POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Enabling Technologies for Android-based Self-Aware Mobile Devices

Relatore: Prof. Marco D. Santambrogio

Tesi di Laurea di:

Alessandra BONETTO

Matricola n. 739372

Anno Accademico 2010–2011

To my family

Contents

1 Introduction **1**

1.1 Introduction to the problem 1

1.2 Research context: autonomic computing systems 3

1.2.1 Definitions and pillars for autonomic computing . . . 3

1.2.2 Realization issues 6

1.3 Related works on adaptive techniques 7

1.3.1 Control Loop models 7

1.3.2 Monitoring and acting 8

1.3.3 Deciding 12

1.4 Related works on complete adaptive systems: the SEEC framework 13

1.5 Target devices and power management 14

1.5.1 Power management: overview and techniques 15

1.6 Summary 18

2 Operating systems for mobile device systems **20**

2.1 OS overview 21

2.1.1 Symbian 21

2.1.2 iPhone and iOS 23

2.1.3 BlackBerry OS 24

2.1.4 Windows Phone 7 24

2.1.5 Android 25
2.2 Market analysis 25
2.3 Power management in mobile OS 26
2.4 The choice of Android 29
2.5 Android OS 29
2.5.1 Android OS Architecture 30
2.5.2 Android kernel vs Linux kernel 32
2.5.3 Dalvik machine 33
2.5.4 Android security 34

3 Proposed Approach 36
3.1 The CHANGE approach 37
3.1.1 Terminology 38
3.1.2 ODA loop 39
3.1.3 Mobile devices: a different approach 41
3.2 Analysis phase: requirements, scenarios and goals 41
3.3 Adaptive System structure 45
3.3.1 Observe 46
3.3.2 Decide 48
3.3.3 Act 49
3.4 Applications 51
3.4.1 Self-adaptive applications 51
3.5 Application monitored by the SC 53
3.6 Services 54
3.7 Summary 56

4 Proposed Implementation 57
4.1 Porting Heartbeats API 57
4.1.1 File based 58
4.1.2 Shared Memory based 59

4.1.3 Binder Interface to implement a shared memory . . . 60
4.2 Native library Implementation 67
4.3 From native library to Java library 67
4.4 Services Coordinator 70
4.5 Frequency scaler actuator implementation 71
4.5.1 Using userspace governor 73
4.5.2 Actuator implementation 74
4.6 Network-type changer actuator implementation 75
4.6.1 Network types 75
4.6.2 Actuator implementation 77

5 Experimental Results 79
5.1 Testing platform 79
5.2 Test system structure 82
5.3 Testing Heartbeat Framework 84
5.4 Case studies: self-adaptive and SC guided applications . . . 88
5.5 Testing self-aware applications 92
5.5.1 Application knobs 92
5.5.2 Implementation changing 95
5.5.3 Reaction times 96
5.6 Testing services 96
5.6.1 FrequencyScaler4Power service 97
5.6.2 FrequencyScaler4Performance service 100
5.6.3 Performance and power tradeoff 101
5.6.4 NetManager4Fluctuation 103
5.6.5 NetManager4Power 103
5.7 Results summarization 106

6 Conclusions 108

List of Figures

1.1 Different usage of heartbeats API. The first scenario represents a self-adaptive application, while in the second one an external observer takes care of monitoring the controlled application. 11
1.2 Areas of power saving technologies 16

2.1 Symbian Architecture Model 22
2.2 iOS Achitecture Model 23
2.3 Q1 2011 global country-level smart phone market data[1]. . 26
2.4 Android Power Management architecture 28
2.5 Android OS architecture 30

3.1 The system model in CHANGE vision 38
3.2 CHANGE ODA loop 40
3.3 System structure: monitors are able to observe specific parameters, the Services Coordinator manages different services that can be based on different actuators or on the same actuator with different policies. Applications are allowed to bypass the SC and self-control themselves. 45

4.1 Communication through the Binder Driver 61
4.2 Binder implementation class diagram 64
4.3 Sequence diagram of a client call 66

5.1 LG Optimus One P500 80
5.2 Test system structure 82
5.3 Comparison between the execution times of heartbeat calls using the file based and the shared memory based implementations 85
5.4 Heartbeat call execution times, both using file and shared memory version, 10 executions 85
5.5 Sequence diagrams of the monitoring process using heartbeats API in MP3 Decoder and PhotoViewer applications . . 91
5.6 Heart rate progress with the introduction of an external disturb, starting at the 100th iteration, without adaptation. . . . 94
5.7 Heart rate progress with the introduction of an external disturb, adaption enabled with buffer resizing. 94
5.8 Heart rate progress with the introduction of an external disturb, adaption enabled with implementation changing. . . . 95
5.9 Heart rate progress of the MP3 decoder application on which the service considered is enabled. Buffer size is 40KB and the final frequency is set to 122MHz 98
5.10 Heart rate progress of the MP3 decoder application on which the service considered is enabled. Buffer size is 160KB and the final frequency is set to 245MHz 99
5.11 Loading time of 10 images of different sizes at different CPU frequencies 100
5.12 Heart rate and buffer size of an application on which are activated both a performance and a power service 102
5.13 Heart rate during the buffering process, using the 2G and the 3G net 104
5.14 Prepare phase, using the 2G and the 3G net 105

6.1 Loading time of a 250KB image at different frequencies. The idle time is computed with respect to the longest execution. 111

List of Tables

3.1 Actuators classes classification 51

4.1 Performance data of different network types 76

4.2 Power consumption of different tasks using different networks 77

5.1 Available frequencies 81

5.2 Actuators available in each service and its target. 84

5.3 JNI overhead over an heartbeat_init function 87

5.4 Impact of the Heartbeats framework over an application execution 87

5.5 Heartbeats rate with different buffer sizes, miniMP3 93

5.6 Reaction times using different actuators 97

Listings

4.1 Heartbeat record type 68
4.2 Heartbeat Initialization 69
4.3 AppHeartBeatInterface 70

List of abbreviations

List of Abbreviations

ACRONYM	EXPANSION
3G	Third-Generation Cell Phone Technology
ADB	Android Debug Bridge
AES	Advanced Encryption Standard
API	Application Programming Interface
ARM	Advance RISC Machine
BAA	Broad Agency Announcement
CDMA	Code Division Multiple Access
CHANGE	Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environment
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency
DES	Data Encryption Standard
DPM	Dynamic Power Management
DRAM	Dynamic Random-Access Memory

ACRONYM	EXPANSION
EDGE	Enhanced Data for GSM Evolution
EKA2	EPOC Kernel Architecture 2
GLA	Generic Log Adapter
GPRS	General packet radio service
GSM	Global System for Mobile Communications
HB	HeartBeat
HSDPA	High-Speed Downlink Packet Access
IBM	International Business Machines
IPC	Inter Process Communication
JIT	Just In Time
JNI	Java Native Interface
JVM	Java Virtual Machine
JVMTI	Java Virtual Machine Tool Interface
JXM	Java Management Extensions
LTA	Log Trace Analyzer
MAPE-K	Monitor Analyze Plan Execute
MIPS	Microprocessor without Interlock Pipeline Stages
NDK	Native Development Kit

ACRONYM	EXPANSION
ODA	Observe-Decide-Act
OOM	Out Of Memory
OS	Operating System
PC	Personal Computer
PDA	Personal Digital Assistant
POSIX	Portable Operating System Interface [for Unix]
QoS	Quality Of Service
RAM	Random Access Memory
RIM	Research In Motion
SC	Services Coordinator
SEEC	SElf-awarE Computational
SEFOS	SElf-aware Factored OS
UI	User Interface
UMTS	Universal Mobile Telecommunications System
USB	Universal Serial Bus
VM	Virtual Machine
WCDMA	Wideband Code Division Multiple Access
WP7	Windows Phone 7

ACRONYM	EXPANSION
XML	eXtensible Markup Language
YAFFS2	Yet Another Flash File System, 2nd Edition

Abstract

Nowadays the complexity of computing systems is skyrocketing. Computing systems have become extremely powerful and devices type can greatly vary from supercomputer to mobile devices, desktops and servers. All of them have different resources and it would be desirable to have the system able to adapt to the mutating conditions, both internal and external. Architecture that makes it possible for a computing system to observe its current status and its current performance, to decide which modifications to undertake, and to act to modify its behavior, to reacting to unpredictable situations are known as self-adaptive or autonomic systems.

These systems have proved to be effective in desktop environment. The objective of this thesis is to prove that such approach can be effective even in a mobile environment. Mobile environment is a constrained and fluctuating environment, resources are limited both in terms of performance (e.g. usually architectures are single-core and CPU frequency is limited) and power (battery's energy is a non-renewable resource). In addition, also the external environment influences the system behavior, e.g. in case of a fluctuating signal strength the power needed to keep a good signal quality can increase. For these reasons, the choice of an adaptive system can help the device to best manage those resources and to react to unpredictable external conditions.

The final aim of this work is to develop a complete adaptive system and run it on a real device, to prove the effectiveness of the adaptive approach

in different scenarios. We will need monitoring systems, both for performance and for power and actuators to change the applications behavior. Applications should be either self-adaptive or controlled by an external observer, that is in charge of deciding what action apply on each application.

The dissertation is organized as follows.

Chapter 1 describes the research context and provides an introduction to the basic concepts involved in autonomic computing. In addition, it reviews related works on both techniques used in adaptive systems and in complete systems.

Chapter 2 gives an overview of the main OSes for mobile devices available in the market. Then, it is presented the OS chosen in the implementation of this work, Android, and the motivations of this choice.

Chapter 3 describes in detail the proposed solution. First it is proposed an analysis of problems and goals of porting an adaptive system on a mobile device. Then, it is identified a minimum set of components and requirements that should be implemented in the adaptive system, in order to prove its effectiveness in various scenarios.

Chapter 4 presents the implementation details of the adaptive system. This chapter is focused on the monitoring (porting heartbeat library) and acting (implement a frequency scaler) phases, that require Android specific implementations.

Chapter 5 proposes many tests to prove the effectiveness of the proposed approach on different scenarios, from self-adaptive applications to controlled applications, and with different targets, whether performance or power. The overhead of the ported heartbeat library is also computed.

Chapter 6 concludes with a final discussion and outlines future works.

Sommario

La complessità dei sistemi informatici è cresciuta esponenzialmente negli ultimi anni, indipendentemente dal tipo di dispositivo considerato, che può variare dai supercomputer ai dispositivi mobili, ai desktop e ai server. In particolar modo, i dispositivi mobili hanno subito una notevole crescita, sia dal punto della vista di potenza di calcolo che di diffusione. Gli attuali dispositivi mobili sono chiamati *smartphones* e uniscono alla capacità di chiamare l'utilizzo di applicazioni come browser, calendari, navigatori GPS, player audio o video. Essendo molto più leggeri e portabili di un PC portatile, possono rappresentare il vero punto di connessione tra l'uomo e l'ambiente che lo circonda dato che, essendo sempre a disposizione in tutti gli spostamenti dell'utente, possono adattarsi ai diversi ambienti per offrire diverse funzioni a seconda delle diverse necessitá.

Per interagire con l'ambiente che lo circonda, in un vicino futuro un dispositivo mobile dovrà essere in grado di osservare il suo comportamento in relazione all'ambiente e di adattarsi alle condizioni per offrire un sempre migliore servizio all'utente. Questa caratteristiche sono note come *self-awareness*, *context-awareness* e *self-adaption* e sono principi basi ella disciplina nota come *autonomic computing*.

Il lavoro di questa tesi si è concentrato nella progettazione e realizzazione delle tecnologie necessarie per abilitare le citate caratteristiche negli attuali dispositivi mobili, per provare che un approccio di questo tipo è utilizzabile con successo anche in questo tipo di dispositivi. Questo lavoro si

inserisce nel progetto di ricerca Computing In Heterogeneous Autonomous aNd Goal-oriented Environments (CHANGE) del Laboratorio di Micro-Architetture del Politecnico di Milano, che si prefigge la realizzazione di un completo sistema operativo adattativo, capace di operare in diversi ambienti, che spaziano dai dispositivi mobili, al sistemi desktop, fino al cloud computing. In questo progetto, questo lavoro di tesi si occuperá di iniziare la transizione dall'ambiente desktop all'ambiente mobile.

Il sistema proposto si propone di estendere gli attuali dispositivi mobili per integrare capacità self-aware, integrando il ciclo Observe-Decide-Act (ODA) all'interno del sistema operativo. Questo modello fornisce al sistema la capacità di osservare sé stesso e l'ambiente che lo circonda, e la capacità di reagire a cambiamenti che possono accadere, internamente o esternamente ad esso, nel migliore dei modi. L'azione migliore dipenderà dagli obiettivi del sistema e dal risultato della fase di osservazione, effettuata tramite diversi sensori. Le azioni vengono concretizzate tramite attuatori, nel nostro sistema utilizzati attraverso dei componenti chiamati *servizi*, a cui è riservato il compito di utilizzare gli attuatori disponibili secondo le politiche decisionali specificate.

Il sistema operativo per dispositivi mobili utilizzato è Android, scelto per la sua ampia diffusione e la possibilità di modificarne i suoi sorgenti. Per provare il reale funzionamento di questa metodologia, il sistema è stato testato su un dispositivo reale basato su questo sistema operativo. I test effettuati mirano a provare l'abilità del sistema nel reagire a diversi eventi inaspettati, sia interni al dispositivo che esterni ad esso.

La tesi è organizzata come segue. Nel **Capitolo 1** viene descritto il contesto di ricerca e vengono illustrati i concetti fondamentali che compongono i sistemi adattativi. Inoltre, sono descritti alcuni lavori relativi allo Stato dell'Arte sia riguardanti singole tecniche utilizzate per implementare sistemi self-aware, sia sistemi completi.

Nel **Capitolo 2** viene proposta una panoramica dei principali Sistemi Operativi per dispositivi mobili che sono presenti ad oggi sul mercato. Questa sezione porterà alla scelta di Android come sistema operativo utilizzato nell'implementazione di questo lavoro di tesi, verranno illustrate le motivazioni di questa scelta e alcuni dettagli relativo a questo sistema operativo.

Il **Capitolo 3** descrive nel dettaglio la soluzione proposta per lo sviluppo di un dispositivo mobile self-aware. Partendo da un'analisi degli obiettivi, è stata identificata una struttura di base del sistema da implementare, costruita sul modello dell'ODA loop e composta da tutti i componenti necessari per provare la validità della soluzione proposta in diversi scenari d'uso.

Nel **Capitolo 4** vengono illustrati alcuni dettagli implementativi relativi allo sviluppo su una piattaforma mobile, riguardanti i componenti che sono stati implementati. Particolare attenzione è stata posta all'implementazione del sistema di performance monitoring basato su Heartbeats e all'implementazione degli attuatori utilizzati nel sistema.

Il sistema di test utilizzato per validare la soluzione proposta viene descritto nel **Capitolo 5**. In questa sezione vengono prima illustrate le applicazioni sviluppate come supporto alla validazione del sistema, successivamente viene proposta una serie di test effettuata per testare la reazione del sistema in diverse situazioni. I risultati provano la correttezza dell'implementazione e l'abilità del sistema nel reagire alle diverse situazioni per ottimizzare diversi obiettivi. Sono riportati inoltre i dati di overhead del sistema di monitoring.

Infine, il **Capitolo 6** conclude la tesi con alcune considerazioni relative al lavoro svolto e a possibili sviluppi futuri di esso.

Chapter 1

Introduction

This Chapter provides an introduction to Adaptive system, the context this work is based on, and the basic concepts used in it. Then, related works on adaptive techniques and on complete adaptive system are presented. Finally, the mobile environment and its problems are introduced.

1.1 Introduction to the problem

Computing systems are rapidly evolving and nowadays devices have become extremely powerful and heterogeneous. Additionally, new device categories have come into being in last years. From supercomputers to mobile devices, through desktops and servers, each kind of device has its own features and resources, and those have to be managed in the best possible way.

In order to make the best possible usage of the available resources, the system first of all must be *aware*. It must be aware of the type of device it is running on and its global condition, past and present. It must be aware of the available resources and how the environment can influence the system. An aware system is able to manage different devices and different situations without the need to change how the system has been designed.

Aware systems are the base of adaptive computing. The stimulus for an improvement or optimization should not only be driven by an external entity but might be advocated by the system itself. Adaptive computing can react to unpredictable changes in the environment (e.g. an external disturb) and react to changes inside the system (e.g. a resource is no more available).

Mobile devices are a constrained and fluctuating environment. Resources are limited, both in terms of performance and power, and their availability can greatly vary over time. The battery energy is an example of a resource that influences the whole system functioning and it has to be carefully managed. Nowadays systems do not provide control or feedback on energy consumption. This limited control and visibility of energy is especially problematic for mobile phones, where energy and power define system lifetime. As mobile devices have evolved from low-function devices to complex systems with applications from different sources, the need of a system able to manage and control energy as a resource has become extremely important.

There is a strong need to make future Operating System (OS) aware of the situation. The importance of different resources should be decided depending on the device type. If e.g. for a smart phone battery energy is a fundamental resource, it is less important for a device always plugged in. The situation in the middle is a device that can run both using battery power and using powerline. In that case a good system should be able to understand the current situation and decide if it is necessary to save anyway energy for future use or if it is best, given the situation, to prioritize another resource over power.

1.2 Research context: autonomic computing systems

This work finds its natural position into the *autonomic computing* or *self-aware adaptive* field, which is an information technology field strictly related to autonomous systems. An autonomic system is defined as a system able to operate and manage itself even in case of environmental changes, without any external intervention. It can involve hardware (e.g. CPUs, Field Programmable Logic Devices), software or both.

Some definitions and requirements of autonomic systems are now presented.

1.2.1 Definitions and pillars for autonomic computing

Among several existing definitions for self-adaptive software, one of the firsts was provided in the *DARPA BAA* in December 1997 [2]:

Self Adaptive Software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.

This definition has strong background in control theory. It implies that the self-adaptive software should have a *sensor-evaluate-adjust* executing loop, just like the feedback control. Furthermore, the adaptation model could have some adjustable parameters and a mechanism to adjust them.

A similar definition has been given later in 1999 by Oreizy et Al. [3]: *Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation*

The first autonomic computing characterization was formulated only in 2001 by IBM with its *Autonomic Computing Manifesto* [4], presented by Dr. Paul Horn's during the National Academy of Engineering meeting of the

same year. From that presentation eight self-* properties that an autonomic system should have were outlined (Section 1.2.1). In 2009 Salehie et Al. proposed an hierarchical division of the eight self-* properties and a set of six questions to elicit adaptation requirements (Section 1.2.1).

IBM Pillars

Horn explained that the inspiration for automating the behavior of a complex system, hardware or software, was found in one of the most complex system available, that is the human body. A human body embeds the autonomic nervous system, that is an obtrusive system that manages many vital functions across a wide range of external conditions, e.g. it monitors the body temperature and it adjusts the blood flow and the skin functions to keep them into correct values. That is the example kept in mind when he defined eight general properties and attributes a system should have to constitute to be able to manage itself:

1. **Self-Awareness** To be autonomic, a computing system needs to *know itself*, it needs detailed knowledge of its components, current status, and all connections with other systems to govern itself.

2. **Self-Reconfiguration** An autonomic computing system must configure and reconfigure itself under varying and unpredictable conditions.

3. **Self-Optimization** An autonomic computing system never settles in a certain situation, it always looks for ways to optimize its workings.

4. **Self-Healing** An autonomic computing system must be able to recover from routine and extraordinary events that might cause some of its parts to malfunction. More than simply responding to a component failure, or running periodic checks for symptoms, an autonomic

system will need to remain on alert, anticipate threats, and take necessary actions.

5. **Self-protection** An autonomic computing system must be an expert in self-protection, it must detect, identify and protect itself against various types of attacks to maintain overall system security and integrity.

6. **Context-awareness** An autonomic computing system knows its environment and the context surrounding its activity, and acts accordingly. It will need to be able to describe itself and its available resources to other systems, and it will also need to be able to automatically discover other devices in the environment.

7. **Openness** An autonomic computing system cannot exist in a hermetic environment, it must function in a heterogeneous world and implement open standards of system identification, communication and negotiation.

8. **Anticipation and transparency** Perhaps most critical for the user, autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden.

Adaptation Requirements Elicitation

Another way[5] to capture the requirements of self-adaptive software is getting help from the six questions *What, Where, Who, When, Why* and *How*.

- **Where** At which level of granularity and at which layer adaptivity should be applied? Information about attributes of adaptable software, dependency between its components and layers have to be collected to locate the problem.

- **When** When does a change need to be applied, and when is it feasible to do so? Can it be applied anytime? Is it enough to perform adaptation actions as needed or do we need to predict some changes?
- **What** This set of questions identifies what attributes or artifacts of the system can be changed through actions, what alternatives are available for the actions, what events and attributes have to be monitored to follow-up on the changes, and what are the resources needed.
- **Why** These questions are used to motivate the building of a self-adaptive application and to found the objectives or goals addressed by the system.
- **Who** This set of questions addresses the level of automation and human involvement in self-adaptive software.
- **How** Determine how to proceed with the changing process, deciding which actions are suitable for any given condition.

These questions have to be answered in two principal phases: while the system is being developed (*developing phase*) and when it is responding to changes (*operating phase*).

1.2.2 Realization issues

After answering to the previously described questions, the next step should be deciding how to develop the system and apply adaption. There are different approaches for incorporate adaptivity into a system:

- **Static/Dynamic decision-making**, how the decision process can be constructed and modified:
  - *static*: the deciding process is hard-coded (e.g. as a decision tree) and its modification requires recompiling and redeploying the system or some of its components;

- *dynamic*: policies or rules are externally defined and managed, and can be changed at runtime;

- **Internal/External adaption**, decide the separation between the adaptation mechanism and the application logic:
  - *internal*: adaption logic is integrated into the application, generally using programming logic features, such as conditional expressions;
  - *external*: an adaption engine external to the application contains the adaptation process;

- **Making/Achieving adaption**, deciding whether self-adaptivity is introduced into the system at the developing phase or it is achieved through adaptive learning.

1.3 Related works on adaptive techniques

As introduced in Section 1.2, all autonomic systems have in common a feedback control implemented as a *sensor-evaluate-adjust* executing loop. In literature, several control loop models and many mechanisms to implement the single phases of the control loop have been proposed.

1.3.1 Control Loop models

The adaption control loop is a component of every autonomous system and it is a notion taken from control theory. A control loop is open (*non-feedback loop*) if the controller uses only the current state and its model of the system to make its decisions or closed (also called *feedback-loop*) if there is also a feedback system that helps the decision phase. In self-adaptive software a closed-loop approach is used, meaning that the software monitor

itself and the surrounding environment while executing. A feedback loop for a self-aware adaptive system is generally described by four processes:

- *monitoring*: it is the phase in which data from sensors are collected and converted into symptoms;
- *detecting*: here symptoms are analyzed to decide whether and when a change is required;
- *deciding*: in this phase the questions about what needs to be changed and how a change has to be performed are addressed;
- *acting*: this process is responsible for applying the actions decided in the previous phase through actuators.

One of the first descriptions of an adaption loop and its phases is presented in [6]. The so called *MAPE-K loop* stands for Monitoring, Analyzing, Planning and Executing, working on a knowledge base K. In this loop the four phases are the same of the phases of the general model previously presented, just with different names. In addition, there is a knowledge base representing the system information.

Another loop model is the *Observe-Decide-Act (ODA) loop*, where the second phase *Detecting* has been integrated into the *Deciding* phase.

1.3.2 Monitoring and acting

A monitoring activity is everything that observe a system for any change inside the system itself or in its external environment. Depending on the system we are working on and which is our goal, several types of monitoring can be applied:

- **Logging**: viewing an application's logs it is one of the simplest techniques to collect data from it. Those logs have to be processed and mined to obtain useful information. The *Generic Log Adapter (GLA)*[7]

and the *Log Trace Analyzer (LTA)*[8] are examples of tools for this purpose.

- **Profiling**: examples of performance profiling tools are:
  - *Oprofile*[9] collects system-wide profile information without requiring any modification to the compiled executables, since it is based on hardware counters;
  - *dproc*[10] is an extension of the Linux's /proc to monitor information about both local and remote cluster nodes;
  - *Java Virtual Machine Tool Interface (JVMTI)*[11] provides both a way to inspect the state and to control the execution of applications running in the Java virtual machine.
- **Signal monitoring**: Pulse monitoring is a technique adopted from Grid Computing and used in [12], while an heartbeats monitor framework has been proposed in [13] and explained in Section 1.3.2.
- **Management framework**: collections of tools for software management and monitoring. Examples are *Java Management Extensions (JXM)*[14], *IBM Tivoli* [15] and *K42* resource management [16].

Monitoring involves obtaining and analyzing performance information from one or more components of the system. Acting involves using the information retrieved in the monitoring phase, to perform actions on the system. Actions that can be applied to a system are strictly related to the nature of the system itself. Some examples of actions include:

- change data quality or data types[17];
- tune parameters in order to meet a particular goal [18];
- change the implementation of an algorithm [3];

- perform Just In Time (JIT) compiler optimizations [19];
- loop perforation [20].

In next Section it is explained in details the Heartbeats monitoring framework, the pulse monitoring technique that it is used in this work to monitor applications' performances.

Heartbeats

A simple way to observe current and target performance of applications is described in [13] and [21]. The framework proposed, named *Application Heartbeats* (or, more simply, *heartbeats*), provides a simple yet effective infrastructure to measure and monitor application progress toward goals. Crucial points in the design of the framework are simplicity and portability, while keeping a standardized interface.

To monitor an application, programmers have first to define application goals using heartbeats API. Then, the basic function *HB_heartbeat* it is used in the application whenever there is a significant point in which application progress toward a goal wants to be monitored. During the initialization phase, goals can be expressed in terms of a window between a minimum and a maximum threshold, such as the lowest and highest heart rate the application should generate or the minimum and maximum latency between two heartbeats.

To reduce programmers effort while inserting heartbeats into their applications, the standard heartbeats API consist of only few functions and, to ensure portability, only standard function calls are used, that do not rely on OS mechanisms.

An heartbeats monitor is created for each monitored application and its task is to collect generated heartbeats and provide information about the application history and its rate over time.

In Figure 1.1 the two scenarios in which heartbeats can be used are shown.

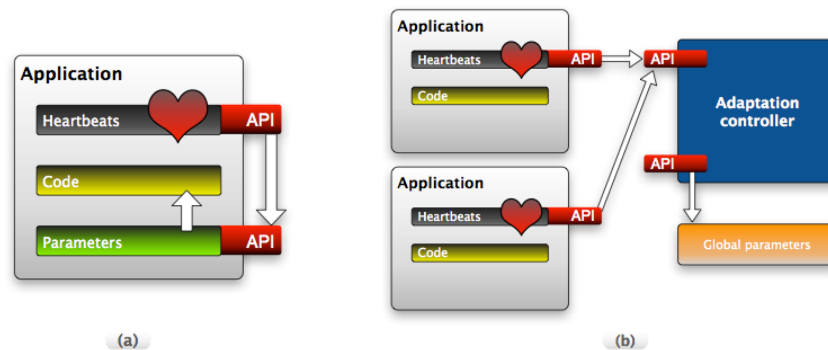


Figure 1.1: Different usage of heartbeats API. The first scenario represents a self-adaptive application, while in the second one an external observer takes care of monitoring the controlled application.

In the first one, the application is self-adaptive and the monitor is embedded into the application itself, which generates heartbeats, monitors its progresses and tunes its parameters, as needed. In the second scenario, the application still generates heartbeats, but this time they are collected by an external observer. Since this time there is a global vision of the system, actions can tune also its parameters.

The heartbeats framework is composed of a set of APIs, which include:

HB initialize Initialized the Heartbeat runtime system and specifies how many heartbeats will be used to calculate the default average heart rate and how many heartbeats to buffer.

HB heartbeat Issue a heartbeat to indicate progress.

HB current rate Returns the average heart rate calculated from the last window heartbeats.

HB get current heartbeat Returns the tag, timestamp, and current heart rate measured the last time a heartbeat was generated.

HB set target rate Called by the application to indicate to an external observer the average heart rate it wants to maintain.

HB get target min rate Called by the application or an external observer to retrieve the minimum target heart rate.

HB get target max rate Called by the application or an external observer to retrieve the maximum target heart rate.

HB set target latency Called by the application to indicate to an external observer the average latency it wants to achieve between two heartbeats with the given tags.

HB get target min latency Called by the application or an external observer to retrieve the minimum target latency.

HB get target max latency Called by the application or an external observer to retrieve the maximum target latency.

HB get history Returns the timestamp, tag, and thread ID of the last n heartbeats.

1.3.3 Deciding

The deciding phase will take care of deciding whether a change has to be performed and, in that case, the set of actions that are suitable given the current situation and the data it receives from the sensors. Many of the approaches used during the deciding phase are taken from Artificial Intelligence. The most common are:

- *Rule based*: this approach is simple and very common[22]. It works well if the system is completely known and rules guarantee a fairly deterministic behavior. They allow actions to be taken in response to an event, usually if a threshold is crossed;

- *Decision tree*: this approach is static and it is generally more hard to extend than the rule based approach, but it is often more efficient.
- *Fuzzy logic*: it is known[23] for working particularly well in the context of conflicting goals and poorly understood optimization spaces.
- *Reinforcement learning*: it is used[24] to implement an agent aiming at maximizing the long-term reward.

1.4 Related works on complete adaptive systems: the SEEC framework

The *Angstrom project* [25] has been recently created to deal with autonomic systems and to meet the challenges of extreme-scale computing.

Project Angstrom's vision to address the four major challenges of extreme-scale computing (energy efficiency, scalability, programmability and dependability) is based on two key foundations: creating a *SElf-awarE Computational* model called SEEC[26], and a *SElf-aware Factored OS* called SEFOS[27].

The *SEFOS* goal is to create a highly scalable operating system for many-cores computing systems while *SElf-awarE Computing (SEEC) framework*[26] is an example of a framework that embeds the ODA loop in both applications and system software. In the SEEC model applications specify their goals, system software specifies a set of possible actions, and the SEEC framework is responsible for deciding how to use the available actions to meet the applications goals. The SEEC framework uses input from applications and systems developers to implement a closed-loop system with three distinct phases: Observation, Decision, and Action and three distinct participants: application developer, system software developer and the SEEC framework itself.

Observation In these phase two entities are involved, the application developer and the SEEC framework. The former has the only task to identify the application goals and its progress toward them and to make those information available to the entire system. The *Application Heartbeats Interface* is used for this purpose. The latter will read the goals and the performance of each heartbeat-enabled application.

Decision This phase is entirely executed by the SEEC framework, with the final purpose of deciding how much to speed up the application. The SEEC control system takes a series of heartbeats observations as input and produces a series of desired speedups which are then used to determine what actions the system should take, to achieve the application's desired heart rate.

Action In this phase, the participants involved are the system developer and the SEEC framework. The system developer needs to indicate a set of possible actions, the speedups associated with these actions, and a function that can take a specified action. On the other hand, the SEEC framework, using those three inputs provided by the system developer, is responsible for mapping speedups into actions and calling the function provided by the system's developer to realize those actions.

Examples of controllers used in the framework are a frequency scaler, a core allocator and a DRAM allocator.

1.5 Target devices and power management

Adaptability is the key aspect in every system that has a large variety of heterogeneous resources to manage, and those resources can vary in type and number and complexity. How to manage them to offer the best ser-

vice depends on which type of system we are considering. Systems can be broadly divided in three categories:

- **Cloud computing** refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those service.
- **Workstations and PC** are high-end computers intended to be used by one person at a time, but nowadays connected to a network and providing a multi-user operating system.
- **Mobile devices** are pocket size computing devices, usually providing a touch screen as input interface.

Especially in mobile devices, but it is true for every energy constrained system, there is an additional resource that have to be taken into consideration, that is energy, and an additional parameter, that is power consumption. For devices that rely on battery power (e.g. mobile devices and laptops), power management has become extremely important to manage at the best the available energy. Since in this work the device used is a mobile device, next Section will give an overview of power management and its most used dynamic techniques.

1.5.1 Power management: overview and techniques

Power management is a feature that turns off the power or switches the system to a low-power state when inactive. Researches in energy saving spans over several areas of interest, Figure 1.2 shows the hierarchy of those areas. The areas below the blue line are related to the power sources: *battery models* and *electrochemistry*. The upper level of the power sources hierarchy, battery modeling, deals mainly with abstracting from the behavior of batteries to predict their discharge times. The lower level of the source

hierarchy is electrochemistry, here new battery families are created, increasing the energy per mass and energy per volume available for mobile systems. The areas above the blue line line are related to the power consumers: *hardware* and *software*. Creating low power software means reducing power consumption by changing algorithms, reducing performance needs, and using lower power instructions. Developing low power hardware means providing new mechanisms for power savings to the higher levels and implementing novel circuit structures or devices. Power management, the explicit scheduling of device accesses and shutdowns to save power, may be implemented in hardware, software, or some combinations of the two. The most important thing is that power management can be applied at runtime.

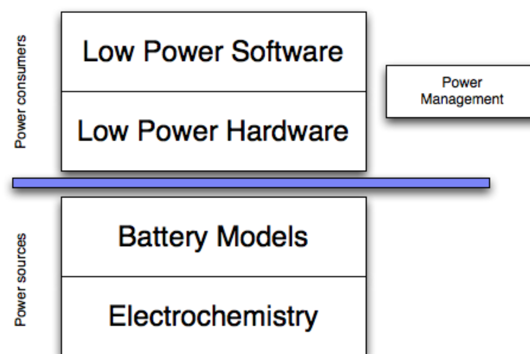


Figure 1.2: Areas of power saving technologies

The main motivation for power management is to reduce the overall energy consumption. This saving has many consequences:

- Extend battery life: an accurate use of the battery energy will make its life last longer.
- Reduce cooling requirements: changing the battery working point can reduce its cooling requirements.
- Reduce noise: reducing cooling requirements will reduce fans speed

and consequently their noise.

- Reduce heat dissipation: heat dissipation depends on the working temperature of the battery, working on this last parameter can reduce also heat dissipation.

Techniques to achieve power management can be applied to several components including, e.g. hard disks, communication devices, display devices and processors. Techniques that operate on processors can be *static* or *dynamic*, the first ones are applied at design time, while the latter ones are applied at run-time. For the scope of this work, we are interested only in *Dynamic Power Management (DPM)* techniques, that can be applied at run-time in order to react to changes

DPM techniques are primarily used to reduce power dissipation. The average power dissipation can be described by the following equation:

$$P_{avg} = P_{dynamic} + P_{short} + P_{leakage} + P_{static} \tag{1.1}$$

$P_{dynamic}$ is the dynamic power consumption, P_{short} is the short-circuit power consumption, $P_{leakage}$ is the leakage consumption while P_{static} is the static power consumption.

The most dominant factor is usually dynamic power consumption, its average is expressed by the following formula:

$$P_{dynamic} = KC_{out}V_{dd}^2 f \tag{1.2}$$

where K is the average number of transitions in a clock cycle, C_{out} is the output capacitance, V_{dd} is the power supply and f is the clock frequency. As the equation illustrates, reducing the operating voltage or frequency, or both, can result in lowering the overall system power consumption, and that is the underlying principle of DPM. DPM identifies low processing requirement periods and reduces operating voltage (*voltage scaling*) and fre-

quency (*frequency scaling*), resulting in reduced average operating power consumption.

It is also possible to act on the output capacitance or the average number of transitions to reduce the power dissipation, but these techniques, since they can be applied only statically, are not investigated in this work.

Dynamic frequency scaling [28, 29] is a technique used to adjust at run-time the clock frequency in order to reduce the power dissipation or the heat. This technique is simple yet effective on energy saving, but as it reduces the number of instructions a processor can issue in a given amount of time, it has as side effect a reduction of performance.

Dynamic voltage scaling [30, 31, 32] is a technique in which the voltage used in a component is increased or decreased depending on needs. If voltage is increased, dynamic scaling is called overvolting, otherwise it is called undervolting. Usually undervolting is performed not only to save power but also to increase stability and reduce temperature. Reducing the voltage in a circuit means reducing the maximum frequency at which that circuit can work. This, in turn, reduces the rate at which program instructions can be issued, which may increase run time for program segments. This is why dynamic voltage scaling is generally done in conjunction with dynamic frequency scaling, at least for CPUs.

1.6 Summary

In this Chapter we have outlined the pillars of autonomic computing system, their characteristics and realization issues. Every adaptive system has integrated a control loop, based on the observing, deciding and acting phases. Different techniques have been proposed, in related works, to deal with each of these phases and Heartbeats API has been described as a

simple and effective way to monitor applications performance toward their goals.

After describing different kinds of devices and their resources, it has emerged that, for energy constrained devices, power consumption is a primary issue. For this reason, we have presented some dynamic techniques related to power management that can be effective in an adaptive system with a power target, in particular those which permit CPU frequency and CPU voltage scaling.

Chapter 2

Operating systems for mobile device systems

A mobile operating system is the operating system that is used on a mobile device. Typical examples of devices running a mobile operating system are PDAs, Smartphones and Tablets. In these Sections are taken into consideration the main operating systems currently available in the market, to see their lacks and necessities and to choose which is the most suitable OS to be used in this work.

In the first Section an overview of each OS and its internal and architectural characteristics is given, when this information is available. Section 2.2 will analyze the current smartphone market situation and Section 2.3 will describe the features introduced by some OSes to deal with power management.

In Section 2.4, the motivations behind the choice of Android as the OS used in this work are explained, while the last Section will describe Android OS in all the relevant details.

2.1 OS overview

With the rapid development of smartphone market, more and more kinds of smartphone operating system (OS) are emerging[33, 34]. There are many different operating systems, but five of them hold the great part of the market: Symbian, Windows Phone 7, RIM Blackberry, Apple iOS and Google Android. Each device or manufacturer has chosen a different OS and they have personalized it with their own custom User Interface. Important differences between them are:

- *license*: Symbian and Android are open source, while iOS, BlackBerry RIM and Windows Phone 7 are proprietary;
- *underlying CPU architecture*: all of them support ARM architecture, while iOS can also run on MIPS, Power Architecture, x86;
- *programming language*: the most used languages are C++ and Java, while iOS is programmed in Objective C;

In the next paragraphs, each one of the most used OSes is presented in details.

2.1.1 Symbian

Symbian[35] is an open source operating system and software platform designed for smart phones and maintained by Nokia. The Symbian platform is the successor of *Symbian OS*, which needed to be integrated with an additional user interface system, while nowadays the Symbian platform includes a user interface component based on *S60 5th Edition*. The latest version, *Symbian 3*[35], was officially released in 2010 and first used in the Nokia N8[36].

In December 2008, Nokia bought Symbian Ltd., the company behind Symbian OS. As a result, Nokia has become the major contributor to Sym-

bian code and on 4 February 2010 the code was published under *Eclipse Public License (EPL)*. Recently, on February 11th 2011, Nokia announced that it would migrate away from Symbian to Windows Phone 7.

The *Symbian System Model*[37] is shown in Figure 2.1 and its structure has remained nearly identical through the different versions of the operating system. The OS is represented as a series of logical layers with the *Application Services* and the *UI framework* layers at the top, a middleware layer with extended services in the middle, and the *Kernel services* and *Hardware Interface* layer at the bottom. Each layer is subdivided in blocks and subblocks by functionality and each block is a collection of components.

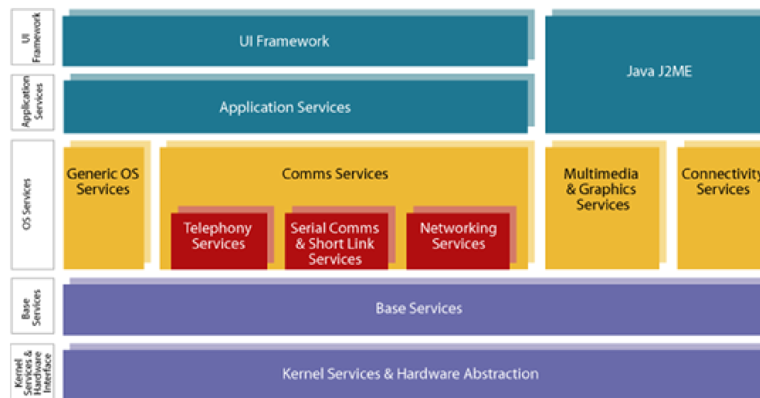


Figure 2.1: Symbian Architecture Model

The Symbian kernel is called *EPOC Kernel Architecture 2 (EKA2)*[38]. Symbian has a microkernel architecture that contains the basic minimum functionalities for maximum robustness, availability and responsiveness. It contains a scheduler, memory management and device drivers. The inclusion of device drivers means the kernel is not a true microkernel. The EKA2 real-time kernel, which has been termed a nanokernel, contains only the most basic primitives and requires an extended kernel to implement any other abstractions. It is single-user, no concept of multiple logins, but it supports preemptively and priority-based multi-tasking.

2.1.2 iPhone and iOS

iPhone operating system (iOS)[39] is a mobile operating system developed and marketed by *Apple Inc*. It is the default operating system for the *iPhone* and the *iPad*. The iPhone OS was derived from Mac OS X and the version history of iPhone OS began in June 2007 with the release of the first iPhone.

The iPhone OS is derived from Mac OS X and is therefore a Unix-like operating system. It is composed of a number of different software layers, each of which provides programming frameworks for the development of applications that run on top of the operating system. These operating system layers are presented in Figure 2.2:

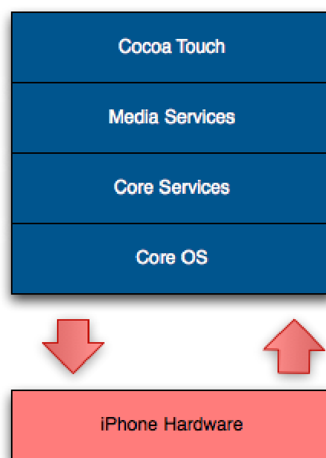


Figure 2.2: iOS Achitecture Model

Top-to-bottom, the layers are:

- The **Cocoa Touch layer** sits at the top of the iPhone OS stack and contains the frameworks that are most commonly used by iPhone application developers. Cocoa Touch is primarily written in Objective-C, is based on the standard Mac OS X Cocoa API and has been extended

and modified to meet the needs of the iPhone.

- The role of the **Media layer** is to provide the iPhone OS with audio, video, animation and graphics capabilities. As with the other layers comprising the iPhone OS stack, the Media layer comprises a number of frameworks that can be utilized when developing iPhone apps, such as Open Audio Library, Media Player framework and OpenGL ES framework.

- The **Core Services layer** provides the fundamental system services that all applications use, such as networking, XML support and SQLite database.

- The **Core OS Layer** is the bottom layer of the iPhone OS stack and sits directly on top of the device hardware. The layer provides a variety of services including low level networking, access to external accessories and the usual fundamental operating system services such as memory management, file system handling and threads management.

2.1.3 BlackBerry OS

BlackBerry OS is a mobile operating system developed by Research In Motion for its BlackBerry line of smartphone handheld devices. Blackberry OS is proprietary and no significant information about its structure is made public.

2.1.4 Windows Phone 7

Windows Phone 7 (WP7)[40] is a mobile operating system developed by Microsoft, and it is the successor to the Windows Mobile platform. There's a minimum set of hardware specifications that all phones must meet in

order to run WP7. They include an ARM7 CPU, a DirectX capable GPU, a camera, and a multi-touch capacitive display.

Windows Phone 7 will employ two different file systems, depending on the fact that it is dealing with a system file or a user file. *IMGFS* is used for system files and *TexFAT*, an extended version of the FAT file system capable of addressing files larger than 4GB, is employed for user files, which can be stored on removable or internal memory.

The shell and application platform reside in user space, while the kernel, drivers, file systems, network, graphics/rendering, and the phone update system run in the kernel space. Since we are talking about a 32 bit operating system, it can only address 4GB of memory, 2GB for processes and 2GB for the kernel.

2.1.5 Android

Android is a mobile operating system initially developed by Android Inc. and then bought by Google in 2005[41]. The Android operating system software stack consists of Java applications running on a Java-based, object-oriented application framework on top of Java core libraries running on a *Dalvik virtual machine* featuring JIT compilation. It is based upon a modified version of the Linux kernel. Section 2.5 will cover in details all the aspect of this OS, since it is the OS chosen during the implementation of this work. This OS was chosen among the others because it is open source, it is currently the most used (as it is shown in next Section) and it is linux based.

2.2 Market analysis

The increasing importance of mobile devices has triggered intense competition among technology giants, like Google, Microsoft, Apple, and Nokia, in order to capture a large portion of the current market. Canalys[1] in May

2011 published its Q1 2011 global country-level smart phone market data, which revealed that Google Android has become the leading platform.

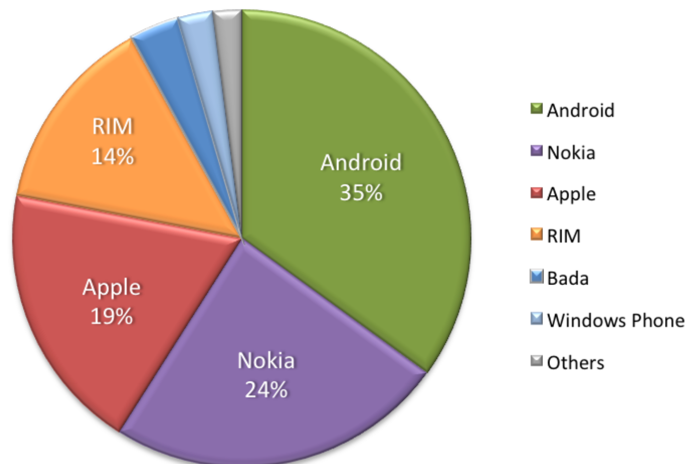


Figure 2.3: Q1 2011 global country-level smart phone market data[1].

Figure 2.3 shows the division of the market between the main OS. As said before, Android is the leader OS with 35%, while the year before its total was just 8.7%, making a huge step ahead. The second OS is Symbian, whose share decreased from 31% to 24%. After the big two, Apple iOS takes the 19% and RIM Blackberry OS the 14%, while Windows phones and Bada phones have the 2.5% and the 3.5% of the market, respectively.

2.3 Power management in mobile OS

This section takes into account how and if power management is taken into account in the previously described mobile OS.

- *iPhone* does not have the power management toolkit as Mac OS does. Instead, the power management system in iOS conserves power by

shutting down any hardware features that are not currently being used.

- *Symbian*, with its EKA2 kernel, introduces a power management framework based on the concept of power domains, where each domain is a set of processes that shares the same power management characteristics. The power manager is embedded into the kernel, which implements the power management executive calls. Device-specific power controllers are implemented as kernel modules to manage the different power states and sleep modes supported by the device.

- *Android* supports its own Power Management (on top of the standard Linux Power Management) designed with the premise that the CPU should not consume power if no applications or services require power. Android allows applications and services to request CPU resources with *wake locks* through the Android application framework and native Linux libraries. A locked wakelock, depending on its type, prevents the system from entering suspend (WAKE_LOCK_SUSPEND) or other low-power states (WAKE_LOCK_IDLE). If there are no active wakelocks, Android will shut down the CPU.

  Figure 2.4 shows the architecture that implements this mechanism inside the Android framework.

  The Android Framework exposes power management to services and applications through the PowerManager class, that allows applications in the upper level to request and release wakelocks. Even user space native libraries should never call into Android Power Management directly, but have to use the PowerManager class. Bypassing the power management policy in the Android runtime will destabilize the system. All calls into Power Management should go through the Android runtime PowerManager APIs.

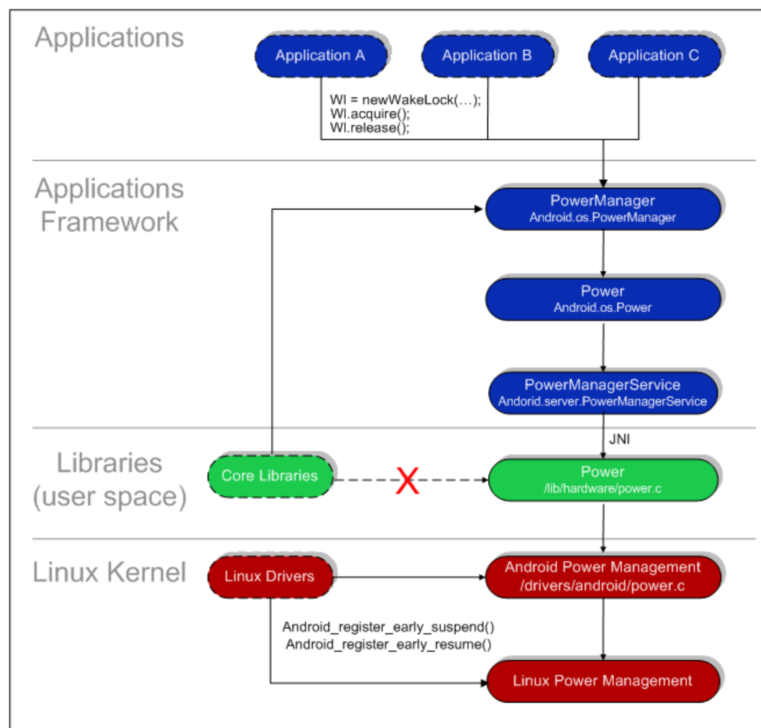


Figure 2.4: Android Power Management architecture

2.4 The choice of Android

The OS for mobile devices chosen for this work is Android, in particular its last version, namely Android 2.3 Gingerbread, is used.

The first reason behind this choice is the fact that Android is an open source project, all its source code it is available via the Apache v2 license through a public accessible repository. The Apache License requires preservation of the copyright notice and disclaimer, but it is not a copyleft license, it allows the usage of the source code for the development of proprietary software as well as free and open source software. This license was chosen to allow manufacturers to innovate using the platform without the requirement to contribute those innovations back to the open source community.

Another reason is that nowadays Android is the most used OS for mobile, with a 33% part of the actual smartphone market (Section 2.2). Additionally, it has a very good community support and this is a great resource since, when this project will be made public, community can contribute to it. The last reason is that it is a linux-based system and both the kernel structure and the programming language (C and Java) are well known.

The only other OS that at the moment is open source is Symbian, but since Nokia is going to abandon this OS in favor of Window 7, Symbian diffusion will considerably decrease due to the loss of its main manufacturer.

2.5 Android OS

Android[42] is a software stack for mobile devices that includes an operating system, middleware and key applications. The first version was Android 1.0, released the 23 September 2008. Current release is *Android 2.3 Gingerbread*, released the 6th December 2010 and based on Linux kernel 2.6.35. On February 22 2011, a tablet-only version was released, *Android 3.0*

Honeycomb. Android has been available under a free software/open source license since 21 October 2008. Google published the entire source code under the *Apache License*, a free software license that allows the use of the source code for the development of both proprietary software and free and open source software. This section will provide many useful details about this operating system and its middleware, the structure of the Dalvik Virtual Machine (VM), the one used in Android, and a comparison between Android kernel and linux base kernel, on which it is based.

2.5.1 Android OS Architecture

The stack architecture of Android OS is shown in Figure 2.5. From top-to-bottom, it is made of four main levels: *Applications*, *Application Framework*, *Libraries* (with *Android Runtime*) and *Linux kernel*.

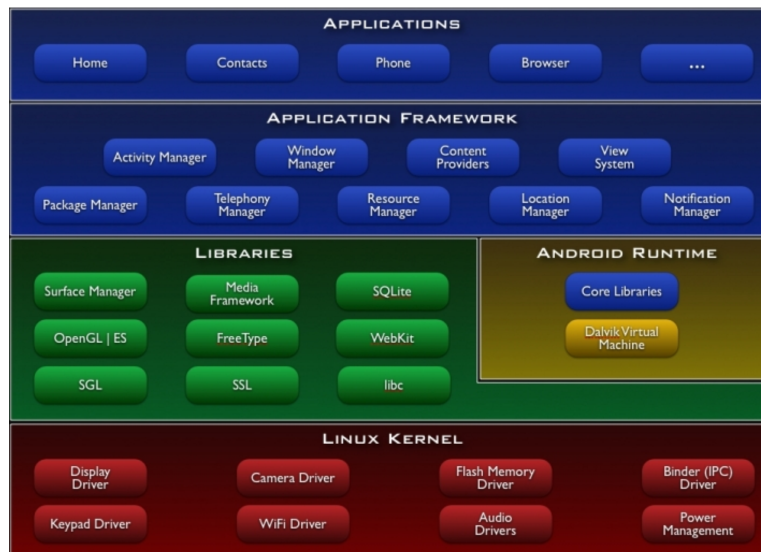


Figure 2.5: Android OS architecture

The **Applications** level contains all the key applications used by a user, like email or SMS client, browser, calendar, maps. All applications are written in Java language and are called Activities. In addition to the core ap-

plications found in a standard Android build, all new user applications are inserted into this level.

The **Application Framework** level is a set of framework APIs used both by core applications and by developers to build new applications. These APIs provide access to the underlying libraries through a set of services and managers such as:

- **views**: a set of components like buttons and grids used to build an application UI;
- **content providers**: a system that allows applications to share data;
- **resource manager**: a system that provides access to extra-resources such as layouts or
graphics;
- **activity manager**: the entity that manages the lifecycle of applications and provides a common navigation back-stack.

The **Libraries** layer contains a set of C and C++ libraries exposed by the upper layer to applications. This level and the kernel level below use C/C++ language, in contradiction to the upper levels that use Java. To make the levels communicate, even if they use different languages, a toolset called *Android NDK* is used, that generates native code libraries from C and C++ sources and wraps those native libraries in *application package files (apk)* using *Java Native Interface (JNI)*. In this layer it is also included the **Android Runtime**, that contains a set of core libraries like data structures, file access, networking and graphics and the implementation of the Dalvik Virtual machine, a VM designed for embedded environments that supports multiple VM processes per device.

The **Linux kernel** layer is based on the Linux kernel with some important differences and enhancements, that are described in details in the next paragraph.

2.5.2 Android kernel vs Linux kernel

Android relies on Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack.

Although Android uses a Linux kernel, there are significant differences between the Android platform stack and the conventional desktop Linux stack. Even starting from the very first version of Android, there are some differences between the two:

- *Yet Another Flash File System, 2nd Edition (YAFFS2)*: Unlike PCs, which store files on disks, mobile phones store files in solid-state flash memory chips. It provides a high-performance interface between the Linux kernel and NAND flash devices. YAFFS2 was already freely available for Linux, however it is not part of the standard 2.6.25 Linux kernel. Starting from Android 2.3, *ext4* filesystem is also supported.
- *Alarm driver*: A driver which provides timers that can wake the device up from sleep.
- *Ashmem*: Standard shared memory is not available in Android, Ashmem has been introduced as an Anonymous or Android SHared MEMory system that adds interfaces to allow processes to share named blocks of memory. The advantage of Ashmem over traditional Linux shared memory is that it provides a means for the kernel to reclaim these shared memory blocks if they are not currently in use.
- *IPC binder*: it is an Inter-Process Communication (IPC) mechanism. It facilitates inter process communication since data can be shared by multiple applications through the use of shared memory. A service registered as an IPC service does not have to worry about different

threads because the binder will handle, monitor and manage them. The binder also takes care of synchronization between processes. It substitutes standard SysV IPC.

- *Power management*: it is built on the top of standard Linux Power Management and introduces the concept of wakelocks, used by applications and services to request CPU resources (see Section 2.3)
- *Low memory killer*: Android adds a low-memory killer that, based on hints from the userspace, can kill off processes to free up memory as necessary. It is designed to provide more flexibility than the Out Of Memory (OOM) killer in the standard kernel.
- *RAM Console and Log Device*: To aid in debugging, Android adds the ability to store kernel log messages to a RAM buffer. Additionally, Android adds a separate logging module so that user processes can read and write user log messages.
- *Android Debug Bridge*: to make debugging easier, Google created the Android Debug Bridge (ADB), which is a protocol that runs over a USB link between a mobile device running Android and a desktop PC.
- *PMem*: the PMem driver is used to manage large physically contiguous regions of memory shared between userspace and kernel drivers.

2.5.3 Dalvik machine

Dalvik[43] is the VM in Google Android operating system, originally written by Dan Bornstein. Before execution, Android applications are converted into the compact Dalvik Executable (.dex) format, which is designed to be suitable for systems that are constrained in terms of memory and processor speed, on an OS without swap space.

Unlike Java VMs, which are stack machines, the Dalvik VM is a register-based architecture. Starting from version 2.2, Dalvik VM in Android has a JIT compiler. JIT compilers represent a hybrid approach, with translation occurring continuously, as with interpreters, but with caching of translated code to minimize performance degradation. It also offers other advantages over statically compiled code at development time, such as handling of late-bound data types and the ability to enforce security guarantees.

Dalvik uses its own bytecode, not Java bytecode. Moreover, Dalvik has been designed so that a device can run multiple instances of the VM efficiently. Multiple java classes are included in a single .dex file and Java bytecode is also converted into an alternate instruction set used by the Dalvik VM. An uncompressed .dex file is typically a few percent smaller in size than a compressed .jar (Java Archive) derived from the same .class files.

2.5.4 Android security

Android is a privilege-separated operating system, in which each application runs with a distinct system identity (Linux user ID and group ID). Linux thereby isolates applications from each other and from the system. In addition, each application runs on a different VM instance, and different instances can communicate only using specific interfaces, like the Binder interface, but the kernel is the only responsible to sandbox each application.

The main point in Android security is that no application, by default, has permission to perform any operations that would adversely impact other applications, the operating system, or the user. Every critical operation, such as reading user's private data or reading another application data has to grant an explicit permission. Application must explicitly share resources and data by declaring the permissions they need to operate, if those operations are out of the sandbox the kernel has applied to each ap-

plication. No permission can be acquired at runtime, instead applications declare the permissions they need statically and those are stored in a file specified for each application.

All android applications have to be signed with a certificate whose private key is held by its developer. This certificate has the only reason to distinguish application authors, it does not need to be signed by a certificate authority.

Practically, Android security policy forces applications that want to communicate to use specific interfaces to implement IPC calls (e.g. the Binder interface) and to expose the permissions they want to obtain in a XML file.

Chapter 3

Proposed Approach

Mobile devices are an example of fluctuating systems, constrained both in computing and power resources. In these particular devices, self-aware systems can help to best manage those resources. The development of self-aware adaptive computing systems is strictly linked with the basic capabilities the ODA loop provides: observation, decision and action. How to implement at the best this feedback loop is the first step in building a self-aware adaptive system.

The first Section of this Chapter shows the vision and the approach proposed by the Computing in Heterogeneous, Autonomous 'N' Goal-oriented Environment (CHANGE) group, in which this work is integrated. One of the main group goal is to build a system able to run on different kinds of computing devices, from server to mobile devices, keeping the same underlying structure.

In the current state of the art of the project, implemented features are build on top of a Linux environment, for desktops and servers. Aim of this thesis work is preparing the enabling technologies to export the CHANGE approach on mobile devices and proving that this approach can be effective even in a mobile environment.

3.1 The CHANGE approach

CHANGE is a research group of the Dipartimento di Elettronica e Informazione (DEI) of the Politecnico di Milano. The research objective of the group is the creation of self-aware devices able to operate in different environments, from mobile devices to desktops and servers. The CHANGE approach takes into consideration the whole system including hardware and software components, divided and grouped in three separate layers. The model for autonomic computing systems, that is extended on all the layers, is the ODA control loop. One of the final aims of the group is to allow application developers to concentrate their efforts only on applications, leaving all the architecture-dependent details to be managed by the autonomic features of the systems where applications will be deployed.

The system model shown in Figure 3.1 explains how CHANGE vision will become a complete adaptive system. Starting from the bottom, the first one is the *Adaptive Hardware Architecture layer*, on top of it there is the *Self-Aware and Adaptive Operating System* which supports a layer composed of a set of *Adapting Applications*.

Each layer has mechanisms to communicate with the adjacent layers and with the external environment.

In the **Adaptive Hardware Architecture** layer we find hardware components divided into categories, e.g cores, memories, devices and reconfigurable devices. Adaptive Applications, on the opposite side, are heartbeat-enabled applications that can monitor themselves or be monitored by the OS. Those applications can rely on a JIT compilation[44].

The architectural and the application layers exchange data with the middle one. The Operating System Layer collects the monitoring information from those two layers, information that is used by an optimization engine based on the ODA loop. The ODA loop is used to optimize resource management through observation and control interfaces that are

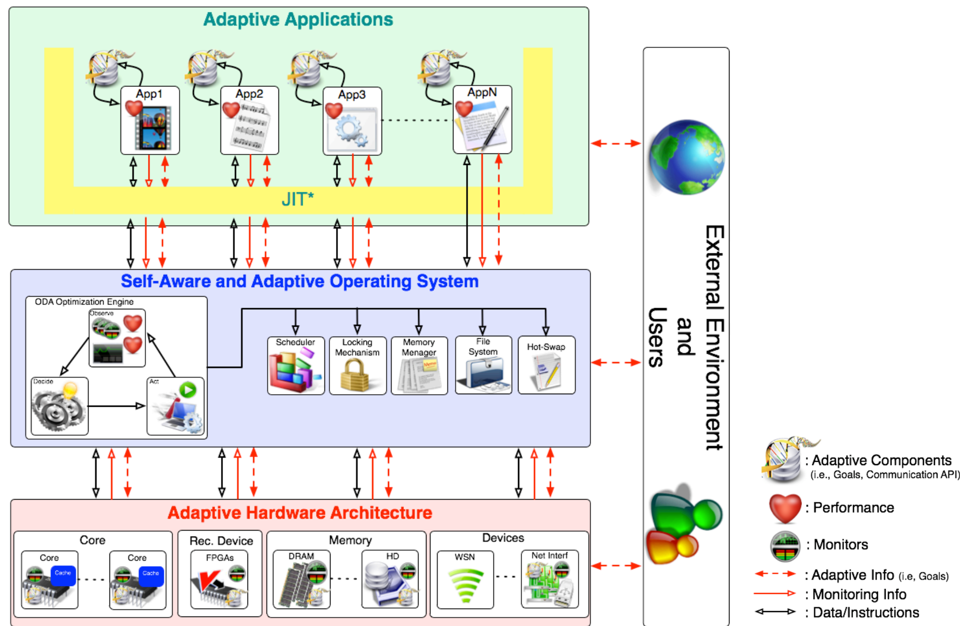


Figure 3.1: The system model in CHANGE vision

added to all applications, to software and hardware components. The OS uses component performance models to decide how to meet a goal given the system global conditions. To reach a goal, the OS can rely on many adaptive mechanisms, e.g. hot-swap[45], adaptive locks[46, 47] and adaptive schedulers[48, 49].

The next section describes the terms and concepts widely used in this Chapter, while Section 3.1.2 describes more in details the adaptive loop integrated in this middle layer.

3.1.1 Terminology

This Section explains the terms that are frequently used in this work, some of them will also be described later in this Chapter into details.

System Device environment, usually composed of the OS and the set of all applications running on it.

Application Software written to accomplish a specific task.

Process Specific instance of an application.

Monitored process (application) Process that is making one or more entities of the system aware of its performance goals and actual progress.

Monitor Entity equipped with sensors able to gather information from the monitored processes.

Actuator Component able to execute an action to modify an application's behavior.

Service Component able to use one or more actuators to perform changes on monitored applications.

Target Objective of a service.

Policy Strategy to be used inside a service to decide how to use an actuator on a target application.

Services Coordinator (SC) Entity that gathers and collects information from monitors, knows the list of monitored applications and available services and decides which service activate on a specific target.

3.1.2 ODA loop

Figure 3.2 illustrates the CHANGE vision on how to implement the ODA loop.

The observe phase collects performance data from all the adaptive applications, that express their intention to be monitored exposing their goals and current performance through a monitor API, e.g. Heartbeats API[13, 21].

Those data are collected by a central element, which has been called Services Coordinator. This entity has a global vision of the system, knowing

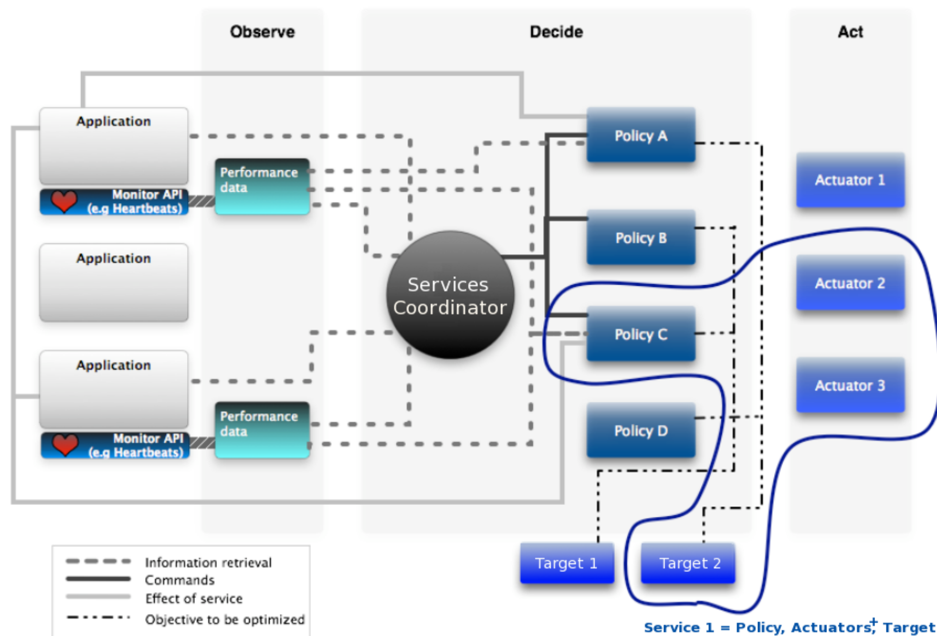


Figure 3.2: CHANGE ODA loop

which applications are currently being monitored, their goals and which policies and actuators are available to meet these goals. Participants in the decide phase are the SC, a set of policies and a set of targets. The act phase is performed by one or more actuators, that are in charge of executing specific actions on the system to vary actual performance.

A combination of a policy, a target and one or more actuators is called a Service. Using this notation, the SC will be aware of all the available services, where hypothetically a service is each combination of components able to vary a system condition and in this way to make a step toward a goal.

Even though in this case it is used a centralized approach, with a single component (the SC) that acts as a decision engine, also a self-adaptive application can be mapped into the proposed approach. In this situation, the SC is bypassed, the application monitors itself and it chooses a suitable actuator; monitor and actuators are integrated in the application itself.

3.1.3 Mobile devices: a different approach

A mobile device is by nature a constrained system: computing resources are usually limited and the life of the entire system is based on the device battery lifetime. Therefore, we consider this scenario as the perfect one for a self-aware approach, since we think that an adaptive system is suitable in every environment where resources are constrained and have to be carefully managed.

The first step necessary to apply the CHANGE approach to mobile devices, is the analysis of the requirements of our mobile system and the identification of its goals, highlighting how they differ from the general case.

The second step will be, given the results obtained from the analysis, to identify a minimum set of elements necessary to prove the effectiveness of this approach, within the ODA loop. Preparing the enabling technologies and validating such approach will be the base for future works in this constrained environment.

3.2 Analysis phase: requirements, scenarios and goals

During the development phase, we were facing the problem to know the characteristics of the external environment to find the requirements our adaptive system should have to respond to our needs.

As we previously said, the environment of our system is *fluctuating* and *constrained* both in terms of computing and power resources. Processing capabilities are reduced, since at the moment most mobile devices are single-core and to meet low power constraints the Central Processing Unit (CPU) maximum frequency is kept low. The energy resource is not renewable by itself and it has to be carefully managed. It is fluctuating, both because internal device conditions can vary over time and because external device conditions can modify our device behavior. The signal strength is a param-

eter that can influence the system performance. In the case of low signal strength or a signal fluctuating between two different nets (e.g. 3G and EDGE), the system will make a huge effort to keep the signal stable, and this causes an unusual power consumption.

These events are usually unpredictable, so what we can do is to use the available resources at their best when we can control them, that includes saving power if there is no need to waste it.

Keeping those needs in mind, we have answered to the 6 questions explained in Section 1.2.1:

Where : the loop will be integrated in the OS, as an extension of it;

When : when a process is under or over performing and whenever there is a a possibility to make a better use of a resource;

What : changes are applied to application specific parameters, or global parameters;

Why : to make a better use of the available resources while keeping a good quality of service in running applications;

Who : applications can manage themselves, if they are self-adaptive, or be managed by an external observer and decision engine, if a centralized approach is used;

How : available actions can improve or decrease performance or resources usage.

In addition, the self-aware system should be lightweight and transparent to the user.

Two scenarios are possible and can coexists in this system. In the first scenario applications are self-monitored and self-adaptive, while in the second one, they are monitored by an external entity (the Services Coordina-

tor) which can enable actuators to perform actions to modify global system conditions. Those two scenarios are not mutually exclusive. The only restriction is that an application is denied to modify global parameters, because it does not have a global vision of the system condition. On the other hand the SC, if necessary, can modify single application parameters. In details:

- a **self-adaptive application** integrates the ODA loop inside the application itself. Such an approach is used when actions are performed to tune application specific parameters, and a global vision of the system is not necessary. In this case, only application specific performance goals can be taken into consideration.

- a **centralized approach** uses an external observer (called SC) to monitor all the applications and activate services on them. With such approach, there is a global vision of the entire system and system wide actions can be applied. An external observer is necessary to monitor and act on power consumption, since changes in power management will reflect changes in all the applications running in the system.

Performance has always been the primary issue for every kind of devices, but since we are dealing with mobile devices, also power consumption is playing a key role. These goals are obviously not isolated the one from the others. Power reduction cannot be blindly performed, just to save as much energy as possible, because this may cause a loss in performance and a discomfort for the user. Performance and power goals have to be taken into consideration at the same time, to reach an adequate tradeoff between the two.

The term **performance** used in the sense of "as-fast-as-possible execution" is not what we want in a mobile environment. Instead a concept of Quality Of Service (QoS) will be used, determined within an heart rate

window between the minimum heart rate and the maximum heart rate. The performance goal will ensure to provide a user an adequate QoS, with respect to the type of applications the user is demanding and the general system conditions. Efforts will be done to ensure a minimum QoS in varying system conditions, not to provide always the best possible performance for each application.

A **power** goal is used to ensure to not use more energy than required. From an energetic point of view, we have divided applications in two main groups:

- *Streaming applications*: those applications require the processor usage for the entire length of their execution. An example is an application that streams a source of data, such as a video stream or an audio stream.
- *One-shot applications*: those applications have to perform an intensive calculation in a specific moment of their execution, while for the remaining time they require few or none processor resources. An example is a PDF reader or an images viewer: once the requested files are loaded from disk, CPU usage is almost zero.

Different services, or different actions belonging to a certain service, are necessary to deal with applications with different needs. Note that these categories are just examples of classes of application needs, used to identify services and tests for this work. Further analysis are necessary to identify more specific categories.

Apart from these two main goals, many other smaller or composed goals can be optimized in a mobile device. Those goals are usually not isolated from the main two goals just described, but they are often parts of them related to a specific problem. An example is the case of network managing, whose target goal can be a mixed goal composed by a power goal

and a stability goal. In this case, the primary goal is to avoid net fluctuation to achieve a better net stability, but the side effect is a reduction of the power dissipation due to the continue change of network.

3.3 Adaptive System structure

Based on the analysis provided in the last Section, some elements are necessary to implement an aware system for a mobile device. The core of the adaptive system is based on the ODA loop (Observe-Decide-Act) and Figure 3.3 represents the situation.

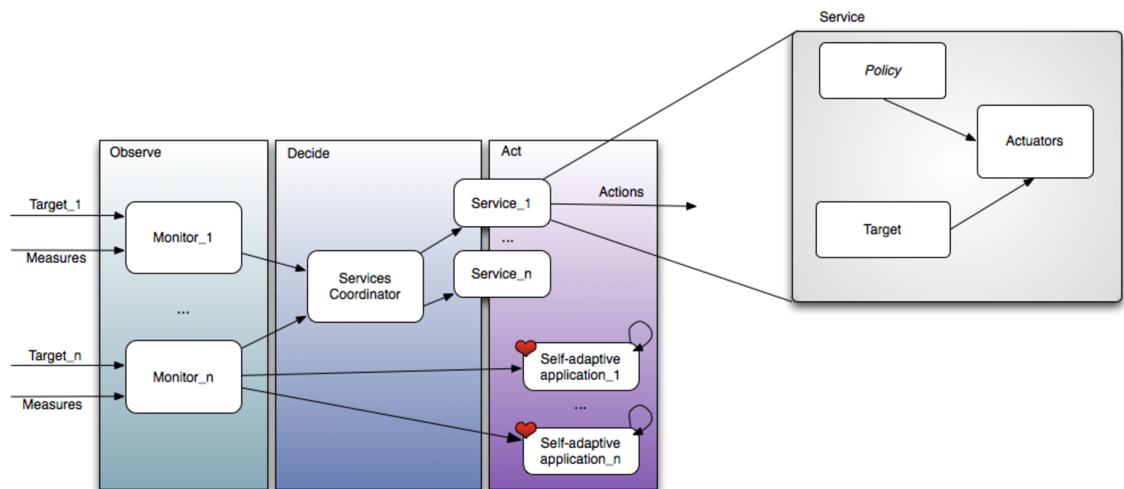


Figure 3.3: System structure: monitors are able to observe specific parameters, the Services Coordinator manages different services that can be based on different actuators or on the same actuator with different policies. Applications are allowed to bypass the SC and self-control themselves.

To prove the effectiveness in different scenarios and situations we may need:

- A set of monitors, both to monitor performance, power consumption or any target that the system wants to optimize;

- A Services Coordinator, to decide which services activate on the applications;
- Self-adaptive applications, that will gather data directly from monitors bypassing the SC;
- Applications controlled by the SC;
- Different services, to respond to different application needs;
- Each service will need a policy and at least one actuator.

These components together are the base of the ODA loop, whose phases are now explained in details.

3.3.1 Observe

Observation is a crucial phase in the loop, and good outcomes relies on the sensors and monitors ability to gather information from the system.

To monitor performance, the proposed solution makes use of the Application Heartbeats API[13, 21]. The principle is the same in both scenarios: each application is responsible for deciding its goals and expressing them in such a way that they can be used by the heartbeat monitor. Then, the application takes care of issuing an heartbeat each time there is a crucial point in the code, in which current performance should be monitored. A monitor is instantiated for each application that have to be monitored. It is integrated inside the application itself, in the case of a self-adaptive application, or inside the SC, if the application is externally controlled.

To monitor power consumption, in this approach there is currently no sensors and no monitors able to gather directly such information. The proposed solution will rely therefore on indirect information, such as the current processor frequency or voltage, or informations relative to the battery discharge provided by Android.

To estimate the variation of the dynamic power consumption caused by actions, it is possible to use a theoretical model and the data received from the sensors available, that include the CPU frequency and voltage.

In Chapter 1 the formula to estimate the CPU dynamic power consumption is is defined as:

$$P_{dynamic} = KC_{out}V_{dd}^2 f \tag{3.1}$$

We know that the output capacitance C_{out} does not vary over time. For our purpose, that is to estimate if a service is able to reduce power consumption, we can assume that the average number of transitions in a clock cycle K is almost constant for different executions of the same process. If we consider two different executions of the same application over the same input data, the difference in power consumption between the two executions will depend only on frequency and voltage, parameters that can be measured by our sensors.

If frequency and voltage values change during the application execution, all the periods of time that have different frequency and voltage values have to be considered separately.

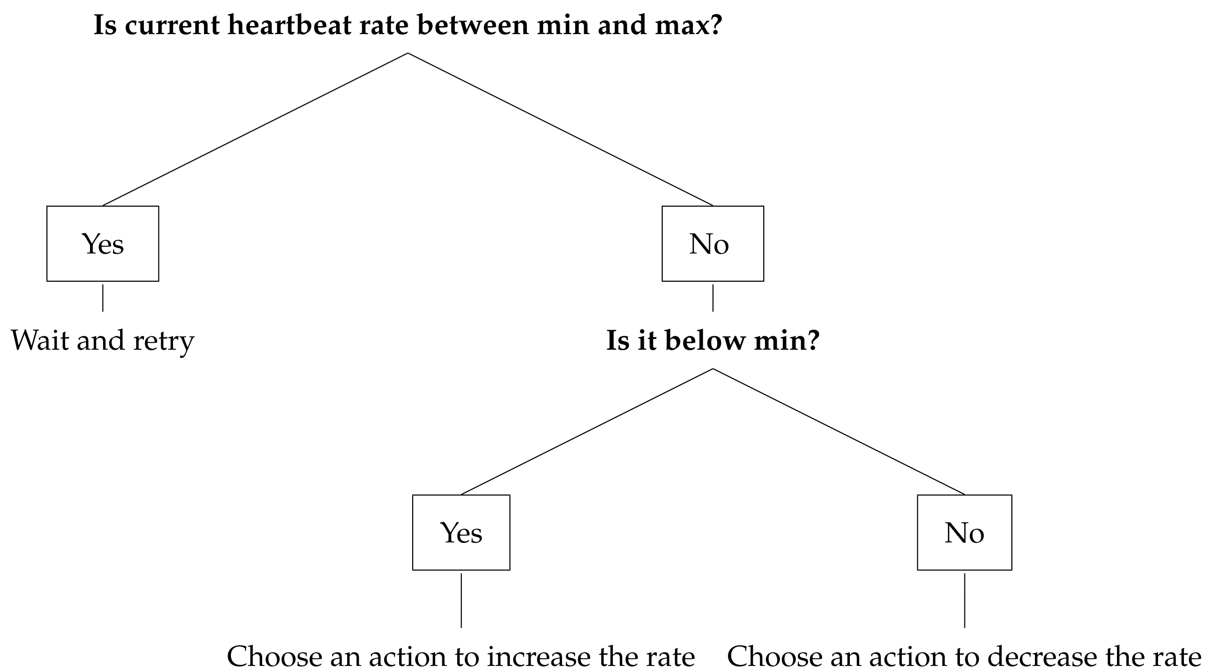
Unfortunately, Android provides little support for knowing the power consumption of the system, and no data are provided about the power consumption of a single application. Apart from information about the CPU current frequency and voltage values, it is possible to retrieve information about the current battery charge status and an estimation of the percentage of battery consumed by certain types of applications during the discharge time. In [50] is shown how both third parts tools, e.g. the Intel's PowerTOP, and the Android built in power meter are demonstrated to be inadequate for accurately measuring the power consumption of Android devices. In [51] it is proposed a precise model and a tool that can be used to have more precise estimations, but still there is no way to know the power consumed by a specific application in a certain amount of time. Future works can in-

vestigate in that directions, since it is fundamental to have a better power monitoring system.

3.3.2 Decide

The decision phase is performed either by the application itself or by an external observer. In both cases, in this phase the data received from the monitoring phase are parsed and analyzed, and a decision of which action to be performed is taken.

In the case of a self-adaptive application, data about the current heartbeat rate are analyzed and the decision follows a simple set of rules. The functioning of the algorithm is based on a static decision tree (shown below). This is not intended to be the best solution, but it is just an enabling technology to prove the effectiveness of the approach.



In the case of a centralized approach, the decision process will use the SC in conjunction with a set of available policies and targets. The first deci-

sion the SC has to perform is on what service to activate. Then, each service provides one or more actions (available through actuators) that can be performed to modify a behavior. It is duty of the policy integrated into the selected service to choose which actuator should be activated on the selected application.

At the moment, there is no way for the engine to know which is the potential of each service (i.e. how much a given service can improve performance) and, more in general, it does not have a description of the service characteristics. The engine only knows if a service is available or not, the list of the actions associated with it and its target.

3.3.3 Act

In the proposed implementation, actuators are integrated into the services and are responsible for performing the action selected by the policy associated with the service. Each service has at least two actions associated with it, one UP action and one DOWN action. Those actions, specific for each service, have the generic purpose of increasing o decreasing a certain parameter. For specific services they can have different meanings, e.g to select the next or previous implementation of an algorithm.

Examples of actuators are:

- **Application knobs**: this group identifies each service able to modify and tune a specific parameter of the application taken into consideration. Examples are: tuning the buffer size in a streaming application, changing the number of rounds in an Advanced Encryption Standard (AES) encryption or its key size.
- **Application implementations**: an application may use different algorithms to complete the same task, usually with different performance in time or quality, depending on the environment conditions. An ex-

ample is whether to use Data Encryption Standard (DES) or AES implementation to perform an encoding procedure.

- **Core allocation**: in a multi-core architecture, assigning a specific core to each process will eliminate time sharing among processes. In addition, if cores are heterogeneous, each process can run on the best core available for its needs, whether they are performance needs or power saving needs.

- **Memory allocation**: if a process requires high memory resources (memory-intensive process), a memory allocator could optimize it to acquire memory easily and efficiently.

- **Niceness adjusting**: modify a process niceness may give this process an higher priority over the other running processes and increase its performance. Even if this enhancement can be done by the application itself, it is a better idea to have anyway a global vision of all the running processes.

- **Frequency scaling**: it is a common way of reducing the power consumption of the overall system, by reducing the processor clock frequency. Among the others, two usages of this actuator are relevant: in the first case, under clocking is used for the entire length of the application, in the second case over clocking is used until an intensive computation finishes, and then the CPU is put in an idle state.

- **Voltage scaling**: this actuator is similar to the previous one, but it acts on voltage rather than frequency. It is hard to implement as a standalone service, but usually voltage is reduce or increased automatically when frequency is changed.

Table 3.1 shows a classification of the previous described actuators depending on whether they act on performance or power, and they are application-

specific or system-wide tuners.

Table 3.1: Actuators classes classification

	Performance	**Power**	**Application specific**	**System wide**
Application knobs	✓		✓	
Application implementation	✓		✓	
Core allocation	✓	✓		✓
Memory allocation	✓	✓		✓
Niceness adjusting	✓		✓	✓
Frequency scaling	✓	✓		✓
Voltage scaling	✓	✓		✓

3.4 Applications

An essential part of the adaptive system is represented by applications. Applications as we saw can be self-adaptive or controlled by the SC. Depending on the application type, only certain actions can be performed and certain actuators can be used. The scope of this Section is to give more details about the actuator types available depending on the application type and about the role of the SC.

3.4.1 Self-adaptive applications

In self-adaptive application, the ODA loop is fully integrated into the application itself. An application, to correctly implement the loop, should both issue and monitor its heartbeats, have at least one service available and a decision system. We decided that a self-adaptive application should

deal only with local actions and modify only internal parameters of the application itself. From Table 3.1 , application knobs and implementation changing are the two actuators classes belonging to this group.

Application knobs are a simple way of changing an application performance while keeping its functionality. Many parameters in an application determine the quality of its results and many of those can be changed at runtime. Taking an example from cryptography, performing an encryption with a different number of rounds or a different length of the key will vary the level of security of the generated encrypted text. But lowering this parameters and still keeping a minimum level of security, that will depend on the application that will use that text, can reduce a lot the time spent performing the operation.

Changing algorithm implementation is another way of changing performance that can be used by applications that perform a task that can be done by different algorithms with different performance. Going back to our cryptographic example, an encoding can be performed using the DES algorithm or the AES algorithm, that provide different security levels and different execution times. Changing algorithm is slightly more difficult than tuning a parameter, because we have to reach a quiescent state before performing the algorithm change.

The mechanism that handles the change of implementation is called hot-swap. Hot-swap can be defined as the dynamic insertion and removal of code in running systems[16]. In this work hot-swap is used only to change between two different software implementations, but other solutions[45] have been proposed to switch between software and hardware algorithms. Many aspects have to be taken into account to ensure that the transition is smooth and guarantees data integrity.

For this to be possible K42[16] researches have identified three essential issues:

- *Quiescent state*: before it is possible to hot-swap, the involved component must be brought into a safe state. The swap can only be done when the component is not currently being used.
- *State transfer*: when it is safe to perform the hot-swap, the system has to decide what state needs to be transferred and how to transfer it to the new component.
- *References swap*: in case the swapped element is referred by its clients (for instance, to allow bidirectional communication), the system has to know how to swap all of the references held by the clients of the component so that the references refer to the new component.

3.5 Application monitored by the SC

In the centralized approach, the SC is the central element that is used an as external observer that collects application monitoring data and decide which sensor to activate. A central element is needed to have a global vision of the system to apply system wide changes. Thus, it needs to be aware of what applications have running processes heartbeat-enabled and what services are available. This is called *discovery phase*.

Then the SC will start monitoring each process, initializing a heartbeat monitor for each of them. This is the *monitoring phase* and it will take care of the way SC and applications can communicate.

For each monitored application, if an action has to be performed, the SC selects a suitable service and executes a service action on the target process. This is called *decision phase*. These steps are continuously performed to detect new processes and to monitor them.

A shared memory area is used to manage all the information flows between the applications and the SC.

Applications will use the shared memory to:

- initialize their intention to be monitored;
- express their goals;
- issue heartbeats.

The SC will use the shared memory to:

- be aware of each process that have to be monitored;
- get the goals and the heart rate of each process.

Each application not only expresses its goals during the heartbeat initialization phase, but it can help the SC to be aware of its needs, using tags associated to each heartbeat. The SC, depending on the service chosen and its policy, can use those hints or ignore them.

3.6 Services

A service has been identified as an entity composed of a Policy, a Target and one or more Actuators. In this section are proposed the structures of different services and their components.

The first set of services is composed by two services that use the same actuator, a frequency scaler. It is a simple yet effective way of influencing the power consumption of a system, since increasing or decreasing the CPU clock frequency will proportionally scale the energy consumed by the system, but this actuator can also be effective in increasing an application performance. Objectives and policies are specific for each service.

Frequency scaling mode varies depending on the type of applications we are dealing with:

- for *streaming* applications, that require constant CPU resources, a good solution is to find the minimum frequency at which every running application with streaming characteristics can keep its minimum QoS.
- for *one-shot* applications, that have to perform mainly only one intensive computational task, a good solution is to over clock the CPU to the maximum frequency possible and keep this frequency for the entire duration of the task. Then, after this operation has completed, the CPU can be set to an idle state, in it which it consumes almost zero energy.

Thinking about those two classes, we have identified two services that use the the frequency scaler actuator with different targets and different policies.

The target of the first service is power. The service tries to minimize the power consumption of the system given the currently running applications. The service policy at first checks the list of frequencies supported by the device. Then, it retrieves the current frequency. Finally, the new frequency value is set to the value previous to the current one in the list.

The target of the second service is performance. Given an application that have to perform a computational task, the policy will make the application compute its task as soon as possible, increasing the frequency at the maximum value available in the device. This policy uses over clocking features, if the device support this capability. When the task has finished, the CPU frequency is scaled down.

While the first set of services is intended to react to and perform internal changes, the second set of services is able to respond to external changes. The first service is able to observe the network fluctuation and, in case of frequent changes between different networks, it will stabilize the connection on a stable net. The target of this service is power/stability, since the frequent change of network will cause an excessive power consumption

and the data connection instability. The second service will ensure that the network type selected is the network that will ensure both a minimum applications performance and the minimum power consumption.

Services that use the same actuator, even if with different targets, cannot be activated at the same time. For this reason the SC has to know which services are available and which ones are not available due to the fact that their actuators are already in use.

3.7 Summary

In this Chapter the proposed approach of the work has been outlined. Following the CHANGE vision of autonomic systems, this work aims to propose the same approach on mobile devices. We have started with an analysis of the goals and the requirements of an adaptive system for mobile devices. The main targets are power and performance and it is important to find an adequate tradeoff between the two. We have identified two possible scenarios for applications. In the first case, applications are self-adaptive and all the phases of the ODA loop are integrated in the applications themselves. Applications that do not have self-adaptive features are controlled by and external entity called SC. The SC can retrieve the information about applications from different monitors. The fundamental monitor is performance monitor, that retrieves data about the applications heart rates. Other monitors can observe e.g. the CPU frequency or the network status. The SC is aware of the list of the monitored applications and of the available services. Several services are currently implemented in the system, each one with different targets and actuators. The SC chooses which service to activate on each application depending on its needs.

Chapter 4

Proposed Implementation

In Chapter 3, we have identified the minimum set of components needed in our system. These components include two monitors, one for performance and one for power, a service manager and two services that use the same actuator, a frequency scaler.

To implement a performance monitor, heartbeats API is used and have to be integrated into the Android framework to be used both from native C applications and from Java applications. Then, to both monitor frequency and implement the frequency scaler actuator, a driver that allows changing the frequency at runtime is needed.

In the next Sections is shown how these requirements have been implemented using Android features.

4.1 Porting Heartbeats API

To implement the monitoring phase, the first step was to port the Heartbeats API to the Android platform. Then, heartbeats API should be made available both to native C applications and to Java applications. In order to accomplish this task, two libraries have been developed, one native shared library written in C and one Java library that uses the native implementa-

tion through the JNI framework.

Heartbeats API is written in C language and is provided in two different versions: one file-based and one shared-memory based. In the initialization phase the memory necessary for maintaining heartbeat states is allocated. In the file-based implementation this memory takes the form of a binary file, while in the shared-memory implementation the initialization function allocates a buffer in the POSIX shared memory using the shmget and shmat functions. Both versions have been ported to the Android platform.

4.1.1 File based

The file-based version of the heartbeat library can be ported to Android without any source code modification, as it uses only compatible functions. These are the steps performed to build the library:

1. For compatibility reasons, C code has to be wrapped into CPP files as it is the standard for Android library source code.

2. The correct namespace has to be added, the standard namespace has been used and it is called *android*.

3. A specific Android makefile has to be created to build the library in the framework.

Android makefiles[52] have a specific syntax and the mk extension, but apart from this, they do not differ from a common unix makefile. The information that have to be provided in the makefile are the module name (that will also be the output name), which are the sources that have to be compiled, the name of the static and shared libraries and a build type for each module. The framework will take care of placing the output of the compilation based on the build type provided. Common build types are: BUILD_EXECUTABLE for a binary file, STATIC_LIBRARY and SHARED_LIBRARY for native libraries, JAVA_LIBRARY for java libraries.

4.1.2 Shared Memory based

The Android kernel does not provide support to POSIX shared memory, so the shared memory version of the heartbeat library as it is cannot be used on the Android platform. The proposed solution to port the shared memory version on Android makes use of ashmem (Android Shared Memory) and the Binder interface.

Android uses its own shared memory implementation, called Ashmem. Ashmem is integrated into the Android kernel, the memory allocated with this implementation is virtual and not physically contiguous. If physically contiguous memory is needed, pmem can be used, but it has no reference counting and is not part of the standard Android kernel. Ashmem has reference counting so that, if many processes use the same area, it will not be removed until all the processes has released it.

First it is created a file descriptor and then this will be used to allocate a memory map through a call to *mmap()* function, as it is illustrated.

```
1 fd = ashmem_create_region("SharedRegionName", size);
2 data = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

The problem is that the memory pointer ("data" in our example) just created is process specific and cannot be shared, and the same occurs to the file descriptor. Additionally, for security reasons, the name assigned to the region in the call to the function *ashmem_create_region()* in the example above is not shared between processes. That means that another process that wants to access the same shared memory area cannot just use a *ashmem_create_region()* with the same name to get access to the same physical memory area.

The solution generally used in Android is to share the file descriptor with the binder interface, since the Binder has special functions that can be used to transfer file descriptors over its interface. Next Section shows how

to use the Binder interface in conjunction to ashmem to allocate a shared memory area.

4.1.3 Binder Interface to implement a shared memory

Binder interface has been used to share a memory area between processes. This shared memory area will be used in the shared memory based version of the Heartbeats library and the same area will be used also by the SC to retrieve the list of processes that have to be monitored.

Binder is a kernel device driver used to achieve efficient, secure IPC. Developers can create their own Binder clients and servers. Servers generally subclass the android Binder class and implement the *onTransact()* method, whereas clients receive a Binder interface as an IBinder reference and call its *transact()* method. Both *transact()* and *onTransact()* use instances of Parcel class to exchange data efficiently.

An implementation of the Binder Interface is used to allocate a shared memory area and to make it available to the Heartbeats API. A client-server architecture is required: applications will act as clients and the service, named HeartBufferService, is the server-side application.

Figure 4.1 shows the client and the server stacks, on the left and on the right respectively, that a generic application and the HeartBufferService must follow to communicate through the Android Binder Driver.

At the highest level, an application that wants to access the shared memory, should get a connection with the HeartBufferService through RPC calls, and then request the memory base address. The memory is physically allocated into the server address space, so the service knows the real memory base address and can communicate it to the client applications.

This high level call is pushed down to the underlying levels, where the objects BpHeartBuffer and BnHeartBuffer, that implement BpInterface and BnInterface, communicate using the *transact()* and *onTransact()* calls. The

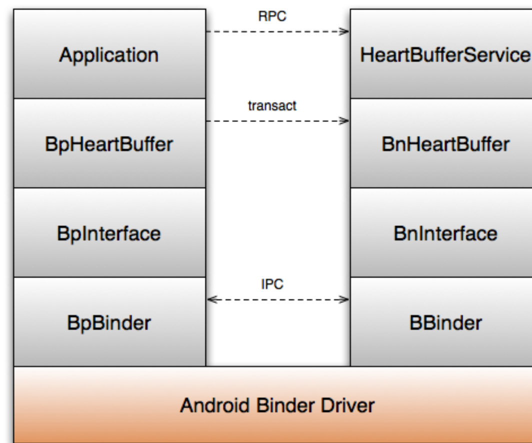


Figure 4.1: Communication through the Binder Driver

lowest levels are BpBinder and BBinder that are allowed to communicate with the Android Binder kernel driver and can rely on IPC calls to exchange the memory address.

Once the client knows the base memory address, it can access the memory area.

This stack structure is implemented into the Binder framework as a set of interfaces. Clients and servers that want to share data using Binder have to implement those interfaces.

The framework has a class named **IInterface** that is the base of all user-defined interfaces. Its principal public method is *asBinder()*, used to retrieve a reference to the IBinder interface.

The **IBinder** interface describes the abstract protocol for interacting with a remote object. The key IBinder API is *transact()* matched by *Binder.onTransact()*. These methods allow to send a call to an IBinder object and receive a call coming in to a Binder object, respectively. The data sent through *transact()* is a **Parcel**, a generic buffer of data that also maintains some metadata about its contents. The metadata is used to manage IBinder object references in

the buffer, so that those references can be maintained as the buffer moves across processes. The Parcel class can be seen as a generic container for a message (data and object references) that can be sent through an IBinder.

BBinder and **BpBinder** are the classes that implement the IBinder interface for the client and the server side, respectively.

Except for these generic classes and interfaces, it is necessary to implement other ones to manage our specific communication through the binder interface. Specifically:

- **IHeartBuffer** it is the interface that implements a generic IInterface, used with BnInterface and BpInterface, client and server interfaces respectively, that have to perform *transact* and *onTransact* operations. The IHeartBuffer interface provide a function called getBuffer, used to retrieve the shared memory buffer address, that is implemented in two different ways on client and server sides.

- **BpHeartBuffer** is the client side class that extends BpInterface through IHeartBuffer interface. It has to implement the *getBuffer()* function on the client side through a transact call, that will give as result a reference to the memory passed through the IHeartBuffer interface.

```
1  sp<IMemoryHeap> getBuffer()
2    {
3      Parcel data, reply;
4      sp<IMemoryHeap> memHeap = NULL;
5      data.writeInterfaceToken(IHeartBuffer::getInterfaceDescriptor());
6      // This will result in a call to the onTransact() method on the
           server
7      remote()->transact(GET_BUFFER, data, &reply);
8      memHeap = interface_cast<IMemoryHeap> (reply.readStrongBinder());
9      return memHeap;
10   }
```

- **BnHeartBuffer** is the server side class that extends BnInterface through IHeartBuffer interface. Its implementation of *getBuffer()* is a simple

return to the available shared memory. More importantly, this class must override the *onTransact()* method, provided in the BnInterface as a pure virtual method. This method in our case has to write a binder to the allocated memory area, to make this binder available on the client side. This action is performed whenever a transact call is performed on client side.

```
CHECK_INTERFACE(IHeartBuffer, data, reply);
     sp<IMemoryHeap> Data = getBuffer();
     if (Data != NULL)
     {
       reply->writeStrongBinder(Data->asBinder());
     }
```

- **HeartBufferService** is the real server class that extends BnHeartBuffer. When the server is firstly initialized, a memory area is allocated as *MemHeapBase*, a library class used to allocate a memory using mmap and ashmem. This class has a specific interface called *IMemoryHeap* that can be transferred through the Binder framework. Additionally on initialization, the service has to register itself to the ServiceManager with a unique name, for its future identification.

Figure 4.2 shows how the presented classes are related.

Classes colored in Orange are defined in binder and are the base classes and interfaces that have to be implemented to make a binder connection. Classes colored in Blue are utilities classes, while classes in Yellow are the classes implemented to share a buffer for Heartbeats API communication.

The **RefBase** class implements basic reference counting facility and the template class *sp* refers to strong pointer implementation. Both RefBase and sp types indicate that binder objects are reference counted.

How can a client use this architecture to retrieve the address of allocated buffer? It can use the function *getBufferMemPointer*, that will execute

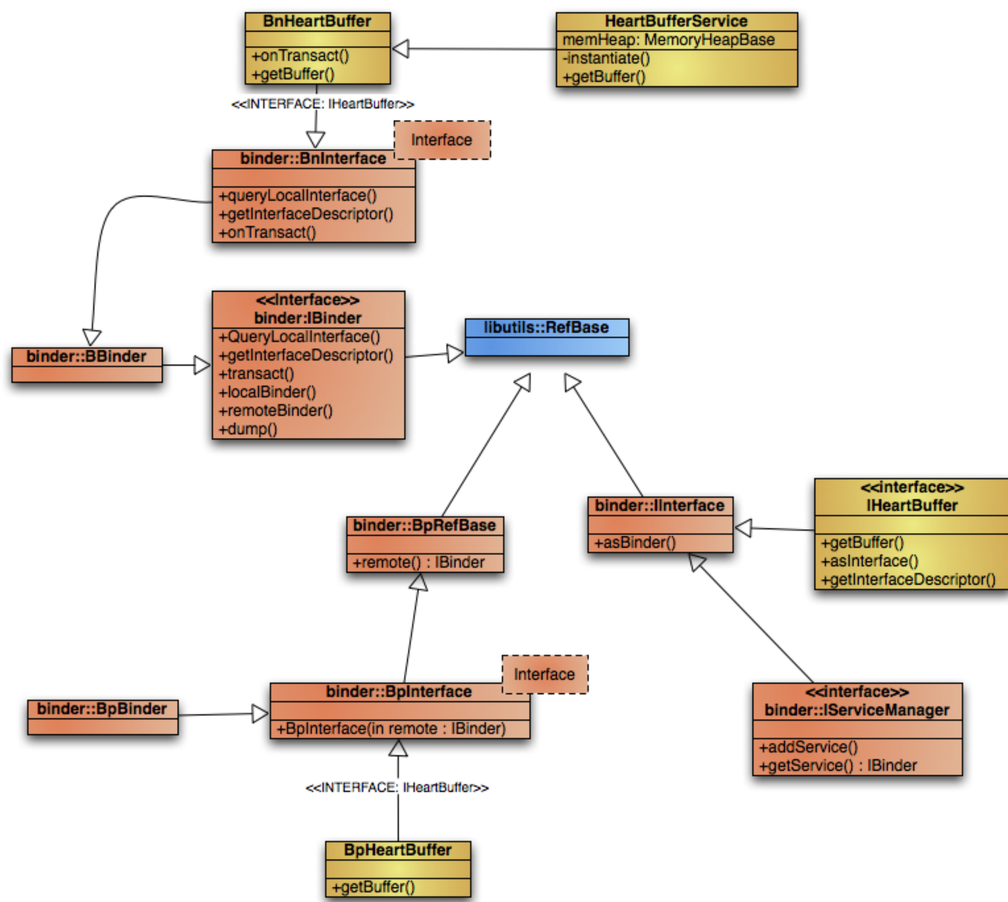


Figure 4.2: Binder implementation class diagram

all the necessary steps. The client gets a service reference from the service manager, it retrieves the binder interface and than calls the transact method using the getBuffer function.

```
1  unsigned int * getBufferMemPointer(void)
2  {
3    static android::sp<android::IHeartBuffer> heartBuffer = 0;
4    if (heartBuffer == NULL)
5    {      android::sp<android::IServiceManager> sm =
        android::defaultServiceManager();
6           android::sp<android::IBinder> binder;
7           binder = sm->getService(android::String16("vendor.heart.Buffer"));
8           if (binder != 0)
9           {     heartBuffer = android::IHeartBuffer::asInterface(binder);    }
10   }
11   if (heartBuffer == NULL)
12   {      LOGE("The HeartBufferServer is not published");
13          return (unsigned int *)-1;
14   }
15   else
16   {
17     receiverMemBase = heartBuffer->getBuffer();
18     return (unsigned int *) receiverMemBase->getBase();
19   }
20 }
```

Figure 4.3 shows the sequence diagram of the initialization phase and of a client call, showing how the different classes interact.

When the memory is allocated by the service, its dimension in bytes has to be chosen. Then, the block of memory is converted to a structure, which is composed of an integer, used to count the registered applications, and a list of heartbeat_state and heartbeat_log for each application. There is no need to store other information about each process in this structure, because some useful information, e.g. the process pid, are already stored in the heartbeat_state of each application.

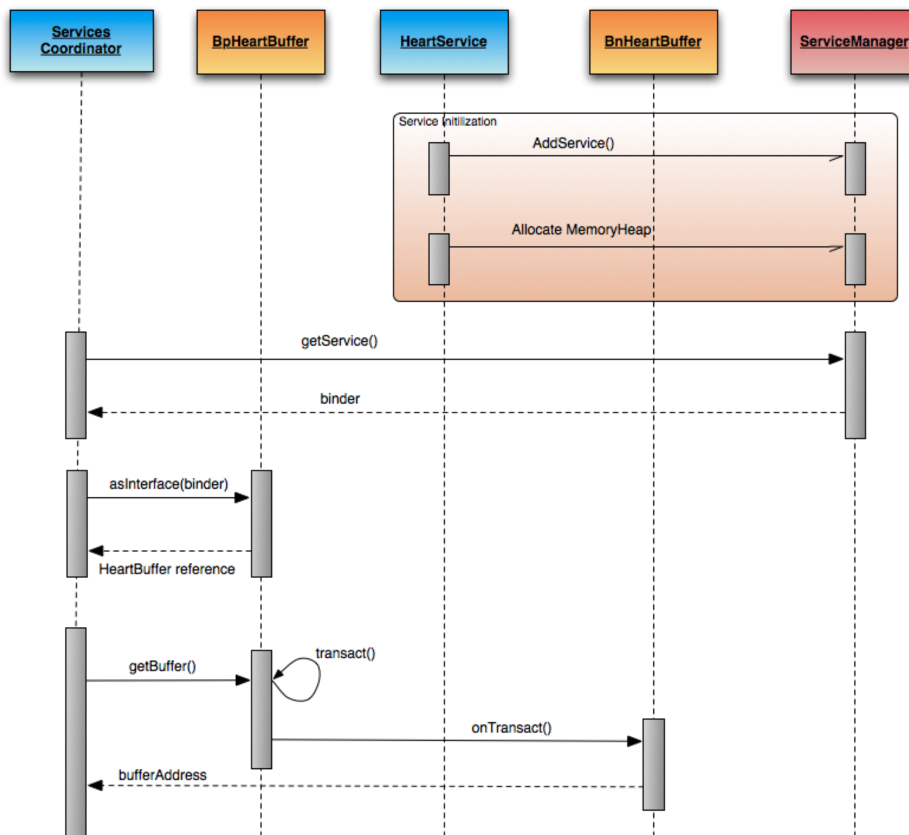


Figure 4.3: Sequence diagram of a client call

4.2 Native library Implementation

To make native C applications use heartbeats API, these functions have to be compiled into a shared library.

Each application that wants to initialize heartbeats API and the SC itself can use this library and they are clients in the process of retrieving the address of the shared memory from the binder interface. This process is anyway transparent to applications, because the functions to access the shared memory have been integrated inside the heartbeat API shared memory implementation. The client function getBufferMemPointer is used directly in the library, the address is retrieved and then it is casted to the structure representing the shared memory. This phase is performed in the heartbeat_init function, and once it has completed, the shared memory can be used in all functions of heartbeats API.

The HeartBufferService has to be available every time a function requires to access the shared memory. It has been implemented as a background service and included in the init.rc list of the processes that are executed at the OS start.

4.3 From native library to Java library

Since our library is written in C++ and compiled as a native shared library, in order to use the Heartbeats API also into Java applications, library functions have to be exported and wrapped into Java functions using NDK and JNI. The Android NDK is a toolset that lets you embed components that make use of native code in your Android applications. Using the JNI framework, native functions are implemented in separate .c or .cpp files. When the JVM invokes the function, it passes a JNIEnv pointer, a jobject pointer, and any Java arguments declared by the Java method.

The *JNIEnv* pointer is a structure that contains the interface to the JVM.

It includes all the functions necessary to interact with the JVM and to work with Java objects. Also native data types can be mapped to/from Java data types. Some conversions are predefined, e.g. integer is mapped to jinteger and float to jfloat, while objects and pointers have to be correctly referenced.

The first step is to wrap the native heartbeats API using JNI. The created file in this step is still in C++.

If a data structure is just used in a library function like it is the case of heartbeat_t type, its pointer is mapped to a jlong and no explicit wrapper for the type has to be created. While, if an object reference is passed to a function to modify the object content, the object data type has to be wrapped into a JNI data structure. That is the case of the type heartbeat_record_t.

Heartbeat_record_t is the structure that stores information about an issued heartbeat, like its timestamp, the tag associated with it or the current rate until this heartbeat. In JNI, it is wrapped in this way:

```
1  static inline jint HBRecordWrapper(JNIEnv* env,
2                  heartbeat_record_t* record, jobject recordObj){
3
4                  //Init wrapper object
5                  jclass recordClass = env->GetObjectClass(recordObj);
6
7                  //Get field beat for the structure
8                  jfieldID field = env->GetFieldID(recordClass, "beat",
                       "J");
9                  //Store field in the object
10                 env->SetLongField(recordObj, field,record->beat);
11
12                 //To be repeated for all field
13 }
```

Listing 4.1: Heartbeat record type

Then, each function in the library has to be wrapped in a JNI function. Here are listed the most significative ones.

```
1  static jlong android_heartbeat_AppHeartBeatInterface_init
2                           (JNIEnv* env, jobject thiz,
3                            jdouble min_target, jdouble max_target,
4                           jlong window_size, jlong buffer_depth, jstring
                                log_name)
5  {       const char *logNameStr = env->GetStringUTFChars(log_name, NULL);
6          heartbeat_t* hb = (heartbeat_t*) malloc(sizeof(heartbeat_t));
7          rc = heartbeat_init(hb, min_target, max_target,
               window_size,buffer_depth,
8                  logNameStr);
9          env->ReleaseStringUTFChars(log_name, logNameStr); }
10
11 static jlong android_heartbeat_AppHeartBeatInterface_heartbeat(JNIEnv*
       env,
12                 jobject thiz, jlong lpHeartbeat, jint tag)
13 {       heartbeat_t* hb = (heartbeat_t*)lpHeartbeat;
14         rc = heartbeat(hb, tag); }
15
16 static jint android_heartbeat_AppHeartBeatInterface_getCurrent(JNIEnv*
       env,
17                  jobject thiz, jlong lpHeartbeat, jobject recordObj)
18 {       heartbeat_t* hb = (heartbeat_t*) lpHeartbeat;
19         heartbeat_record_t* record = (heartbeat_record_t*)malloc
20                 (sizeof(heartbeat_record_t));
21         hb_get_current(hb, record);
22         HBRecordWrapper(env, record, recordObj);
23         free(record); }
```

Listing 4.2: Heartbeat Initialization

This JNI interface has to be registered to be called from Java side. Each function has to be registered into a static array of JNINativeMethod, with its signature. Than those methods are registered in the declared Class Name.

Than, Java code must wrap those JNI functions. A Java class has been created as *HeartBeatRecord.class* to store the heartbeat record type, and another class *AppHeartBeatInterface* to wrap the API functions. The HeartBeatRecord class is just made of one function and a constructor. The AppHeartBeatInterface has to load the JNI library (called heartbeat_jni) and declare

native functions that can be used in Java and match exacly the signature of the JNI functions declared in the library.

```
1 static {        System.loadLibrary("heartbeat_jni");           }
2
3 public static native long heartbeat_init(double min_target, double
      max_target,
4         long window_size, long buffer_depth, String log_name);
5 public static native int heartbeat_finish(long lpHb);
6         ...
```

Listing 4.3: AppHeartBeatInterface

The same methodology has been used to wrap the HeartbeatMonitor native library, resulting in a JNI library heart_rate_monitor_jni and a Java class HeartRateMonitorInterface. The three generated classes (AppHeart-BeatInterface, HeartRateMonitorInterface and HeartBeatRecord) are compiled into the same JAR archive. This Java library can be imported in any Java project to use Heartbeats API into an Android Activity.

4.4 Services Coordinator

The SC has been identified as the entity in charge of monitor applications and activate services on them.These are the most important functions available in the SC implementation:

GetBufferMemPointer it is the function used to retrieve the base address of the shared memory to be used in heartbeats operations. It is the same previously defined and it is used in the SC initialization phase. If the buffer service is not published, execution quits. The pointer retrieved with this function is casted to the type representing the memory area.

updateAppList is used to retrieve the list of applications that are running and have requested to be monitored. Heartbeat-registered applica-

tions have a state registered into the shared memory, associated with their pid.

updateRegistryInfo is a function used to initialize a monitor for each application to be monitored. At each iteration, current information retrieved from the heartbeat monitor are saved locally to be used by the decision engine.

ServicesRegistration at the moment it is not based on shared memory. The list of possible services is hard-coded in the engine and this function checks the list to see what services are really published and available to be used.

ExecuteActions is the function that implements the decision engine and it activates a service when needed.

The SC should implement a system to retrieve information from the OS, in this particular case we have methods to read information about the CPU current state.

getCpuUsage this function parses information from /proc/stat file to retrieve information about CPU kernel, user-space and idle time.

4.5 Frequency scaler actuator implementation

In order to implement a frequency scaler, a driver that enable changing the CPU frequency at runtime is required. One of these drivers is called CPUfreq and it is the one used. A CPUfreq policy consists of a couple of values (min, max) that are the minimum and maximum possible frequency that can be set. Governors are used to decide which frequency within the CPUfreq policy should be used. Usually CPU can work only at specific frequencies, that are device dependent and are stored in a system file. The

CPUfreq governor decides (dynamically or statically) which frequency to set within the limits of the policy. The frequency is set to the feasible value nearest to the chosen one, if it is within the policy limits, otherwise no changes are applied.

The available governors are the following:

Performance The CPUfreq governor "performance" sets the CPU statically to the highest frequency within the borders of the policy.

Powersave The CPUfreq governor "powersave" sets the CPU statically to the lowest frequency within the borders of the policy.

Userspace The CPUfreq governor "userspace" allows the user, or any userspace program running with UID root, to set the CPU to a specific frequency.

Ondemand The CPUfreq governor "ondemand" sets the CPU depending on the current usage. To do this the CPU must have the capability to switch the frequency very quickly. Parameters like the sampling rate (how often to look at CPU usage and to make a decision) and the up_threshold (the average CPU usage needed to make a decision to increase the frequency).

Conservative similarly to the "ondemand" governor, it sets the CPU depending on the current usage. It differs in the fact that it gracefully increases and decreases the CPU speed rather than jumping to max speed the moment there is any load on the CPU, like it is done in the ondemand governor.

The CPUfreq driver stores its system files into the directory <sysfs root>/devices/system/cpu/cpuX/cpufreq/, where <sysfs root> is the device root and cpuX is the number of the core into consideration. The most important configuration files are now listed:

scaling_available_governors is list of available governors;

scaling_available_frequencies is list of available frequencies, device dependent;

scaling_min_freq corresponds to the minimum frequency of the policy;

scaling_max_freq corresponds to the maximum frequency of the policy;

scaling_cur_freq shows the current frequency of the system;

scaling_governor is used to set the governor to be used;

scaling_setspeed is used to set a frequency value and can be used only with the userspace governor;

stats provides frequency statistics.

4.5.1 Using userspace governor

In order to dynamically change the frequency of the selected CPU, some steps have to be performed. Here are listed the commands, in bash code for better readability, that have to be issued to scale frequency. These commands can be converted in I/O operations on the same files.

1. **Set the governor** Userspace governor has to be selected
   `echo userspace > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor`

2. **Set the policy min value** To set e.g. 800MHz as the maximum frequency:
   `echo 800000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq`

3. **Set the policy max value** To set e.g. 122.8MHz as the minimum frequency:
   `echo 122800 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq`

4. **Set a specific frequecy** To set e.g. a 300MHz frequency:
   echo 300000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed

The list of device-specific available scaling frequencies and the current frequency can be retrieved using the commands:
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies
and
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq

Those actions have to be repeated every time the system is rebooted.

4.5.2 Actuator implementation

The actuator used in the frequency service is a frequency scaler and it has the ability to perform two actions, an UP action and a DOWN one. Both actions at first make sure that the correct governor is selected and read the current frequency. Then, if it is an UP action, the frequency selected is increased to the next frequency available greater than the current one. On the other hand, if it is a DOWN action, the frequency is decreased with the same principle. Those action accept a flag: if it is set, the frequency is set to the maximum possible in the UP action, or to the minimum possible, in the DOWN action.

```
1  void Upfreq(bool flag){
2          //get available frequencies list
3          list=..
4
5          //get current frequency
6          fd=fopen("/sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq","r");
7          fgets(currentS,7,fd);
8
9          //find current frequency in the list and select new frequency
10         if(flag)
11                 newFrequency=maxFrequency;
12         else
13                 newFrequency= nextInTheList;
14
15         //write and set new frequency
16         fd=fopen("/sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed","a");
17         fputs(newFrequency,fd);
18 }
```

A similar implementation is used also for the DOWNfreq() function.

4.6 Network-type changer actuator implementation

In order to implement an actuator able to change at runtime the network that have to be used for data connection, there is not necessity for a specific driver, since the Android SDK provides all the necessary functions. At first it is provided a description of the network types that can be accessed by current devices, then some details about the actuator implementation on the Android platform are provided.

4.6.1 Network types

Current phones can belong to the *CDMA enabled* category or *GSM enabled* category. Global System for Mobile Communications (GSM) is the mobile communications technology originating in Europe that it is now the

most popular mobile standard in the world, at about 80% of the world's more than 4 billion cell phone users. Conversely, competitor Code Division Multiple Access (CDMA) is a mobile technology that originated in the U.S. that serves upwards of 10% of the world's cell phones. In this work we are dealing only with GSM phones.

The GSM family of data technologies include GPRS, EDGE, UMTS - WCDMA and HSDPA, each one with different performances and power consumptions. Usually the GSM/GPRS/EDGE technologies are considered 2G networks, while WCDMA/HSDPA are considered 3G networks.

GPRS is a globally available network that makes many applications feasible, including messaging, e-mail, Web browsing and some multimedia applications. EDGE significantly expands the capability of GPRS, enabling richer Internet browsing, streaming applications and more multimedia applications. Then, with UMTS and HSDPA, users can video call, listen to high-fidelity music and use rich multimedia applications. These different network types are characterized by different performance, different power consumption and different coverage.

Data about different networks performances are reported in Table 4.1 and they are taken from [53]

Table 4.1: Performance data of different network types

Type	Peak Network Downlink Speed	Average User Throughputs
GPRS	115 kbps	30 - 40 kbps
EDGE	473 kbps	100 - 130 kbps
WCDMA	2Mbps	220 - 320 kbps
HSDPA	14 Mbps	550 - 1100 kbps

With respect to power consumption, some data about the different consumptions of 2G nets and 3G nets are provided in [54] and reported in Table

4.2.

Table 4.2: Power consumption of different tasks using different networks

Task	2G	UMTS
Receiving a voice call	612.7 mW	1224.3 mW
Making a voice call	683.6 mW	1265.7 mW
Idle mode	15.1 mW	25.3 mW

The device network choice is influenced by the preferred network type selected. Types of preferred network type modes than can be selected are:

WCDMA preferred : The GSM phone is capable of using both 2G and 3G data communication and when signal strength is low 3G is favored more.

GSM only : The GSM phone is capable of using only 2G data communication. When the 2G signal is too low, no connection is established.

WCDMA only : The GSM phone is capable of using only 3G data communication. When the 3G signal is too low, no connection is established.

GSM auto : The GSM phone is capable of using both 2G and 3G data communication and when signal strength is low 2G is favored more.

Usually, current phones are set to *WCDMA preferred*, if another mode is not forced by the user. In our actuator, to have complete control on the network selected, we are using only the *GSM only* mode, to have a 2G net, or the *WCDMA only*, to have a 3G net.

4.6.2 Actuator implementation

This actuator has the ability to change the network to which the phone is connected. The change is performed not directly changing the connec-

tion, but changing the phone preferred network. The actuator has to belong to the process android.phone in order to call these functions, because they are internal to the phone management system and has to access the method *setPreferredNetworkType* specifying the *GSM only* constant to force the connection to use the 2G network, or specifying the *WCDMA only* constant to force the 3G network usage.

Chapter 5

Experimental Results

In this Chapter the results of the various tests performed to prove the effectiveness of the approach in a mobile environment are described, and in the very first Section the device used in all tests is presented . Tests performed aim to prove that both versions of the heartbeats library (file based and shared memory based) are correctly integrated into Android framework and that the shared memory based implementation is faster than the the file based implementation.

Then, tests are performed to prove that, in this approach, applications, both self-adaptive and managed by the SC, are able to respond to changes to keep a good QoS over time. Self-adaption is tested using as actuators both an application knob and an actuator able to perform a change in the implementation of an algorithm. The SC has different services available that use different actuators with different targets. Tests prove that these services can be used with effectiveness to reach different goals.

5.1 Testing platform

The Android SDK provides a mobile device emulator to be used to test applications and libraries on a generic Android system. Using the emulator

for testing is very easy and handy, but it is not precise as a real device and some features are not available (e.g. frequency cannot be changed at runtime). To test our aware system, we used a real device equipped with an Android OS patched to integrate self-aware capabilities. The device used in all our tests is the LG Optimus One P500. This device is equipped with an Android OS, version 2.2 Froyo and relevant characteristics are:

- 600 MHz ARM 11 processor, Adreno 200 GPU, Qualcomm MSM7227 chipset
- 418MB RAM
- FT capacitive touch-screen, 256K colors, 320 x 480 pixels, 3.2 inches
- Battery Li-ion 1500mAh

Figure 5.1: LG Optimus One P500

This device does not run the last Android version, Gingerbread. An official update for this device has not been released yet by LG and probably will never be made available. Apart from the official versions, there exists alternative and customized ROM developed by community users and made public. Usually these ROMs are experimentally but provide additionally features the official update does not have.

Our first configuration step will be to upgrade the OS of the device to Android Gingerbread 2.3. The ROM used is a version of Cyanogen Mod for LG P500 device, the only one at the moment available for this device and still in alpha release. This ROM provides a 2.3 Android version and a kernel 2.6.32.9. This kernel has been patched to integrate the CPUfreq driver and

to support over clocking.

A preliminary step is to grant root access to the device, a privilege that is normally denied. After that, the selected ROM has been flashed on the device using a custom recovery image.

Using the CPUfreq driver, the available frequencies of this device have been extracted. They range from 122 Mhz to 844 MHz as shown in Table 5.1.

Table 5.1: Available frequencies

	Frequency (Hz)
Standard	122880
	245760
	320000
	480000
	600000
Over clocked	729600
	748800
	768000
	787200
	806400
	825600
	844000

The maximum frequency allowed in this processor is 600MHz, frequencies over this threshold are due to CPU over clocking and have to be tested to see if the device is still stable at all these speeds. Tests show that this device can handle frequencies until 806 MHz.

For each frequency, the device assigns a voltage, defining the working

point of the CPU. We have tested all frequencies and for each one the voltage is almost costant. Maximum and minimum values allowed are 4200 and 3200, respectively. All tests have been performed first on the emulator and then on the device, but since results on the device are more relevant, they will be the only ones taken into consideration in this Chapter.

5.2 Test system structure

In order to perform a set of test cases, needed to verify the correctness of the proposed approach on a mobile devices, a test system has been implemented on the previously described device. The structure of the system, based on the structure exposed in Section 3, and its components are shown in Figure 5.2.

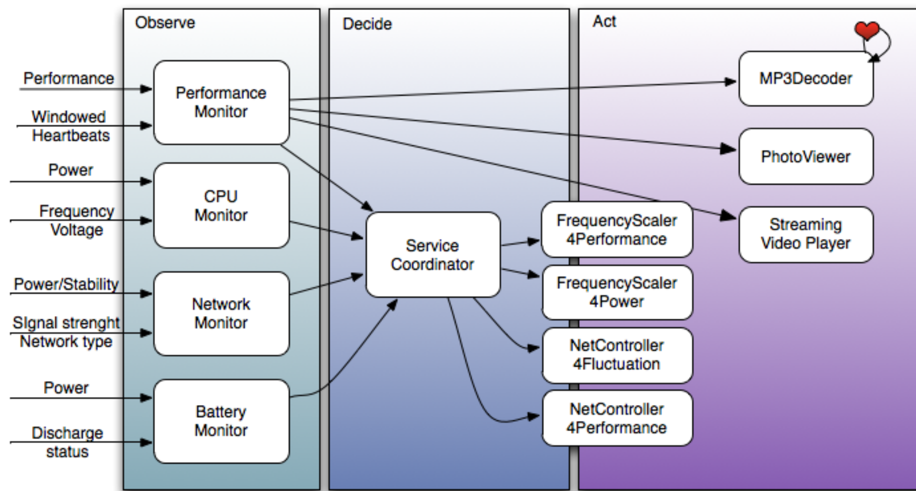


Figure 5.2: Test system structure

The monitoring phase is composed by:

- **Performance monitor**: based on the Application Heartbeats framework and used to retrieve informations about the applications performances.

- **CPU monitor**: used to retrieve informations about the CPU current frequency and voltage values, through the CPUfreq driver.
- **Network monitor**: used to retrieve information about the mobile data connection status, in particular the network type and the signal strength.
- **Battery monitor**: indicates the percentage of battery charge.

The SC has the availability of different services, that include:

- **FrequencyScaler4Performance**: a service that uses the CPU frequency scaler actuator with a performance target. Its policy is to increase the CPU frequency to an higher value to help an application compute an intensive task as soon as possible, than restore the frequency to normal values.
- **FrequencyScaler4Power**: a service that uses the CPU frequency scaler actuator with a power target. Its policy is to decrease the CPU frequency to reduce the device power consumption.
- **NetController4Fluctuation**: a service that uses the actuator able to change the preferred network type to avoid network fluctuation. Its policy is to stabilize the network type on a lower performance network if the fluctuation between different networks is too frequent.
- **NetController4Power**: a service that uses the actuator able to change the preferred network type to reduce the device power consumption with respect to the applications performance. Its policy is to try to set the connection on a low power network to reduce the power consumption and check the application performance to ensure that the applications are having good performance even after the change. If they are not, connection is set back to its initial type.

Table 5.2 summarizes the use of different actuators in services and their targets.

Table 5.2: Actuators available in each service and its target.

Service	Target	Actuators
FrequencyScaler4Performance	Performance	CPU frequency scaler
FrequencyScaler4Power	Power	CPU frequency scaler
NetController4Fluctuation	Stability/Power	Network type changer
NetController4Power	Power/Performance	Network type changer

The system runs a set of test applications, both self-adaptive and controlled, that includes an MP3decoder, a photo viewer and a streaming video player. Those applications will be described in Section 5.4.

5.3 Testing Heartbeat Framework

This Section is intended to analyze the impact of the Heartbeats framework into an Android system. Such investigation is extremely important in the context of autonomic systems, since the monitoring infrastructure must be as lightweight as possible. Both versions of Heartbeats API, file-based and shared memory based, have been ported into Android framework. The first test is performed to decide which one provides the better performance, to choose which implementation is more suitable to be used in our final system.

The most frequently used function of the API set is the heartbeat() function and it requires constantly access to files or to the shared memory.

In this test, the execution times of those eighty calls of this function are measured, both using the file based and the shared memory based implementation. Results are shown in Figure 5.3.

The solid line is the execution using the file based implementation. Its average is 2813473,48 ns and execution times suffer from a great vari-

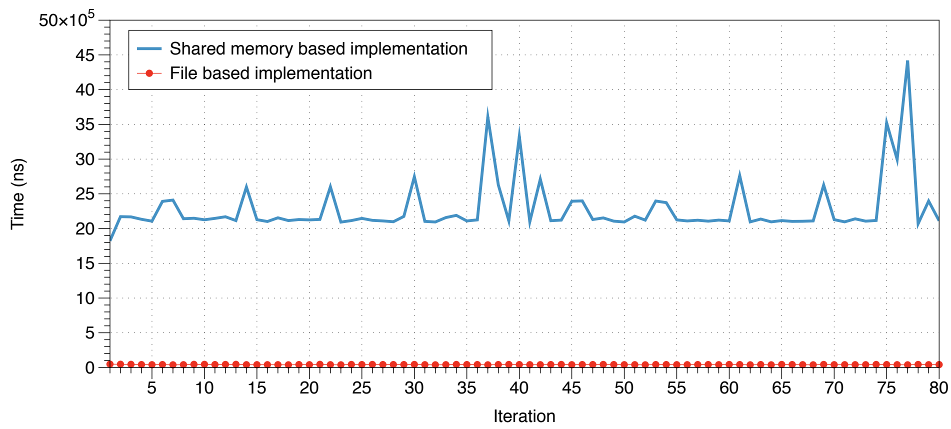


Figure 5.3: Comparison between the execution times of heartbeat calls using the file based and the shared memory based implementations

ance, due to the unpredictability of disk accesses. The same calls, using the shared memory based version, have an average execution time of 42538.46 ns, 66 times faster than the file based implementation. In addition, the variance characterizing the execution times is very low.

Execution times of ten calls to the HB_heartbeat() function are also shown in Figure 5.4, using a logarithmic scale.

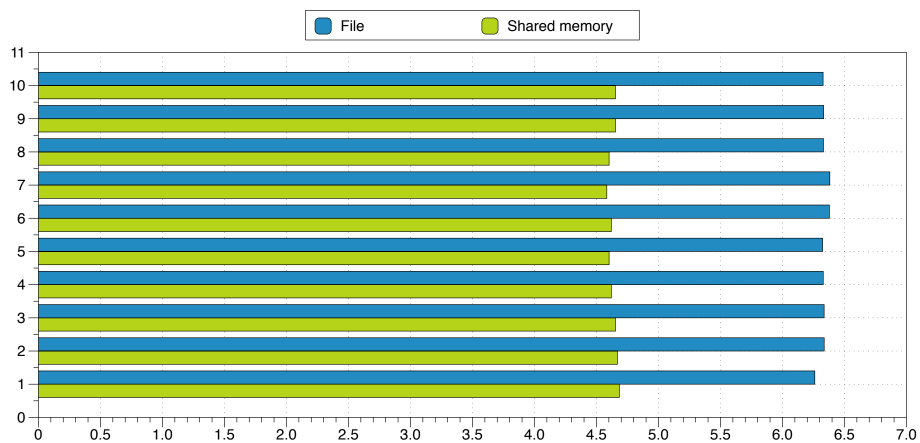


Figure 5.4: Heartbeat call execution times, both using file and shared memory version, 10 executions

It is easy to understand that the best choice is to use the shared memory version rather than the file based one. From now on, all tests refer always to the shared memory based implementation.

The Heartbeats framework has been tested also from the Java layer, to check the correctness of the class from Java to C and reversed and to see the overhead introduced by JNI. Execution times of native calls through JNI are influenced by the number of input parameters, because those parameters have to be correctly wrapped an passed to the lower levels. Due to this fact, we have tested the overhead on the function in the Heartbeats API that has the greatest number of input parameters, that is the heartbeat_init function, to compute the worst case overhead.

Its Java signature is *long heartbeat_init(double min_target, double max_target, long window_size, long buffer_depth, String log_name)*.

Table 5.3 shows the results of ten calls to the heartbeat_init function, both from Java and natively from C, and their average.

Note that Java and C execution times have been recorded separately. That is, on a single row of the previous Table, Java and C execution times are not related. Related execution times cannot be easily measured because the Java call does not provide information on the native function it wraps.

JNI introduces an overhead, in the worst case, that duplicates the time necessary to perform the initialization call. Nevertheless, to have a better idea of the impact of the framework, we should measure its impact on applications execution.

To test the overhead introduced during an execution, we have used the MP3 decoder application. Several tests have been performed using input streams of different sizes. The buffer size is the same in all tests performed and it has been set to 40KB. Table 5.4 shows the results of the tests performed. The overhead introduced is, in average, 0.15% of the execution time.

Table 5.3: JNI overhead over an heartbeat_init function

	Java	C
Execution time (ns)	15435000	5770000
	13883334	7093333
	10726667	5613333
	20070000	7933334
	14726667	6551667
	14456666	6636666
	8440000	6113333
	11203333	6473333
	14886667	6530000
	14631667	6499999
Average time (ns)	13846000,1	6521499,8
Average JNI overhead	7324500,3	
Average JNI impact (%)	52,9	

Table 5.4: Impact of the Heartbeats framework over an application execution

Input size (MB)	Heartbeats time	Total execution time	Impact (%)
1.1	74178334	45636491690	0.16254171
2.8	281066678	176528440028	0.159218921
3.4	259400057	158665350013	0.163488788
4.2	279718317	175273043354	0.159590038
5.5	364231606	235572626693	0.15461542

5.4 Case studies: self-adaptive and SC guided applications

Three applications have been developed to be used as case studies for many of the tests following in this Chapter. The first one is an MP3 decoder and it is a self-adaptive application. It is written in C++ and therefore it runs in the low level of the Android stack. The second one is a photo viewer, this time written in Java and wrapped in an Android activity.[1] This application has not self-adaptive characteristics and it can be controlled by the SC. The last one is a video player, able to stream and play a video file from a server. No self-adaptive features have been integrated inside this application.

Adaptive Mp3Decoder One of the applications developed as a case study, is an adaptive MP3 decoder. This application is a streaming application, since it has to decode a new buffer of data when the previous one has been played and so it requires constantly CPU resources over its execution. This application is written in C++, and it has been chosen to do not write this application in Java in order to report times and overheads without the additional overheads caused by JNI. This application does not have a graphical user interface, but a demo of the tests performed using this application with self-adaption has been proposed using Java to provide a simple GUI.

This application goal is to keep a good QoS, monitoring its performance and using an actuator to modify its internal parameters. We have developed two different versions of this application, that differ from the actuator used during the adaptation phase. In the first case, the actuator belongs to the class of application knobs while, in the second one, the actuator performs an implementation change.

In both the two versions, the monitor phase is performed through

[1] An activity represents a single screen with a user interface

the usage of the Heartbeats API, both to define goals and to monitor progresses toward them. For an MP3 decoder application, the minimum QoS should guarantee is to provide a stable data stream, so that the song can be played without any interruption. Given this goal, we have decided to generate an heartbeat every time a buffer of data has been decoded. The metric used to decide if a change is needed will be the rate of heartbeats issued in a window of time. This parameter can be extracted using the Heartbeats API with the function *get_windowed_rate()*. Goals are expressed in terms of the minimum and maximum heart rate that the application should keep. These parameters depend on the device the application is running on and on the system conditions. Usually these data are hard coded into the application, but in the future a learning phase, in which the application learns its optimal minimum and maximum rate, can be implemented.

MP3 data read from input file have to be decoded into PCM samples, in order to be played using the Android audio framework. We have included two decoding algorithms into the Android framework: the *MPG123* algorithm and the *MiniMP3* algorithm. These algorithms are written in C and they provide different output performance, in terms both of the decoding speed and the quality of the audio played.

ReadSamples is a generic function that decodes a buffer of data, using the selected implementation of a decoding algorithm, from the input MP3 file. The buffer size can vary. The monitoring phase is the same in both versions of our application. Instead, the deciding and acting phases are specific for each version, given the fact that different versions have different actuators. The common point is that since both versions are self-adaptive, they can only modify internal parameters of the application itself.

PhotoViewer Another application used in these tests is a photo viewer and it has been developed in Java, to test the Heartbeats framework exported with JNI and the ability of the Services Coordinator to interact also with Java applications. The goal of this application is to load a set of photos and visualize them on the device display. The photo are preloaded first, and then they are all visualized at the same time. This allows tests on execution times of an intensive task at different frequencies. During this preloading phase, an heartbeat is issued each time a photo has been loaded. Then the application acts only as a visualizer and its CPU usage is almost zero. Ten photos of different sizes are used, and photo paths are hard coded into the application. This application is not self-aware but instead it is monitored by the SC, that will record its monitoring data and decide which service activate on it, if necessary.

Audio-Video streaming player The last application implemented is an audio-video player, written in Java, that is able to stream an audio file or video from a web server. While audio files are streamed correctly without any limitation of format, Android has the ability to play video only using the HTTP progressive download technique, not implementing a real video streaming. This technique provides an end user experience similar to streaming media, since the user is able to watch the video while the rest of the video is loading into the player, caching the video into the users temporary internet files. The application has not self-aware abilities and heartbeats are issued during the buffering phase.

Figure 5.5 shows the interaction of the first two applications with the Heartbeats Framework during the monitoring phase. The interaction between the Audio-Video Player application and the Heartbeats framework is similar to the mp3decoder one.

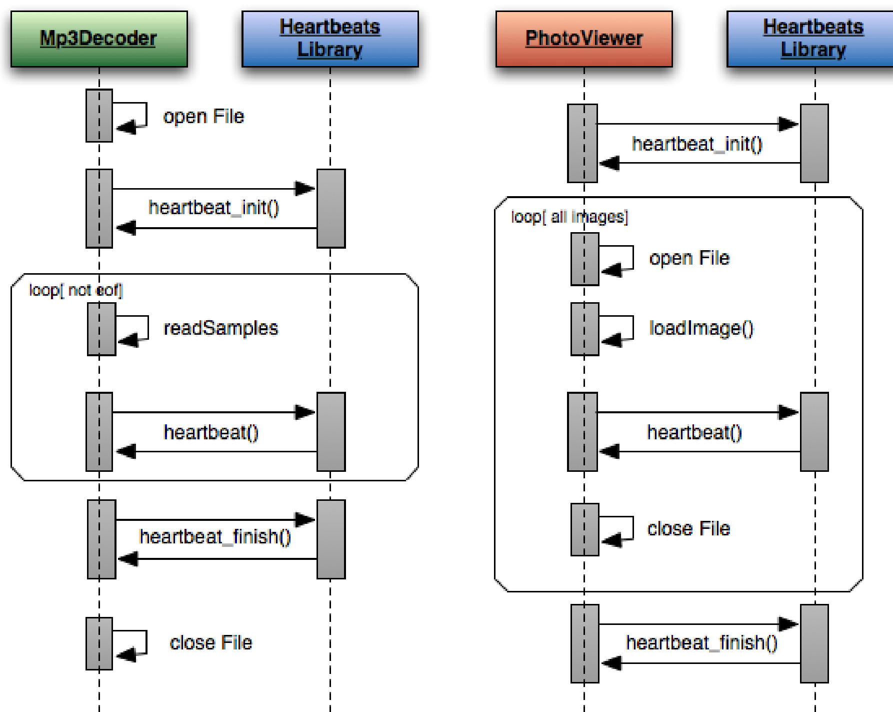


Figure 5.5: Sequence diagrams of the monitoring process using heartbeats API in MP3 Decoder and PhotoViewer applications

5.5 Testing self-aware applications

This section will test the ability of self-aware applications to keep a good QoS. In the first test, the actuator is an application knob, while in the second one a changing in the algorithm implementation is performed. After these two tests have been performed, a comparison between the reaction times of these two self-adaptive applications is shown.

5.5.1 Application knobs

The first version of the MP3 decoder reacts to changes by modifying an application parameter. The parameter chosen is the buffer size, a parameter used in the readSample function to know the amount of data to be decoded from the input file. The buffer size influences the time needed to decode and send a block of data to the player. A smaller buffer size will prevent the CPU from being blocked for a long time and so it is better to reduce the buffer size in periods where the CPU is heavily loaded. On the other hand, a larger buffer size helps in avoiding streaming skips. Table 5.5 shows how the heartbeat rate varies depending on the buffer size. Data have been retrieved using the MPG123 algorithm. In the first column is shown the average heart rate in the first thirty iterations using different block sizes. The first ten iterations are a settlement, due to the window filling process, while a more accurate measure is provided considering the last twenty iterations. Its average is shown on the second column.

Results show that acting on the buffer size modifies the application heart rate and therefore its performance. The decision system has been implemented using the decision tree shown in Section 3.3.2. If the heart rate is within the decided limits of a good QoS, therefore within the minimum and maximum heart rate limits, no action is performed. While, if the current heart rate is below the minimum acceptable heart rate, the buffer

Table 5.5: Heartbeats rate with different buffer sizes, miniMP3

Block size (Byte)	Avg over 30 iters	Avg over last 20 iters
8	18.662591	21.574667
16	9.535203	10.780328
24	6.608230	7.176518
32	4.954121	5.377346
40	4.013479	4.304207
64	2.532456	2.692246
80	1.041693	1.076222

size is decreased by a predefined amount called *buffer_change_step*. Otherwise, if the heart rate is over the maximum rate, the buffer is increased by a *buffer_change_step*. This step has been set to 1KB, to slowly decrease or increase the buffer size at each iteration. Changing the size of the buffer step will modify how fast the application reacts to changes. Desired heart rate is within 3 and 5. The policy reduces the buffer by a *buffer_change_step* every time the rate is under the minimum threshold but it is allowed to increase the buffer size only if for ten iterations the heart rate is over the maximum rate.

To test our adaptive application, we have manually introduced a controlled disturb that simulates a falling heart rate. The disturb was introduced starting from the hundredth iteration and it lasts fifty iterations. Figure 5.6 shows the rate progression in the application, without adaptivity enabled.

When the hundredth iteration is reached, the heart rate starts dropping and no change is performed to increase the heart rate. The blue area in the graph indicates the desired heart rate. Figure 5.7 shows the heart rate

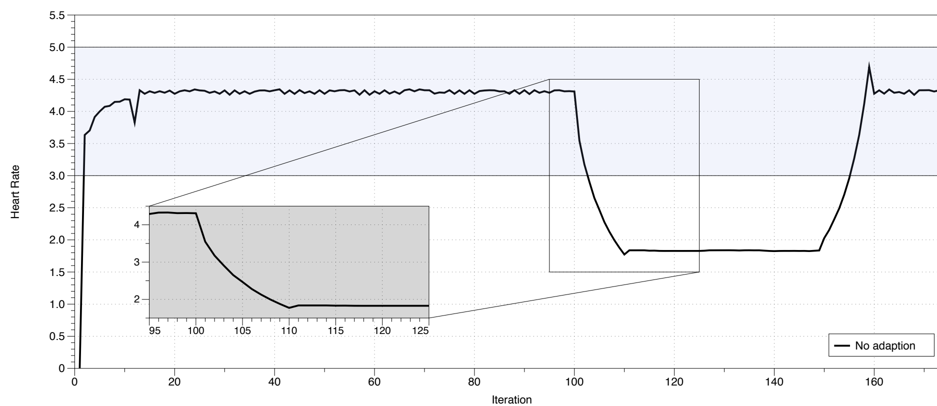


Figure 5.6: Heart rate progress with the introduction of an external disturb, starting at the 100th iteration, without adaptation.

progression, with the adaptive system enabled. At the hundredth iteration the heart rate starts dropping and, when it is considered under threshold, the buffer size is decreased at each iteration to make the heart rate increase. After few iterations, a correct heart rate is restored to regular values.

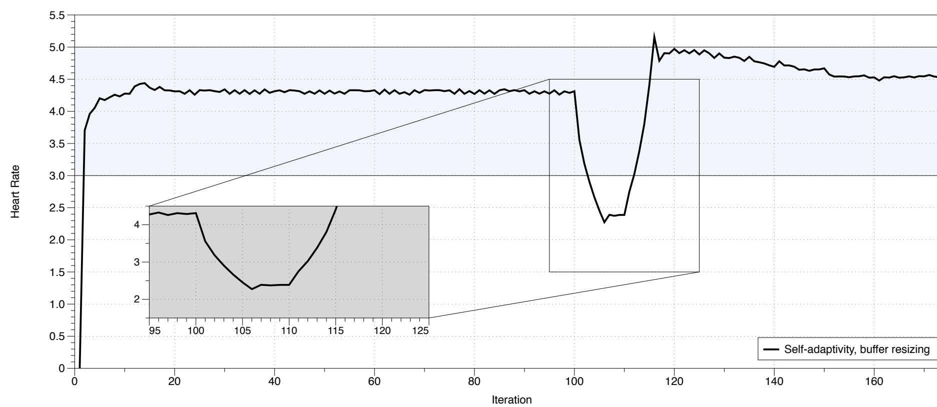


Figure 5.7: Heart rate progress with the introduction of an external disturb, adaption enabled with buffer resizing.

5.5.2 Implementation changing

The second version of the self-adaptive MP3 decoder makes use of an implementation changing as actuator to perform adaptivity. The miniMP3 decoding library is an implementation of a decoding library that provides a slightly better quality in the music played but the decoding process for each buffer is slower. Changing implementation will increase the buffer decoding process, that will consequently increase the heart rate. Note that in this test the maximum desired heart rate has been increased to 9, while the minimum desired heart rate is set to 3, like in the previous test.

Figure 5.8 shows the heart rate progress using the adaptive system with implementation changing.

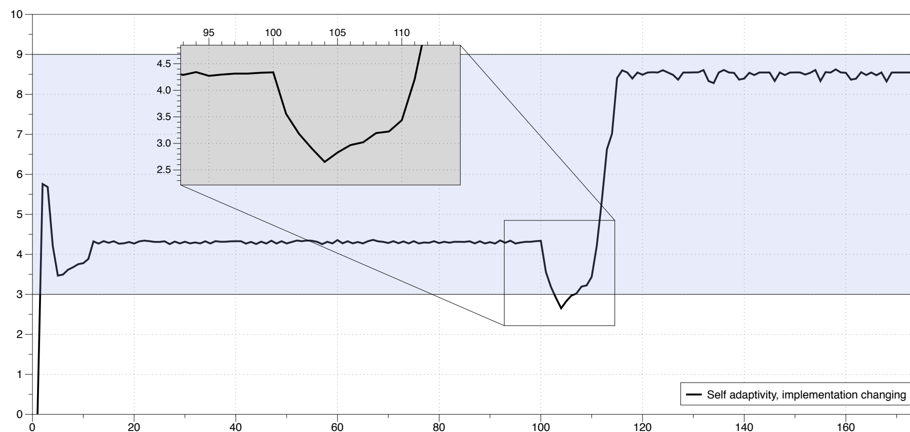


Figure 5.8: Heart rate progress with the introduction of an external disturb, adaption enabled with implementation changing.

As in the previous test, if the application rate is within the limits no changing is performed. If the rate is below the minimum threshold, the algorithm implementation is changed from miniMP3 to mpg123. However, if the heart rate is over the threshold, the implementation is not changed back to miniMP3, because we have seen that if the algorithm is changed frequently the application becomes unstable and not able to keep a good

QoS.

Since to change implementation it is required a quiescent state, we have decided that a change is possible only when the data in the current buffer have finished to be decoded with the previous implementation. Then, the change is performed, and the next data in the buffer will be decoded using the new algorithm.

5.5.3 Reaction times

The test previously explained shows that both two versions of our self-aware adaptive application are able to react correctly to an external disturb, that causes a falling heart rate. On these tests we have measured the time needed to the application to react to a falling heart rate. This time has been calculated as the interval between the first heart rate identified as low (under threshold) and the first heart rate within the QoS limits. We have taken into account ten executions of the application with the same characteristics, with both the self-adaptive methods.

Table 5.6 shows the results. The average reaction time of the adaption system with implementation changing is nine times faster than the adaptive system that uses the application knob. Anyway, the implementation changing technique has a drawback. Once the second implementation is chosen the implementation cannot be changed back and the system is vulnerable to new disturbs. A good approach can mix both techniques and apply first an implementation changing and then, if needed, a buffer resizing.

5.6 Testing services

In this Section services available in the system are tested. Services are tested independently and also a test on how services can be applied by the

Table 5.6: Reaction times using different actuators

	Buffer Resizing	Implementation Changing
Reaction time (s)	0.208	0.022
	0.212	0.021
	0.212	0.024
	0.207	0.024
	0.211	0.024
	0.211	0.022
	0.206	0.022
	0.206	0.023
	0.207	0.024
	0.209	0.022
Average time (s)	0.209	0.023

SC at the same time is shown.

5.6.1 FrequencyScaler4Power service

In this test we have used the MP3 decoder application, a streaming application, to test the service that uses the frequency scaler actuator with the goal to save as much power as possible. In this case the MP3 decoder, even if it has been designed as a self-adaptive application, is controlled by the SC and its self-adaptive features are blocked. The SC monitors the application and, if possible, activates this service on it.

The frequency scaler tries to change the frequency every ten iterations to perform a gradual change. Anyway lowest values are possible and depend on the application taken into consideration.

In the first test, the initial frequency has been set to 600MHz and the

MP3 decoder uses a 40KB buffer.

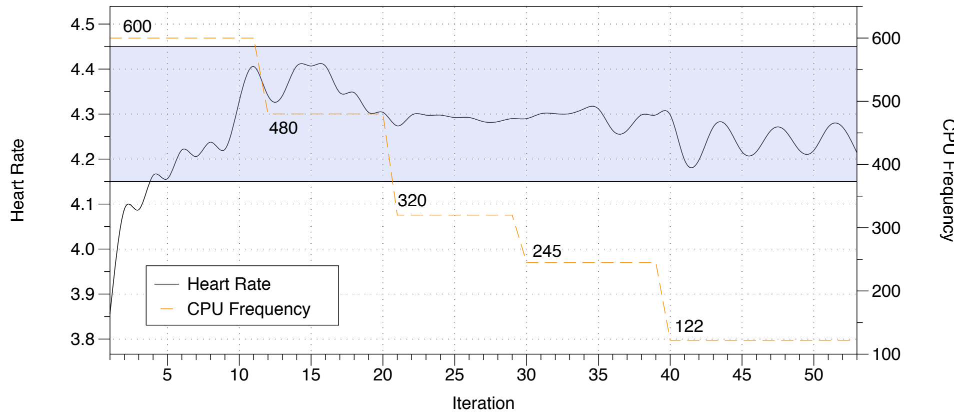


Figure 5.9: Heart rate progress of the MP3 decoder application on which the service considered is enabled. Buffer size is 40KB and the final frequency is set to 122MHz

Figure 5.9 shows the heart rate of the application while the service is acting on it. Starting from the initial frequency value, the frequency is decreased every ten iterations until it reaches the lowest value, 122MHz. Desired heart rate is within 3 and 5, for the entire execution the heart rate is always within this range.

A second test has been performed, keeping the initial frequency to 600MHz but changing the buffer size to 160KB. In this case, desired heart rate is within 1.058 and 1.090. This time, when the frequency is set to 122MHz, the heart rate starts dropping.

In Figure 5.10 are shown the results using a slightly different policy, in this case the execution of the service will not blindly scale down the frequency, but it will be guided by the heartbeat monitor to keep the application performance over its QoS minimum value. It is an example of a power target with a performance feedback.

This service tries to lower the CPU frequency to its lowest value, starting from the current frequency, at each step monitoring how good are the application performance. If this performance is too low (the heart rate is

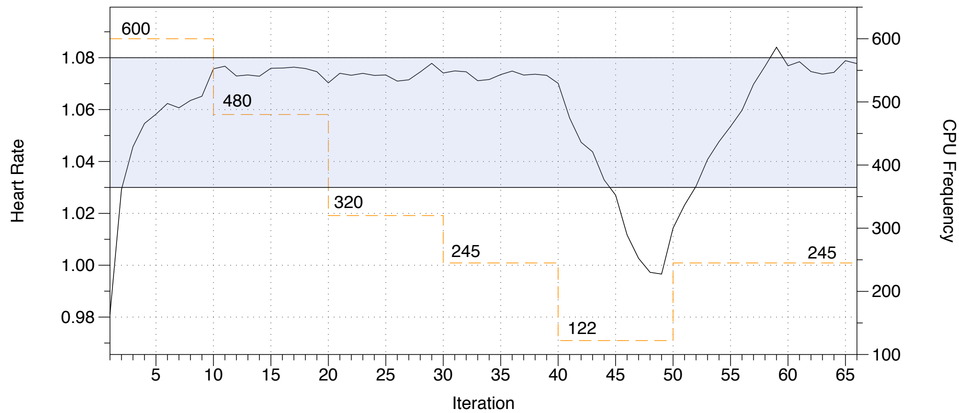


Figure 5.10: Heart rate progress of the MP3 decoder application on which the service considered is enabled. Buffer size is 160KB and the final frequency is set to 245MHz

below the threshold), the frequency is increased until a frequency stable for the application is reached.

When the frequency is set to 122MHz, the heart rate starts dropping. The application cannot keep its minimum QoS and so the service decides to increase the frequency one step at a time until the minimum heart rate is restored. Only one step is necessary, the final frequency is set to 245MHz.

In the first case, the frequency was changed from its initial value of 600MHz to a value of 122MHz. Using the theoretical model shown in Section 3.3.1 and given the fact that the voltage is constant and the execution time is the same in both executions, the power consumption can decrease of a factor up to 79%. In the second case, the frequency changed from 600MHz to a value of 245MHz and the decrease can be up to 50%. For a more precise estimation, we should compute the time the application executed in each frequency from the initial value to the final value.

5.6.2 FrequencyScaler4Performance service

In this service, performance is the primary issue. An application has to perform an intensive task and this task has to be performed as soon as possible. The application used in the following tests is the photo visualizer and the most intensive task it has to perform is to load into memory all the images before visualizing them. The ten images have different sizes. Each photo has been loaded separately to measure the loading time at different frequencies. Figure shows the results.

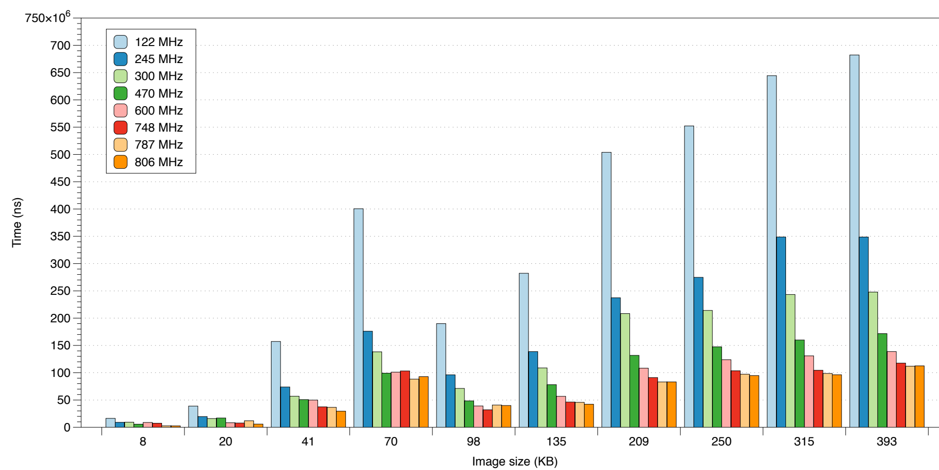


Figure 5.11: Loading time of 10 images of different sizes at different CPU frequencies

Images sizes vary from 8KB to 393KB and frequencies taken into consideration are all the available frequencies between 122MHz and 806MHz. Note that frequencies over 600MHz are over clocked frequencies. The obviously result is that the highest the frequency is, the lowest is the execution time needed to finish the task. Therefore the service, when activated, sets the frequency to the highest value possible. This value is kept until the intensive task has finished its computation. Since an heartbeat is issued each time a photo is loaded, when no more heartbeats are issued, it means that the task has completed. At this time, cpu can be put in an idle state as no

more computation is required for this application.

Considering an average value of 320MHz of frequency, increasing the cpu clock speed using over clock until the value of 806MHz will result in a reduction of the execution time of 54 %, in average. Considering the highest value of frequency possible without over clock, that is 600MHz, the reduction is 22%, in average.

5.6.3 Performance and power tradeoff

This test aims to prove the ability of the system in finding a tradeoff between performance and power. The application used in the test is the mp3 decoder with a buffer size of 160KB, and power target has priority over performance during the choice of services.

The SC starts searching for a service whose target is power. The only service available with this target is the service that uses the frequency scaler to reduce power dissipation and this one is chosen to be activated on the application into consideration.

Then, it is searched for a service able to control application performance. The SC has two options: selecting the service that uses the frequency scaler to adjust performance or delegating the performance management to the application, since it has self-adaptive features. Since the frequency scaler actuator is already in use in the service applied for power management, it cannot be activated. The obliged choice will be then to use application self-adaptivity features.

The actuator that implements a buffer reduction is the one used in this test. As it is shown in last Section, using a buffer size of 160KB, the power service is not able to set the frequency to the minimum value (122MHz) without making the application lose its QoS. Introducing a performance service in the test in conjunction with the power one, we are expecting to be able to set the frequency to 122MHz and keep the desired QoS, reducing

the buffer when needed. Desired heart rate is within 1.058 and 5. Note that the upper threshold has been increased with respect to the previous test. With the introduction of the buffer resizing technique, higher heart rates are allowed.

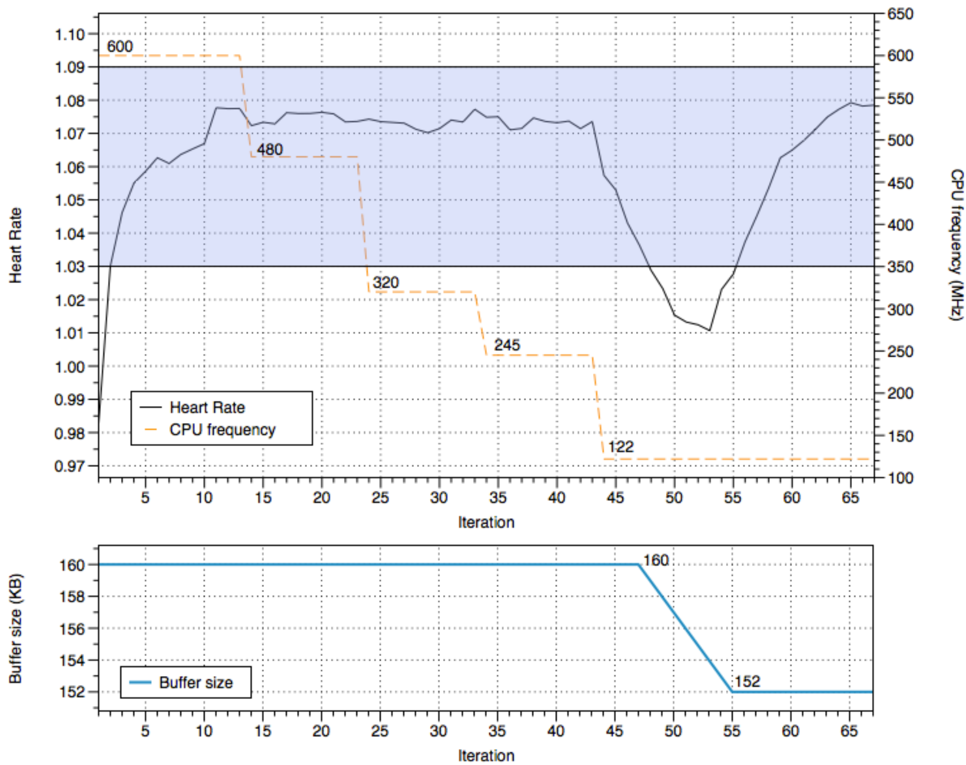


Figure 5.12: Heart rate and buffer size of an application on which are activated both a performance and a power service

Figure 5.12 shows the results.

Starting from 600MHz, the frequency is scaled each ten iterations. When the frequency is set to 122MHz, the application heart rate drops below the minimum threshold. The application recognizes that the performance is below the minimum QoS and starts decreasing the buffer size. After this action, the application is able to keep its QoS and the frequency is set to the minimum value.

5.6.4 NetManager4Fluctuation

This test aims to prove the correct functioning of the service used to avoid the network fluctuation. The rationale of this service is the elimination of the unnecessary power consumption due to the changing between different networks. Usually, the preferred network type in current devices is set to *WCDMA preferred*, meaning that the device will always try to connect to the 3G net, if available. In a situation in which the user is changing many times its position in a short period of time, e.g. during a train journey, the probability that the connection will continuously fluctuate between the 3G and the EDGE net is very high. This situation causes an high power consumption and the network instability.

This service is activated when the SC recognizes a number of network changes higher than usual and the service stabilizes the connection on a lower-speed but available network. Periodically the SC has to check if a stable high-speed connection has become available.

The rationale of this service is based on data provided in [54], since at the moment is missing a precise power monitoring system. In this work data relative the power and the energy consumed when the network switches between the 2G and the 3G and vice versa are provided. The handoff between 2G and 3G requires 1389,5 mW and 2,4J, while the switching between the 3G and the 2G net requires 591,9 mW and 2,5J. It is clear that the continuos switching between different nets will cause an useless power consumption.

These data justify the usage of this service to manage the frequent changes of network type, to reduce a useless power consumption.

5.6.5 NetManager4Power

This service has the goal to select the best network to be connected to, with respect to the running applications. The best network is intended to

be the network with the lower power consumption that will ensure a minimum QoS for all the running applications. Given the list of the available networks, from GSM (2G) to HSPA (3G), they are characterized by an increasing speed but also an increasing power consumption. At he moment the WiFi connection is not taken into consideration, therefore we are dealing only with the network associated with the *rmnet* interface.

When this service is activated, the SC has to check the running applications and the current network status. The service will scale down the current network to a lower-speed network to try to reduce power consumption. As a consequence of this action, a running application may not able to provide an adequate QoS. In this case, the service will restore an higher-speed connection to ensure the QoS of all the applications. Applications performance are monitored, as usual, using the Heartbeats framework.

In this test case, we have executed on the device the VideoPlayer application, previously instrumented. The size of the video it has to download and play is 11,3 MB. Figure 5.13 shows the different heart rate of the two different execution, using only the 2G net or only the 3G net.

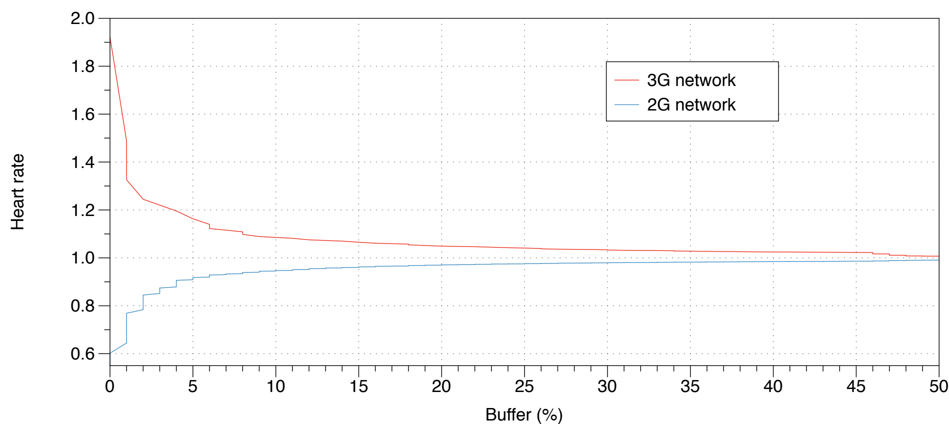


Figure 5.13: Heart rate during the buffering process, using the 2G and the 3G net

Both executions using different network are able to provide a good QoS,

the only different visible to the user is in the *prepare* phase. The MediaPlayer of the Android framework has to follow some specific steps in order to play a video. In particular, a MediaPlayer object must first complete the prepare phase before playback can be started. In this phase an initial buffer is filled, which size depends on the file size, and in this amount of time the user just sees a black screen, so it is important to reduce as much as possible this time. Figure 5.14 shows the times needed to complete the prepare phase on files of different sizes, using the 3G network and using the 2G network.

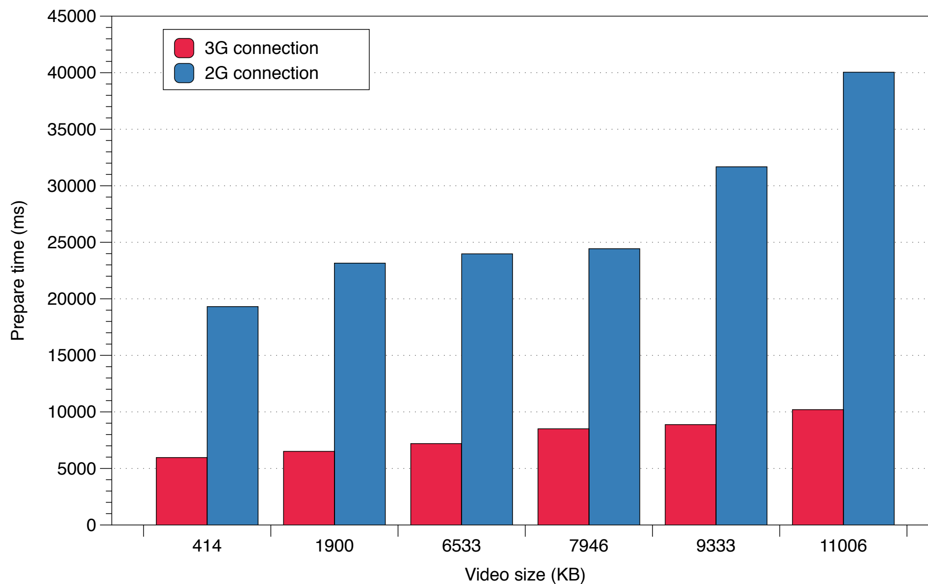


Figure 5.14: Prepare phase, using the 2G and the 3G net

While, as seen in Figure, 5.13, one the initial buffers has been filled, the user will not note any difference in the playback using 3G with respect to the playback using the 2G network.

Due to these conclusions, the policy used in this service is to check the availability of the 3G network and in that case complete the prepare phase and the first 5% of the execution through this network connection. Then, the service will complete the remaining of the buffering process using the

2G network, to save power.

Note that the service NEtManager4Fluctuation take priority over this service.

5.7 Results summarization

In this Section, the results provided in this Chapter are summarized. At first, the performance of the file based and shared memory based implementation of the heartbeats API have been compared, and results show that the shared memory based version is the best one to use.

After this choice, tests have been performed on the heartbeats framework, both using its API compiled as a native shared library and compiled as a Java library through the use of JNI. Results of these tests have proved that this monitoring framework has a low overhead in both cases with respect to the application execution times.

The set of tests on self-adaptive applications aimed to prove the effectiveness of such approach in reacting to environmental changes, in that case an external disturb. Both tests, that used different actuators, have proved the ability of such applications to keep a good QoS even in fluctuating conditions.

The last set of tests wanted to test the ability of a Services Coordinator, an external entity, to monitor both native C and Java applications, to react to both internal and external conditions and to perform adaptivity enabling a service on them, when needed. Two services employ the frequency scaler actuator, to perform different adaption actions with different goals on internal changes, while other two services use the actuator able to change the network type to react to different external conditions.

In the first group of services, the first one concentrates its efforts in saving as much power as possible, alway monitoring the applications perfor-

mance to keep a minimum QoS in all applications, while the second one wants to boost performance in applications that perform intensive tasks that have to be completed as soon as possible. Thanks to over clocking capabilities, the execution time is drastically reduced. Both services worked as expected and they have been proved effective in reaching their goals.

The second group of services is used to control the device connection status, the first service is used to avoid the network fluctuation that causes an excessive power consumption, while the second one is used to select the network with the lower power consumption given the QoS of the running applications. Both services have proved to be effective in power reduction, even if a more precise power monitoring service is needed to know the amount of power saving.

Finally, the last test shows that a tradeoff between performance and power can be reached, applying two services with different targets at the same time.

Chapter 6

Conclusions

The aim of this thesis has been to prove the effectiveness of an adaptive approach in constrained and fluctuating systems, specifically in the case of mobile devices. In this context the proposal was to implement an autonomic architecture capable of monitoring its current status and its progress towards specific goals, capable of performing optimizations on itself, and capable of adapting to unpredictable, unknown, and unfavorable conditions. To prove the effectiveness of the approach, a set of tests in different system conditions, had to be performed.

After stating the problem and its related works in Chapter 1 and showing in Chapter 2 that the best Operating System for mobile devices on which integrate our adaptive system is Android, the remaining Chapters have outlined the proposed solution, its implementation and the results obtained.

In Chapter 3 the work has been divided in two steps. The first is an analysis of the goals and different scenarios that it is possible to find in a mobile environment and what characteristics should have an autonomic system on such devices. Primary targets have been identified as performance and power and possible scenarios include self-adaptive applications and applications controlled by an external observer called Service Coordi-

nator. The second step aims to identify a minimum set of requirements and components that we need to implement in order to have a complete adaptive system. Those components include two monitors, one for performance and one for power. The former uses heartbeats API to control application performance, while the latter monitors the CPU frequency.

Usually monitors send the information to the Service Coordinator, the one in charge of monitor applications, but self-adaptive application can use the performance monitor directly, bypassing the Service Coordinator. The SC reacts to changes by activating services on application. Services currently available are two and they use the same actuator, a frequency scaler, with different targets. One acts on power consumption, the other on application performance. Self-adaptive application are not allowed to modify system parameters, so they can use only application knobs or implementation changing actuators.

All those components have been implemented and in Chapter 4 are discussed Android-specific implementation details. To port heartbeats API inside the Android framework, the Binder interface has been used to create and share a memory area between processes, since standard POSIX shared memory implementation is not available in Android. Then, since usually Android applications are written in Java, heartbeats API has been compiled into a Java library through JNI. To build a frequency scaler, the actuator used in two of the implemented services, a driver that allows changing at runtime the CPU frequency was necessary. The driver used is called CPUfreq and the actuator implementation makes use of its configuration files to change the frequency when needed.

In Chapter 5 the results of tests performed on the final system were shown. Three applications have been developed as case studies: an mp3 decoder, self-adaptive and written in C, a photo viewer and an audio-video player, Service Coordinator controlled and written in Java. Those applica-

tion have been employed to test the correct functioning of heartbeats API in both native C code and Java code. Since both versions of heartbeats API, file and shared memory based, have been ported to Android, those two versions were compared in terms of efficiency and the shared memory version resulted the fastest one.

The self-adaptive application was tested using two different actuators to prove its ability to keep a good Quality of Service in varying conditions. In the first test it was employed an application knob to change the buffer size, in the second one the actuator has the ability to change the algorithm used to decode the stream. Both tests were successful.

Services were tested using the Service Coordinator. The service FrequencyScaler4Power, used to control power consumption, was tested in conjunction with the mp3 decoder application, this time used without self-adaptive capabilities. Reducing the CPU frequency while using heartbeats information to control Quality of Service resulted in a reduction of the power used. The FrequencyScaler4Performance service, used to maximize performance, was tested using the photo viewer application. Since this application has to perform an intensive task and it has to be completed as soon as possible, the frequency scaler was used to increase the frequency at its maximum possible value, thanks to over clock. The task execution was visibly reduced. The NetManager4Fluctuation service was used to control the network fluctuation between different net types, to avoid unnecessary power consumption, while the NetManager4Power service was used to reduce the power consumption of the video player application on a streaming playback, with respect to the application QoS.

In addition, the last test shows that it is possible to reach an adequate tradeoff between performance and power targets, applying different services at the same time.

Even if there is still a lot of work to be done before the described ar-

chitecture can be considered completed to a public audience, we have designed and built an enabling technology for adaptive systems in mobile environments. An interesting future work can be shown using one of the tests of the previous Chapter, the one that uses the frequency scaler in a service whose target is performance. In that test, the frequency value was set to the maximum feasible value in order to complete a task as soon as possible. Setting the CPU in an idle state after the task completion can reduce the total application power consumption. Taking into consideration the loading time of an image with size 250KB, Figure 6.1 shown the idle time than can be gained using higher frequency values, with respect to the average case in which the frequency is set to 320MHz.

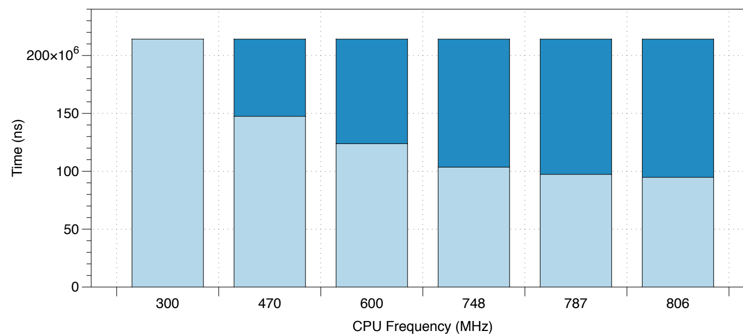


Figure 6.1: Loading time of a 250KB image at different frequencies. The idle time is computed with respect to the longest execution.

At the moment we have no means to estimate the power reduction due to setting the CPU to an idle state with respect to an execution at a specific power, but further analysis in that direction can prove if this method is useful to reduce power consumption.

An important future work will be the implementation of a power monitor able to retrieve precise information about the power consumption of a single process. Then other ideas may include implementing an actuator able to scale the CPU voltage or developing a system able to detect the kind

of device on which it is running and apply policies consequently.

Bibliography

[1] Canalys. Q1 2011, May 2011.

[2] R. Laddaga. Self-adaptive software. *DARPA BAA 98-12*, December 1997.

[3] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14:54–62, May 1999.

[4] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology. *Computing Systems*, 2007(Jan):1–40, 2001.

[5] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009.

[6] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, January 2003.

[7] IBM. [a practical guide to the ibm autonomic computing toolkit. 2004.

[8] IBM. Log and trace analyzer for autonomic computing. *AlphaWorks Release*.

[9] Cohen. Multiple architecture characterization of the linux build process with oprofile. *Proceedings of the IEEE International Workshop on Workload Characterization*, 2003.

[10] Jasmina Jancic, Christian Poellabauer, Karsten Schwan, Matthew Wolf, and Neil Bright. dproc - extensible run-time resource monitoring for cluster applications. In *International Conference on Computational Science (2)'02*, pages 894–903, 2002.

[11] D. Gonzalez-Pea and F. Fernandez-Riverola. Understanding jpda (debugging) and jvmti (profiling) java apis within javatraceit.

[12] Roy Sterritt. Pulse monitoring: Extending the health-check for the autonomic grid, 2003.

[13] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats for software performance and health. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 347–348, New York, NY, USA, 2010. ACM.

[14] J. Steven Perry. *Java Management Extensions*. O'Reilly, Beijing, 1. edition, 2002.

[15] Günter Karjoth. Access control with ibm tivoli access manager. *ACM Trans. Inf. Syst. Secur.*, 6:232–257, May 2003.

[16] Andrew Baumann, Jeremy Kerr, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in k42. In *IN 6TH LINUX.CONF.AU*, 2005.

[17] Arun Mukhija and Martin Glinz. The casa approach to autonomic applications, 2005.

[18] Gabor Karsai, Akos Ledeczi, Janos Sztipanovits, Gabor Peceli, Gyula Simon, and Tamas Kovacshazy. An approach to self-adaptive software based on supervisory control. In *Proceedings of the 2nd international conference on Self-adaptive software: applications*, IWSAS'01, pages 24–38, Berlin, Heidelberg, 2003. Springer-Verlag.

[19] J Aycock. A brief history of just-in-time. *ACM Comput. Surv*, 2003.

[20] S. Sidiroglou A. Agarwal H. Hoffmann, S. Misailovic and M. Rinard.

[21] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceeding of the 7th international conference on Autonomic computing*, ICAC '10, pages 79–88, New York, NY, USA, 2010. ACM.

[22] Hua Liu and Manish Parashar. Rule-based monitoring and steering of distributed scientific applications. *Int. J. High Perform. Comput. Netw.*, 3:272–282, December 2005.

[23] A. Boulkroune, M. Tadjine, M. M'Saad, and M. Farza. How to design a fuzzy adaptive controller based on observers for uncertain affine nonlinear systems. *Fuzzy Sets Syst.*, 159:926–948, April 2008.

[24] Frank L. Lewis and Draguna Vrabie. Reinforcement learning and adaptive dynamic programming for feedback control. *Cir. and Sys. Mag.*, 09:32–50, September 2009.

[25] Various Authors. The mit angstrom project: Universal technologies for exascale computing. Mar 2011.

[26] Santambrogio Leva Hoffmann, Maggio and Agarwal. Seec: A framework for self-aware computing. *Technical Report MIT-CSAIL-TR-2010-049, CSAIL, MIT*, October 2010.

[27] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43:76–85, April 2009.

[28] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 35–44, New York, NY, USA, 2002. ACM.

[29] Kihwan Choi, Karthik Dantu, Wei-Chung Cheng, and Massoud Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ICCAD '02, pages 732–737, New York, NY, USA, 2002. ACM.

[30] Trevor Pering, Tom Burd, and Robert Brodersen. Dynamic voltage scaling and the design of a low-power microprocessor system. In *In Power Driven Microarchitecture Workshop, attached to ISCA98*, 1998.

[31] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, MobiCom '01, pages 251–259, New York, NY, USA, 2001. ACM.

[32] Thomas D. Burd and Robert W. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of the 2000 international symposium on Low power electronics and design*, ISLPED '00, pages 9–14, New York, NY, USA, 2000. ACM.

[33] Steven J. Vaughan-Nichols. Oss battle in the smart-phone market. *Computer*, 36:10–12, June 2003.

[34] Feida Lin and Weiguo Ye. Operating system battle in the ecosystem of smartphone industry. In *Proceedings of the 2009 International Symposium on Information Engineering and Electronic Commerce*, pages 617–621, Washington, DC, USA, 2009. IEEE Computer Society.

[35] Symbian Foundation. The symbian blog.

[36] Nokia. *Nokia N8 Technical Specifications*.

[37] Ben Morris. *The Symbian OS Architecture Sourcebook*. John Wiley and Sons, 2007.

[38] Jane Sales. *Symbian OS Internals: Real-time Kernel Programming*. John Wiley and Sons, 2005.

[39] Apple Inc. ios overview.

[40] Microsoft. Windows phone 7 guide for iphone application developers.

[41] Ben Elgin. Google buys android for its mobile arsenal. *Businessweek*, 08 2005.

[42] Google Inc. What is android?

[43] Dan Bornstein. Presentation of dalvik vm internals.

[44] A. Krall. Efficient javavm just-in-time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 205–, Washington, DC, USA, 1998. IEEE Computer Society.

[45] F. Sironi, M. Triverio, Henry Hoffmann, M. Maggio, and Marco D. Santambrogio. Self-aware adaptation in fpga-based systems. In *FPL'10*, pages 187–192, 2010.

[46] Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. Smartlocks: lock acquisition scheduling for self-aware synchronization. In *Proceeding of the 7th international conference on Autonomic computing*, ICAC '10, pages 215–224, New York, NY, USA, 2010. ACM.

[47] Jonathan Eastep, David Wingate, Marco D. Santambrogio, and Anant Agarwal. Smartlocks: lock acquisition scheduling for self-aware synchronization. In *Proceeding of the 7th international conference on Autonomic computing*, ICAC '10, pages 215–224, New York, NY, USA, 2010. ACM.

[48] Paul Richardson, Larry Sieh, and Ali M. Elkateeb. Fault-tolerant adaptive scheduling for embedded real-time systems. *IEEE Micro*, 21:41–51, September 2001.

[49] J. H. Abawajy. An efficient adaptive scheduling policy for high-performance computing. *Future Gener. Comput. Syst.*, 25:364–370, March 2009.

[50] Joshua A. Wise Wei Lin. Precise power characterization of modern android devices. October 2010.

[51] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM.

[52] Android Open Source Project. Android build system.

[53] P. Rysavy. Data capabilities: Gprs to hsdpa and beyond. *Whitepaper, Rysavy Research*, September 2005.

[54] Gian Paolo Perrucci, Frank Fitzek, Giovanni Sasso, Wolfgang Kellerer, and Joerg Widmer. On the impact of 2G and 3G network usage for mobile phones' battery life. In *European Wireless 2009*, Aalborg, Denmark, 5 2009.