** is the energy consumption. We have done an additional experiment to test what is the impact of the offloading mechanism on the energy consumption. As stated earlier, the battery is an important feature of mobile devices and we want to be sure that our system is capable to preserve energy. We have done the experiment using the same application and a constant image. We have extend the time of execution in order to see how

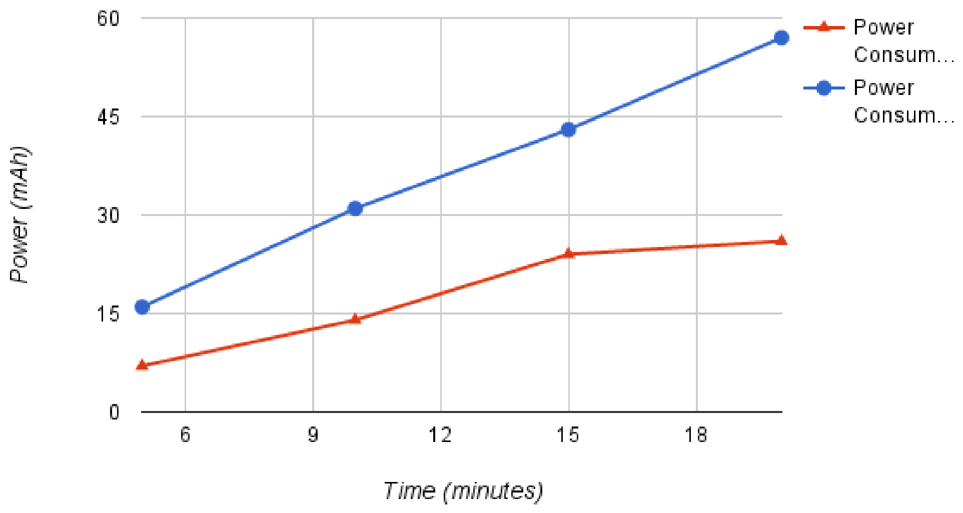


Figure 5.7: Power consumed by the image manipulation app with the support of the offloading software and without it.

the offloading energy consumption behaves in increasing time. We have applied a Gaussian filter function on the same picture repeatedly in order to have stronger results, not driven by unpredictable variables.

The **Figure 5.7** shows that the power consumed by the offloading supported execution is lower than the simple local one. In addition, the energy consumption of the local scenario increases linearly in time, inversely the offloading behaviour is sub-linear. Therefore, the offloading technology saves battery and it particularly behaves better in case of durable executions.

Moreover, it is important to realise how much the offloading functions impact on the regular execution. We want to measure the migration overhead weight on the total time. This delay is introduced mainly by two factors: interception of the methods invocation and the language reflection, and the communication time introduced by the Avatar system. We have collected this measurements using the same image manipulation app. In this case also we have used different images with incremental size. The

Table 5.1 shows the collected results and it is possible to see that the overhead introduced by the offloading mechanism is generally low. Before the code execution of the actual method, the Interceptors and the Execution Manager generates some delay in order to pick up the method execution and also to extract and prepare the state of the run-time object.

Execution Time Offloading (milliseconds)	**Overhead Interception and Run-time State Extraction Time (milliseconds)**	**Overhead State Update (milliseconds)**	**Percentage Overhead on Total Execution Time**
3212	2.11	0.06	0.06%
664	2.75	0.07	0.40%
490	2.93	0.08	0.61%
145	2.79	0.06	1.97%

Table 5.1: Results introduced by the offloading functions.

The last column of **Table 5.1** shows the percentage of the total overhead respect to the total time of execution. The percentage is negligible compare to the total time of execution, and this consideration is particularly suitable for big heavy computation, as the first row of the table.

The communication overhead introduced by the offloading mechanism is another important factor to check. We have collected this information during the execution of the image manipulation app. The data transferred and the time necessary to transfer it on a WiFi network is shown in the **Table 5.2**. Like the previous parameters, the communication cost

Execution Time Offloading (milliseconds)	Communication Time (milliseconds)	Transferred data
3212	13	237.312Kb
664	4	61.362Kb
490	2	36.437Kb
145	0.1	6.701Kb

Table 5.2: Table with communication information collected during the image manipulation app.

of the experiment is generally negligible even if the size of the data is increased. Naturally, if the data sizes are too big the gain obtained with the offloading would be mitigated or in the worst cases cancelled.

5.2.4 Macro-Benchmarks Results

This section focuses on the macro-benchmarks evaluation. In the **Section 5.2.3** there are listed the results obtained from a single user application. We have done experiments in order to evaluate the Offloading Middleware Service in a distributed scenario. The sample application used for this experiments is FaceDate, already presented in the **Section 5.2.1**.

The tests have been conducted using up to four devices: a LG Nexus 5, a LG Nexus 5X, a Oneplus One and a Motorola X Pure. As mentioned in the **Section 5.2.3**, the Nexus 5 is equipped with a Quad-core Krait 400 @ 2.3 GHz and 2 GB of RAM. The Nexus 5X has a Quad-core Cortex-A53 @ 1.82 GHz supported by a Dual-core Cortex-A57 @1.82 GHz and 2 GB of RAM. The Oneplus One device is equipped with a Quad-core Krait 400 @ 2.5 GHz and 3 GB of RAM. Finally, the Motorola X Pure adopts a Hexa-core Qualcomm Snapdragon 808 @ 1.8 GHz and 3 GB of RAM. We have decided to use different devices in order to have a strong feedback about the software in a heterogeneous system.

We have run three different experiments using a variable number of devices. As stated in the **Section 5.2.1**, we introduced the update feature in the FaceDate app. The experiments aim to show how the Offloading Middleware Service reacts and modifies the app performances based on the developer policy. We played the roles of developer in this case and we have defined two different policies already presented in the **Section 5.2.1**, a privacy and a low-latency policy. This policies have two different behaviours about the offloading decisions: the privacy policy avoids every type of transferring of sensible pictures like preference images; the low-latency policy aims to optimise the execution time needed to accomplish a date search, so it probably would prefer the offloading of heavy task with small data transferring instead of running locally. As explained in the **Section 5.2.1**, when the user adds new preference images the app does not update immediately the recogniser state but it waits for the next search in order to decide if it is better to offload or not based on the ruling policy specified by the initiator user.

The first experiment has been led with two devices: a Nexus 5X and a Motorola X Pure. The Motorola X Pure has been selected as user who requests for a date, the Nexus 5X has played the rule of a normal member of the group who replies to the request. On the Nexus 5X we have varied the number of pictures to update in order to see how the relationship between the low-latency execution and the privacy one.

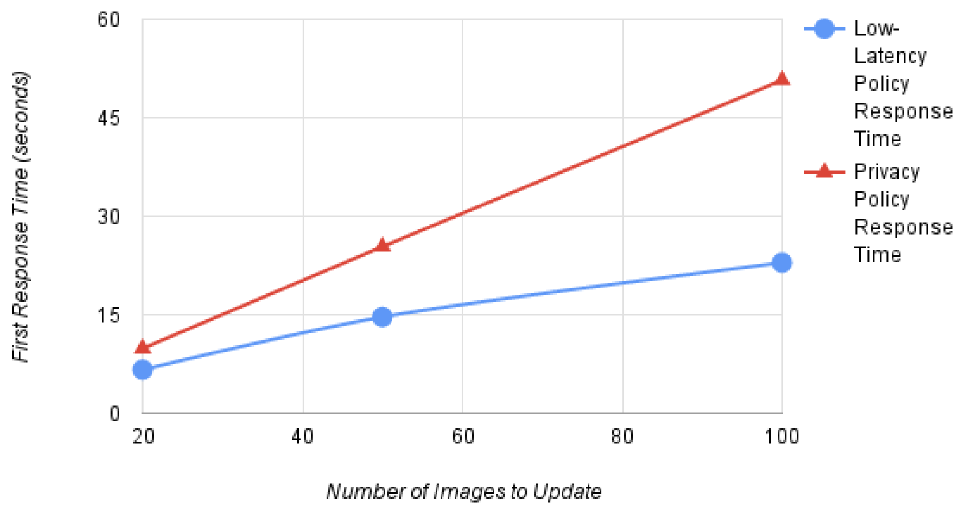


Figure 5.8: Experiment results of the FaceDate application using two devices and varying the number of pictures to update.

The results are similar to the image manipulation ones because in the low-latency policy scenarios the Decision Maker component decides to run the code remotely in order to improve the execution time. The **Figure 5.8** shows the results collect which support the previous considerations. In the privacy policy, the Decision Maker avoids the offloading mechanism for training and updating of the recogniser because it cannot transfer the pictures through the network. However, the face matching functions are executed remotely because they do not need the real pictures.

The increase of the picture numbers highlights the performances of the offloading scenario against the simple local execution. The time represented on the Y axis is the first response received from the participant devices, in this first case there is only one participant.

We have increased the number of participants to three in order to verify if the results would be similar or strongly different. We have selected a Nexus 5X, a Motorola X Pure and a Nexus 5. The results collected are similar to the previous experiment. The low-latency policy drives the Decision Maker component to run the code remotely, since the connection is fast and the performances gain in terms of computation is bigger than the introduced communication delay. The results described are shown in the **Figure 5.9**

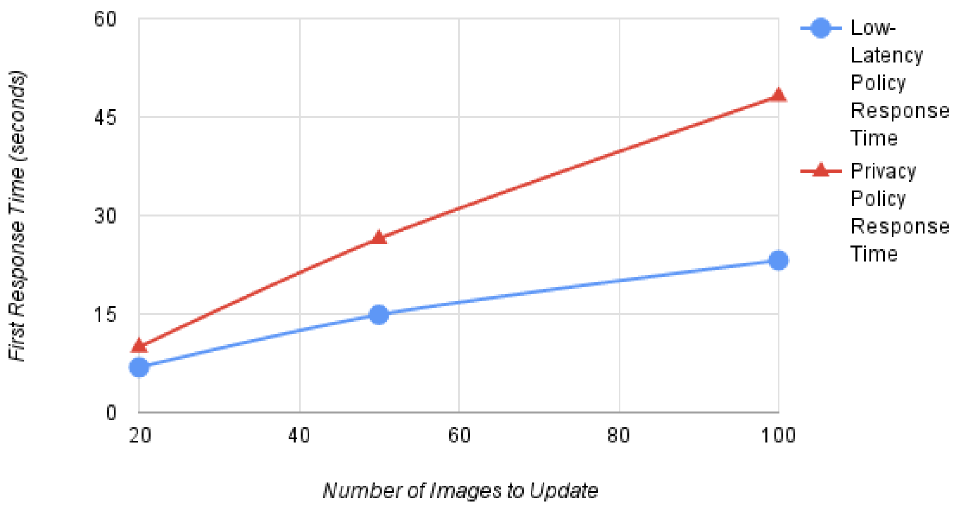


Figure 5.9: Second FaceDate experiment using three devices.

The number of devices does not modify the result in a strong way. The distributed nature of the application supports this consideration and it is also true that increasing the number of devices would not affect strongly the first response time. It is the first device to response the real factor to influence the results and not the number of them.

We have run a last experiment with four devices, so the same devices including a Oneplus One. As expected the results have been coherent with the previous affirmation and they are shown in the **Figure 5.10**. The collected results have confirmed that the scalability of the offloading functions is guaranteed and the overall execution time for a search request in scenarios with a different number of devices is almost constant. The communication layer needs to use a broadcast communication in order to propagate the user request. The Avatar communication introduce some delays in order to deliver the information to the other members, therefore an improvement of the communication layer will also affect the results shown in the graphs, improving certainly the time of execution of a research. It is also possible to unplug the Avatar Communication Manager and replace it with a different solution in order to obtain better results. The offloading software could work in many systems which provide VMs as cloud infrastructure supports. In this case, naturally the communication constraints will vary and so the overall time of execution.

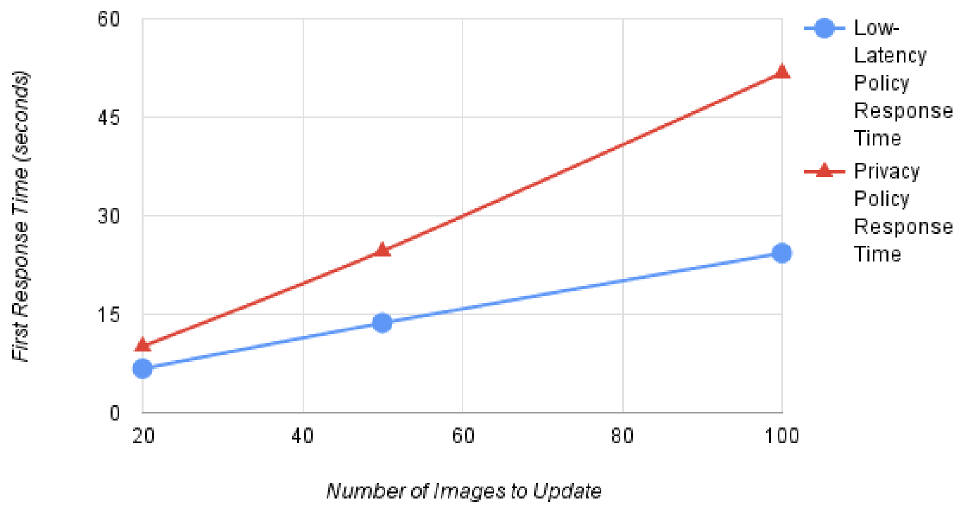


Figure 5.10: Last Experiment using four devices.

The results collected in the last two experiments have highlighted another important consideration. The offloading technology in a distributed scenario is capable to balance the device hardware specifications. The first response in the privacy policy scenario is always from the Motorola X Pure because it is the most powerful device used in the experiments. In the low-latency cases the *winners* are almost random. In order to clarify this concept it is shown the **Table 5.3**.

Number of devices	Number of Images to Update	Device of First Response for Privacy Policy	Device of First Response for Low-Latency Policy
3	20	Motorola Pure X	Nexus 5
3	50	Motorola Pure X	Motorola Pure X
3	100	Motorola Pure X	Motorola Pure X
4	20	Motorola Pure X	Nexus 5X
4	50	Motorola Pure X	OnePlus One
4	100	Motorola Pure X	Motorola Pure X

Table 5.3: This table shows the first device to response to the initiator user in the FaceDate app comparing the low-latency policy with the privacy one.

5.2.5 Results Considerations

The results collected during the experiments have driven us to many considerations.

On the one hand, from the micro-benchmarks data we have confirmed that the offloading technique it is really capable to improve the performances particularly with high computation tasks. On the other hand, the development choices adopted have guaranteed good results in terms of system overhead on the distributed scenario.

The battery consumption observed is really a positive feature of the thesis project, since it can improve the user experience and it can also support longer execution for mobile applications. The offloading software designed could meaningfully help the distributed application ideas of the **Section 3.3**. In addition, the developer could use the policies definition in order to improve desired parameters, for instance we have improved the time of execution in the experiment scenarios. For the input data provided to the policies classes the developer could use every kind of profilers or monitors and he can develop his own new policies to have ad-hoc behaviours.

As mentioned earlier, the execution supports by virtual machines could balance the user devices hardware specification making the application result fairer. For instance, in scenarios similar to the FaceDate application if two users are suitable for the face match the winner user would be random and not always the same with a more powerful device. In this case for the participant users the offloading also changes the logic of the application giving more balanced results.

Firstly, the thesis project provides a easy-to-use programming tools based on Java annotations to allow the developer to simply define Plain Old Java Objects without changing the original code. Furthermore, the programmer might want to define custom policies to feed the decision engine of the thesis project. In this way, the programmer can influence the offloading decision. A new policy could be defined as a simple Java class and then the system can identify it through the annotation applied on them. The scope of the policies is method-level, therefore the developer can assign a policy to a single method or to a defined class.

On the one hand, we designed an offloading mechanism able to evaluate rules based on various parameters and migrate parts of code executions. This is a common way to augment the Mobile Computing resources and the first step in order to check the performances of the thesis project has been to collect results using sample applications and varying their computation load. We have observed that not only the time of execution has benefits of the cloud support but also the battery has been better preserved. Finally, we have tested the offloading of C/C++ code in order to verify the behaviour of the system using the Android NDK.

On the other hand, we have designed the system to be modular, it will be indeed possible to modify or replace entire components in future. For example, we have used simple profilers for the local mobile state but in future applications the system may need more accurate tools, such as battery estimation algorithms, location based information and computation load monitoring. In addition, it is possible to provide some standard policies for the developer, such as simple policies for common conditions, such as battery preservation, bandwidth usage or location based policies.

Finally, the project is equipped with a simple cache memory used for the eager state synchronisation between a mobile device and its avatar. This component is extensible with more powerful features in order to entirely manage the state synchronisation between the endpoints. This new feature could furthermore improve the performances of the system and it also could supply fail-tolerance support, enabling the execution resuming after a device de-synchronisation.

Conclusions

The solution proposed by the thesis aims to provide a computation offloading middleware service for the Avatar system. Our project is not only able to migrate code from mobile devices to Avatar virtual machines but it is also able to evaluate rules in order to decide whether to offload a piece of code or not.

Firstly, the thesis project provides a easy-to-use programming tools based on Java Annotation. The developer can simply define Plain Java Old Object without changing the original code but just marking specific methods or classes with the provided annotations. The programmer might similarly want to define custom policies to feed the decision engine of the system and in this way influencing the offloading decisions. The developer can define a custom policy defining a simple Java class, he can also mark a method with it and then the system will automatically associate the method to the defined annotation.. The scope of the policies is method-level and so the developer could assign a policy to a single method or to entire class.

We also want to enlarge the concept of computation offloading to mobile distributed applications. The custom policies definition feature helps in this goal providing a simple way to evaluate information related to different members of the distributed group. Therefore, our purpose is to provide different offloading decisions based on run-time user requests and/or global distributed resources. The selection of different policies has shown an improvement of the target parameter at run-time, for example the low-latency policy has improved the overall time of execution. In addition, we learned that the offloading technology applied to a distributed environment is able to reduce the hardware specifications gap between different mobile devices.

Finally, the thesis project has been designed and implemented to be compatible with the Avatar system. However, it could be possible to use the software without the Avatar support replacing the communication layer. The project could be applied to new systems, the unique hypothesis is that the VM would execute the same application of the mobile device. We indeed designed the system to be modular and so it is simply to unplug components and replaces them with new implementations. It is possible to unplug the thesis project from the Avatar system and to connect it to other solutions. The offloading software indeed could work with many commercial solutions which provides virtual machines and related management tools. The performances of the thesis project partially depends on the communication layer and also on the virtual resources of the VM. As stated previously the modular design of the thesis project enables to replace components. In this way, it is possible to adapt the system to new scenarios and to include new features.

Bibliography

[1] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile computing middleware. In *Advanced lectures on networking*, pages 20–58. Springer, 2002.

[2] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[3] David E. Bakken. Middleware. `http://www.eecs.wsu.edu/~bakken/middleware.htm`.

[4] BELLAVISTA P.; CORRADI A. EDS, editor. *Mobile Middleware: Definition and Motivations*, NEW YORK, 2006. Auerbach (CRC press).

[5] C. Borcea, Xiaoning Ding, N. Gehani, R. Curtmola, M.A. Khan, and H. Debnath. Avatar: Mobile distributed computing in the cloud. In *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2015 3rd IEEE International Conference on*, pages 151–156, March 2015.

[6] Alan LaMont Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[7] Aniruddha Gokhale and Douglas C Schmidt. Principles for optimizing corba internet inter-orb protocol performance. In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, volume 7, pages 376–385. IEEE, 1998.

[8] P Reynolds and R Brangeon. Service machine development for an open longterm mobile and fixed network environment. *Project deliverable, DOLMEN Consortium*, 1996.

[9] Timm Reinstorf, Rainer Ruggaber, Jochen Seitz, and Martina Zitterbart. A wap-based session layer supporting distributed applications in nomadic environments. In *Middleware 2001*, pages 56–76. Springer, 2001.

[10] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service. *Sun Microsystems Inc., Santa Clara, CA*, 2002.

[11] Andrew T Campbell. Mobiware: Qos-aware middleware for mobile multimedia communications. In *High Performance Networking VII*, pages 166–183. Springer, 1997.

[12] Yating Wang, Ray Chen, and Ding-Chau Wang. A survey of mobile cloud computing applications: Perspectives and challenges. *Wireless Personal Communications*, 80(4):1607–1623, 2015.

[13] Saeid Abolfazli, Zohreh Sanaei, Erfan Ahmed, Abdullah Gani, and Rajkumar Buyya. Cloud-based augmentation for mobile devices: motivation, taxonomies, and open challenges. *Communications Surveys & Tutorials, IEEE*, 16(1):337–368, 2014.

[14] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[15] Shadi Ibrahim, Hai Jin, Bin Cheng, Haijun Cao, Song Wu, and Li Qi. Cloudlet: towards mapreduce implementation on virtual machines. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 65–66. ACM, 2009.

[16] Saeid Abolfazli, Zohreh Sanaei, Muhammad Shiraz, and Abdullah Gani. Momcc: market-oriented architecture for mobile cloud computing based on service oriented architecture. In *Communications in China Workshops (ICCC), 2012 1st IEEE International Conference on*, pages 8–13. IEEE, 2012.

[17] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003.

[18] Margaret van Steenderen. Universal description, discovery and integration. *SA Journal of Information Management*, 2(4), 2000.

[19] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, 2013.

[20] 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 *INFOCOM, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[21] Z Yang. Powertutor-a power monitor for android-based mobile platforms. *EECS, University of Michigan, retrieved September*, 2:19, 2012.

[22] 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*, pages 49–62. ACM, 2010.

[23] Common language runtime (clr). `https://msdn.microsoft.com/en-us/library/8bs2ecf4`.

[24] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri Bal. Cuckoo: a computation offloading framework for smartphones. In

Mobile Computing, Applications, and Services, pages 59–79. Springer, 2010.

[25] Android interface definition language (aidl). `http://developer.android.com/guide/components/aidl.html`.

[26] Mohammad A. Khan, Hillol Debnath, Nafize R. Paiker, Naharain Gehani, Xiaoning Ding, Reza Curtmola, and Cristian Borcea. Moitree: A middleware for cloud-assisted mobile distributed apps. In *The 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, 2016.

[27] Android developers. `https://developer.android.com/sdk/index.html`.

[28] Apple's ios. `http://www.apple.com/ios/`.

[29] AspectJ Team. The aspectj programming guide, 2003.

[30] Marc Fleury and Francisco Reverbel. The jboss extensible server. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 344–373. Springer-Verlag New York, Inc., 2003.

[31] Spring. `https://spring.io`.

[32] Jonas Bonér. Aspectwerkz–dynamic aop for java. In *Invited talk at 3rd International Conference on Aspect-Oriented Software Development (AOSD)*. Citeseer, 2004.

[33] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97—Object-oriented programming*, pages 220–242. Springer, 1997.

[34] LÁSZLÓ Lengyel and TIHAMÉR Levendovszky. Introduction to aspect-oriented programming, 2005.

[35] G Chavez Christina von Flach and Carlos JP de Lucena. A theory of aspects for aspect-oriented software development, 2010.

[36] Jeremy Blosser. Explore the dynamic proxy api. `http://www.javaworld.com/article/2076233/java-se/explore-the-dynamic-proxy-api.html`, Nov 2000.

[37] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse aspectj: aspect-oriented programming with aspectj and the eclipse aspectj development tools*. Addison-Wesley Professional, 2004.

[38] The aspectjtm 5 development kit developer's notebook. `https://eclipse.org/aspectj/doc/released/adk15notebook/index.html`.

[39] The java^{TM} tutorials. `https://docs.oracle.com/javase/tutorial/java/`.

[40] Gary Bradski et al. The opencv library. *Doctor Dobbs Journal*, 25(11):120–126, 2000.