CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Repositório Científico do Instituto Politécnico do Porto

Departamento de Engenharia Informática

Instituto Superior de Engenharia Informática

Code offloading on Real-time Multimedia Systems

A Framework for handling code mobility and code offloading in a QoS Aware Environment

Guilherme Rios de Sousa e Silva

Dissertação para obtenção do grau de Mestre em

Engenharia Informática.

Área de Especialização em **Sistemas Gráficos e Multimédia**

Orientador

Professor Doutor Luís Lino Ferreira

Júri

Presidente: Professora Doutora Maria de Fátima Coutinho Rodrigues,Professora Coordenadora no Departamento de Engenharia Informática do Instituto Superior de Engenharia do Porto

Vogais: Professor Doutor Filipe de Faria Pacheco Paulo, Professor Adjunto no Departamento de Engenharia Informática do Instituto Superior de Engenharia do Porto

Professor Doutor Luis Miguel Moreira Lino Ferreira. Professor Adjunto no Departamento de Engenharia Informática do Instituto Superior de Engenharia do Porto

Porto, Outubro de 2011

Acknowledgments

It would not have been possible to write this thesis without the help and support of the people around me, to only some of whom it is possible to give particular mention here.

Foremost, I would like to express my sincere gratitude to my supervisor Prof. Luís Lino Ferreira the continuous support of my MSc study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my MSc dissertation.

I would also like to thank all the other people at CISTER Research Center that guided and provided me with all the required resources during the course of this thesis. I want express my deepest thanks to everybody whom I worked with in the writing of the published papers, with special attention to Prof Luís Pinho, Prof. Luís Nogueira, Claudio Maia and Joel Gonçalves.

I am grateful to my professors and colleagues at ISEP for all their support, advises and knowledge along the past six years from the begging of my licenciature to the end of my masters degree.

Finally, I would like to thank all my family and friends for being there in the good and bad times. I would like to specially thank my grandparents, my parents, my two brothers and sister who have given me their unequivocal support throughout, as always, for which my mere expression of thanks likewise does not suffice.

Resumo Alargado

Actualmente, os smartphones e outros dispositivos móveis têm vindo a ser dotados com cada vez maior poder computacional, sendo capazes de executar um vasto conjunto de aplicações desde simples programas de para tirar notas até sofisticados programas de navegação. Porém, mesmo com a evolução do seu hardware, os actuais dispositivos móveis ainda não possuem as mesmas capacidades que os computadores de mesa ou portáteis.

Uma possível solução para este problema é distribuir a aplicação, executando partes dela no dispositivo local e o resto em outros dispositivos ligados à rede. Adicionalmente, alguns tipos de aplicações como aplicações multimédia, jogos electrónicos ou aplicações de ambiente imersivos possuem requisitos em termos de Qualidade de Serviço, particularmente de tempo real.

Ao longo desta tese é proposto um sistema de execução de código remota para sistemas distribuídos com restrições de tempo-real. A arquitectura proposta adapta-se a sistemas que necessitem de executar periodicamente e em paralelo mesmo conjunto de funções com garantias de tempo real, mesmo desconhecendo os tempos de execução das referidas funções. A plataforma proposta foi desenvolvida para sistemas móveis capazes de executar o Sistema Operativo Android.

Palavras-chave: Mobilidade de Código, Execução remota de código, Jogos para telemóvel, motores de Física. Sistemas Adaptativos, Sistema Operativo Android

Abstract

Smartphones and other mobile devices are becoming more powerful and are capable of executing several applications in a concurrent manner. Although the hardware capabilities of mobile devices are increasing in an unprecedented way, they still do not possess the same features and resources of a common desktop or laptop PC. A potential solution for this limitation might be to distribute an application by running some of its parts locally while running the remaining parts on other devices. Additionally, there are several types of applications in domains such as multimedia, gaming or immersive environments that require soft real-time constraints which have to be guaranteed.

In this work we are targeting highly dynamic distributed systems with Quality of Service (QoS) constraints, where the traditional models of computation are not sufficient to handle the users' or applications' requests. Therefore, new models of computation are needed to overcome the above limitations in order to satisfy the applications' or users' requirements.

Code offloading techniques allied with resource management seem very promising as each node may use neighbour nodes to request for help in order to perform demanding computations that cannot be done locally.

In this demanding context, a full-fledged framework was developed with the objective of integrating code offloading techniques on top of a middleware framework that provides QoS and real-time guarantees to the applications.

This paper describes the implementation of the above-mentioned framework in the Android platform as well as a proof-of-concept application to demonstrate the most important concepts of code offloading, QoS and real-time scheduling.

Keywords: Code Offloading, Code Mobility, Adaptative Systems, Mobile Games, Physics Simulation, Android Operating System.

Table of Contents

Acknowledgments iii
Resumo Alargado v
Abstract vii
Figure Index xiii
Table Index xv
Chapter 1. Overview 1
1.1 Introduction 1
1.1 The Smartphone market 2
1.2 Motivation 3
1.3 Real-time Code Offloading Solution Overview 4
1.4 Thesis Overview 5
Chapter 2. State of The Art 7
2.1 Introduction 7
2.2 Multimedia Applications 7
2.3 Mobile Gaming 8
2.3.1 Game Engines 9
2.3.2 Physics Engines 10
2.4 Real-time Systems 12
2.5 Mobile Code 12
2.6 Code Offloading 14
2.7 Mobile Operating Systems 16
2.7.1 iOs 16
2.7.2 Windows Phone 16
2.7.3 Android Platform 17
Chapter 3. Support Components 19

3.1 Introduction 19
3.2 MobFr (Mobile Framework) 20
3.3 CooperatES (Cooperative Embedded Systems) 22
3.4 Component interaction 23
3.5 Summary 24
Chapter 4. Code Offloading in Real-time Systems 25
4.1 Introduction 25
4.2 System Architecture 26
4.3 Real-time Offloading 28
4.4 Code Offloading Algorithm 29
4.5 Timing Parameters 32
4.5.1 Predicting $t_{xMaxCap}$ 32
4.5.2 Estimating t_{mob} 33
4.5.3 Measuring t_{core} 34
4.6 Summary 34
Chapter 5. Framework Implementation 35
5.1 Introduction 35
5.2 Offloading Library Class Diagram 35
5.3 Application Implementation Overview 39
5.4 Summary 39
Chapter 6. Framework Demonstrator Implementation 41
6.1 Introduction 41
6.2 Application Overview 41
6.3 Game Engine 42
6.4 Physics Engine 43
6.5 Physics World Partition 44
6.6 Application Structure 46
6.7 Performance Optimization Techniques 47
6.7.1 Serialization 47
6.7.2 Linear Regression Optimization 49
6.7.3 General Code Optimization 49
6.8 Summary 49
Chapter 7. Tests and Results 51
7.1 Benchmark Tests Mobile Framework 51
7.1.1 APK Transfer and installation Test 51
7.1.2 Remote Intent Execution Test 52

7.2 Benchmark Tests Offloading Library ... 53
7.2.1 Execution ... 54
7.2.2 Devices Execution ... 55
7.2.3 Data Transfer ... 56
7.2.4 Offloaing Framework Delay ... 56
Chapter 8. Conclusion and Future Work ... 59
8.1 Research Context and Objectives ... 59
8.2 Future Work ... 60
Papers and Technical Reports ... 61
Papers ... 61
TRs ... 61
Bibliography ... 63

Figure Index

Figure 1. Performance Comparison 3
Figure 2. Physics Object Analysis 4
Figure 3. Physics simulation with offloading 5
Figure 4. Game Engine Life Cycle 10
Figure 5. Android Software Stack 18
Figure 6 - MobFr Architecture 20
Figure 7. Mobile Library Class Diagram 21
Figure 8. CooperatES Architecture 22
Figure 9. Component Interaction 24
Figure 10. Offloading Structure 27
Figure 11. Offloading algorithm example 29
Figure 12. Real-time Offloading Sequence Diagram 32
Figure 13. Framework UML Model 38
Figure 14. Offloading Manager UML Diagram 38
Figure 15. Example application UML Diagram 39
Figure 16. Application Screen 42
Figure 17. Game Engine Class Diagram 42
Figure 18. Box2D Class Structure 44
Figure 19. Physics World Division 45
Figure 20. Application Class Diagram 46
Figure 21. Serialization Comparison 47
Figure 22. Deserialization Comparison 48
Figure 23. Device Comparison 48
Figure 24. Remote APK Transfer and Install Test Results 52
Figure 25. Execution results 53
Figure 26. Offloading Library Test Results 54
Figure 27. Execution Test Results 55
Figure 28. Total Execution Test Results 55
Figure 29. Data transfer Test Results 56
Figure 30. Pre Offload Test Results 57

Table Index

Table 1. Physics Engines Comparison 43
Table 2 - Device Specification 51

Chapter 1. Overview

Smartphones and other Internet enabled devices are now common in our everyday life, thus unsurprisingly a current trend is to adapt desktop PC applications to execute on them. However, since most of these applications have Quality of Service (QoS) requirements, their execution on resource-constrained mobile devices presents several challenges. One solution to support more stringent applications is to offload some of the applications' services to neighbour devices nearby. Therefore, in this thesis, we propose an adaptable offloading mechanism which takes into account the QoS requirements of the application being executed (particularly its real-time requirements), whilst allowing offloading services to several neighbour nodes.

1.1 Introduction

Smartphones are nowadays an essential part of our lives, executing a multitude of applications, connecting us to social networks, online games, or Internet calls. Some of these applications are monolithic, only being able to execute locally on the device, while others are distributed, being able to execute some parts locally and to execute (existing) services on other nodes in the network. Such computing paradigm is nowadays supported by the availability of high bandwidth networks. In the case of mobile devices the accessibility to high bandwidth wireless networks is of particular importance.

Although the performance of mobile devices is increasing in an unprecedented way, they still do not possess the same features and resources of a common desktop or laptop PC.

Nevertheless, a more dynamic and flexible solution to solve the performance gap, is to allow mobile devices to dynamically offload some of the applications' services to neighbour devices, taking advantage of collaborative environments, such as at home or in the car, or of infrastructures providing value-added services.

In comparison with more traditional distributed approaches, supported by "fat" network servers, the offloading solution has the following advantages:

i) The code to execute is available in the client application;

ii) The nodes to which computations are offloaded are nearer, consequently communications, usually, have less delays and better QoS levels;
iii) The changes required on the original code are usually less significant.

Several different types of solutions for code offloading have been provided, motivated by the need to obtain access to additional resources, like memory, power or more computation capabilities. Examples of code offloading frameworks are: Cuckoo (Kemp, Palmer, & Bal, 2010)and MAUI (Cuervo, et al., 2010).Other solutions adopt a more automatic approach, where the offloading framework is able, by itself, to analyze the code and determine which parts/classes can be offloaded, e.g. CloneCloud (Chun, et al, 2010). Furthermore, some of these algorithms are adaptive, i.e. they are able to dynamically, in run-time, determine an adequate application partitioning (Gu, Nahrstedt, et al., 2003)

Applications like multimedia, control applications, image processing and gaming, inherently have real-time requirements associated with high computational demand. But none of the frameworks discussed before is capable of handling the application's real-time requirements; they mostly provide a best effort solution. The adaptive solutions also present the additional burden of calculating, in run-time, the application partitioning.

It is in this context that in this paper we put forward a code offloading approach, allowing applications to offload some of their services to neighbour nodes. The goal is to support adaptable applications, which present variable QoS requirements.

1.1 The Smartphone market

Devices such as Apple iPhone, Google Android devices, Rim Blackberry phone and Windows Phone 7 devices, as well Apple iPad are some examples of devices which have achieved high commercial success. Studies show that in the third quarter of 2010, worldwide mobile phones sales increased by 35% and smartphones' sales increased by 96% in comparison with the previous year (Egham, 2010). Meanwhile, mobile industry analysts estimate that tablets will outsell netbooks by 2012 and "will constitute nearly a quarter of all PCs by 2015" (Cush, 2010). This success and the interest from the general public allowed such embedded systems to start appearing in other environments such as Vehicles and TVs (examples of this applications are Ford Sync (Cunningham, 2009) and Google TV (Patel, 2010)).

The smartphone boom made that companies producing these devices started to release new devices with better hardware in shorter periods of time, which caused the older smartphones, even ones that entered the market less than 6 months became unfit to run newer applications. This fact, accentuated the performance problem since the user does not want to buy a new phone each time a more resource hungry application arrives to the market.

As the new smartphones entered the market, their mobile operating system also became more the focus of the user's attention. Operating systems such as Apple iOs, Android OS, Rim BlackBerry OS, webOs, Windows Phone OS became an important factor decision of what smartphone the user would buy. Of all the discussed operating systems, it's important to highlight the Android Operating System. Android is an open source operating system designed for resource constrained devices, which has be chosen for this dissertation.

The Android OS development started in 2003 by a company named Danger which was later purchased by Google and is now administrated by the Open Handset Alliance (Open Handset Alliance).

1.2 Motivation

Although the performance of mobiles devices is increasing in an unprecedented way, they still don't possess the same features and resources as a common desktop or laptop system.

In order to illustrate how much faster a computer is when compared a smartphone, tests were performed using a PC and two devices: one HTC Magic and one Samsung Galaxy S. The computer is a Dell Pc with a 3GHz Intel Pentium 4 processor with 1 GB of RAM with Windows XP Operating System while the HTC Magic has the default 258 MHz Snapdragon processor with 512 MB of RAM, running Android 1.5 Operating System. The Samsung Galaxy S has the default 1GHz ARM Cortex processor and 512 MB of RAM, and runs the Android 2.1 Operating System.

The tests consisted in counting the number of milliseconds it would take for a device to compute an iteration on an object movement simulation composed by 1000 box shaped objects bouncing on screen. Figure 1 depicts the tests results. As it is possible to observe, the smartphone's performance is almost 0.3 % and 0.8 % of a PC, from early 2000s, in the case of the HTC Magic and the Samsung Galaxy S, respectively.

In this test the PC is able to execute the physics simulation in an average of 96 ms while the Samsung Galaxy S needs on average 11000 ms and the HTC Magic needs on average of 25000 ms.

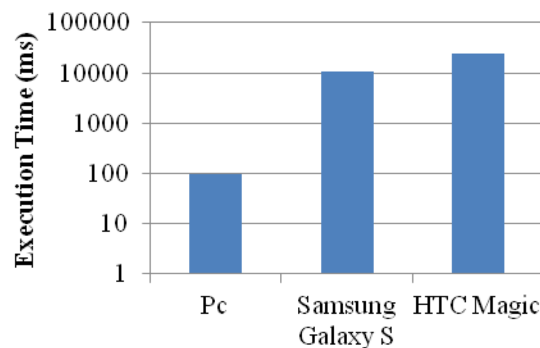

Figure 1. Performance Comparison

The discrepancy between what the device is capable of and what the user expects from his/her device led to the research and design of a framework capable of increasing its performance by harvesting resources from neighbour devices.

Another test consisted in determining the time required to calculate an iteration on the same physics' simulation application as a function of the number of screen objects. These results are an average of 100 runs, which have been performed on a HTC Magic Android device running a 258 MHz Snapdragon Processor with 512 MB RAM. Figure 2 clearly shows that the execution time of each iteration increases with the number of objects.

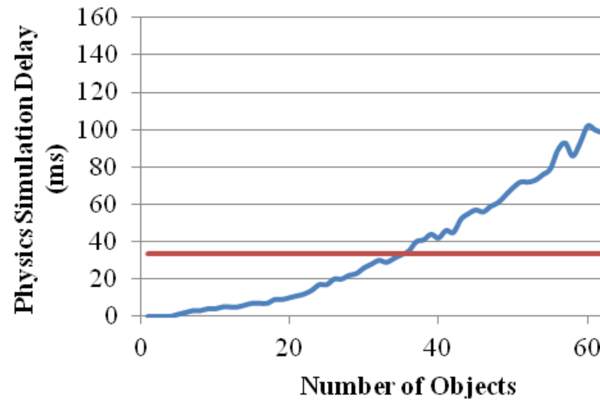

Figure 2. Physics Object Analysis

As a consequence, any real-time visualization of the simulation results, using a frame rate of 30 frames per second (represented by the red line in Figure 2), would not be possible for more than 38 objects.

A solution to solve this performance gap is to offload some of the object computations to neighbour nodes, as described in the following section.

1.3 Real-time Code Offloading Solution Overview

The solution proposed in this dissertation aims at creating an offloading architecture for highly dynamic distributed systems with Quality of Service constraints, where traditional, single core, models of computation are not sufficient to handle the users' applications' request. The target applications range from flexible sensor and control applications to multimedia applications, gaming applications and other applications which periodically run services with variable execution time.

The offloading approach involves a constant monitoring of the time required to execute a service, this service is hereafter called as **core service**, on the main device. Based on past execution times of the core service, the offloading algorithm predicts the future ones. If the algorithm determines that the required execution rate cannot be maintained, then the offloading procedure is triggered in advance, in order to minimize the occurrence of timing errors. Therefore, it is possible to timely offload some of the services to neighbour nodes and execute them there without reducing the rate or the supported quality.

When it is not advantageous to continue executing the offloaded services in other nodes, migration can once again take place, and these services can return to be totally executed on the original device.

If these principles were applied to the real-time physics simulation described in Section 1.3, then it would be possible to offload some of the computations of the core service, in this case the piece of code which calculates the trajectories of objects, to other nodes available in the network. Figure 3 illustrates the expected behaviour if a real-time offloading solution is applied.

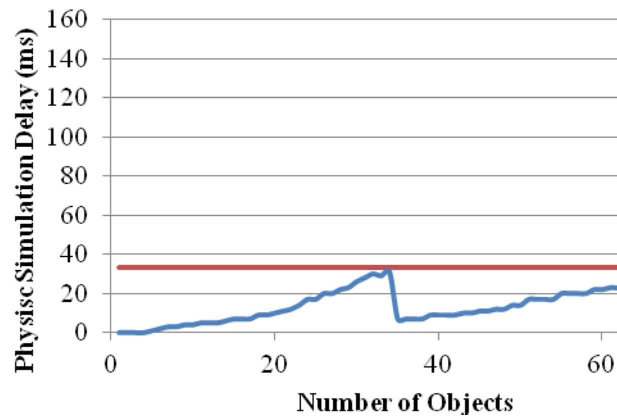

Figure 3. Physics simulation with offloading

The proposed solution would be able to prevent crossing the maximum admissible core execution time, therefore being able to maintain the frame rate of the real-time visualization of the simulation results. Details on the algorithm are discussed in Chapter 4.

The solution proposed in this document requires the support from additional framework(s) that are able to handle operations such as detecting neighbour devices, performing code migration, and managing the available resources in all the devices. Therefore, this work is build upon previous works at CISTER Research Centre the **MobFr** (Gonçalves, Ferreira, & Silva, 2010) and **CooperatES** (Nogueira & Pinho, 2009) frameworks.

The **MobFr** framework is responsible for code mobility, which enables the dynamic offloading of the core service to neighbour devices. Additionally, it also supports the seamless communication between neighbour devices and the main device. The **CooperatES** framework handles resource reservation and allocation on the overall system, including CPU and network resources. Since both are components of structural importance to the framework, they are described in detail in Section 3.

1.4 Thesis Overview

This thesis is divided in 9 Chapters. Chapter 2 introduces the state of the art in the context of this dissertation. Then Chapter 3 gives an in-depth view of components required to support the framework proposed in this thesis. Chapter 4 discusses the main theoretical details of the proposed code offloading framework for real-time systems, whose implementation is described in detail in Chapter 5. Chapter 6 discusses a proof-of-concept application which validates the proposed framework, then Chapter 7 analyses the results of the proof-of-concept application. Chapter 8 draws some conclusions about this thesis.

Chapter 2. State of The Art

This thesis presents a code offloading solution designed to increase the performance of applications that run services periodically with variable execution time in resource constrained devices. Examples of these applications range from flexible sensor and control applications to multimedia applications, gaming applications.

2.1 Introduction

This thesis has as the core the research and documentation of guidelines to increase the performance of mobile applications, more precisely mobile multimedia applications, using offloading techniques. This means that it covers two specific areas of computer science, multimedia applications, more precisely, mobile games and mobile code execution.

Both areas seem to be the subject of a great number of scientific papers. Multimedia applications are used in a large range of different areas such as education, network architectures, 3D graphics, performance optimized game engines and many other areas where the target user is not positioned in a fixed location. Mobile code execution is used different areas of research such as using mobile code to extend battery life, increase performance, in monitor mobile agents through the network and other.

That said, although both topics seems different, it’s actually very common to find nowadays multimedia when researching for code offloading papers. That is interesting because both concepts are not new, if fact some argue that their first appearance date more than 30 years old, and yet only recently they have being seen in together in scientific researches. The combination of both concepts has generated very important papers which are analyzed in the next sections.

2.2 Multimedia Applications

The term multimedia application is used to identify a large group of different applications, which may be very different from one to another. Some of these applications include movie players, audio players, electronic games, virtual worlds, video streaming applications, and others.

Multimedia applications are resource driven applications whose execution is continuous over time. Multimedia applications, especially mobile multimedia applications, take advantage of the heterogeneous environment where they are being executed. For example, a smartphone application can use both the cellular network such as CDMA or Wireless LAN when wanting to retrieve a video from a remote server in a wired network.

According to the authors of (Bolliger & Gross, 1998), network-aware applications have two basic aspects:

- They must have the ability to monitor or get information from the network monitors about the current status of the underlying network (network awareness).
- They must be able to adjust their behaviour based on the collected information (network adaption)

The interconnectivity of different networks makes pervasive computing an exciting reality, but it also poses many challenges for application developers. In any application transparent to network changes, its data is generated and transmitted at a fixed rate and there can be only two results:

- The quality of the data is reduced so that even a client with low bandwidth access can receive data with little delay.
- The data is received in high quality so that the clients with high bandwidth access experience satisfactory levels of performance.

The solution relies on the application being able to adapt to changes in the network. This requirement is even more critical to mobile multimedia applications, because the multimedia content, such as audio and videos higher peak bandwidth as illustrated by the framework of (Krikellis, 2000). If an application does not change the data quality to be delivered according to the network changes, a huge amount of multimedia data send from a wired network will encounter unbearable delay or error when transmitting in a wireless network with limited bandwidth (Kim & Jamalipour, 2001).

2.3 Mobile Gaming

Mobile gaming refers to the area in computer science that studies electronic games running on mobile devices. Mobile games development began in the first stages of the mobile devices with the appearance of the first cell phone (Schilling, 2011). During their first generation these games used to be very basic without any composed graphics or any other feature users now take for granted. As the time passed, these games started to became each time more similar to the ones on desktop computers or gaming consoles.

Since their appearance many scientists have used them, not only as a proof of concept for their research but also as well the base of their works (Cuervo, Balasubramanian, & Cho, 2010).

The most researched topics in mobile gaming include education, 3D graphics, its use in peer to peer networks, game engine implementation and others.

Yang and Zhang (Yang & Zhang, 2010) analyze the challenges that mobile games running on lower power devices have to face, the design of different optimization techniques capable of increasing their performance and the implementation of a mobile game, more specifically a billiard game that takes advantages of those techniques. The techniques presented in the paper are divided in two categories: **Collision Detection Optimization** and **System Optimization**.

The **Collision Detection Optimization** techniques promote speed over accuracy and uses octree-based multi-level collision detection and dynamic multi-resolution grid subdivision to reduce the number of objects to be calculated. By using this technique, the authors conclude that

the overall performance of the physics calculation is decreased and the complexity of the collision algorithm is reduced to $O(n \log n)$.

The **System Optimization** techniques consist in three methods: Rendering Optimization; by buffering the data in a local buffer and writing directly to the video buffer rather than using the SDK rendering callbacks; Computation Optimization; by promoting the use of fixed-point function over software implemented floating point ones; and Language Optimization techniques; by programming directly in assembly language rather than using compiled languages.

Xin (Xin, 2009) analyzes the different mobiles games and creates taxonomy in order to group them in different types, proposing the following types:

- **Embedded Games:** Games that are programmed to run natively on the phone chipset usually installed by the phone's manufacture.
- **Messaging Games:** Games that are played by exchanging messages with the server. The player can send the messages in SMS format, which are processed by the server.
- **Browser Games:** Games are played by submitting the data to the server and the results are viewed in web browser.
- **Interpreted Language Games:** Games that are executed in a virtual machine. These types of games tend to be the most abundant ones and are generally distributed through a digital store.
- **Compiled Language Games:** Games that are executed in native machine code.

Games can be classified in two categories: Real-time games and turn based games.

The term Real Time Games refers to electronic games whose action is continuous throughout the game. Some examples include car racing and sport simulation games.

The opposite of real time games is turn based games, where the game action moves forward by the player's input. Some examples include Chess and Card based games.

2.3.1 Game Engines

A game is an electronic application that runs a set of instruction repeatedly, in a pre-defined interval of time. The life cycle of the game is managed by the game engine.

Game engines manage the game life cycle using a state machine algorithm based on four states: Initialize, Update, Draw and Finalize. Figure 4 shows the four states of a game engine and how they interact.

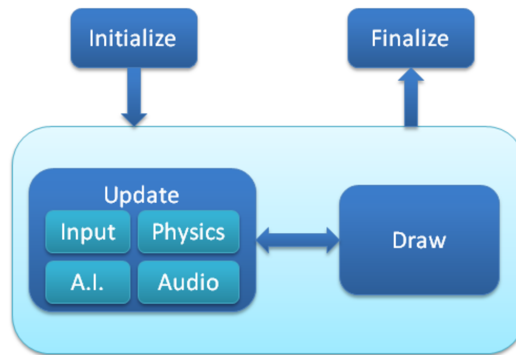

Figure 4. Game Engine Life Cycle

The **Initialize** method is called at the beginning of the game and allows the game to prepare. This includes instantiate variables, load content from the hard drive to memory, set the desired frame rate and connect to the any required services.

The **Update** and **Draw** methods are called repeatedly, not necessarily in sequence, but multiple times per second, according to the frame rate defined in the initialization phase.

The **Update** method is responsible for all the game's logic. That involves retrieving the user's input, calculating the game's physics, playing audio, managing the game screens, updating the artificial intelligence among other tasks.

The **Draw** method is responsible for all the rendering calls to the graphics API. These calls include drawing primitives, textures and applying rendering effects. The rendering API is responsible for the interface between the software and the graphics device. In the most visual appealing games and most 3D games, the rendering is not executed completely in the Central Processing Unit but rather executed cooperatively with the Graphics Processing Unit and Central Processing Unit.

The **Finalize** Method is called after the game is over and is used to dispose the instantiated variables and unload the content from memory.

In most professional video games, game engines are assisted by other engines such as physics engines, which calculate object movement and collision.

2.3.2 Physics Engines

Fiedler (Fiedler, 2006) describes physics engines as computer software capable of simulating the interaction of geometric objects in a confined space, denominated, physics world. Physics engines are used in a vast array of different areas, and they can be classified in many different ways, such as by type of physics system, geometry or precision.

The physics system is the way the shape of a body is affected when it collides with another body. This can be rigid body dynamics if the shape of the body does not chance when it collides, soft body dynamics if the shape of the body is deformed by the collision or it can be fluid dynamics if the shape of the object depends on the bodies colliding with it.

The geometry is the dimensional system supported by the physics engine, more precisely, 2D or 3D, depending on the geometry model of the objects the physics world contains.

Finally, the precision refers to the accuracy of the physics. There are two types of precision: high precision and real-time. High precision physics engines (or dynamic simulations) require more processing power to calculate very precise physics whereas real-time physics engines use simplified algorithms that increase the performance but lower the accuracy of the results.

For example, movies use high precision physics engines which focus on creating realistic scenes while video games and scientific simulations use real-time physics engines that focus on performance.

Independent on the type of physics engine, all physics engines possess two core mechanisms: **collision Detection** and **Collision Response**.

The **Collision Detection** mechanism calculates if an object overlays another. This method varies depending on the object shape, that is, calculating the collision between two circles is different from calculating the collision between a square and a triangle.

The **Collision Response** mechanism calculates what happens when two bodies collide. In the most accurate physics engines, Newton laws are used to calculate result.

The physics calculations are divided in steps.

The **first step** is to calculate the new position for each body in the world. The new position is calculated using the object's current position, velocity and direction.

The **second step** is to apply collision detection methods to detect if any object is colliding with another.

The **third step** is to apply collision result methods to calculate the new direction and velocity of all bodies colliding.

The **final step** is to calculate the new positions of all the colliding objects based on the values from step 3.

Physics engines are developed and analyzed in different fields ranging from physicians trying to validate their theories to semiconductor companies trying to execute hardware-accelerated physics engines. One example of a research company is Havok; an Irish software company specialized in the development of the Havok physics engine. Recently Intel purchased the company and its intellectual proprieties in order to compete with Nvidia's PhysX.

Chabukswar and Lake (Chabukswar & Lake, 2005) document the use of a multi-threaded environment to increase the performance of a physics simulation game. The work specifies the implementation of a game engine where the rendering code and the physics simulation code are executed in two separated threads.

The main objective of the work is to demonstrate that it is possible to increase the performance of the physics calculations by executing them in different threads in parallel while the application is performing all the rendering instructions. The author promotes the use of different threads to perform different tasks based on the fact that future processor architectures are moving to multiple cores.

The results state that executing the scenario in a single thread, the processor requires 99-100% of its resources while executing in different threads results in the processor requiring 15% of its resources for the rendering and 85% of the resources.

The Authors state that separating different tasks in threads has the advantage of distributing the load by all the cores of the processor and proposing that way as a standard for future multi core application design.

2.4 Real-time Systems

A real-time system is a system which its correctness depends not only on the results it provides, but also on the time instance at which the results are produces (Stankovic, 1996).

In real-time systems, the notion of time is relative, as the response time requirements may vary from application to application, but a late action might cause an expected behaviour that could lead to a system failure. For instance, in an air control system, if a task does not actuate at the time in which it is expected, it may cause the air plane to lose control and in the worst case scenario even crash. Other examples where real-time systems are fundamental include nuclear power plants systems, medical applications and intelligent vehicle highway systems.

The time instant at which is expected to provide its results is denominated deadline. Therefore, the task's deadline is the maximum time allows for a task to complete its execution. The term task refers to a unit of work that is schedulable and then executed. Schedulable means that it can be assigned to the processor unit in a particular order to be executed, i.e. when the selected time slot becomes available.

Not all real-time tasks may cause critical failure. For instance, in a multimedia system, if the system fails to present to display a certain amount of frames belonging to a movie at requested frame rate, due to a task missing a deadline, the user might notice it. Although the scenrario does not induce catastrophic consequences, the dead line miss may clearly cause performance degradation which may affect the user's perception of the movie.

In order to distinguish the above system, the consequences of a deadline miss are used for classification proposes, thus is a real-time task missing causes performance degradation, without jeopardizing system behaviour, that task is considered a soft real-time task. In the other hand if a task missing a deadline may cause catastrophic consequences, the task is considered hard real-time task.

Incidentally, the solution present in this thesis is not appropriate for Hard Real-Time System, as it cannot guarantee the deadline for all operations.

2.5 Mobile Code

Code mobility is a paradigm of computer science in which a computer program running on a certain device executes part of its code remotely in another device connected through a network. Code mobility is a fertile research field that through the last 20 years has generated and continues to generate a growing body of scientific literature and industrial development. Many authors have different definition of code mobility, such as:

“Paradigm in which computing resources such as processing, memory, and storage are not physically present at the user´s location.”

(Kumar & Lu, 2010)

“The capability to reconfigure dynamically, at runtime, the binding between the software components of the application and their physical location within a computer network.”

(Carzaniga, Picco, & Vigna, 1997)

Mobile devices have limited battery and wireless bandwidth. Remote code execution can provide energy savings. Several studies have identified longer battery lifetime as most desired feature in smartphones. In 2005, a study from users around the world found longer battery life to be more important feature in smartphones that camera or storage (Various, 2005), while Change wave research revealed short battery life to be the most disliked characteristic in IPhone 3GS (Radwanick, 2011).

Code mobility is one of many ways to increase battery life in mobile devices. Many researchers have described these techniques. The authors of (Kumar & Lu, 2010) have described four different approaches to save energy and extend battery lifetime in mobile devices:

i) Adopt a new generation of semiconductor technology;
ii) Avoid wasting energy through implementing standby or sleep mode in the whole system and in individual components;
iii) Execute program slower;
iv) Perform the computation in the cloud.

Fuggetta, Pietro and Vigna (Fuggetta, Pietro, & Vigna, 1998) divide Mobile code techniques in four different paradigms. The author uses the term *host* to refer the network node where the application is running, *server* to refer to a neighbour node, *know-how* to refer the instructions of the service and *resources* as the data used by the service.

In the **client-server** paradigm, the server offers a set of services. That node has all the resources and know-how need for the service execution. When the application running on host device needs to execute a service, it interacts with the server which prompts it to execute the service and return the results.

In the **Remote Evaluation** paradigm, the host node has the know-how necessary to perform the service, but lack resources required which are located in a server. In this paradigm the node that wants to execute the service sends the know-how, which is executed in the remote node and then return to original one.

In **Code on Demand** paradigm, the node that wants to execute the service has the resources it requires, but not the know-how. In this paradigm that node has to request the know-how from a server, which is then be used to execute the service.

In the **Mobile Agent** paradigm, the host node has the know-how, but lacks resources, which are located in a server. This triggers the migration of the application from its original node to the server. This paradigm differs from the Client-Server because instead of sending the data and receive the results, the hole application migrates to the server and is executed there.

Sommer (Sommer, 2010) proposes an offloading system for sensor networks. The system is based on mechanisms which are capable of handling service migration and service updates. Code migration can be triggered by the user or by a monitoring agent. The system uses two components, a **Migration Facility** and a **Migration Coordinator**. The **Migration Coordinator** is responsible for coordinating the migration according to network and the application needs. It has an in-depth knowledge of the application service's requirements, and the data paths of the network. The **Migration Coordinator** only allows the migration if all the requirements are fulfilled. The **Migration Facility** is present on the device where the application is being executed and the network devices. In the device where the application is being executed, the network facility is responsible for checking if the required service is available and if it implements all the requirements. The network facility in the network node is responsible for the state and service migration.

2.6 Code Offloading

It's important not to confuse code mobility with code offloading. In Code Offloading, the code that is going to be executed in the remote server can already be installed prior to the application start.

Several different solutions for code offloading have been provided, motivated by the need to obtain access to additional resources, like memory, power or more computation capabilities. Some of these solutions rely on the user to determine which parts of the code to offload; for instance, Cuckoo (Kemp, Palmer, & Bal, 2010), which provides an offloading environment for Android-based systems using the available inter-process communication mechanisms, or MAUI (Cuervo, Balasubramanian, & Cho, 2010), where the user is responsible for the annotation of the methods which can be executed remotely, being power savings the main objective.

Other solutions adopt a more automatic approach, where the offloading framework is able, by itself, to analyze the code and determine which parts/classes can be offloaded. That is the case of CloneCloud (Chun & Maniatis, 2010) which permits the execution, in the cloud, of the application in an almost exact virtual machine. Furthermore, some of these algorithms are adaptive, i.e. they are able to dynamically, in run-time, determine an adequate application partitioning ((Gu, Nahrstedt, Messer, Greenberg, & Milojicic, 2003) and (Xian, Lu, & Li, 2007)). Code offloading also usually relies on libraries or frameworks that support the mobility of code or services. For instance, the work presented in (Sommer, 2010) describes several service migration scenarios for embedded networks.

The authors of (Xian, Lu, & Li, 2007) propose a solution for adaptive offloading systems in Java. The system focus on two components: a **Distributed Offloading Platform** and an **Offloading Interference Engine**. The **Distributed Offloading Platform** is responsible for monitoring the application, managing the resources, deciding the application partition and supporting Remote-Procedure-Call (RPC) between virtual machines. The **Offloading Interference Engine** has two decision making modules to address the problems of triggering offloading and selecting the partition The triggering offload module decides based on resource consumption, resource availability in the pervasive computing environment and by using Fuzzy Control Model (Li & Nahrstedt, 1999). Based on the output, the system decides if the application continues as is, needs to start offloading or needs to stop offloading. The module responsible for selecting the application partitioning selects it from a group of candidates partition plans generated by the offloading platform. Additionally the user can specify multiple

offloading requirements such as minimizing the wireless bandwidth overhead or minimizing average response time stretch or minimizing total execution time.

Spectra, a remote execution system of resource constrained systems, described in (Flinn, Park, & Satyanarayann, 2002), dynamically decides how and where to offload computation, depending on the application resource usage and availability in the environment. Spectra regularly monitors the CPU, the network and the battery and bases its decision on three goals: Performance, Energy Consumption and Quality. In cases where there is a conflict between the goals, spectra increases priority of the one that best fits the current execution.

Scavenger (Kristensen & Bouvin, 2010), a computation offloading system designed to allow easy development of mobile code offloading applications, delivers efficient use of remote computing resources through the use of a custom built mobile code execution environment and an adaptive dual-profiling scheduler. The scheduler uses history-based profiling and selects the tasks based on several factors: network capabilities, data locality device strength, task complexity. Tests made to the system report that it can greatly increase the performance of the application (even on small tasks) and the system can provide energy savings in the applications.

The solution proposed in (Cuervo, Balasubramanian, & Cho, 2010) presents MAUI, a system that enables fine-grained energy aware offloading of mobile code to the cloud through virtual machine virtualization. MAUI was developed as an alternative code offloading solution, that rather than heavily relying on the programmer, perform a full virtualization of the process or perform the virtualization of the virtual machine, uses the advantages of the .NET framework to perform the code offloading without the least possible burden from the developer.

MAUI is required to address some challenges in order to be partitioned across multiple machines, such as:

- MAUI must distinguish which methods are going to be executed remotely and which are going to be execute locally;
- MAUI must automatically identify and migrate the necessary program state from the running program on one machine to another;
- MAUI must, based on the current environment, dynamically select whether to run a method locally or remotely;
- MAUI must detect and tolerate possible failures in the system without having any critical consequence to the original program.

Like the framework presented in this thesis, MAUI decides dynamically during the runtime of the application which parts of the code should be executed locally and which parts of the code should be offloaded to the infrastructure. One of the main differences between the proposed Real-time Offloading Framework and MAUI is that MAUI objective is to achieve the maximum energy savings possible unlike the Real-time Offloading Framework whose objective is to offload the parts of the code the framework considers that it doesn’t possess enough resources to execute locally. Another difference is that MAUI is integrated with the .NET where the proposed solution is implemented as middleware. This was chosen because as middleware, can be easily extended and ported to different application with causing much effort to the developer.

During compilation time, the framework generates two proxies: one that runs on the smartphone and other that runs on the dedicated server. These proxies automatically execute the code remotely or locally depending on the decision made by a component named MAUI Solver. This

component decides if the code should be performed locally or remotely based on the input it receives from a resource management component. Whenever the MAUI Solver decides to offload code, the proxies handle all the control and data transfer operations.

Nimmagadda (Nimmagadda Y. K., 2010) describes a system capable of increasing the performance of real time systems for computation-intensive tasks in resource constrained robots. The system is designed as an alternative to the existing systems were the computation is executed exclusively on the robot or in specialized servers. The system based on different independent modules each responsible for a different task. Some of the modules can only be executed on the robot and other can be executed in either the robot or externally. The offloading decision is based on two factors: the resources available and the communication involved between the modules and the offloading.

Although the majority of studies promote the idea of using code offloading in a WAN environment, it's also relevant to discuss the possibilities of using code offloading in a LAN environment. While WAN covers a significantly larger area, LANs have proved to be significantly faster, more secure, less expensive and more flexible than WANs.

2.7 Mobile Operating Systems

In recent years, mobile Operating Systems have become a target of public and media attention (Adams, 2011). Unlike desktop OSs, mobile OSs run in resource constrained devices, where resource management is a priority.

Of the present Mobile operating systems, the ones that stand out are Apple iOs, Google Android, Rim Blackberry, HP WebOs, Nokia Symbian and the recent Microsoft Windows Phone.

2.7.1 iOs

The iOs is Apple's mobile operating system. iOs is used in Apple's iPhone, iPod touch and iPad and Apple TV. Apple reported iOs being based on Mac OS X which uses the Darwin foundation, a Unix-like OS.

iOs application development uses the Xcode integrated development environment using the iOs SDK. The programming language promoted by apple is the Objective-C although recently many third party companies have developed alternative solutions such as Adobe Flash/ActionScript, Unity using C# or JavaScript, and Unreal Engine using C++.

The iOs possesses some limitations when it comes to application development that limits the development of research work, such as: I) the operating system does not allow multiple applications to run in the same device as in normal desktops, II) the iOs is not an open source OS , and III) Apple does not allow alterations to OS source code.

2.7.2 Windows Phone

Windows phone is the operating system used in Windows Phone 7 devices. This operating system was developed by Microsoft as the successor of the Windows mobile platform.

Windows Phone 7 was announces in February 2010 in the World Mobile Conference 2010 and launched in October of the same year.

Windows Phone stands out from other mobiles due to its simplified user interface codenamed Metro previously used by Microsoft on the Zune media player devices.

Windows phone application development uses the Visual Studio IDE and can be done using either Silverlight or XNA frameworks. Silverlight is a rich interface based application framework which uses Microsoft XAML language for the interface and the control specification uses C# or Visual Basic as the programming language. XNA is a framework designed to facilitate video game development using C#. The XNA framework is built on top of the DirectX API and allows easy multiplatform development between Windows, XBOX 360 and Windows phone.

The use of Windows Phone OS has the same limitations as the iOs when it comes to the ability to be used for research.

2.7.3 Android Platform

The Android is an operating system designed for smartphones and other low-power mobile devices whose development is administrated by the Open Handset Alliance (Open Handset Alliance), a group of 80 Organizations whose objective is to create open standards for mobile devices.

The architecture of the real time offloading framework present in this document is developed for mobile devices with the android operating system. The Android Operating System was chosen based on other factors such as:

- Android allows complete application multitasking.
- Android is easily extendible/extensible.
- Android application development uses a familiar programming language (Java and C++).
- Android is well documented and there are large amount resources available online.
- Android has a large user base.
- Cister was already using Android for other projects.

The Android Software stack (What is Android?, 2009) is structured in 4 layers and several modules, as shown in Figure 4. These layers can be organized in two groups. The static unchangeable part that all mobile devices are required to have and a changeable part whose device manufactures can remove, change and alter features. The unchangeable part, the system image, is composed by the **Linux Kernel** layer, the **Libraries** and **Android Runtime** Layer and the **Application Framework** Layer. The changeable part is composed by the Applications layer.

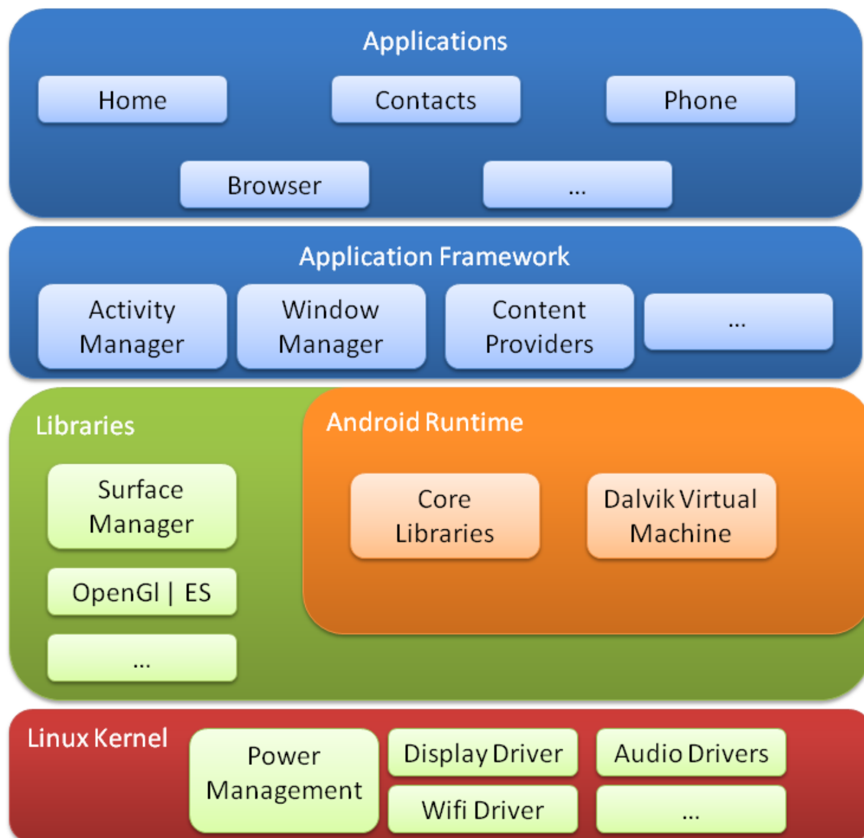

Figure 5. Android Software Stack

The bottom layer is the **Linux Kernel**. The kernel is responsible for memory and process management. One of the main advantages of using the Linux Kernel is that it has a wide community support and allows easily adding device drivers for specific hardware components.

The second layer is composed by two parts, the **Libraries** and **Android Runtime**. The **Libraries** are a collection of resources that can be used by any application to access the hardware or other services. These include the OpenGL API, SQLite libraries, among others. The **Android Runtime** is composed by the core components of the system such as the **Core Libraries** and **Dalvik Virtual Machine**. Unlike the Java Virtual Machine, the Dalvik Virtual Machine is register-based process virtual machine designed for mobile devices and other resource constrained devices.

The third layer is the **Application Framework** layer. This layer is composed by a collection of libraries that facilitate application development. These libraries provide the interface between the application and the underlying system resources and classes that improve the compatibility of the application in different Android devices.

The last layer is the **Applications Layer** and holds all the applications installed in the device. This application can be home application, contacts manager application, Internet browser, games, and email application among others.

Chapter 3. Support Components

Code offloading techniques rely on the mechanisms to determine which nodes are available on the network to perform code mobility operations and to communicate with those nodes. The first mechanism can be supported by the CooperatES framework which is described in section 3.2. Both the second and the third mechanisms are supported by the MobFr framework, which enables code migration and controls communication between distributed nodes (section 3.3). To contextualize the chapters we start by discussing the general architecture of the offloading framework.

3.1 Introduction

In order to an application to offload its computation, the application must implement mechanisms that: I) Monitor the available neighbour nodes and their resources, II) Remote code migration and execution and III) Communication between network nodes.

Those mechanisms are provided by two supporting frameworks, a code mobility framework – **MobFr -** and a real-time cooperative framework - **CooperatES**.

MobFr is responsible for the code mobility operations such as transferring and installing services from a device to another, executing them, and the communication between devices. MobFr is composed by two parts: a set of modules which are responsible for the code mobility operations that run on the device background, and a library which is used by applications to communicate with those services.

CooperatES handles the resource and network management operations such as detecting new devices on the network, grouping the network nodes and monitoring their available resources. CooperatES is composed by three modules, two of which are implemented in the operating system kernel and are responsible for managing the available network devices and the system resources; and another module which groups network nodes and determines the best candidate to offload code based on the resources required by the service implemented in the Android Runtime.

MobFr is also the interface between application and the CooperatES framework, thus simplifying the development of the solution proposed.

3.2 MobFr (Mobile Framework)

MobFr (Gonçalves et al., 2010), or Mobile Framework, is a software platform designed to increase the performance of applications in resource constrained devices by allowing them to scavenge resources thought service migration to available neighbour nodes and by supporting interconnection between nodes.

MobFr is composed by two parts: a set of modules which run in the device background, responsible for code mobility and a library that serves as the interface between the modules and the application.

Figure 6 depicts the components of MobFr: the Application which runs on the users' device, the service which runs on the neighbour nodes and the mobile services and the modules which run in the background of all devices.

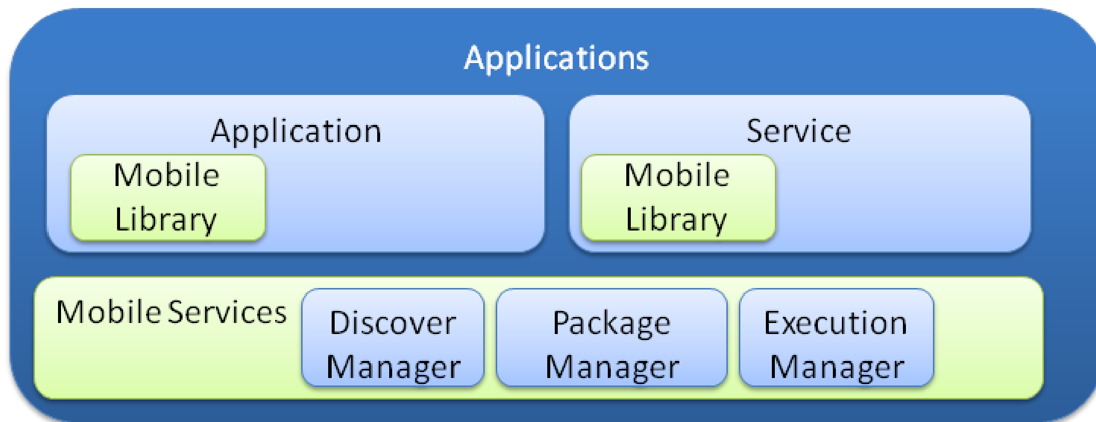

Figure 6 - MobFr Architecture

The **Discovery Manager** module discovers neighbour devices on a local network, advertise the host's resource availability and gathers information about the resource availability in the other network nodes. These functionalities can be used by interacting with other Discovery Manager components on the network or using any service provided by the system.

The **Package Manager** module transfers, installs and uninstalls services. This module also specifies the resources a node requires to compute the service which can be used to identify the best neighbour candidate.

The **Execution Manager** module allows executing services in neighbour nodes through the exchange of Android Intents, thus allowing the development of transparent applications (regard it to its distribution).

MobFr provides a library, referred as Mobile Library that allows any application to communicate with the Mobile Services. Each Service has an interface class. The classes `StandardDiscoveryManager`, `StandardackageManager` and `StandardExecutionManager` classes handle the Discovery Manager, Package Manager, and Execution Manager modules, respectably.

The Mobile Library also is designed to automate the process of code offloading. It achieves that by providing the `MobileActivity` and `MobileService` abstract classes which are an

extension of the Android Activity and Android Service classes, respectively, and are designed to abstract the developer from the inherent distribution of the services, thus reducing the effort involved in porting standard applications to the new paradigm.

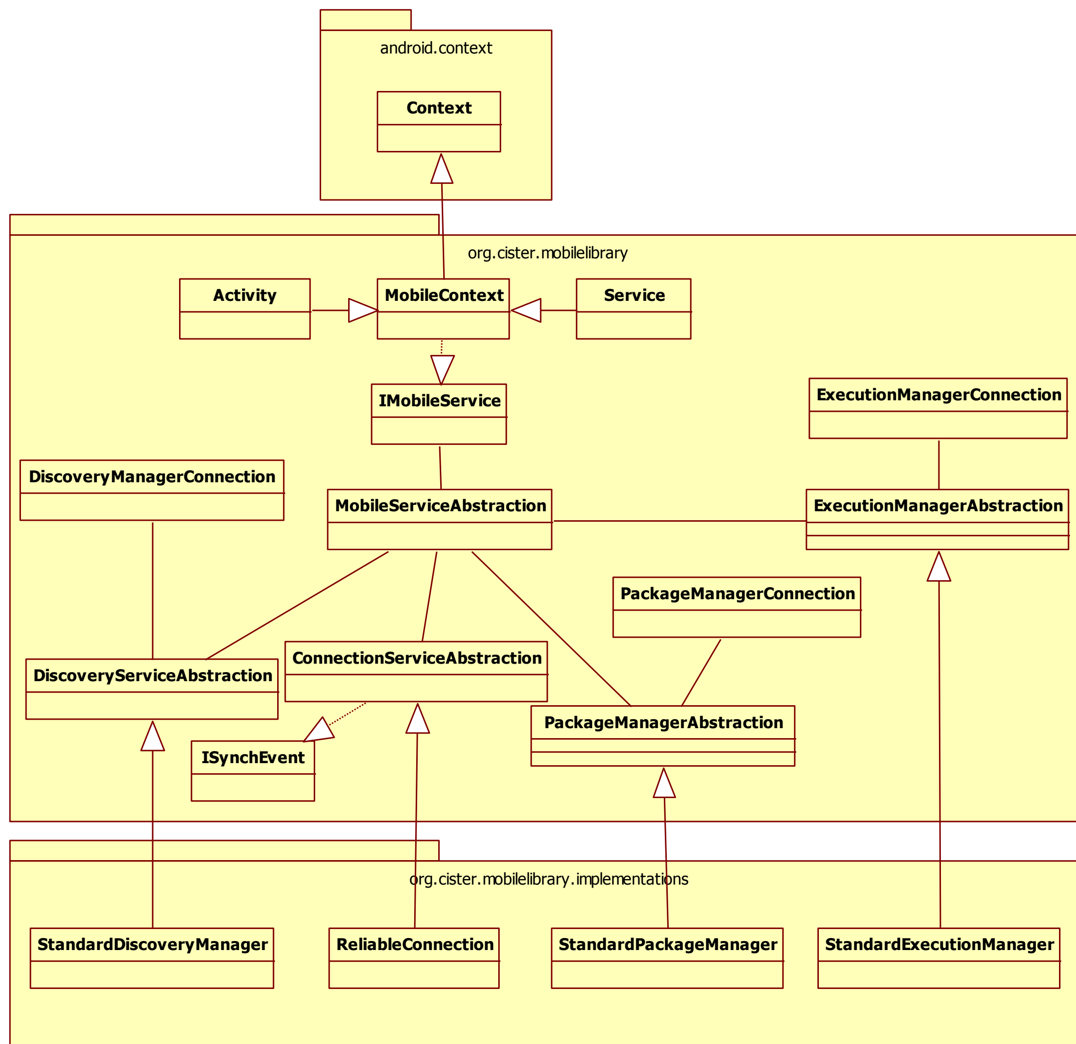

Figure 7. Mobile Library Class Diagram

The `IMobileService` class extends is the Android Context, which provides a well defined interface with the other classes. All these classes are abstract, therefore, developers can use the standard implementations provided in the Mobile Library.

This library also helps programmers to organize and develop service-based applications by provident already implementations of the required service interfaces. The implementations are contained in the `org.cister.MobileLibrary.Implementations package`, which can be used by the programmer. Otherwise the developer can extend the base classes to construct new implementations.

The MobFr is design to handle QoS requirements of the application, but its enforcement is a task of other components residing on the operating System.

3.3 CooperatES (Cooperative Embedded Systems)

The CooperatES (Nogueira & Pinho, 2009) is a framework that enables services to be executed in a distributed cooperative environment. This framework allows resource constrained devices to collectively execute services with their more powerful neighbours, meeting non-function requirements that otherwise would not be met by an individual execution. This framework is capable of managing nodes and grouping them into coalitions, allocate resources to each new service and establish an initial Service Level Agreement (SLA).

The CooperatES framework is based on the concept of resource management. When a device estimates that it does not possess enough resources to compute a set of instructions in a certain period of time, it reserves the required amount of resources on other neighbour devices. This has some challenges such as timing requirements which are unknown until runtime, making accurate optimization more difficult. Those challenges are overcame by making these new real-time systems adaptable to the environment, thus capable of reacting to changes in the operating conditions by acting on the applications' and system's parameters.

CooperatES is composed by three main modules: **QoS Provider**, **System Manager** and **Resource Manager** as shown in Figure 6.

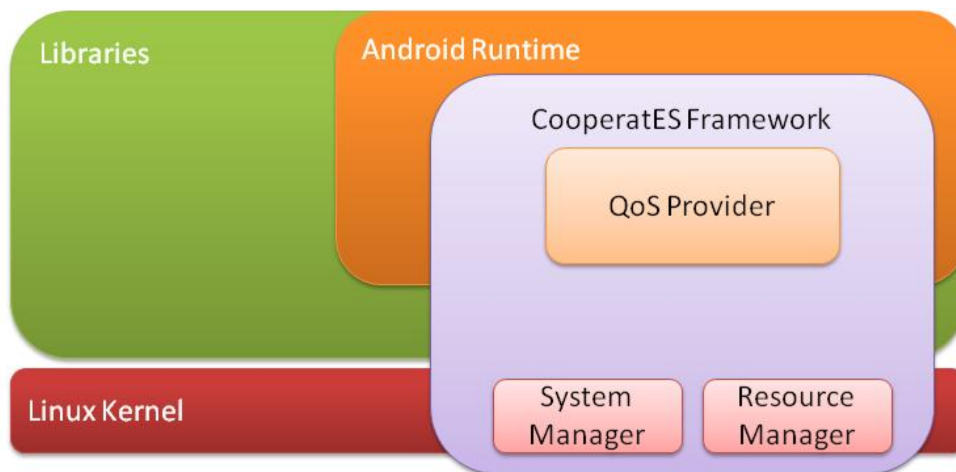

Figure 8. CooperatES Architecture

The **System Manager** is a module implemented in the Linux Kernel and is responsible for the connection interface that is able to detect which nodes are available and which are willing to participate in a new coalition.

The **Resource Manager** module is implemented in the kernel layer and is responsible for monitoring the available resources such as memory, CPU, network, disk, etc.

The **QoS provider** is a module implemented in the Android Runtime and is responsible for deciding whether to form a coalition of cooperative devices, or not. The decision is based on the data from the **Resource Manager** and the **System Manager**. In order for a device to be eligible to form coagulation, it has to be willing to do so and has to prove that it has enough resources.

The CooperatES Framework specifies the QoS requirements as group of parameters that are used by the system for adaptation proposes. These parameters, also called QoS Dimensions, are

related to a single aspect of the service quality or attribute which are associated to a value. This specification is structured as:

$$QoS = \{ Dim, Attr, Val, DAr, AVr, Deps\} \quad (1)$$

Where *Dim* is the set of QoS dimensions, *Attr* is the set of attributes of one or more dimensions, *Val* is the set of attribute's values, *DAr* is the set of relationships between dimensions and attributes, *AVr* is the set of relations between attributes and values and *Deps* is the set of dependencies between attributes.

Video Streaming applications are a good example of applications that can take advantage of CooperatES. In such applications, the domain could be specified shown in Listing 1.

Listing 1 - Example Application Domain

```
1. Dim = {Video Quality, Audio Quality}
2. Attr = {compression index, color depth, frame size, frame
   rate, sampling rate, sample bits}
3. Val = {1, integer, discrete}, {3, integer, discrete}, …, {
   [1,30], integer, continuous, …}
4. DA Video Quality = {image quality, color depth, frame size,
   frame rate}
5. DA Audio Quality = {sampling rate, sample bits}
6. AV compression index = {[0, 100]}
7. AV frame rate (per second) = {[1, 30]}
8. AV sampling rate (kHz) = {8, 11, 32, 44, 88}
9. AV sample bits (bits) = {4, 8, 16, 24}
```

This domain when specified in an application, enables users and service providers to reach an agreement on service provisioning; and the system to map dimensions to resources and perform quality trade-offs.

In each service the range of QoS preferences is provided by the user and can range from the desired QoS level to the maximum tolerable service degradation, independently of the service internals. It is important to note that the system tries to map specified levels in resource allocation in a decoupled manner (the QoS is not specified in terms of resource usage) according to the input QoS dimensions.

It is important to highlight that the CooperatES framework is not used in the development of the proof of concept application. Instead, the MobFr will use its resources to best monitor the network devices and their resources. For that reason, the final application is not capable of having absolute guarantees on its real-time behaviour.

3.4 Component interaction

MobFr also serves as the interface with between the Real-time Offloading Framework and CooperatES, thus it encapsulates CooperatES functions in order to be easily accessed by the framework. Figure 8 shows the interactions between the MobFr and the CooperatES.

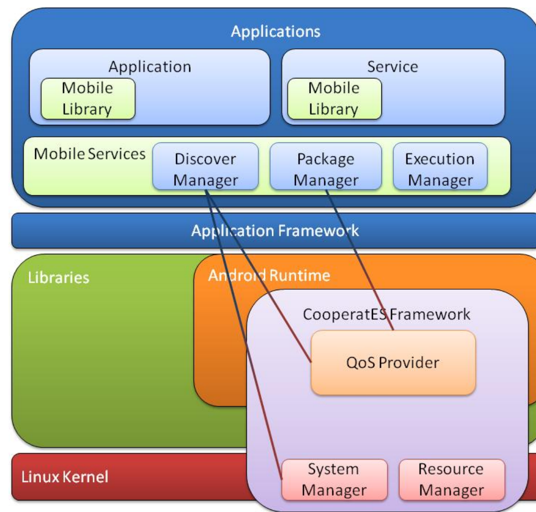

Figure 9. Component Interaction

The **Package Manager** sends data to the **QoS Provider** when it needs to consult the Services' Application Domain.

The **Discovery Manager** communicates with the **System Manager** in order to monitor the available neighbour devices. This includes managing when a new device enters or leaves the network or the list of the available devices.

The **Discovery Manager** provides a method `GetDevice()` which returns the best offloading candidate chosen by the **QoS Provider**. This method is used by any application or library that wants to get the best offloading candidate to start offloading.

3.5 Summary

This chapter introduces the components that support the Real-Time Offloading framework, the code mobile framework, MobFr and the real-time cooperative framework CooperatES.

MobFr supports the Real-time Offloading Framework by providing support for code mobility operations and communication between devices.

CooperatES supports the Real-time Offloading Framework by providing resource and network management operations such as determining the best candidate device to offload the code, the amount of resources available in the device, and monitoring the available devices in the network.

These components are completely transparent during the development of any application that uses the Real-time Offload Framework, the only requisite for the developer is to specify the resources required in the application domain.

The next chapter details the focus of this thesis, the design of the real-time code offloading system.

Chapter 4. Code Offloading in Real-time Systems

In this document we propose an adaptive code offloading framework for soft real-time systems. The system uses the components detailed in Chapter 3 as support for the code mobility and resource management. The framework is implemented as software application middleware depicted in Section 4.2. The framework is capable of deciding when to start offloading dynamically during the run time using a mechanism detailed in Section 4.3.

4.1 Introduction

The code offloading techniques proposed by other authors support adaptable applications that have variable QoS requirement, ranging from flexible sensor and control applications to multimedia applications, gaming applications and other applications. Our proposal is adequate for applications which periodically run services with variable execution time. In some aspects the solution addresses a similar problem to that of load balancing in real-time distributed systems (Lonnie, et al., 2000).

Although this dissertation uses a physics model for a game engine as a proof of concept, the designed framework supports most of the generic requirements of real-time applications. These engines usually run periodically, with a period that depends on an application specific rate, with a set of Core Services related with 3D/2D object movement, collisions, rendering, etc. When the size of data (number of simulated objects) is low, the mobile device handles all computations, but when the number of data items, or the computation requirements increases, only local execution may not be possible. In this case, instead of reducing the quality provided to the application (e.g., by reducing the quality of some of its computation, for instance the accuracy or simply by reducing the rate), the nearby nodes are sought to execute parts of the computation.

The offloading approach works by constantly monitoring the time required to execute the core services (t_{core}). Based on a set of past t_{core} times, the algorithm predicts its evolution; if it determines that the required rate cannot be achieved in the future, the offloading procedure is triggered in advance, minimizing the occurrence of timing errors. Therefore, it is possible to timely offload some of the core services to other nodes and executed there without reducing the rate or the supported quality.

Complementary, when it is not advantageous to continue executing the offloaded services in other nodes, migration can once again take place, and these services can return to be executed on the device.

To our best knowledge, this proposal is the first framework which is able to integrate dynamic real-time requirements into service offloading. The timing behaviour of the solutions proposed in (Nimmagadda, 2010) and (Sommer, 2010) has not, yet, been studied, and additionally, the application architecture proposed in (Sommer, 2010) is not generic thus not easy to adapt to other types of applications.

4.2 System Architecture

The architecture of the proposed system relies on the MobFr and the CooperatES frameworks (described in chapter 3), which are responsible for:

i) Detecting the neighbour devices;
ii) Determining the best candidate where to run the offloaded code, according to the QoS requirements of the application and the available resources on the neighbour nodes;
iii) Migrating the code and initial state;
iv) Controlling remotely the code execution;
v) Handling the transfer of data between nodes.

Figure 11 presents the structure of the code offloading framework, showing in the lower part the modules of the CooperatES framework, which manages the neighbour devices resources and coalitions and the MobFr which is responsible for supporting code mobility. The core modules provided by the MobFr framework are the Discovery Manager, Package Manager, and Execution Manager. Additionally, the MobFr framework also relies on the CooperatES framework for assuring that the QoS requirements of each module can be met.

The Real-time Offloading Framework is designed to simply the code offloading process, allowing application that require code offloading to be easily developed.

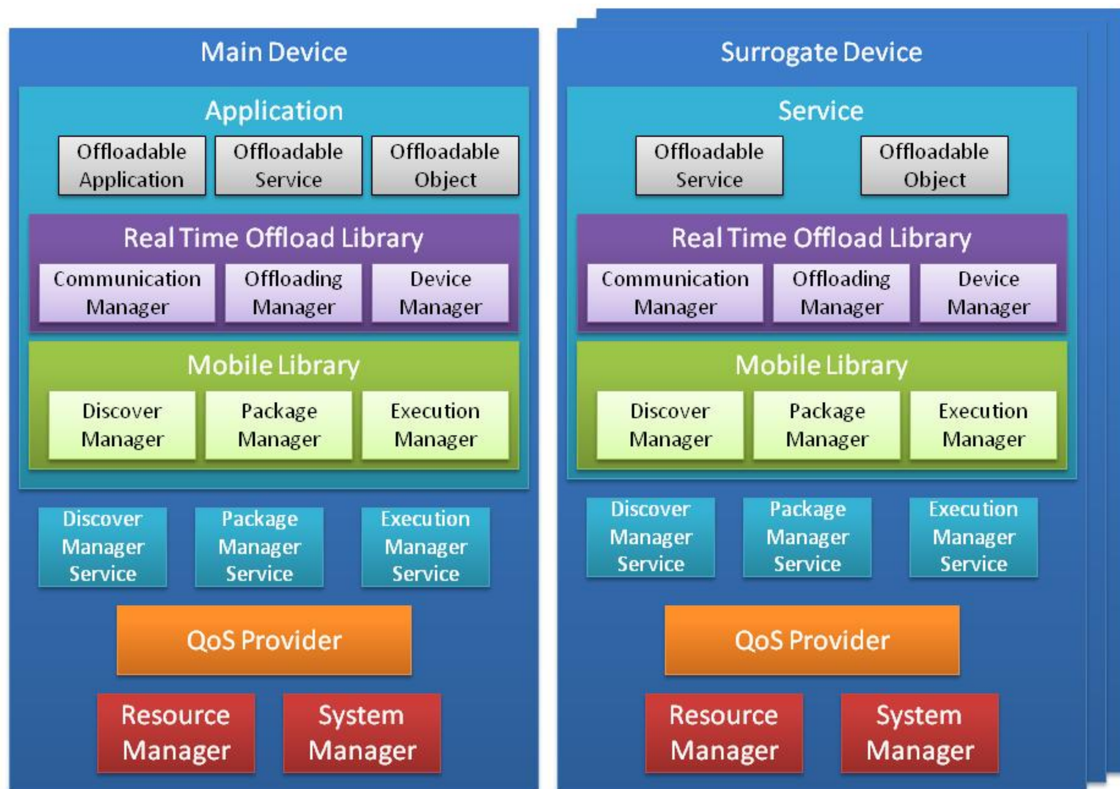

Figure 10. Offloading Structure

The **Offloadable Application** is the user application and a regular android application that is responsible for executing the **Offloadable Service**.

The **Offloadable Servic**e is the service whose code is distributed. This class inherits from an abstract class from the Real-time Offloading Framework which provides all the abstract methods the programmer needs to override. Since that class also inherits from the Android `Service` class, no additional code is required when executed in the Surrogate Device.

The **Offloadable Object** is the data used by the Offloadable Service. When the offloading library detects that the service needs to be executed remotely, these objects are distributed among all the available devices.

It is the responsibility of the **Offloading Manager** to take care of the initial configuration, monitoring the core execution times of the neighbour devices.

The **Communication Manager** takes care of communications between the main and neighbour device which include sending, receiving and aggregating the configuration data and the objects. This class also handles all the communication between the Real-Time Offloading Library and the MobFr framework. The communication manager uses the components provided by the mobile library to access the Services from the MobFr, the **Discovery Manager**, the **Package Manager** and the **Execution Manager**.

The code mobility process is the responsibility of the MobFr modules: the Discover Manager Service, The Package Manage Service and the Execution Manager Service.

The **Discovery Manager Service** is discovers neighbour devices on a local network, advertising the host's resource availability and gathering information about the resource availability on

neighbour devices. The Discovery Manager can use its functionalities or access the ones provided by the underlying QoS framework, more specifically the System Manager

The **Package Manager Service** installs, uninstalls and transfers services. This module is also responsible for the interaction with the underlying operating system QoS Manager, the QoS Provider, in order to request specific QoS levels for the service being transferred.

The **Execution Manager Service** executes services on a neighbour node through the exchange of Android intents.

The **QoS Provider** is responsible for managing the network device coalition. This component is capable is also capable to determine the best candidate neigbour device to offload based on the service's application domain and data from the Resource Manager and System Manager.

The **Resource Manager** administers the system resources, either locally or in a distributed environment. Consequently, this module can interact with Resource Manager in order to choose the most appropriate nodes where to run the offloaded services.

The **System Manager** monitors the available devices in the network, thins includes detecting when a new device enter or an old one leaves.

4.3 Real-time Offloading

The decision to offload code is performed dynamically in runtime and adapts to the current state of the system. Therefore, the main device must determine when to start offloading the code, when it needs additional resources and when to stop offloading. The framework decides when to offload code depending on the data gathered from past executions, the information it has of its neighbours, and the information of its surround environment, using a real-time offloading algorithm.

The main objective of the real-time offloading algorithm is to dynamically adapt to the varying execution times by offloading computation to neighbour nodes in a timely way. The term timely refers to the notion that the user should not notice any disruption on the application behaviour. To that purpose, the offloading algorithm tries to predict the forthcoming core execution times, based on past execution times. Figure 10 illustrates the algorithm operation. The core services' execution time on the main and neighbour device are represented by square and triangle marks, respectively. The continuous line, without marks, shows the linear regression that best approaches the evolution of t_{core} on the local device. The line is obtained by considering the 6 points, from 0 to 165 ms. The example assumes an application which is executing periodically with a period of 1000/30 ms.

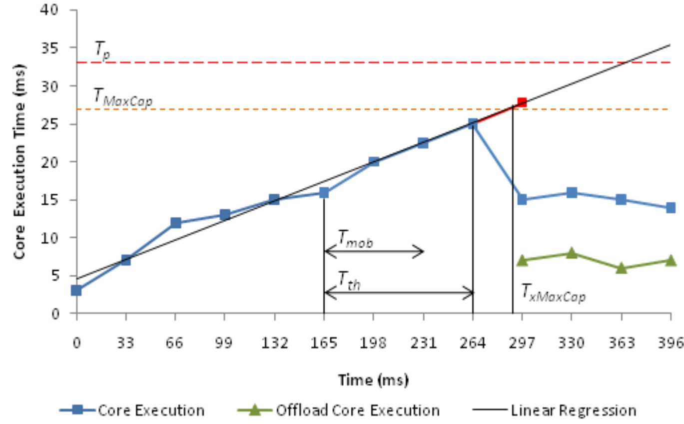

Figure 11. Offloading algorithm example

Based on the linear regression parameters, at 165 ms, it is possible to estimate the instant, $t_{xMaxCap}$, at which the core execution time will exceed the maximum capacity of the original device – t_{MaxCap}, which for the case of Figure 2 occurs in the interval between 264 ms and 297 ms. Obviously, to fulfill the operating objective of the framework, no QoS disruption should occur, consequently the code mobility operation, which precludes the offloading procedure, should be completed prior to 264 ms.

The device must offload the code before the execution time reaches that threshold. If the time required for code mobility is equal to t_{mob} then the offloading decision can be expresses as:

$$t_{th} = \left\lfloor \frac{t_{xMaxCap} - t_{actual}}{T_p} \right\rfloor \times T_p \leq \left\lceil \frac{t_{mob}}{T_p} \right\rceil \times T_p \qquad (2)$$

If this expression is true then the system should start the offloading procedure of its selected services in parallel with its current operations at 165 ms. The floor and ceiling functions, used in Eq. (2), normalize all results to multiples of T_p. Such as other approaches, such as (Kemp et al., 2010) and (Cuervo et al., 2010) it is up to the programmer to determine, while developing, the application services to be offloaded.

Figure 2 shows that prior to 299 ms the necessary code is transferred to a neighbour device, consequently at 297 ms there is a noticeable reduction on the core execution time on the main device, since the neighbour device enters into operation. In this example, there is only one neighbour device, but, if required, the middleware can handle offloading to a group of devices. Note that it is up to the programmer to determine how to parallelize the application code, while the framework handles all run-time operations, guaranteeing that:

i) Each of the neighbour devices receives new data from the main device;
ii) Executes the calculation over that data and returns the results;
iii) After receiving all responses the coordinating device aggregates the responses from the neighbour devices with its local calculations.

4.4 Code Offloading Algorithm

The offloading algorithm pseudo-code is shown in Listing 2. This algorithm assumes that the application has a set of interface methods available which can be used by the underlying offloading framework. Nevertheless, some methods are periodically called by the application

itself, such as the `update` method, which must be periodically executed by the application with a periodicity of T_p. The update method starts by determining if the services had already been offloaded to other neighbour devices (line 2). If the condition is true then it tests to determine if an additional neighbour node is needed (lines 23– 31). That is done by testing if the execution time on each device reaches the maximum execution capacity in that node using Eq. (2). The `requiresNewSurrogate` invokes the method `tryRebalance`, an application dependent method determines if it is possible to avoid adding a new device by rebalancing the load between nodes. It is important to note that this capability has not been implemented nor studied in detail.

Listing 2. Offloading Library Methods

```
1.  Method update() {
2.    If isOffloading()
3.      If requiresNewSurrogate()
4.         new Thread(addAdicionalDevices())
5.         runOffloaded()
6.      else {
7.        if needsToStopOffloading() {
8.          runLocally()
9.        } else
10.         runOffloaded()
11.       }
12.     }
13.   else //if is not offloaded
14.     If requiresNewSurrogate() {
15.       new Thread(addAdicionalDevices())
16.       runLocally()
17.     } else
18.       runLocally()
19.     }
20.   }
21. }
22. Method requiresNewSurrogate() {
23. Output result:bool - determines if a new neighbour node is
    required.
24.
25.   ForEach(dev in devices) {
26.     If eq (1) is true
27.       If !tryRebalance() Return true
28.     Return False
29.   }
30. }
31.
32. Thread addAdicionalDevice() {
33.   newDev = DiscoveryManager.getDevice()
34.   If (newDevice != null)
35.     Devices.add(newDev)
36.     OffloadCode(newDev)
```

```
37.   Else
38.     signalError()
39. }
```

If an additional neighbour device is needed then the `addAdditionalDevice` thread is started (lines 33–40). This thread might require a few cycles to complete due to the time consuming sequence of operations that it must execute when using the underlying code mobility framework (Gonçalves et al., 2010). It starts by determining if there is a node, with available resources in the neighbourhood (line 34), after it offloads the required code to it (line 37), otherwise an error is signaled to the application. It is important to note that, once started, the offloading process is not stopped, although the main node may choose not to execute offloaded services.

The algorithm then runs the core services in offloaded mode (Listing 2). Basically, it starts by partitioning the data to be computed by neighbour devices. This operation is done by an application specific method and it can be adjusted on every cycle, e.g. for load balancing proposes. Afterwards, the data is sent to neighbour devices, computed and the results are returned, using the method `sendData&Execute`. The last step (line 8) is related to the aggregation of results on the main node in order to compute the final results.

Listing 3. Method runOffloaded

```
1. Method  runOffLoadded(offloadService)
2. Input offloadService: the code that can be executed in
   offloading.
3. {
4.   parts = offloadService.dataPartitioner()
5.   sendData&Execute(devices, parts)
6.   Result[0] = offloadService.runLocally(parts[0])
7.   receiveData(devices, results)
8.   offloadService.aggregateResults(results).
9. }
```

Listing 1 also accommodates the case when the main node is not offloading any computations (lines 15 – 20). In this case, it determines if another neighbor device is needed, and, if needed, it releases a thread that runs the method `addAdditionalDevice` to prepare the offloading of code. Meanwhile, the code is executed locally. Another situation occurs when the load no longer justifies the offloading procedure (tested in line 7 of Listing 1), but handling this run-time adaptation is outside the scope of this dissertation.

For a better understanding of the offloading algorithm, Figure 9 depicts the sequence diagram of its execution which more clearly shows the relation between the different actors: `OffloadableApplication`, `DeviceManager` and the `OffloadableService`.

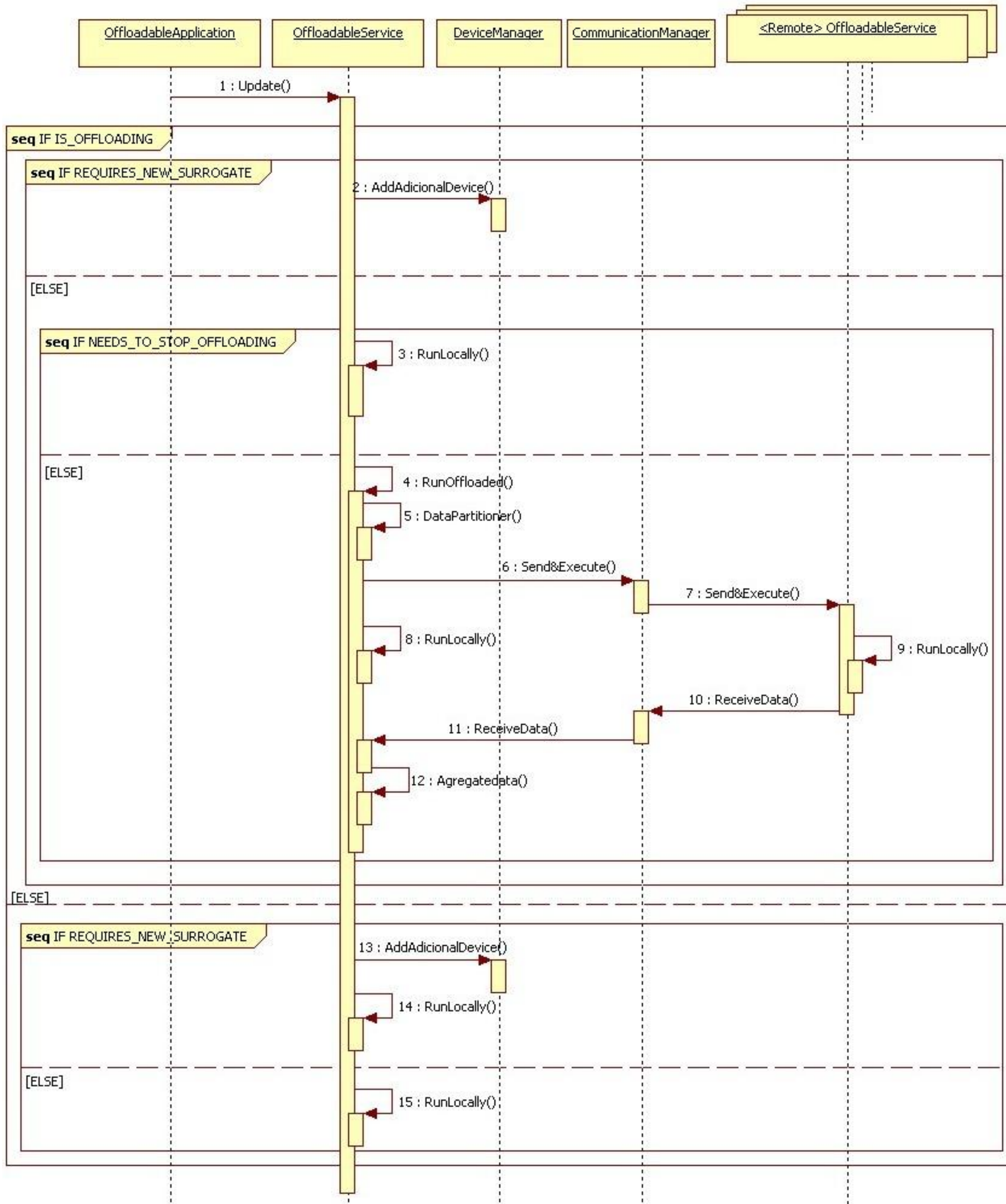

Figure 12. Real-time Offloading Sequence Diagram

4.5 Timing Parameters

The real-time offloading algorithm operations depend on several timing parameters. In this section, we illustrate how to determine $t_{xMaxCap}$, t_{mob} and how to measure the core execution time.

4.5.1 Predicting $t_{xMaxCap}$

To determine when to start the offloading procedure, Eq. (2) requires the knowledge of $t_{xMaxCap}$ time, the time at which an estimated value for the core execution time (t_{core}) reaches the maximum capacity (t_{maxCap}).

$t_{xMaxCap}$ can be calculated using any statistic regression approach such as linear regression, exponential regression, polynomial regression. The choice of the best approach should have in consideration that:

- The more precise it is, the more resources the calculations consume.
- Most mobile devices can only calculate float-point equations by software.

The solution we propose is to use linear regression to determine the line which best approaches the evolution of the core execution times and determine when that line crosses the maximum capacity line. Each point i of the estimation line is expressed by the formula $t'_{core,i} = m.t + b$. Where m, the line slop, is calculated by solving the following equation:

$$m = \frac{n\sum_{i=0}^{n}(t_i \times t_{core,i}) - \sum_{i=0}^{n} t_i \times \sum_{i=0}^{n} t_{core,i}}{n\sum_{i=0}^{n}(t_i)^2 - \left(\sum_{i=0}^{n} t_i\right)^2} \tag{3}$$

In this equation n is the number of past core execution times being considered. Parameter t_i is the time at which the core execution time ($t_{core,i}$) had been measured.

The setting of n has a big impact on the behaviour of the algorithm. If n is set to a small value then the algorithm becomes more sensitive to rapid changes on the t_{core} value, otherwise the algorithm is slower to react. Parameter b is calculated by solving the following equation:

$$b = \frac{\sum_{i=0}^{n} t_{core,i} - m\sum_{i=0}^{n} t_i}{n} \tag{4}$$

After having calculated m and b it is possible to determine $t_{xMaxCap}$ as follows:

$$t_{xMaxCap} = \frac{t_{maxCap} - b}{m} \tag{5}$$

Obviously, other regression algorithms could be used, like a polynomial regression, but the computation power required would be much higher, although the results could potentially also be more precise, particularly, when the variation of the core execution time does not follow a linear rule. The main advantage is that the proposed algorithms can be executed with minimum overhead in devices with limited computation capabilities.

4.5.2 Estimating t_{mob}

Another value required to determine when to start offloading is the period of time that elapses from the time when the offloading decision is taken until the new device is ready to start executing the offloaded code – the code mobility time (t_{mob}).

We support this calculation on the formulations proposed in (Ferreira L. , 2011), which can be adapted to this specific case. Therefore, t_{mob} can be calculated by:

$$t_{mob} = t_{conf} + t_{code} + t_{ist} + t_{start} \tag{6}$$

Where t_{conf} is the time required to find a feasible system configuration, i.e. a neighbour node where to offload the code. To obtain a system configuration the framework uses the functionalities offered by the CooperatES framework. t_{code} is the time required to transmit the offloaded code. Some configuration data can also be sent along with the code, which requires a time of t_{ist} to be transmitted. Finally, the code must be installed on the neighbour node and started, prior to be ready to start processing items sent from the source node, thus requiring a time of t_{start}.

It is important to note that these timings are not worst-case timings, but they represent just average values or any other kind of stochastic value.

4.5.3 Measuring t_{core}

An essential part of the estimation of the t_{mob} is to measure the core execution time of all neighbour nodes ($t^{s}_{core}, s \in \{1, 2, \dots, nSurr\}$) and on the local node (t^{0}_{core}). On the local node this time is simply the execution time of method `runLocally()`. The measurement of the core execution time on the neighbour nodes is performed at the source node, since it must also take into account the communication delays, consequently:

$$t^{s}_{core} = t^{main \to s}_{req} + t^{s}_{exec} + t^{s \to main}_{res} \quad (7)$$

Where $t^{main \to s}_{req}$ represents the time required for the data to be sent from the source to the destination node. Time t^{s}_{exec} is the execution at neighbour node s and $t^{s \to main}_{res}$ is the time required to transmit the response from the neighbour node back to the source node.

4.6 Summary

This chapter introduces the reader to the offloading mechanism used in the Real-time Offloading Framework. The mechanism takes in account the QoS requirements of the application being executed, whilst allowing offloading services to several neighbour nodes.

The offloading framework relies on an algorithm to predict the instance of time when the device will not have enough resources uses a statistic regression approach based on the results from past executions is used.

When the algorithm predicts that the device does not have enough resources in the near future, the underlying components of the architecture, the MobFr and CooperatES framework, are signaled in order to choose and prepare a neighbour device to start offloading.

The next chapter presents how the proposed computing model can be implemented in an Android environment.

Chapter 5. Framework Implementation

The Real-time Offloading Framework is implemented as middleware. The solution simplifies the development of code offloading application by providing the all the classes and methods the programmer has to override.

5.1 Introduction

Developing distributed applications is not a simple task (Vigna, 2004). The Real-time Offloading Framework simplifies the process by providing a more dynamic and flexible solution to solve the performance gap, and allows the mobile applications to dynamically offload some of the applications' services to neighbour nodes.

The Offloading library automates all the offloading process, leaving the programmer to specify two components: the service that is executed using code offloading and the data used by the service.

5.2 Offloading Library Class Diagram

The Real-time Code Offloading Framework is designed to allow easy development of applications that uses code offloading. The programmer only specifies the core service and the data used by it. The framework provides the `OffloadableServiceAbstraction` abstract class for the service to extend (Figure 13). This class already implements some methods which are the same for each application using Real-time Offloading Framework while leaving others for the programmer to override. An application using the framework executes the service through the method `update()` which decides based on the history of past executions and the resource availability if the application requires additional resources, and if it is going to be executed locally or offloaded. The `runLocally()` method executes the required instructions and is defined by the programmer. The `runOffloaded` method is invoked when the core service is execute using code offloading, as detailed in Figure 11 of Chapter 4. The core service must also override the abstract methods required for data partitioning (`dataPartitioner`), data aggregation (`agregateData`). The Method `tryRebalance()` is a method that the programmer can override and is used by the framework as an attempt to avoid adding additional resources.

Listing 4 details the `runOffloaded` method.

Listing 4 - runOffloaded Method

```
1. public void runOffloaded(float elapsedTime, OffloadableObject[] offloadableObjects)
2. {
3.   try {
4.
5.     int numDevices = _deviceManager.getNumAvaiableDevices.Lenght();
6.
7.     ArrayList<OffloadableObjectsDevices> parts = dataPartitioner(data, numDevices);
8.
9.     _communicationManager.SendAndExecute(parts);
10.
11.    OffloadableObjectAbstraction[] localData = parts[0].offloadableObjects;
12.    runLocally(elapsedTime, localData);
13.
14.   ArrayList<OffloadableObjectsDevices> resultsTemp = _communicationManager.buffer.getAll();
15.
16.    ArrayList<OffloadableObjectsDevices> results = new ArrayList<OffloadableObjectsDevices>();
17.
18.    results.add(localData);
19.    for(int i =0; i < resultsTemp.length; i++)
20.     collections.add(collectionsTemp[i]);
21.
22.    data = agregateData(results);
23.
24.
25.   } catch (InterruptedException e) {
26.    Constants.LogError("Error Getting the offloadableObjectCollections from buffer");
27.   }
28.   _isOffloading=true;
29. }
```

The data is distributed using the method `dataPartitioner()` shown in line 7. This method returns an array of `OffloadableObjectsDevices` objects which contain an array of `OffloadableObjectAbstraction` objects and the identifier of the device which is responsible for their execution. The first element of the `OffloadableObjectsDevices` array is executed in the local device and the others are executed in the neighbour devices. The method `send&Execute()` from the `CommunicationManager` sends these objects to the neighbour devices (line 9). All devices then execute the `runLocally()` method. The last part of the code, the framework receives all the results by calling the method `getAll()` from the

buffer in the communication Manager. This method is synchronous and waits until all the devices have returned the data or after the time out.

The timeout is used in case a device disappears from the network, which means that the data is lost. The programmer decides what to do with the lost data. He can either use the data from the previous `update` methods or discard it.

The last step of the `runOffloaded` method is the aggregation of the data (lines 13-22). This method is the reverse of the data partition in terms of the data flow. The data first grouped in a single list of `OffloadableObjectsDevices`, which is used by the method `agregateData (Line` 22) to produce the final result.

The data distributed in the Real-time Offloading Framework extends the `OffloadableObjectAbstraction` class and the programmer does not need to override any method.

The `DeviceManager` class assists in the device monitoring. This class manages the devices being used by the Real-time Offloading Framework and is used during the `update()` method to add new devices for the code to be offloaded using the algorithm . That device is chosen using the method `getDevice()` from the `StandardDiscoveryManager` class.

The `CommunicationManager` class handles all interactions between the Framework and MobFr. This class has access to a list of devices in the network provided by the Discovery Manager Service. The `CommunicationManager` manages the communication between the main and neighbour devices which includes sending data, receiving data and buffering it. The method `send&ExecuteData()` sends the offloadable objects to the neighbour devices through the `sendData()` method.

The `CommunicationManager` also serves as the interface between offloading library and the `StandardPackageManager` and the `StandardExecutionManager` classes which are responsible for installing and executing services remotely. The method `transferAPK()` has as the parameter the name of the APK where the service is located and the name of the device and invokes the Package Manager service in order to install the required APK In the remote device. The method `executeActivity()` receives the service Intend and the device name and triggers the Execution Manager service to communicate with the remote device Execution Manager in order to start the service. All the communications between network devices afterwards uses the method `sendData()`. The data is received using a separated thread whose function is to wait for new data and storing it in a thread-safe bounded buffer.

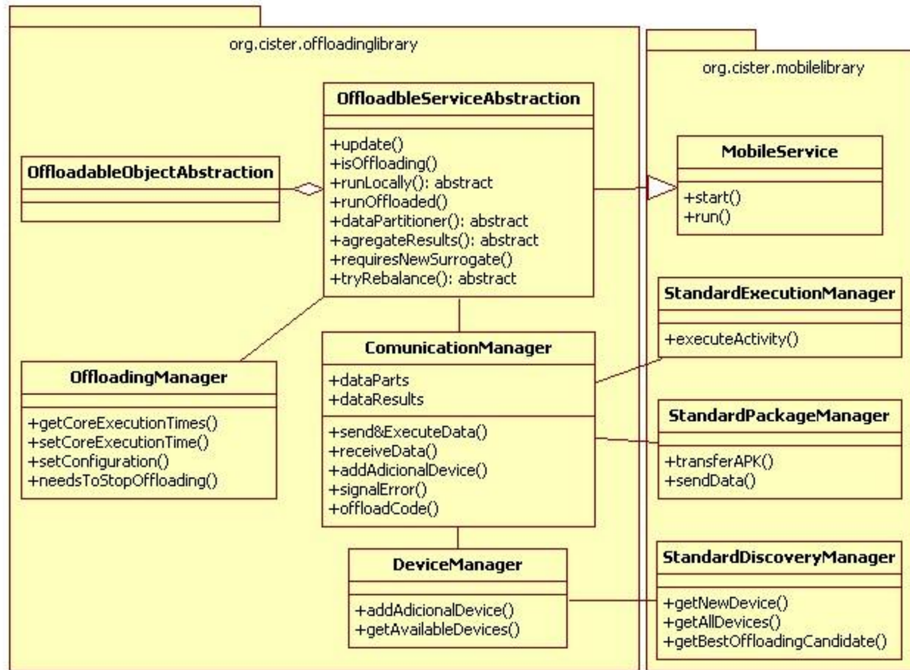

Figure 13. Framework UML Model

It is the responsibility of the `OffloadingManager` to take care of the initial configuration, monitoring the core execution times and control the creation of new neighbours by applying the algorithms proposed in (Ferreira, et al., 2011) as detailed in Chapter 3.

The `OffloadingManager` stores the previous core execution time data and uses it to predict when the service will reach the maximum capacity using the algorithm detailed in Chapter 4. The data is structured as in figure 14.

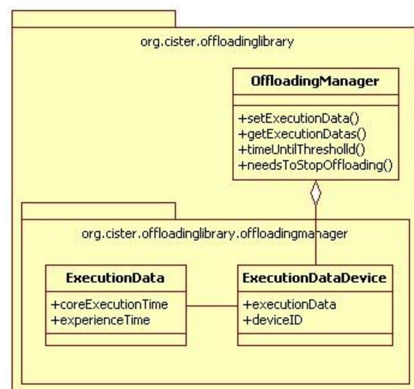

Figure 14. Offloading Manager UML Diagram

The `ExecutionData` class holds the core execution data of one instance of the application. Both values are expressed in nanoseconds.

The `ExecutionDataDevice` represents the device and a list with its `ExecutionData`.

Additional classes are present in the final application, but their only purpose is to support the ones present the specified architecture.

5.3 Application Implementation Overview

The Real-time Offloading Framework allows the easy development of code offloading applications. To do so, the application has to implement two classes: the `OffloadableService` and the `OffloadableObject`.

Figure 14 shows the minimum that an application required when using the Real-time Offloading Framework

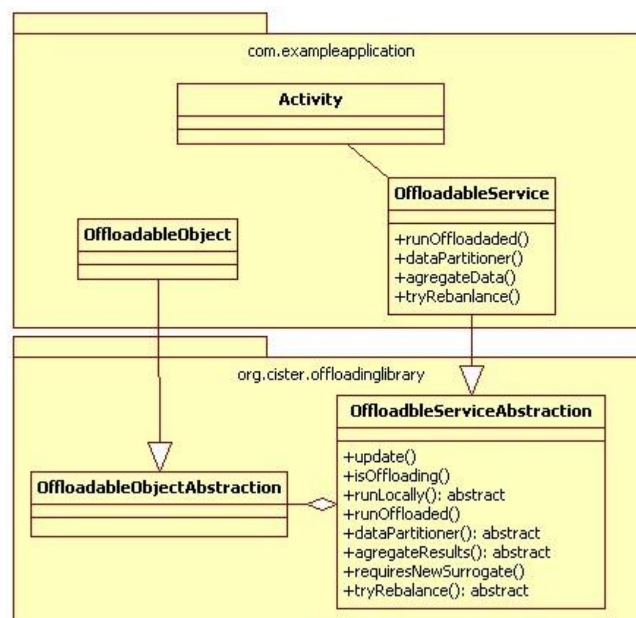

Figure 15. Example application UML Diagram

The `Activity` class invokes the service. The only requirement it has is to invoke the `update()` method from the `OffloadableService` in a pre defined interval of time.

`OffloadableService` is the class that implements the service whose code is distributed This class has to override four methods: `runOffloaded()`, `dataPartitioner()`, `agregateData()` and `tryRebalance()` whose functions are detailed in Section 5.2

`OffloadableObject` represents the objects used by the service. The objects are distributed by all the available devices during the code offloading procedure.

5.4 Summary

This chapter documents the implementation of the offloading library using the offloading design pattern detailed in the previous chapter.

The Real-time Offloading Framework automates all the code offloading process, leaving the programmer to specify the service and its data.

The next chapter details the development of an application capable of using the offloading library

Chapter 6. Framework Demonstrator Implementation

This chapter describes the implementation of the application created as a proof the concept for the Real-time Offloading Framework, the external libraries that supports it and performance optimization techniques used.

6.1 Introduction

The application described is this chapter serves as proof of concept of the Real-time Offloading Framework. The application is an interactive physics simulation for the Android OS and is designed to test the main functions of the framework and to evaluate its performance.

The application uses two additional libraries: a game engine and a physics engine. The game engine is responsible for drawing objects on the screen and managing the application life cycle and the physics engine is responsible for calculating the movement of the objects and their collisions.

In the context of the Real-time Offloading Framework, the physics engine is the service, the physics objects are the objects of the service and the game engine is responsible for executing the service.

6.2 Application Overview

The application starts with blank area. Each time the user presses the screen, the application generates a random object (Box, Triangle, circle or hexagon). The object moves using the physics calculations. Whenever the application predicts that will not have enough resources to continue executing locally, it then distributes the physics computations by the neighbour devices.

All the objects are affected by gravity and can collide with themselves or with the boundaries of the physics world. When a collision occurs, their direction and velocity change.

Figure 16 shows the application UI during the execution of the application.

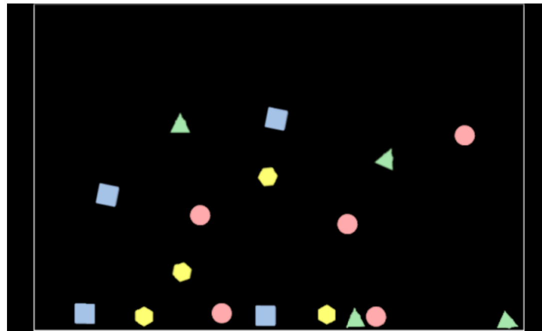

Figure 16. Application Screen

When the device needs to offload, the physics engine distributes the physic objects by all available devices using an algorithm that divides the physics world in small areas and distributes all the areas by all the available devices.

A typical physics engine in a gaming application is executed in each frame of the game but because of the network latency makes that impossible, the engine rather than running every frame, it run only 8 frames per second, therefore we specify the t_{MaxCap} as 128 ms The application uses interpolation to smooth the object translation in order avoid to the user to notice any disruption on the screen.

The application shares some similarities with work in (Chabukswar & Lake, 2005). In this paper the author specifies how to implement a game where the physics are calculated in a thread while the rest of the game is calculated in another. In this application, the physics is being offloaded in the cloud while the rest of the game is being executed in the device. Executing an application in concurrent threads and in concurrent devices is different as communication between threads is simpler than communication between devices.

The game engine used by the application is specified in the next section.

6.3 Game Engine

The game engine manages the application life cycle. In the application context, the game engine is responsible for invoking the `update()` method from the service. The game engine used in this application is based implementation of the AndEngine (Gramlich, 2009), an open source game engine and the engine used in Replica Island (Pruett, 2010). The main focus of the game engine is `Game` class as represented in figure 17.

Figure 17. Game Engine Class Diagram

The method setFrameRate() is used by the application to specify the frame rate. This will indicate the frequency of which the engine will invoke the methods draw() and update().

The game uses a physics engine that is responsible for moving the objects and manages their collisions. Next section introduces which are used in the application.

6.4 Physics Engine

The application being developed uses a real-time 2D physics engine based on rigid Body dynamics. The decision on which was the best physics engine required a careful consideration and analysis on the advantages and disadvantages of each available engine.

Some of the prerequisites of the physics engines were:

- Implementation in Java or C++;
- Ease of integration with the game engine;
- Performance;

After analyzing various engines, the ones that best fit the application were:

- Custom Engine: Simple physics engine developed for testing proposes and used to achieve the results visible in the introduction chapter.
- Box2D. A very famous open source physics engine used in many commercial projects. (Catto, 2007)
- Phys2D. A fork of an early version of box2d. (Phys 2D, 2008)

Table 1 shows the advantages and disadvantages of each physics engine.

Physics Engine	Advantages	Disadvantages
Custom Engine	Already implemented. Easy to change.	Only capable of simulating the interaction between rectangular shapes.
Box2D	Capable of simulating the interaction between any non convex polygon and circles. Better documentation and support.	Difficult to change.
Phys2D	Capable of simulating object as well as Box2D Easier to implement than Box2D.	Worst performance than Box2D

Table 1. Physics Engines Comparison

The physics engines described in Table 1 are not prepared for distributed systems. Consequently, the adaption of the software has to take into account the following aspects:

- How to distribute the physics computation by all devices
- How to monitor the execution times.

After much consideration, and based on performance and ease of integration, the chosen engine was the **Box2D Physics Engine**.

The Box2D engine is structured as in figure 18.

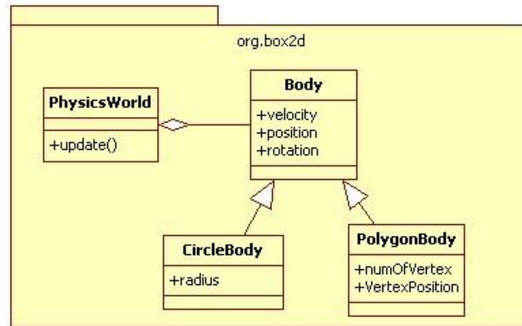

Figure 18. Box2D Class Structure

The `PhysicsWorld` is composed by a collection of `Body` elements. Each `Body` element represents a physics object that can have a circular or polygon shape. Each time the `update()` method is invoked, the physics engine calculates the new position, velocity, direction and rotation of all the objects. This is achieved by using the collision detection and collision response detailed in Chapter2.

It is important to highlight that Box2D is composed by large number of classes, but for simplification purposes, only these ones are presented.

6.5 Physics World Partition

The development of the physics world division takes in consideration researches of distributed physics in multithreading environments (Chabukswar & Lake, 2005) and distributed server-client (Fiedler, 2006)

The physics world division implemented in this application is divided in two phases and assumes that there is a physics simulation running with many different objects distributed in the physics world.

The first phase is choosing the number of areas in which the world is divided. The number of areas must be higher than the number of available devices so that each can have at least one area. The number of part impacts the how realistic and how balanced the distribution is.

- The bigger the number of areas, the most balanced the calculations area is.
- The smaller the number of areas, the most realistic the results are.

The second phase is assigning each object to a part. Any object that is on the border of two or more areas, it assigning those areas. The duplication of the object is solved during the aggregation of the physics world. Since each object has a unique id, when the main device receives all the objects, if any object is duplicated, then both objects are re-joined.

Figure 19 illustrates an example of the results before and after the physics world partition.

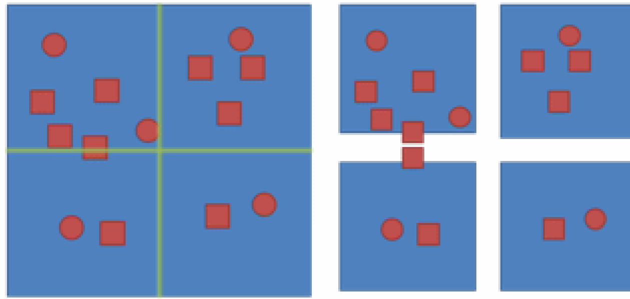

Figure 19. Physics World Division

The example in Figure 19 shows a world being divided into four parts where there is one object in the border of two parts. Listing 5 further details the pseudo-code of the division algorithm.

Listing 5 - Physics World Partition

```
1. Method WorldDivision(world, nDevices)
2. Input world: physics world to divide; nDevices – number of
   devices
3. Output areas: areas of the physics world.
4. Begin
5.   nParts = FindCorrectNumberOfParts(world, nDevices)
6.
7.   partWidth = world.width/ nParts.x
8.   partHeight = world.height/ nParts.y
9.   Area[] areas = new part[parts.x * parts.y];
10.        Foreach(object b in physicsObjects)
11.          x = b.x / partWidth
12.          y = b.y/partHeight
13.          areas [x + y * nParts.y].add(b)
14.          If(CollisionUpperBorder(b, PartHeight) AND X >=
   0)
15.            areas [x+1 + y * nParts.y].add(b)
16.          ElseIf(CollisionBottomBorder(b, partHeight) AND X
   < nParts)
17.            areas [x-1 + y * nParts.y].add(b)
18.          EndIf
19.          If(CollisionLeftBorder(b, PartWidth) AND Y >= 0)
20.            areas [x + (y-1) * nParts.y].add(b)
21.          ElseIf(CollisioRightBorder(b, partWidth) AND Y <
   nParts)
22.            areas [x + (y+1) * nParts.y].add(b)
23.          EndIf
24.        EndForEach
25.      End
```

The function `FindCorrectNumberOfParts` in line 5 calculates the number of areas in which the physics world is divided and returns a variable that represents the number of columns and the number of lines. For example, if there are 3 available devices, this number should be

bigger than 3, or else some devices would not perform any calculation. The Area class is composed by a list that stores physics objects which are added using the `add(object)` function (Line 13) . The `Foreach` block assigns each object to its respect parts. If an object is in the border of two parts, the both parts receive the object (Lines 14-22). The function finishes by returning the bi-dimensional array containing the parts containing the objects.

Another important aspect of the data partition is the aggregation. In the aggregation phase all object are placed in the original physics world. Any object that resided on the border that was split is joined together in this phase. This is possible because the `OffloadableObjectAbstraction` class has an id member which is a unique number and that identifies every object. The result object takes in consideration the transformations occurred to both objects that created it.

6.6 Application Structure

The application is structured as detailed in Figure 20.

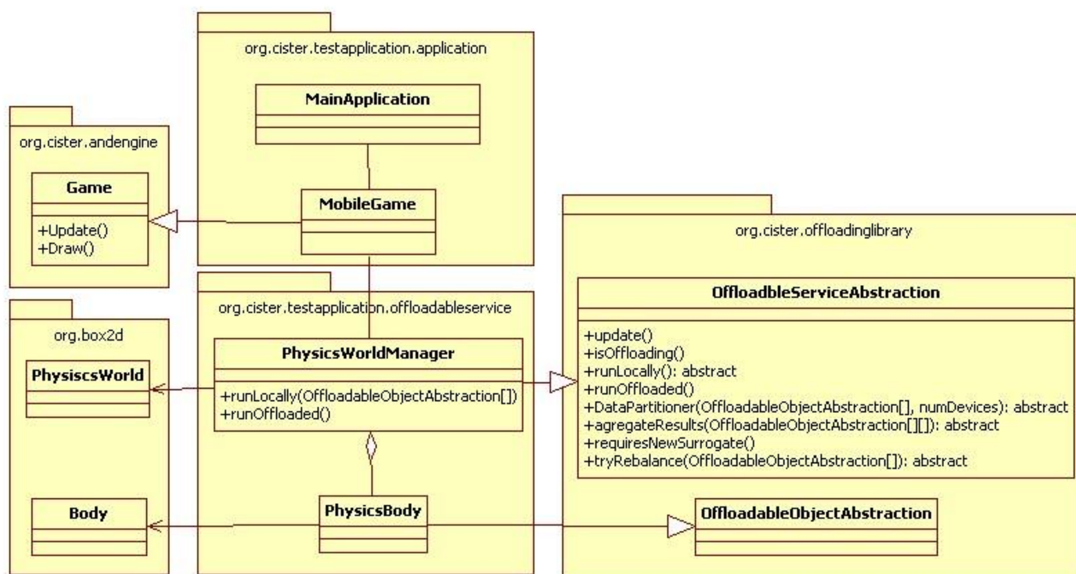

Figure 20. Application Class Diagram

The `MobileGame` class extends the game engine's `Game` class. This class provides a method named `setFrameRate()` that sets the number of times the update method is invoked per second it also support methods to draw objects in the canvas, which in this project are the ones from the `PhysicsBody` class.

The `PhysicsWorldManager` class is responsible for all the physics calculation and his code can be executed using code offloading. This class extends the `OffloadableServiceAbstraction`. The only methods that are required in order to allow code offloading are: `runLocally()`, `PartitionateData()`, `AgregateData()` and `tryRebalance()`. The `runLocally()` method executes the code in the `PhysicsWorld` provided by Box2D using the device's share of physics object, in case of being offloading or all the objects is running locally. The `PartitionateData()` method distributes the physics objects by all the available devices, as detailed in Listing 5. The

AgregateData() joins the all the transformed objects and the tryRebalance() method redistributes the physics objects by the devices. Since the offloading library has a transparent design philosophy; this class runs on the neighbour devices without any required change.

In the runLocally method inserts the PhysicsBody objects in the PhysicsWorld and invokes the method Step.

6.7 Performance Optimization Techniques

The optimization techniques were one of the main focuses during the implementation and generated some technical reports (Silva & Ferreira, 2010). The next section present some optimization made to the serialization algorithm, network latency and other general ones.

6.7.1 Serialization

Object serialization and deserialization is the process of converting a data structure into a sequence of bytes, in order to be stored in a memory buffer, or to be transmitted across a network connection, and the reverse.

Object serialization and deserialization can be achieved using different methods. In this subsection we analyze the serialization/deserialization process using java built-in methods, described as "Standard" methods, and using custom methods created, described as "Manual" methods.

Standard methods use the ByteArrayObjectOutpuStream, which is able to serialize any variable, as long as its class inherits from object and implements Java.io.Serializable. The manual method involves parsing each Fundamental data types in is respective byte format. For example, image an object composed by 3 floats and a string. Each float is parsed in a four byte array and the string is parsed in a byte array, whose size is the same as the number of characters and those array are aggregated to form the serialized object.

Figure 21 shows the results of serializing a collection of 10 simple objects each composed by 3 objects each composed by 2 floats using java standard method and using the manual method. Each test was performed 1000 times.

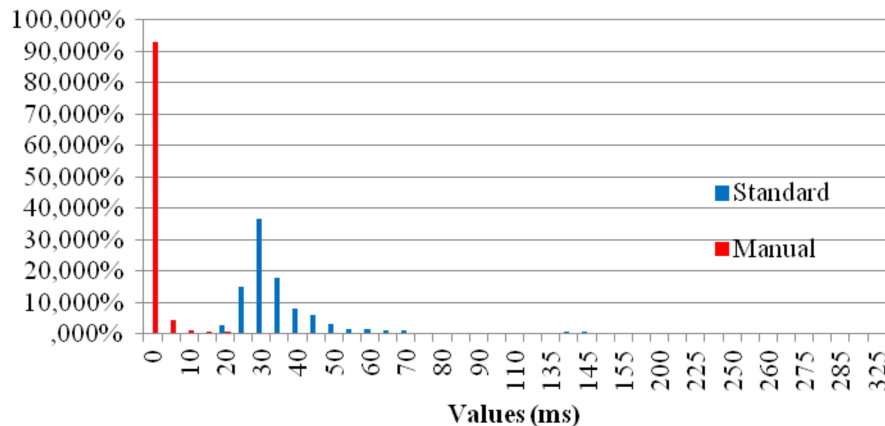

Figure 21. Serialization Comparison

The results show that in average the standard serialization required an average of 43.44 ms with a confidence interval of 2.2 ms for 95% of the samples while the manual serialization required an average of 2.28 ms with a confidence interval of 0.55 ms for 95% of the samples. Aside from the quicker process it is also possible to observe that that serialized object length is 761 bytes for the standard serialization while in the custom serialization, the size of the object is 240 bytes.

Figure 22 shows the results of deserializing the serialized objects using the standard method and using the manual one.

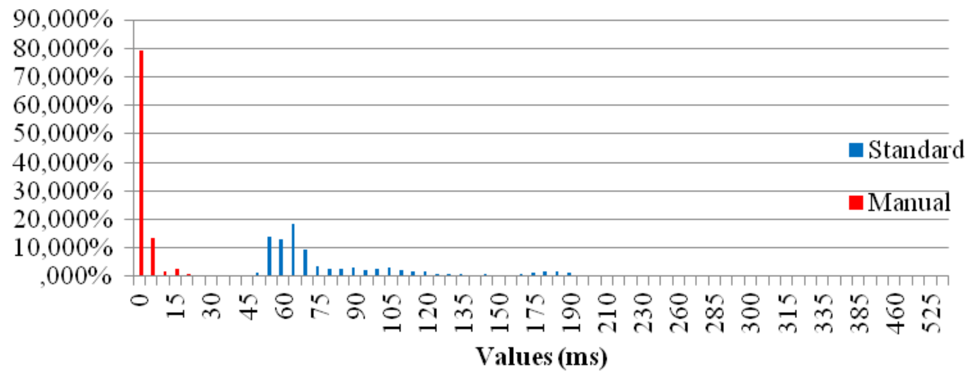

Figure 22. Deserialization Comparison

The results show that the manual deserialization process is faster than the standard one requiring in average 5.41 ms with a confidence interval of 1.04 ms for 95% of the samples while the standard method required in average 93.83 ms with a confidence interval of 3.93 ms for 95% of the samples.

After evaluating both methods is possible to conclude that the manual serialization has the advantage of begin faster and creating small object while having the disadvantage of not being as flexible.

It's also important to know that the performance of the serialization is affected by the device in which is performed.

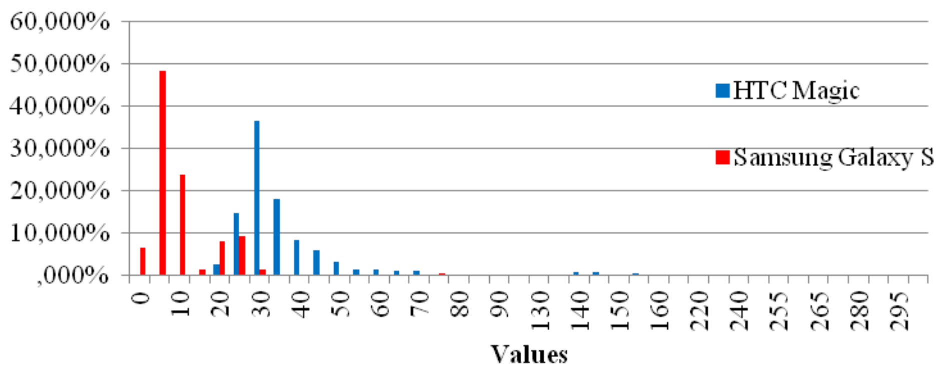

Figure 23. Device Comparison

Figure 16 shows the results of serializing a collection of 10 objects in a HTC Magic and a Samsung Galaxy S, both with their default configurations.

6.7.2 Linear Regression Optimization

It is possible to increase the performance of the linear regression parameters calculation by:

i) If m is negative then calculating b is not necessary since the line will not cross the maximum capacity line in the future;
ii) The summations which are required to be calculated in Eq. (3) and (4) can use previously calculated values. As an example, the calculation of $a = \sum_{i=0}^{n} t_i$, can be done using the following recurring formulation:

$$a_x = a_{x-1} - a_{x-n} + a_x \qquad (7)$$

where, a_0 is the summation of the first *n* values.

6.7.3 General Code Optimization

Aside from the techniques described in chapter 5, other techniques were used in order to increase the performance of applications. Some of those techniques include the techniques denoted in the paper "Mobile Game Development: Object Oriented or Not?" (Zhang, Han, Kunz, & Hansen, 2007) which were used in the development of this application and the offloading library, more specifically:

- Initialize the object when it's first used and not when it's created.
- Prioritize local variables rather than class members.
- Increase the methods access time by declaring it final or static when possible.
- Increase the performance of the garbage collector by set an object to null when it is not going to be used anymore.
- Return null rather than throwing exceptions.
- Reduce resource consumption by reusing exception objects and delegates when possible.

6.8 Summary

This chapter documents the architecture and design of an application that uses the provided offloading framework.

The application uses a game engine, which is responsible for the application life cycle, which includes calling the `update()` method in the service, which is a physics engine.

In this application, the service is the physics engine and the offloadable object is the physics objects.

The next chapter presents the results of the tests performed in the application in order to evaluate the framework.

Chapter 7. Tests and Results

In this chapter we describe the tests performed to the real-time offloading framework. These tests evaluate the overhead introduced by the MobFr and also the timings related to several experiences performed with the framework.

7.1 Benchmark Tests Mobile Framework

The tests described in this section aim to evaluate the timings of the code mobility operations. These include the time to transfer an APK between devices, the time it takes to execute an intent remotely and the time of service migration from a device to another, also known as service rebinding.

All the tests were performed using one or more of the devices presented in Table 2.

Device Name	Android OS Version	Process Speed	RAM	Processor	Quantity
Emulator	2.2	3000 MHz	1024 MB	Pentium 4	1
HTC Magic	1.5	528 MHz	288 MB	Qualcomm MSM7200A	2
Samsung Galaxy S	2.1	1000 MHz	512 MB	Arm Cortex A8	2
Toshiba Folio 100	2.2	1000 MHz	512 MB	Nvidia Tegra 2	1

Table 2. Device Specification

7.1.1 APK Transfer and installation Test

This test measures the time the Package Manager requires to send and install an APK in a remote device .This test is fundamental for the basic functionality of the framework. If the mobile framework requires too much time to transfer and install an APK, then it cannot guarantee the required QoS.

Listing 6 present the source code used to perform this test. The time it takes to transfer and install an APK is calculated by measuring the time before the call of the package manager and after it (lines 3 and 5). The method `transferAPK()` from the `StandardPackageManager` class (line 4) issues the command to transfer and install an APK remotely. Since this method waits for the confirmation from the remote device, the difference of both times should give an approximate of time it toke to install the APK remotely (line 6).

Listing 6 - APK Transfer and Install Source Code

```
1. public long transferAndInstallService(String ServiceName,
   CooperativeDevice dev)
2. {
3.   long timeInit = System.nanoTime();
4.   StandardPackageManager.transferAPK(ServiceName, dev);
5.   long timeFin = System.nanoTime();
6.   return timeFin-timeInit;
7. }
```

Two devices were used in this experiment: a Samsung Galaxy S and a HTC Magic. The Samsung Galaxy S issued the transfer and install order to the HTC magic. The communication involved a wireless network at 54 Mbps and the transferred APK's size is 116 KB. The results of 1000 executions of the test are depicted in figure 21.

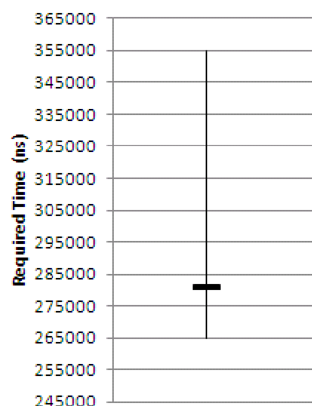

Figure 24. Remote APK Transfer and Install Test Results

The APK transfer from a device to another requires in average 280 ms with a confidence interval of 0.21 ms for 95% of the samples with 198 ms being the shortest time and 398 ms being the longest.

The values shown in the results are higher than the value expected in a framework that installs the service exactly before its execution. For this reason the neighbor device must be initialized prior to the start of code offloading procedure.

7.1.2 Remote Intent Execution Test

As detailed in chapter 3, MobFr is capable of executing an Intent remotely using the Execution Manager module. In this test we have recorded the time elapsed from moment when the Intent is issued on the main device until the confirmation is received.

This test is of fundamental importance to the architecture being proposed because if the execution manager requires too much time, than there is the possibility of the application not being able to guarantee the required real-time performance

Listing 7 presents the source code used in this test. The operation used to execute and activity remotely is the `executeActivity()` from the `StandardExecutionManager` class.

Since that operation waits until the confirmation is received, the difference of the time before its execution and the time after should result in an approximate of the real value.

Listing 7 - Remote Intent Execution Source Code

```
1. public long executeActivity(Intent i, CooperativeDevice
   dev)
2. {
3.   long timeInit = System.nanoTime();
4.   StandardExecutionManager.executeActivity(i, dev);
5.   long timeFin = System.nanoTime();
6.   return timeFin-timeInit;
7. }
```

The test was executed in two devices: a Samsung Galaxy S and a HTC Magic as presented in table 2. The Samsung Galaxy S issues the remote execution command that will be received by the HTC Magic. The communication involved a wireless network at 54 Mbps. Figure 19 shows the results of 1000 experiments.

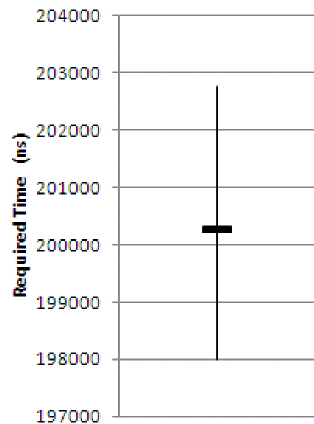

Figure 25. Execution results

The results of the experiment show that the remote intent execution requires in average 200 ms with a confidence interval of 0.39 ms for 95% of the samples. The highest recorded result is 274 ms and the lowest is 20 ms.

The results, as predicted, reveal that executing an Intent remotely requires more time than $t_{xMAxCap}$. For this reason, the neighbour device is initialized as soon as the framework detects that it needs to start offloading the code.

7.2 Benchmark Tests Offloading Library

The offloading tests analyze the behaviour and also the delay of the offloading library during the life cycle of an application. These tests include measuring the core execution time, the execution time in the offloading library requires before the offloading, and others.

Testing the offloading framework is much different than testing the MobFr operations because the experience requires much more time and is easily affected by external factors which alter the behaviour of the application. For example, if the main device is executing another application in

parallel, than is possible that the offloading framework predicts that the application needs start offloading sooner that if no application was being executed simultaneously. The same is true for the network latency. For this reason, the test application is executed once.

The test consisted in generating a physics object every 128000 ns. When the offloading library detects that the device will not be able enough resources to execute the service, part of it is offloaded to a neighbour.

This test was executed in two HTC Magic devices as specified in Table 2. The devices communicated using a wireless network at the speed of 54 Mbps.

Figure 26 details the general times of the experience. These times are explained in detailed in the next sections.

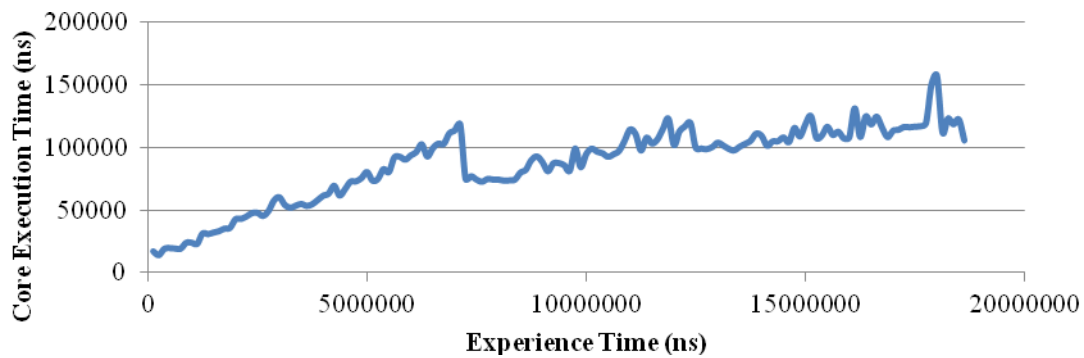

Figure 26. Offloading Library Test Results

As the figure shows at 3.712 ms the offloading decided start preparing the neighbour device and the application starts offloading at 7.808 ms.

7.2.1 Execution

The execution test analyzes the core execution time through the entire experience.

This test is the most important of the experience as it proves that using the specified code offloading architecture, indeed reduces the load from the device in which the application is running to the nodes in the network.

In order to calculate the total of the execution time, the time was measured before and after the invoke of the `update()` method and the difference was recorded. Figure 27 details the results of this test.

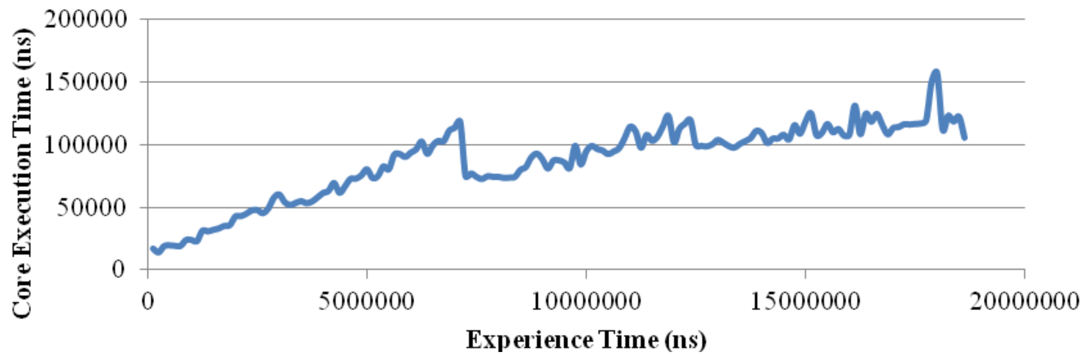

Figure 27. Execution Test Results

The data shows that after starting offloading the execution time decreases significantly. This proves that offloading does benefit the application. It is also possible to observe that after the offload process, the data starts increasing again at a slower rate. It can be attributed to the fact that both devices share the load.

The next section analyses the core execution time in both devices.

7.2.2 Devices Execution

The total execution test consists in measuring the execution time of both all devices during the experience.

This test provides the opportunity to analyze how the computation load is distributed among all the devices in the network. The test corresponds to the step 3 in Figure 26.

In order to have the most accurate measures possible, in all devices, the time before executing the data and the time after were measured and then the difference was interpreted as the time. The main device stores the result locally while the neighbour device transmits it when they transmit the altered objects in step 4 which then the main device also stores in the buffer. Figure 28 shows the results of the experience.

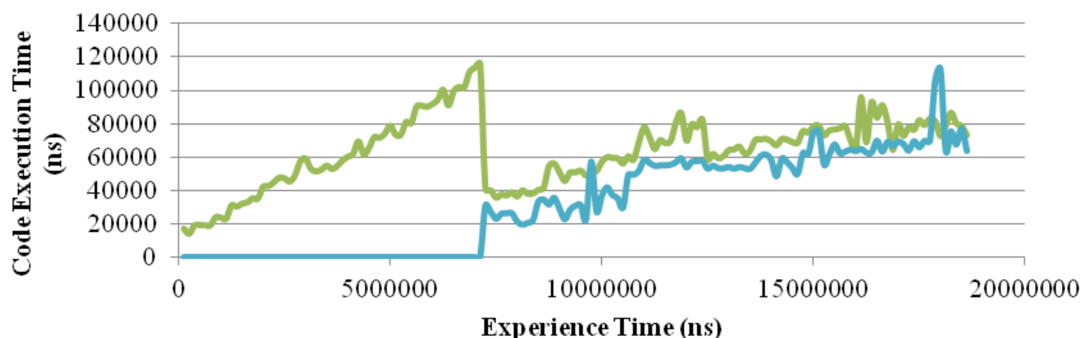

Figure 28. Total Execution Test Results

The second device starts executing during at 7.808 ms and both execution times are somehow similar during the rest of the experience. Both devices have the same characteristics, the

difference discrepancy between the core execution times can be related to the data partitioner algorithm.

7.2.3 Data Transfer

The data transfer test analyzes the time required to transport the offloadable objects between the main device and the neighbour device and their return

This test analyses the approximate delay caused by the network latency when the Offloadable objects are transferred from a device to another.

In order to measure its time, the time is measured before sending the data to the neighbour device and after receiving it. The difference between those times results in a number which is then subtracted the execution time of device and the number obtained is used as the correct result. Figure 29 shows the results of the experience.

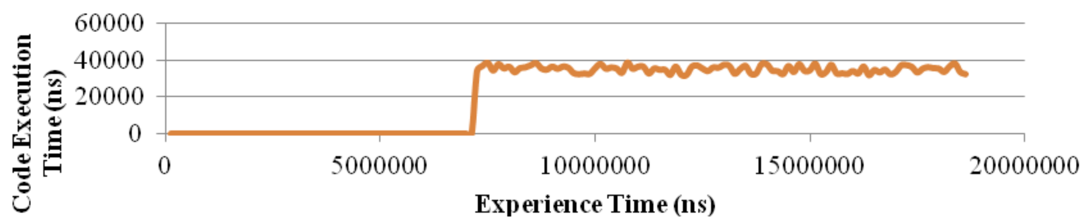

Figure 29. Data transfer Test Results

The network latency is in average 40 ms. As detailed in chapter 6, usually, in a electronic game, the physics engine is executed in every frame. If a game runs at 30 frames per second, that means that the update method has 33.0 ms to executed. Since the network latency is in average 40 ms, it is not possible in the proof-of-concept application, hence the decision to execute the physics engine every 128 ms.

7.2.4 Offloaing Framework Delay

The offloading delay test analyses the delay caused by the offloading library.

This test consists in evaluating the time to execute the three following operations:

- Calculate if the application requires more resources;
- Offload code to the neighbour device;
- Data partitioning between all the devices in the network in case the service is going to be executed remotely;
- The aggregation of the data;
- Registering the core execution times of all the devices.

In order to calculate this time, the time at the beginning of the update method and at the after finishing all the preparations is measured and then subtracted to achieve the end result. It's also important to observe that when the device is not offloading its data, the offloading library does not partitionate the data. The test is better detailed in Figure 31.

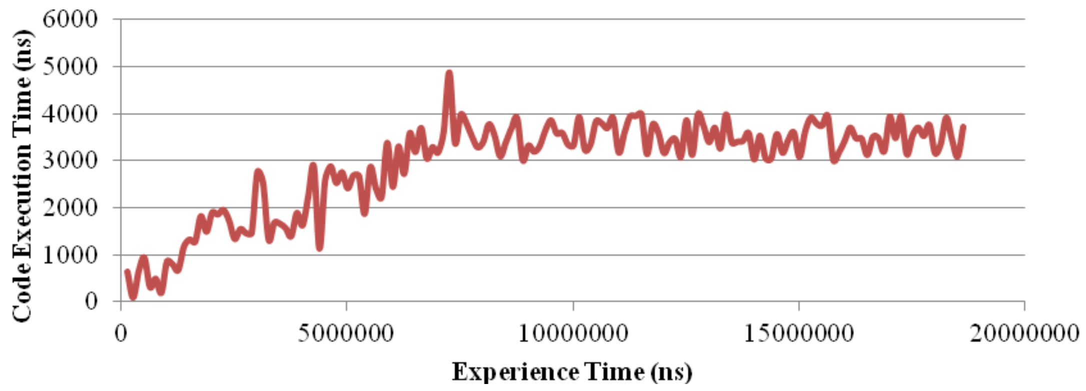

Figure 30. Pre Offload Test Results

The results show that the delay in average is 3.82 ms with a confidence interval of 0.04 ms for 95% of the samples. This information is important during the development of an application that uses the Real Time Offloading Framework because if the service' t_{MaxCap} is small, then this delay may cause consequences in the execution of the service.

Chapter 8. Conclusion and Future Work

This chapter analyses the context and objectives of this thesis and highlights the possible directions of future work.

8.1 Research Context and Objectives

The use of smartphones and other Internet enabled devices is changing the habits of users, which more and more require that their desktop applications are seamlessly supported in these resource-constrained devices. One solution to support these requirements is to offload some of the applications' services to devices nearby, taking advantage of high-capacity local networks. Code offloading techniques have proven to be useful in increasing the performance or the battery life of mobile devices.

The goal of this thesis is to create an offloading mechanism that considers the QoS of the applications, offloading services to neighbour nodes and, at the same time, adapting to changing real-time execution parameters of the application.

The offloading mechanism involves a constant monitoring of the time required to execute a core service, on the main device. Based on past execution times of the core service, the offloading algorithm predicts the future ones. If the algorithm determines that the required execution rate cannot be maintained, then the offloading procedure is triggered in advance, minimizing possible timing errors.

There is no feasibility analysis without an implementation of an application that can demonstrate the main mechanisms being proposed. Therefore, an application is presented in Chapter6 which evaluates the feasibility of the approach.

The evaluation of the algorithms and the framework is elaborated through the test and analysis of the application behaviour.

8.2 Future Work

Although the proposed framework is implemented and analyzed it is possible to propose future work.

The proposed implementation uses linear regression as the statistic method for predicting future core service execution times. This approximation has been chosen since it is one of most simple with low complexity. An alternative solution is to use exponential regression or polynomial regression. It would be interesting to compare different approaches and estimate the precision and computation trade-offs of each one.

The network latency is an important factor in code offloading applications. The proposed system is design in LAN networks. It would be interesting to modify the underlying components to provide full support for WAN topologies and analyze the impact.

The distribution algorithm used in this application uses very specific guidelines when distributing the data. It would be appropriate to design and implement different distribution algorithms that take in consideration the resources of the devices.

No framework can truly be analyzed when tested on a small number of applications. The final proposal for future work consists in testing the framework in different applications with different services and requirements.

Papers and Technical Reports

Papers

Handling Mobility on a QoS-Aware Service-based Framework for Mobile Systems (Gonçalves et al., 2010) - Paper published in IEEE/IFIP International Conference on Embedded and Ubiquitous Computing that details the design and implementation of MobFr, the code mobility framework used in this dissertation. The conference took place in Hong-Kong, China in December 2010.

Service Offloading in Adaptive Real-Time Systems (Ferreira et al., 2011) – Paper published in the 6th IEEE International Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE2011). The paper details the design of the offloading architecture defended in this dissertation.

Offloading QoS-enabled Applications in the Android Platform (Maia, et al., 2011) – Paper submitted as an entry to the RTSS @ Work 2011 competition. The paper served as an introduction a project using on the Real-time Offloading Framework that is currently under development.

TRs

Physics Distribution using Code offloading (Silva, 2011) – Technical report that documents the design of the physics distribution algorithm.

An analysis on Object serialization Methods in Java (Silva & Ferreira, An analysis on Object Serialization Methods in Java, 2010) – This technical report analyses the use of different serialization methods in java.

Bibliography

Ahn, Y., Cheng, A., Back, J., & Fisher, P. (2009). A multiplayer Real-Time Game Protocol Architecture for Reducing Network Latency. *IEEE Transactions on Consumer Electronics* (pp. 1883-1889). IEEE.

ALRahmawy, M., & wellings, A. (2007). A Model for Real Time Mobility Based on the RTSJ. *JTRES'07* (pp. 155-164). Vienna: ACM.

Bai, J., & Leong, B. (2008). Is it Pratical to Offload AI over the Network? *The 16th IEEE International Conference on Network Protocols* (pp. 1-6). Orlando: IEEE.

Ballinger, D., Turner, D., & Concepcion, A. (2011, March). Artificial Intelligence Design in a Multiplayer Online Role Playing Game. *Eight International Conference on Information Technology: New Generations* , pp. 816-821.

Bernier, Y. (2001). Latency Compansation Methods in Client/server In-game Protocol Design and Optimization. *Proceedings of the Game Developers Conference* (pp. 1-13). Orlando, FL: Game Developers Conference.

Bolliger, J., & Gross, T. (1998). A Framework-Based Approach to the Development of Network-Aware Application. *IEEE Transactions and Software Engeering*. IEEE Computer Society.

Botsch, M., & Kobbelt, L. (2003). High-Quality Paoint-Based Rendering on Modern GPUs. *11th Pacific Conference on Computer Graphics and Applications (PG'03)* , pp. 1-9.

Brandt, D. (2009). *Accelerating Online Gaming*. Reykjavik : Reykjavik University.

Cao, J., Zhang, D., McNeill, K., & Nunamaker, J. (2004). An Overview of Network-Aware Applications for Mobile Multimedia Delivery. *37th Hawaii International Conference on System Sciences* (pp. 1-10). Hawai: IEEE Computer Society.

Carzaniga, A., Picco, G., & Vigna, G. (1997). Designing Distributed Applications with Mobile Code Paradigms. *Proceedings of the 19th International Conference on Software Engneering (ICSE 97)* (pp. 22-32). Boston MA, USA: ACM.

Carzaniga, A., Picco, G., & Vigna, G. (2004). Is Code Still Moving Around? Looking Back at a Decade of Code Mobility. *29th International Conference on Software Engineering (ICSE'97 Companion)* (pp. 1-10). Minneapolis: IEEE Computer Society.

Catto, E. (2007, 09 10). *Box2D*. Retrieved 02 02, 2011, from Box2D: http://box2d.org/

Chabukswar, R., & Lake, A. (2005). *Multi-threaded Rendering and Physics Simulation.* Intel, Intel Software Solutions Group. Chicago: Intel.

Chen, G., Kang, B.-T., Kandemir, M., Vijaykrishnan, N., Irwin, M., & Chandramouli, R. (2004). Studying Enery Trade Offs in Offloading Computation/Compilation in Java-Enabled Mobile Devices. *EE Transactions On Parallel And Distributed Systems. 15*, pp. 1-16. Essex: IEEE Computer Aociety.

Chun, B.-G. I. (2011). Clonecloud: Elastic execution between mobile device and cloud. *EuroSys 2011.*

Chun, B.-G., & Maniatis, P. (2009). Augmented Smartphone Applications Through Clone Cloud Execution. *12th Workshop on Hot Topics in Operating Systems (HotOS XII)* (pp. 1-5). Monte Verità, Switzerland: USENIX.

Chun, B.-G., & Maniatis, P. (2010). Dynamically Partitioning Applications between Weak Devices and Clouds. *MCS '10 Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond* (pp. 1-5). New York: ACM.

Chun, B.-G., Ihm, S., Maniatis, P., & Naik, M. (2010). *CloneCloud: Boosting Mobile Device Applications Through Cloud Clone Execution.* Orlando: ARXIV.

Cuervo, E., Balasubramanian, A., & Cho, D.-k. (2010). MAUI: Making Smartphones Last Longer With Code Offload. *MobiSys 10* (pp. 49-62). San Francisco, California: ACM.

Cunningham, W. (2009, January 8). *Ford Sync Version 3.0*. Retrieved February 15, 2011, from Cnet: http://reviews.cnet.com/8301-13746_7-10138116-48.html

Cush, J. (2010, July 18). *Analysts: Tablets to Outsell Netbooks by 2012*. Retrieved February 15, 2011, from TabletPC Review: http://www.tabletpcreview.com/default.asp?newsID=1460&news=apple+ipad+tablet+computer+netbook

Egham. (2010, November 10). *Gartner Says Worldwide Mobile Phone Sales Grew 35 Percent in Third Quarter 2010; Smartphone Sales Increased 96 Percent*. Retrieved February 23, 2011, from Gartner: http://www.gartner.com/it/page.jsp?id=1466313

Eichelkraut. (2002). *An Architecture for Real-Time Mobile Agent systems.* Science Applications International Corporation.

Ferreira, L. L., Silva, G., & Pinho, L. M. (2011). Service Offloading in Adaptive Real-Time Systems. *6th IEEE International Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE)* (pp. 1-6). Toulouse, France: IEEE Conputer Society.

Ferreira, L. (2011). On the use of Code Mobility Mechanisms in Real-Time Systems. 1-5.

Ferreira, L., Silva, G., & Pinho, P. (2011). Service Offloading in Adaptive Real-Time Systems . *International IEEE Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE)* (pp. 1-6). Toulouse, France: IEEE Conputer Society.

Fiedler, G. (2006, 09 02). *Networked Physics*. Retrieved 02 14, 2011, from Gafferon Games: http://gafferongames.com/game-physics/networked-physics/

Flinn, J., Park, S., & Satyanarayann. (2002). Balancing Performance, Energy; and Quality in Pervasive Computing. *22nd International Conference on Distributed Computing Systems (ICDCD'02)* (pp. 1-10). IEEE Computer Society.

Fritsch, T., Ritter, H., & Schiller, J. (2006). CAN mobile gaming be improved? *The 5th Workshop on Network & System Support for Games 2006 - NETGAMES 2006* (pp. 1-4). Singapore: ACM.

Fuggetta, A., Pietro, G., & Vigna, G. (1998). Understanding code Mobility. *IEEE Transactions On Software Engeneering. 24.* IEEE.

Geoffray, N., Thomas, G., & Folliot, B. (2006). Transparent and Dynamic Code Offloading for Java Applications. (pp. 1-17). Paris: Springer-Verlag Berlin Heidelberg.

Glinka, F., Plob, A., Muller-Iden, J., & Gorlatch, S. (2007). RTF: A Real-Time Framework for Developing Scalable Multiplayer Online Games. *NETGAMES 2007* (pp. 81-86). Melbourne: IEEE.

Gonçalves, J., Ferreira, L. P., & Silva, G. (2010). Handling Mobility on a QoS-Aware Service-based FrameWork for Mobile Systems. *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, 2010* (pp. 97-104). Hong Kong, China: IEEE Computer Society.

Gu, X., Nahrstedt, K., Messer, A., Greenberg, I., & Milojicic, D. (2003). Adaptative offloading Interference for Delivering Applications in Pervasive Computing Enviroments. *Proceedings of the first IEEE International Conference on Pervasive Computing and Communications (PerCom'03)* (pp. 1-8). Fort Worth: IEEE Computer Society.

Gui, N., Vincenzo, F., Sun, H., & Blondia, C. (2008). A framework for adaptative real-time OSGi component model. *ARM '08 Proceedings of the 7th workshop on Reflective and adaptive middleware* (pp. 35-40). New York: ACM.

Hassan, Z. (2008). Ubiquitous Computing and Android. *Third International Conference on Digital Information Management, 2008. ICDIM 2008.* (pp. 166-171). London, England: IEEE Computer Society.

He, Z., & Liang, X. (2006). A Point-Based Rendering Approach for Mobile Devices. *16th International Conference on Artificial Reality and Telexistence (ICAT'06)* , pp. 1-5.

Kemp, R., Palmer, N. K., & Bal, H. (2010). Cuckoo: a computation Offloading Framework for Smartphones. *MobiCASE '10: Proceedings of The Second international Conference on Mobile Computing, Applications, and Service* (pp. 182-184). San Diego: MobiCASE.

Khan, S., Khan, S., & Banuri, S. (2009). *Analysis of Dalvik Virtual Machine and Class Path Library.* Technical Report, Institute of Management Sciences , Security Engineering Research Group, Peshawar, Pakistan.

Kim, J., & Jamalipour, H. (2001). Traffic Management and QoS Provisioning in the Future. *IEEE Personal Computation* (pp. 46-55). IEEE Computer Society.

Knutsson, B., Lu, H., Xu, W., Hopkins, & Bryan. (2004). Peer-to-Peer Support for Massively Multiplayer Games. *IEEE INFOCOM 2004* (pp. 1-12). Hong Kong: IEEE.

Kremien, O., & Kramer, O. (1992). Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Systems* (pp. 747-760). Florida: IEEE Computer Society.

Krikelis, A. (1999). Mobile Multumedia Considerations. *IEEE Concorrency* (pp. 85-87). IEEE Computer Society.

Krikellis, A. (2000). Considerations for new generation of mobile communication systems. *IEEE Concorrency* (pp. 80-82). IEEE Conputer Society.

Kristensen, M., & Bouvin, N. (2010). Scheduling and development support in the Scavenger cyber foraging System. *Pervasive and Mobile Computing* (pp. 677-692). Elsevier.

Kumar, K., & Lu, Y.-H. (2010). Cloud Computing for Mobile Users: Cam Offloading Computation Save Energy? *IEEE* (pp. 1-14). West Lafayette: IEEE Computer Society.

Kundu, T., & Paul, K. (2011). Improving Android performance and energy efficiency. *24th Annual Conference on VLSI Design* (pp. 256-261). Chennai: IEEE COmputer Society.

Kurki-Suonio, R. (1994). Real Time: Further Misconception (or Half-Truths). (pp. 71-75). IEEE Computer Society.

Kwok, T. (1992). Wireless Network Requirements of Multimedia Applications. (pp. 1-5). Cupertino, Ca: IEEE Computer Society.

Li, B., & Nahrstedt, K. (1999). A Control-Based Middleware Framework for Quality-of-Service Adaptations. (pp. 1632-1650). IEEE Computer Society.

Linear Regression and Excel. (2000, 04 18). Retrieved 02 10, 2011, from Physics Laboratory: http://phoenix.phys.clemson.edu/tutorials/excel/regression.html

Lonnie, R., Welch, R., Behrooz, P., Shirazi, A., Cavanaugh, D., Fontnot, C., et al. (2000, November). Load balancing for dynamic real-time systems. *Cluster Computing* , pp. 125-138.

Maia, C. (2011). *Cooperative Framwork for Open Real-Time systems.* Porto, Portugal: Instituto Superior de Engenharia do Porto.

Maia, C., Silva, G., Ferreira, L., Pinho, M. N., & Gonçalves, J. (2011, June 23). Offloading QoS-enabled Applications in the Android Platform. pp. 1-6.

Messer, A., Greenberg, I., Bernadat, P., Milojicic, D., Chen, D., Giuli, T., et al. (2002). Towards a Distributed Platform for Resource-Constrained devices. *Proceedings of the 22nd International Conference on Distributed Computing systems (ICDCS'02)* (pp. 1-9). Vienna: IEEE Computer Society.

Mininel, S., Vatta, F., Gaion, S., Ukovich, W., & Fanti, M. (2009). A Customizable Game Engine for Mobile Game-Based Learning. *IEEE International Conference on Systems, man, and Cybernetics* (pp. 2445-2450). San Antonio: IEEE.

Mogul, J. (2003). TCP offload is a Dumb idea whose time has come. *9th Workshop on Hot Topics in Operating Systems (HotOS IX)* (pp. 1-6). Lihue, Hawaii, USA: USENIX.

Monkkonen, V. (2010, 09 06). *Gamasutra*. Retrieved 02 16, 2011, from Multithreaded Game Engine Architectures: http://www.gamasutra.com/view/feature/1830/multithreaded_game_engine_.php

NI Developer Zone. (2010, January 8). *What is a Real-Time Operating System (RTOS)?* Retrieved May 23, 2011, from Nation Instruments: http://zone.ni.com/devzone/cda/tut/p/id/3938#toc0

Nimmagadda, Y. K. (2010, October). Real-time moving object recognition and tracking using computation offloading. *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)* , pp. 2449-2455.

Nimmagadda, Y., Kumar, K., Lu, Y.-H., & Lee, C. S. (2010). Real-time Moving Object Recognition and Tracking Using Computation Offloading. *The 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 2449-2455). Taipei, Taiwan: IEEE Computer Society.

Nogueira, L., & Pinho, L. (2009). *Time-bpimded Distributed QoS-Aware Service Configuration in Heterogeneous Cooperative Environments.* Porto, Portugal: IPP Hurray!

Open Handset Alliance. (n.d.). *Android Overview*. Retrieved September 12, 2011, from Open Handset Alliance: http://www.openhandsetalliance.com/android_overview.html

Ou, S., Yang, K., & Zhang, J. (2007). An effective offloding middleware for pervasive services on mobile devices. *Pervasive and Mobile Computing 3 (2007)* (pp. 1-24). Chicago: Elsevier B.V.

Ouliafito, A., Ricconene, S., & Scarpa, M. (2007). An analytical Comparison of the client-Server, romete evaluation and mobile agents paradigms. (pp. 1-20). IEEE Computer Society.

Paczkowski, J. (2009, Agust 21). *iPhone Owners Would Like to Replace Battery, AT&T*. Retrieved January 11, 2011, from All Things D: http://allthingsd.com/20090821/iphone-owners-would-like-to-replace-battery-att/

Patel, N. (2010, October 29). *Google TV review*. Retrieved February 18, 2011, from Engadget: http://www.engadget.com/2010/10/29/google-tv-review/

Phys 2D. (2008, 10). Retrieved 02 11, 2011, from Phys 2D: http://phys2d.cokeandcode.com/

Pruett, C. (2010, 07). *Replica Island*. Retrieved 09 22, 2010, from Google Code: http://code.google.com/p/replicaisland/

Radwanick, S. (2011). 2010 Mobile Year in Review. (pp. 4-11). comScore.

Rho, S. (2004). *A Distributed HArd Real-Time Java System for High Mobility Components.* PhD Thesis, Texas A&M University, Office of Graduate Studies of Texas A&M University, San Antonio. Texas.

Rich, C., & Claypool, M. (2000). *Basic Game Physics.* Chicago: IMGD.

Schilling, C. (2011, August 23). *From Snake to Tegra: the evolution of mobile phone gaming*. Retrieved September 2, 2011, from Recombu.com: http://recombu.com/news/from-snake-to-tegra-the-evolution-of-mobile-phone-gaming_M14965.html

Shi, Y., Gregg, D., Beatty, A., & ertl, A. (2005). Virtual Machine Showdown: Stack Versus Registers. *VEE'05* (pp. 153-161). Chicago, Illinois, USA: ACM.

Silva, G. (2011). *Physics Distribution using Code offloading.* Porto: Instituto Superior de Engenharia do Porto.

Silva, G., & Ferreira, L. L. (2010). An analysis on Object Serialization Methods in Java., (pp. 1-8). Porto, Portugal.

Soh, J., & Tan, B. (2008). Mobile Gaming. *Communications of the ACM. 51*, pp. 35-39. Association for computing Machinery.

Sommer, S. S. (2010). Service Migration Scenarios for Embedded Networks. *2010 IEEE 24th Intl. Conf. on Advanced Information Networking and Applications Workshops* , pp. 502-507.

Stankovic, J. (1988). A Serious Problem for Nest-Generation Systems. (pp. 10-19). IEEE Computer Society.

Stankovic, J. (1996). Strategic Directions in Real-time and Embeded Systems. *ACM Computing Surveys* (pp. 751-763). ACM.

Vaughan-Nichols, S. (2003). OSs Battle in the Smartphone market. *IEEE procedings on Industry Trends* (pp. 10-12). Los Alamitos: IEEE Computer Society.

Vigna, G. (2004). Mobile Agents: Ten Reasons For Failure. *IEEE International Conference on Mobile Data Management, 2004* (pp. 1-2). Santa Barbara: IEEE.

Wang, A., Sorteberg, E., Jarrett, M., & Hjermas, A. (2008). Issues Related on Mobile Multiplayer Real-time Games over Wireless Networks. *International Symposium on Collaborative Technologies and Systems* (pp. 237-246). Chicago: IEEE and ACM.

Weinsberg, Y., Dolev, D., Anker, T., & Wyckoff, P. (2007). Hydra: A Novel Framework for making High-Performance Computing Offload Capable. *Proceedings 2006 31st IEEE Conference on Local Computer Networks* (pp. 1-9). Tampa, FL: IEEE.

Weinsberg, Y., Dolev, D., Wyckoff, P., & Anker, T. (2007). Accelerating Distributed Computing Applications Using Network Offloading Framework. *IEEE International Parallel and Distributed Processing Symposium* (pp. 1-10). Long Beach, California, USA: IEEE Computer Society.

What is Android? (2009, 10 10). Retrieved 02 11, 2011, from Android Official Website: http://developer.android.com/guide/basics/what-is-android.html

Xian, C., Lu, Y.-H., & Li, Z. (2007). Adaptive Computation Offloading for Energy Conservation On Battery-Powered Systems. *ICPADS '07 Proceedings of the 13th International Conference on Parallel and Distributed Systems. 01*, pp. 1-8. Washington, DC, USA: IEEE Computer Society.

Xin, C. (2009, September). Artificial Intelligence Application in Mobile Phone Serious Game. *First International Workshop on Education Technology and computer Science* , p. 10931095.

Xin, W. (2009). Discussions on Mobile Phone Game Implemented. *ISECS International Colloquium on Computing, Communication, Control and Management* (pp. 514-516). Sanya: IEEE.

Yang, B., & Zhang, Z. (2010). Design and Implementation of High Performance Mobile Game On Embedded Device. *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)* (pp. 196-199). Taiyuan, China: IEEE Computer Society .

Yang, K., Ou, S., & Chen, H.-H. (2008, January). On Effective Offloading Services for Resource-Constrained Mobile Devices Running Heavier Mobile Internet Aplication. *IEEE Communications Magazine* , 56-63.

Zhang, W., Han, D., Kunz, T., & Hansen, K. (2007). Mobile Game Development: Object-Orientation or Not. *31st Annual International Computer Software and Applications Conference (COMPSAC 2207)* (pp. 1-8). IEEE.

Zhang, Y., Yang, J., & Li, W. (2008). Towards Energy-Efficient Code Dissimination in Wireless Sensor Networks. *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.* (pp. 1-5). Pittsburgh: IEEE Computer Society.