POLITECNICO DI MILANO
SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

COMPUTER SCIENCE AND ENGINEERING
MASTER'S THESIS

ENERGY-AWARE RUN-TIME MANAGEMENT OF DISTRIBUTED MOBILE DEVICES

Author:
dott. Michele Zanella

Student ID (Matricola):
836604

Supervisor (Relatore):
Prof. William Fornaciari

Co-Supervisor (Correlatore):
Ph.D. Giuseppe Massari

A.Y. 2015/2016

Beauty is more important in computing than anywhere else in technology because software is so complicated. Beauty is the ultimate defence against complexity

---

David Gelernter

Di imparare non si finisce mai, e quel che non si sa è sempre più importante di quello che si sa già

Gianni Rodari

Contents

List of Figures

List of Tables

Acknowledgments

Al termine di questo percorso vorrei innanzitutto ringraziare tutto il laboratorio *HeapLab* del Politecnico di Milano, gruppo di ricerca con il quale ho lavorato nel contesto e durante la stesura di questo lavoro di tesi, soprattutto per come mi ha accolto e supportato: in particolare il relatore *prof. William Fornaciari*, il correlatore *Ph.D. Giuseppe Massari* per la sua notevole e non comune disponibilità ad aiutarmi, incoraggiarmi e a sopportare settimane di correzioni, *Simone Libutti*, *Federico Terraneo* e *Federico Reghenzani*, con il quale in parte ho condiviso anche il percorso di studi.

Inoltre vorrei ringraziare i miei genitori, *Antonella* e *Lucio*, che hanno creduto nelle mie capacità, mi hanno sempre supportato con ogni mezzo durante gli studi e mi hanno dato questa possibilità. Mia nonna *Maria Luisa*, mia zia *Santina* e i miei parenti che mi hanno sempre incoraggiato in questi anni.

Un ringraziamento particolare vorrei dedicarlo a *don Ivano Tagliabue* che è stato ed è tutt'ora una guida per il mio cammino e con cui ho avuto modo di confrontarmi in questi anni. E sicuramente non posso mancare di ringraziare *Daniele*, *Davide* e *Ilaria*, *Michele*, *Simone* e *Ilaria*, innanzitutto per la loro amicizia, per i momenti e le bevute di questi anni e poi per il loro continuo supporto e sostegno, specialmente negli ultimi mesi.

Infine vorrei ringraziare anche coloro con i quali ho condiviso direttamente questi cinque anni di studio, lezioni, esami e progetti, in particolare *Alessandro*, *Andrea*, *Claudio*, *Elena*, *Massimo* e *Walter*.

Certamente se sono arrivato fin qui è anche grazie a tutti voi che siete in queste poche righe e anche a tutte le altre persone che qui non ho nominato ma che sono state presenze attive in questi anni. E come disse una volta una mia professoressa al termine del liceo: *"Ora inizia l'avventura più bella"*.

Sommario

NEGLI ultimi anni, si è osservata una crescita esponenziale del mercato dei dispositivi mobili come smartphone e tablet. In questo contesto, è rilevante osservare con quale frequenza gli utenti di smartphone sostituiscono il proprio dispositivo. Secondo uno studio dell'International Telecommunication Union, infatti, questo accade mediamente ogni 20 mesi. L'effetto collaterale di questo trend è la gestione dello smaltimento di un numero sempre crescente di dispositivi elettronici in molti casi ancora perfettamente funzionanti.

Questa tesi si pone l'obiettivo di proporre una soluzione per mitigare tale problema. Più in dettaglio crediamo che attraverso un cambio di paradigma, sia possibile perseguire un duplice obiettivo: 1) estendere la vita dei dispositivi mobili riducendo la quantità di rifiuti elettronici prodotti; 2) avere la possibilità di aumentare le prestazioni delle applicazioni attraverso l'esecuzione parallela su più dispositivi.

In quest'ottica, il paradigma proposto è caratterizzato da due elementi: 1) l'interconnessione di più dispositivi mobili in una rete locale (domestica o aziendale); 2) la presenza sui dispositivi di un software di gestione delle risorse computazionali al fine di gestire questo tipo di sistema distribuito secondo un obiettivo di massimizzazione dell'efficienza energetica. Dal punto di vista dell'utente ciò che accade è che l'interazione con un singolo dispositivo viene estesa attraverso l'utilizzo trasparente di più dispositivi.

Questa tesi presenta una estensione del framework Barbeque Run-Time Resource Manager per gestire l'esecuzione di applicazioni Android su più dispositivi. A tal riguardo viene proposta anche una estensione dei concetti di *device capability* e *features* connessi all'esecuzione di applicazioni su sistemi Android.

Il concetto centrale è quello di *Capability*, ossia una rappresentazione aggregata dello stato delle risorse, delle *Features* hardware esposte dai sistemi Android e delle condizioni energetiche e termiche del dispositivo. Questo modello introduce, quindi, una dipendenza tra le funzionalità offerte da un dispositivo e il suo stato a runtime. In tal senso la nozione di *Capability* ci permette di individuare una serie di dispositivi candidati all'esecuzione di una applicazione.

Il framework BarbequeRTRM si fa carico della selezione dei dispositivi e della gestione delle loro risorse. A tal riguardo il framework è stato esteso per poter operare in maniera distribuita.

Un ulteriore contributo è stato lo sviluppo di una politica di selezione orientata all'efficienza energetica, integrata in BarbequeRTRM. Tale politica sfrutta una caratterizzazione a priori dell'efficienza energetica delle singole applicazioni per ogni dispositivo. Per fare ciò sono state utilizzate due suite di benchmark attraverso le quali, in una seconda fase, si è effettuato inoltre un processo di classificazione di applicazioni reali.

L'implementazione attuale della soluzione proposta rappresenta una "proof-of-concept" dell'esecuzione di applicazioni mobili in maniera distribuita. I risultati sperimentali ottenuti dimostrano inoltre che, oltre ad aumentare l'utilizzo dei molteplici dispositivi mobili di cui un utente può disporre, l'approccio proposto ci permette di incrementare sensibilmente la durata della batteria dei dispositivi tra il 10% e il 30%, al costo di una perdita di performance tollerabile per applicazioni che non hanno vincoli real-time.

Abstract

In the latest years, we observed an exponential growth of the market of the mobile devices. In this scenario, it assumes particular importance the rate at which the smartphones are replaced. According to the International Telecommunicaton Union, in fact, this happens every 20 months, on average. The side effect of this trend is to deal with the disposal of an increasing amount of electronic devices which, in many cases, are still working.

This thesis aims to propose a solution to mitigate this problem. More in detail, through a change of paradigm, it is possible to achieve a two-fold objective: 1) extend the mobile devices lifetime; 2) enable a new opportunity to deploy mobile applications on distributed devices.

In this sense, the proposed paradigm is characterized by two elements: 1) the interconnection of mobile devices in a local network (domestic or corporate); 2) the availability of a resource manager software to manage the distributed system according to an energy-efficiency maximization objective. From the user-stand point the interaction with a single device is extended through a transparent use of multiple devices.

This thesis presents an extension of the Barbeque Run-Time Resource Manager framework to manage the execution of Android applications over multiple devices. In this regard, we presented an extension of the *device capability* and *features* concepts related to the execution of applications in Android systems.

The key aspect is the concept of device *Capability* which is an aggregated representation of specific resources status, available hardware Android *Features* and power conditions of the device. This model introduces a dependency between the device functionalities and its run-time status. In this sense the defini-

tion of *Capability* allows to identify a set of available devices for the execution of an application.

The BarbequeRTRM framework is the software layer in charge to select the devices for the application execution and to manage their resources. In this regard the framework has been extended to operate in a distributed fashion.

A further contribution of this thesis is the development of an energy-efficiency oriented device selection policy, implemented as a framework plugin. The policy exploits an a-priori energy-efficiency characterization of the single application execution over different devices. In order to do this, we exploited two benchmark suites from which we have: 1) extracted energy-efficiency profiles; 2) performed a classification process of real applications.

The current implementation represents a "proof-of-concept" of the execution of mobile application in a distributed context. Moreover, the obtained experimental results show that, as well as increasing the utilization of multiple mobile devices available to the single user, using an energy-efficiency approach considerably increases the device lifetime between 10% and 30%, despite a tolerable performance loss for applications that do not have real-time constraints.

CHAPTER 1

Introduction

1.1 The era of mobile devices

From 1945 to about 1985 computing systems was very expensive and bulky, their performance were very low compared to the current computers. However since 80s, development of these systems has been subject to a great speed up mainly due to two technological improvements. The first one has been the progress of the silicon industry thanks to which microprocessors with increasing capabilities have been made available with reduced costs and size with respect to previous decades. The second improvement has been the introduction of high-speed wired and wireless networks, which allowed different machines in the same building to exchange information in a few microseconds. With the passing of years, the performance gains and the miniaturization of processors have followed the *Moore's Law*[1] shown in Figure 1.1 accordingly leading to high speed technological improvements never reached in other fields.

In the latest decades, the hardware miniaturization has led to the possibility of integrating complex digital systems into battery-powered devices. From the fusion between the first mobile phones and personal mobile computers of 90s a new type of computing device has been created.

[1] Moore's law is the observation, made by Gordon Moore, that the number of transistors in a dense integrated circuit doubles approximately every eighteen/twenty-four months

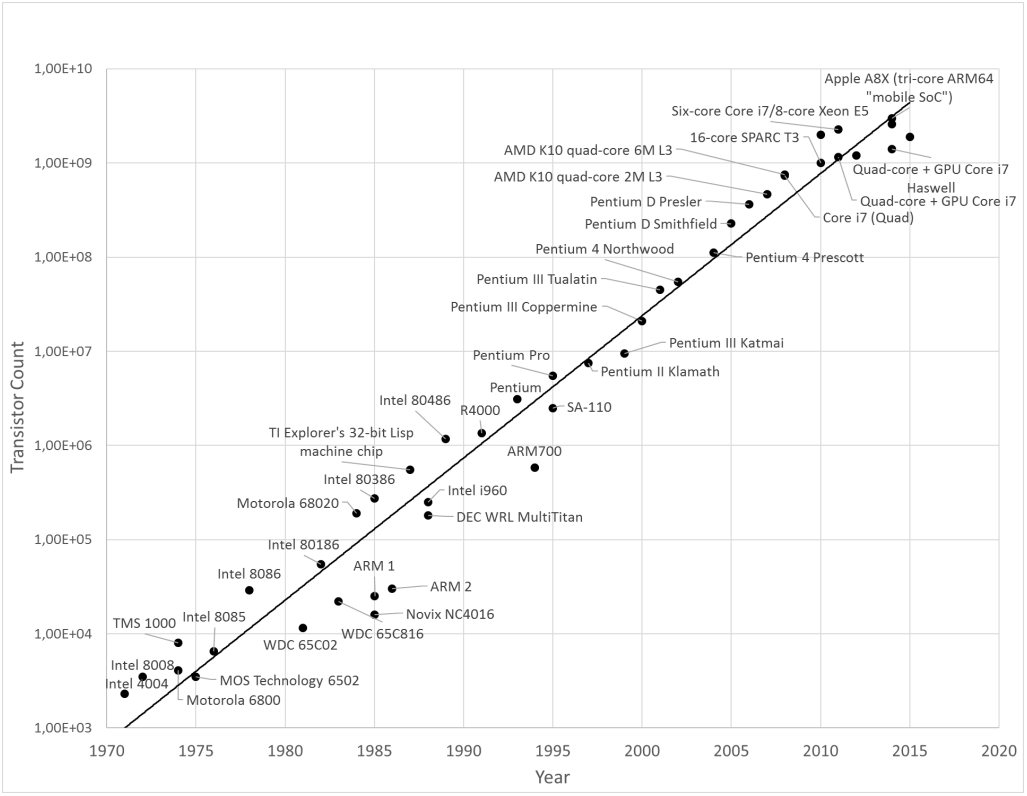


Figure 1.1: *Transistor counts for integrated circuits plotted against their dates of introduction. The curve shows Moore's law - the doubling of transistor counts every two years. The y-axis is logarithmic, so the line corresponds to exponential growth.*

Definition 1.1.1. *A **Mobile Device** is an highly portable battery powered computing system that can include a screen, some sensors, wireless network adapters and a software stack.*

The first devices, known as PDAs[2], were mainly pocket-size tablet computers, often equipped with touchscreen input panel and a wireless network interface. In the late 2000s the first smartphones made their appearance. They put together the telephony and PDA's worlds and starting a more and more expansive and competitive market.

These devices have some features in common that we can summarize as follow:

- *Portability*: mobile computing systems should enable ease of use in movable environment
- *Connectivity*: the possibility of connecting to other devices through a wireless network without affecting the Quality of Service

[2]Personal Digital Assistants

- *Interactivity*: this feature can be viewed both as the possibility that an user can interact with the device and as the possibility to enable social interaction with other users

- *Self-operativity*: each device is a self-contained system, including all the technology to supply the expected functionalities.

Nowadays mobile and personal devices are becoming more pervasive than ever. According to Gartner institute[3], global smartphones sales reached 1.423 millions units in 2015, 1.495 millions units in 2016 and 70.5% of market share. Similarly wearable devices sales counts 232 millions units in 2015 and are estimated to grow up to 322.69 millions units in 2017. Conversely PCs sales went down to 275.8 millions units from 363.8 millions in 2011.
Mobile devices represents an interesting field of study not only for its pervasiveness but also for security challenges and social factors. In fact, people use their personal mobile devices for many aspects of their life and in a lot of various context. They incentive social interaction because of their interconnection through the Internet. This latter point can be exploited by different perspectives, both in a communication-related fashion and in a computational-based one. We decided to focus on this last point, in fact the increased tasks that these devices have to perform requires manufacturers to design and implement more and more powerful and efficient System-on-Chip (SoC). The hardware and the computing resources currently available on such devices make them capable of executing multi-threaded performance-hungry applications. However, despite the grown computational capabilities, the mobile nature makes them constrained by battery technology and capacity.

Another problem that arises from such pervasiveness of mobile devices is their heterogeneity: both hardware and functional. They are present under different forms, such as smartwatch or tablets, and they are composed by a widely variety of processors, sensors and GPUs[4]. This heterogeneity is addressed by Operating Systems developers and vendors which take care of managing such hardware variety in a transparent way for end-users and applications developers. In spite of hardware market, Operating Systems (OS from now on) field is less heterogeneous and three main products dominate the global mobile market: Android OS, iOS and Windows. A worth to be considered fact is that, according to International Data Corporation (IDC)[5] Smartphone OS market Share, up to second quarter 2016 87.6% of smartphones share the Android Operating Sys-

[3] www.gartner.com
[4] Graphics Processing Unit
[5] https://www.idc.com/

tem, while considering also tablets, according to NetApplications statistics[6], the percentage is about 66.36%. Started by Andy Rubin, Rich Miner, Nick Sears and Chris White as an advanced operating systems for digital cameras it was acquired by Google Inc. in July 2015, Android OS is part of the Android Open Source Project[7] and it is based on the Linux kernel. Nowadays it is available in different releases targeting on smartphones, tablets, TVs, automotive and wearable devices.
This large spread of the same Linux-based OS across multiple devices is an enormous advantage both for applications developers and especially for our purpose.

The rapid increase of sales is connected partially to the rate at which users replace their mobile devices. Regarding this aspect a worth to be considered data comes from the International Telecommunication Union. According to them, the average rate at which smartphones are replaced by the respective owners is around 20 months. The consequence of this is an increasing amount of unused devices populating our houses or destined to disposal. In fact people often replace their personal device even if they are still working and could be exploited for some computational work.

1.1.1 The evolution of computational power in mobile devices

Model	Release Date	Clock Freq (MHz)	# of cores
Core i7-2600S (Low Power)	January 2011	2800	4
Core i7-2960XM (Notebook)	September 2011	2700	4
Core i7-2700K	October 2011	3500	4
Core i7-3960K	November 2011	3300	6

Table 1.1: *A list of some available general purpose Intel CPU's clock frequency in 2011.*

In the previous section we mentioned the high speed at which processors performance increased in few years. In particular for mobile devices the performance improvements involve different aspects.
Looking at Figure 1.2 we can see the trend of the maximum clock frequency in smartphones CPUs over the years. It appears quite clear that until 2011 the CPU clock frequency was relatively low respect to standard desktop or laptop PC processors shown in Table 1.1. Since 2012 onward, clock speed considerably increased of a factor of 1.5 and than it remained stable.

The graph in Figure 1.2 shows another change in the CPU architecture. Starting from 2011 in fact, as it happened for processors of PCs and servers, the power

[6] http://www.netapplications.com/
[7] https://source.android.com/

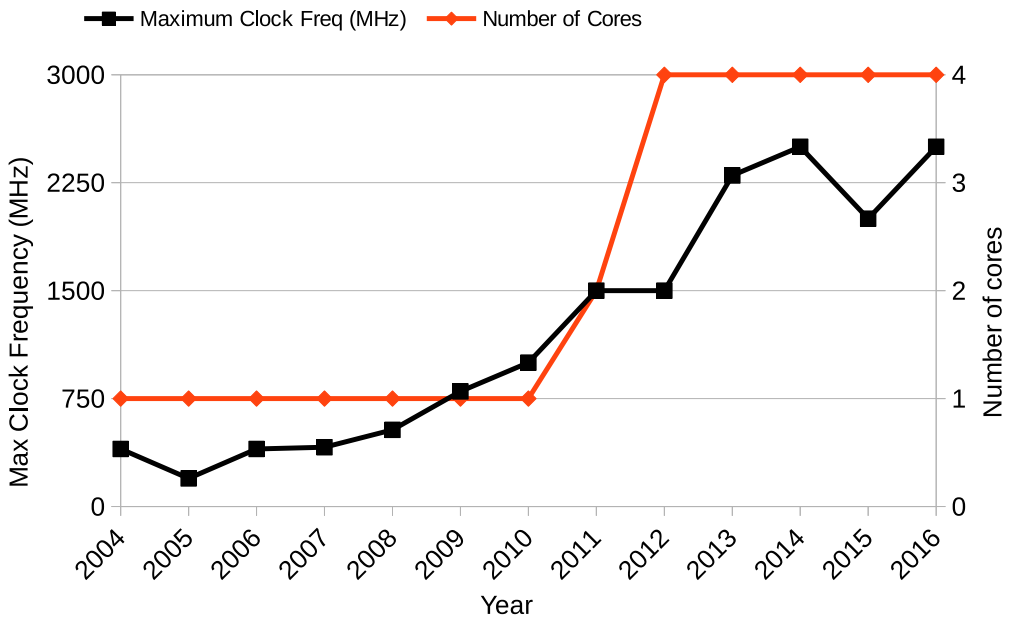


Figure 1.2: *The evolution of maximum clock frequency and number of cores of smartphone's CPUs over years.*

consumption connected to high clock speed overcame the benefit of the achieved performance. For this reason processors manufacturers changed the approach by designing and implementing parallel architectures.
Moreover, for energy efficiency purpose, some manufacturers started from 2012 to implement a SoCs[8] integrating two heterogeneous single-ISA multi-core processors: a low power (usually called "LITTLE") and a high-performance one (called "big"). An example of this implementation is the Samsung Exynos 5 Octa SoC, featuring a 1.8 GHz quad-core ARM Cortex-A15 as a big CPU cluster and a 1.2 GHz quad-core ARM Cortex-A7 as LITTLE CPU cluster (e.g., ARM big.LITTLE [2]).
Table 1.2 shows a list of smarthpones with the related SoC and processors specifications.

The frequency clock increase was accompanied by improvements on chip architectures that augmented the Floating-Point Operations per cycle (from now on FLOPs). These two improvements enable the possibility to make smartphones processors able to perform billions FLOPs per second consuming less power. For example, known the equation of FLOPs per second [3]

$$FLOPS = sockets * \frac{cores}{socket} * clock * \frac{FLOPs}{cycle} \tag{1.1}$$

[8] A System-on-a-Chip is a circuit that integrates all components and peripherals into a single chip [1].

Model	Year	SoC	Big Little	CPU	Clock Freq (MHz)	# of cores
Samsung Galaxy S7	2016	Snapdragon 820	✓	2.15 GHz Kryo	2150	2
				1.44 GHz Kryo	1440	2
Huawei P9	2016	HiSilicon Kirin 955		2.5 GHz Cortex-A72	2500	4
Apple iPhone 7	2016	Apple A10 Fusion	✓	2.34 GHz ARMv8-A	2340	2
				Apple "Zephyr"		2
HTC One M9	2015	Snapdragon 810	✓	1.5 GHz Cortex-A53	2000	4
				2 GHz Cortex-A57	2000	4
Apple iPhone 6S	2015	Apple A9		1.85 GHz ARMv8-A "Twister"	1850	2
Samsung Galaxy S6	2015	Samsung Exynos 7 Octa 7420	✓	2.1 GHz Cortex-A57	2100	4
				1.5 GHz Cortex-A53	1500	4
HTC One M8	2014	Snapdragon 801		2.5 GHz Krait 400	2500	4
Apple iPhone 6	2014	Apple A8		1.4 GHz ARMv8-A "Typhoon"	1400	2
Samsung Galaxy S5	2014	Samsung Exynos 5 Octa 5422	✓	1.9 GHz Cortex-A15	1900	4
				1.3 GHz Cortex-A7	1300	4
LG Nexus 5	2013	Snapdragon 800		2.3 GHz Krait 400	2270	4
Apple iPhone 5S	2013	Apple A7		1.3 GHz Apple Cyclone	1300	2
Samsung Galaxy S4	2013	Samsung Exynos 5 Octa 5410	✓	1.6 GHz Cortex-A15	1600	4
				1.2 GHz Cortex-A7	1200	4
LG Nexus 4	2012	Snapdragon S4 Pro		1.5 GHz Krait	1500	4
Samsung Galaxy S3	2012	Samsung Exynos		1.4 GHz Cortex-A9	1400	4
Apple iPhone 4S	2011	Apple A5		800 MHz ARM Cortex-A9	800	2
Samsung Galaxy S2	2011	Samsung Exynos		1.5 GHz dual-core ARM Cortex-A9	1500	2
LG Nexus One	2010	Snapdragon QSD8250		1 GHz Qualcomm Scorpion	1000	1
Nokia C5	2010	/		600 MHz ARM11	600	1
Apple iPhone 3GS	2009	Samsung S5PC100		600 MHz ARM Cortex-A8	600	1
HTC Dream	2008	/		528 MHz Qualcomm MSM7201A ARM11	528	1
Apple iPhone	2007	/		412 MHz ARM11 76JZF-S	412	1

Table 1.2: *The bestseller smartphones SoC and processor architecture specification from 2007 up to 2016*

a Quad-core Cortex-A57 with 2.1 GHz clock frequency executes 8 Single Precision FLOPs at cycle, that theoretically is $67.2\text{x}10^9$ FLOPS.

The performance reached by these devices can be transposed also for tablets and partly to other mobile devices, so that they can be exploited to perform also computational intensive workloads as it will be explained in the next chapters.

1.2 Distributed systems

Over years a lot of definitions of distributed system has been provided, Tanenbaum and Van Steen [4] proposed the following one that we took for our treatment:

Definition 1.2.1. *A **Distribued System** is a collection of independent hardware systems that appear as a single coherent system to the user.*

The main characteristic that is specified by this definition is that the distributed nature of the system is transparent to the user. This last does not know the architecture or the heterogeneous components of the system, because it seems to act as a whole, unique system.

As well as the definition of a distributed system, other two related concepts can be introduced from the literature: the notion of Distributed Computing and Distributed Networking.

Definition 1.2.2. ***Distributed Computing** also refers to the use of distributed*

systems to solve computational problems. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers. [5]

Definition 1.2.3. ***Distributed Networking*** *is a distributed computing network system, said to be "distributed" when the computer programming and the data to be worked on are spread out over more than one computer. Usually, this is implemented over a network. [6]*

The study of distributed systems started in 1960s, where concurrent processes communicated by message-passing infrastructures. The development of technologies for the creation and the configuration of local-area networks (LAN) represented an important factor that contributed to the spread of these kind of systems.
The advent of powerful multi-core microprocessors slew down the interest in distributed systems to achieve parallelization and multi-threading, due to the fact that they were more complex to manage and had some critical issues about data consistency and power consumption. However, when the physical and power limits of silicon manufacturing process began to appear and, as already mentioned in Section 1.1, parallel architectures gained interest, the idea of distributing computational tasks over different interconnected systems returned fully in the spotlight of both research and industry.
Moreover, the advent of Internet with the possibility of having a unique global network increased drastically the development of distributed systems. The Internet itself indeed can be seen as a huge distributed system composed by registered Web Servers and hosts. The Figure 1.3 shows the growth of DNS[9] advertised hosts over years according to the latest Internet Domain Survey of the Internet Systems Consortium (January 2016) [7], while in 1993 there were about 2 million hosts registered, up to the survey's date there were about 1 billion hosts, with a growth rate of about 10% per year until 2015.

Getting actual advantages from distributed systems requires some properties to hold:

- *Accessibility*: it refers to the possibility to access and share "resources" in an user-friendly, efficient and controlled way
- *Transparency*: as aforementioned it means that the system does not show its distributed nature and appears as a single system to the users. The transparency affects different sides of a distributed system, so we can have different types of transparency: *access, location, migration, relocation, replication, concurrency, failure, persistence*

[9] Domain Name System

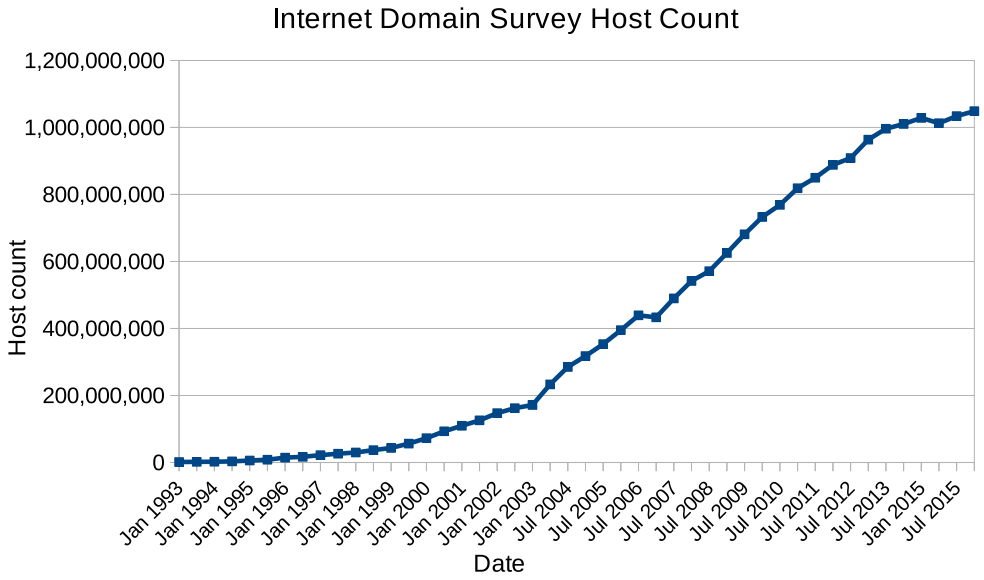


Figure 1.3: *Number of hosts advertised in DNS through years. Data source: Internet Systems Consortium [7].*

- *Openness*: it relies on the concept of interface definition. Interfaces allow to define only the service that are available into a system without further information on how the service is implemented, so that different entities can expand the system by re-implementing services in another ways. This kind of property enables entities interoperability and applications portability
- *Scalability*: at first it involves the capability and ease of the system to adapt its size according to the number of resources or users. Secondly it requires that a system can be scalable in term of geographic and administrative span. The major challenge on this objective is to find a good trade-off between scale and performance
- *Maintainability*: it requires an high grade of ease at which a system can be updated, upgraded or fixed. How a complex system is designed and its modularity degree have a severe impact on this objective.

Since nodes of a distributed systems can be heterogeneous both on the software and the hardware side, the aforementioned properties require the implementation of several abstraction layers. In Figure 1.4 we can see that the upper layer is made of applications. This is the layer that interacts with the users. Then the lower layers are constituted by the operating systems and low-level communication modules. Halfway between these layers the *middleware* represents the logical implementation of the distributed system from the application perspec-

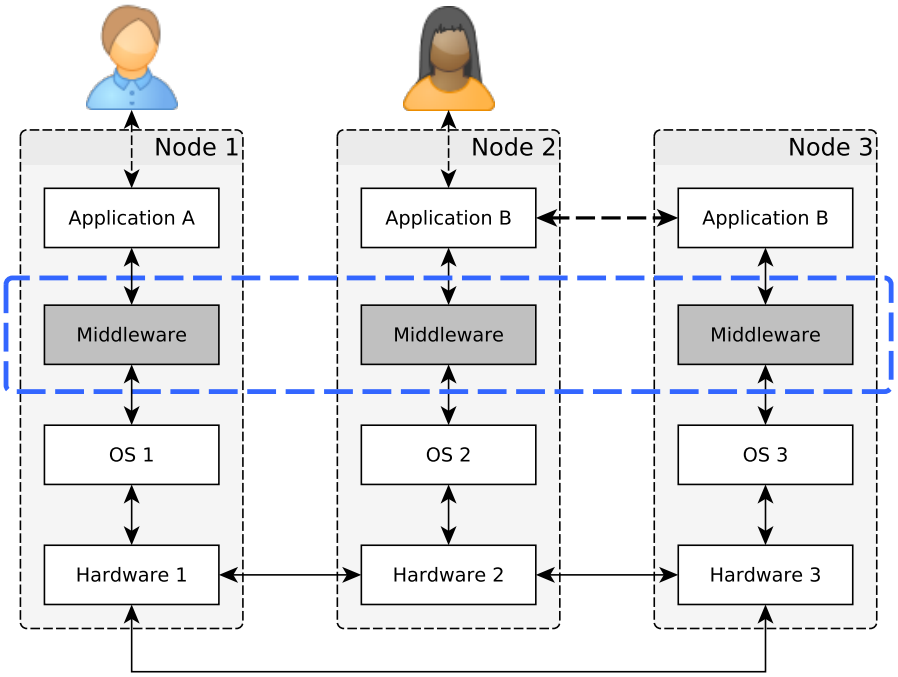


Figure 1.4: *The layers of a distributed system with a middleware that exposes the same interface to all applications on different nodes.*

tive. It provides in fact the communication interface to the applications and hides the underlying hardware and OS heterogeneity. The figure shows different nodes featuring different hardware and OSs, running a not distributed application (Application 1) and a distributed one (Application 2). The user of the second application is not aware of the fact that the execution of the application is performed on multiple nodes. Furthermore, the instances of the second application communicates to each other through the same interface exposed by the middleware layer which directly interact with the underlying OSs.

Among the different typologies of distributed systems, for our purpose we will focus on and combine two of them. The first type is the distributed *computing* system, which main goal is to execute computational intensive tasks with high performance requirements. Generally speaking such a system can be *cluster-based* or *grid-based*, depending on whether the set of connected machines is in the same network, sharing the same hardware and OS configuration, or it is spread across different administrative domains with heterogeneous hardware and software configurations [8].

The second type we consider is the so-called *pervasive* (or ubiquitous) distributed system: the main characteristic is that the system is part of the environment and the computing can occur using any device, anytime and everywhere without a

direct human control a part from the initial setting. Grimm et al. [9] indicate three main requirements for pervasive systems:

1. *Context-awareness*: the system is aware of the continuous changes of the environment in which it works and reacts consequently
2. *Versatility and flexibility*: different users can use different devices in different ways
3. *Sharing as default*: easiness of accessing, storing and sharing the acquired information.

Typical scenarios for these systems are the domestic and health care ones. It is increasingly common to have a lot of interconnected devices (such smartphones, PDAs, smart TVs, lights and sensors network) in people's domestic context that collaborate as an unique system to control and manage the environment or to assist the user through, for example, *recommender* systems. Part of this technology exploits the well-known *Internet of Things*.
On the other hand the low cost and size of such devices allow them to be exploited for health care purpose to constantly monitor the health status of patients without the need of hospitalization, maintaining them in a life context that it is as normal as possible. Anyway, one of the main challenges of the pervasive systems is the instability of its components, because they are mobile or embedded devices.
Finally, the other types of distributed systems that we do not treat include, among others, *distributed information* systems to manage large quantity of data.

For our work we combined the two aforementioned types to exploit the advantages of a *Pervasive Distributed Computing System*.

1.3 Mobile devices and Pervasive Distributed Computing System

As previously said the pervasiveness of personal mobile devices is very strong and their number is continuously growing. This leads to an high rate of devices renewal and abandonment as well as an electronic device disposal problem.
Despite their different hardware capabilities (e.g., screen resolution, camera quality, sensor availability, ...) a set of these devices can share the same operating system (e.g., Android) such that it is possible to rely on the same software stack on a wide variety of devices, allowing us to seamlessly deploy applications on a device or another.

In such a scenario we should also take into account the typical constraints of mobile devices, concerning thermal and energy management. Most of them are

powered by batteries which have a limited charge-time and they need to be used sparingly, or maybe they consume a lot of power and they should be used only in case of real need. Furthermore mobile devices are usually fan-less devices so there is also a thermal management mandatory task to be considered. Finally, as addressed by Subsection 1.1.1, modern devices include multi-core CPUs and embedded GPU that can be exploited at different levels to perform computational tasks.
In this regard, a distributed rearrangement of mobile devices can lead to improvements in terms of energy efficiency and heat dissipation, thanks to task offloading and dynamic load balancing.

These considerations lead to a change of paradigm in which different personal devices are interconnected to configure a single distributed mobile systems to achieve a two-fold objective:

1. Extend the device lifetime by reusing old and abandoned devices to embrace *Green computing* purposes

2. Enable a new opportunity to deploy mobile applications on distributed devices.

This vision aims at maximize the device utilization but also to scale application performance through multiple nodes (i.e., devices) exploiting by smart device management policies taking into account also thermal and energy budget management objectives. To meet these requirements we formulated our definition of a *Pervasive Distributed Computing System* starting from the definition of Distributed System, Distributed Computing and Distributed Network presented previously:

Definition 1.3.1. *A **Pervasive Distributed Computing Network System** is a distributed system of interconnected personal devices of a network, in which the main goal is to distribute or migrate the computational load between them.*

The Figure 1.5, shows an implementation of such a system exploiting different personal devices such as smartphones, Desktop and laptop PCs, Android TV and tablets.

In a scenario like this, we could start addressing problems like defining a novel programming paradigm and design a inter-device run-time layer in charge of handling the task placement in a distributed fashion. In Chapter 2 we will discuss the *task offloading* that is a typical approach that could be enhanced by our work.

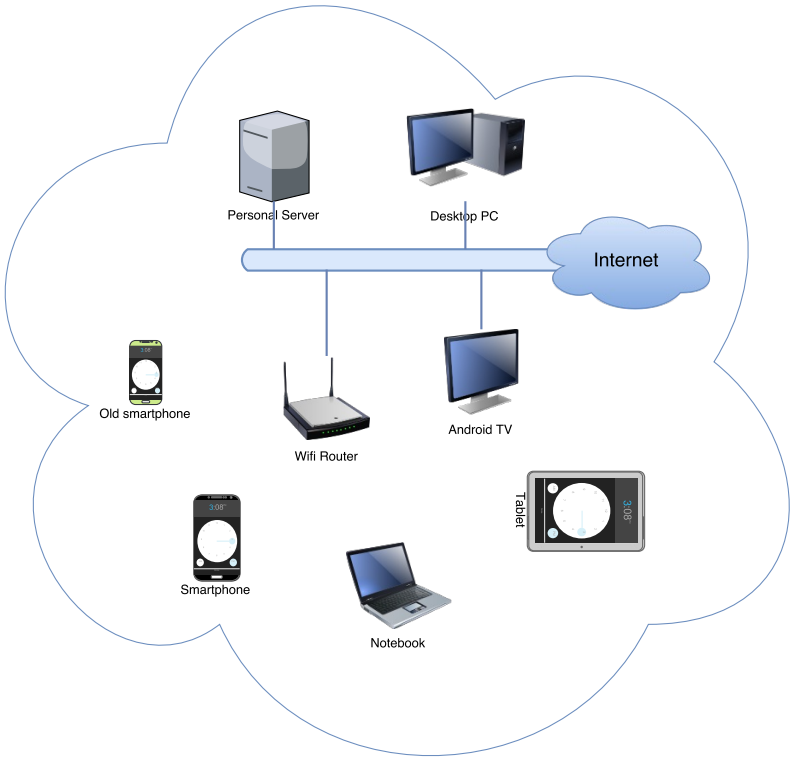


Figure 1.5: *The representation of a Pervasive Distributed Computing System in a Local Area Network context.*

1.3.1 Challenges of a Pervasive Distributed Computing System

There are some practical challenges and technical issues that we need to overcome in order to effectively exploits devices in a distributed manner.
First of all the distributed nature of the system requires a specific management strategy. In particular how the device network is created, how devices communicate in the network and, last but not least, how the decisions are taken are three requirements that play an important role in the system design.
The network creation involves the devices discovery and negotiation protocol and the topology of the network. When the network is created and a set of devices are connected they should be able to communicate through a specific protocol (e.g., RPC[10], Message-Passing, stream, multicasting ...).
Then, from the run-time management perspective it is important to define if the system must operate according to a centralized, a fully-distributed or an hybrid architecture. While the first is less complex to manage because one node acts

[10]Remote Procedure Call

as a coordinator for other nodes, the others two solutions are more flexible but challenging. As it will be exposed in Subsection 4.2.4 we choose an hybrid architecture for our purposes, but we expect to develop our run-time management strategy in order to support all the three configurations.

The aforementioned challenges are in common among all the distributed system, being each mobile device a remote system (node). However, for distributed systems based on mobile devices, the run-time management strategy must take into account the not uniform distribution of *capabilities* and the current *status* of all the available devices.
Moreover, we must guarantee a proper synchronization (e.g., avoiding resource starvation, efficient election algorithms), consistent executions and fault- tolerance, given the unreliable nature of battery-powered devices [10, 11]. All this without breaking the security requirements, which must be considered an hard constraint.

1.4 Android Operating System

In Section 1.1 we mentioned the Android OS as the most spread mobile OS. In this section, we want to give a briefly overview of its architecture, since it is the software platform on top of which we developed our approach.
The Android OS was initially developed by Andy Rubin, Rich Miner, Nick Sears and Chris White by October 2003 and then it was acquired by Google Inc. in July 2005, that made its source code released under an open source license to encourage large community to develop community-driven projects and for bug or flaws finding purpose.
The Android OS has an high flexibility and portability due to the fact that its kernel is Linux-based. This is also one of the motivation behind the choice to use Android device for our purposes.
In May 2015 Google announced Brillo, a lightweight version of Android that uses only lower levels of the software stack without providing a GUI to target on the "Internet of Things" (IoT) embedded systems.

As we can see in Figure 1.6 there are four main layers in the Android software stack. The lower one is the aforementioned *Linux kernel* with Android-specific extension for memory, power and device drivers management.
On top of the kernel there are the *middleware layer*. This layer is composed by native Libraries and APIs. For example we can find a custom *libc* library developed by Google specifically for Android within the *Bionic* library project[11] and the *SQLite* library.

[11] https://android.googlesource.com/platform/bionic/

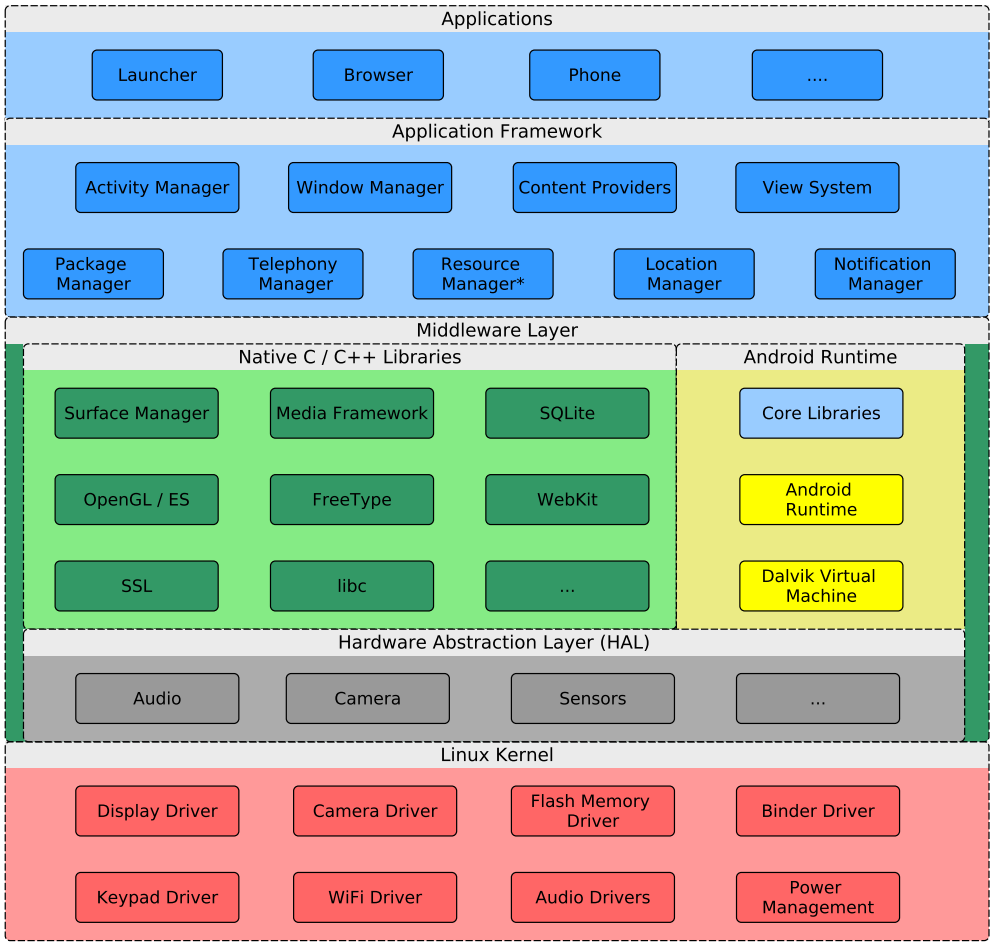


Figure 1.6: *The Android software architecture. The marked* Resource Manager *is a singular case of homonymy, because it refers to application's images or strings resources and not to system hardware resources.*

Although the *Android Native Development Kit (NDK)* enabled support for applications written completely in C or C++ languages, the OS does not support the full set of standard GNU libraries of standard Linux systems.
Included in the middleware layer there is also the run-time environment that is based on a set of virtual machines architecture that are in charge to run the Java-coded applications. Until version 5.0, the process virtual machine was the Dalvik one, which performed a just-in-time compilation of the Java bytecode to a Dalvik Executable code. Since version 4.4 (optionally) and version 5.0 (officially), the *Android Runtime (ART)* uses an ahead-of-time compilation of the application bytecode into machine code at the installation stage. At time of writing the Java Core libraries of the virtual machine are still based on the Apache Harmony project [12].

[12] https://harmony.apache.org/

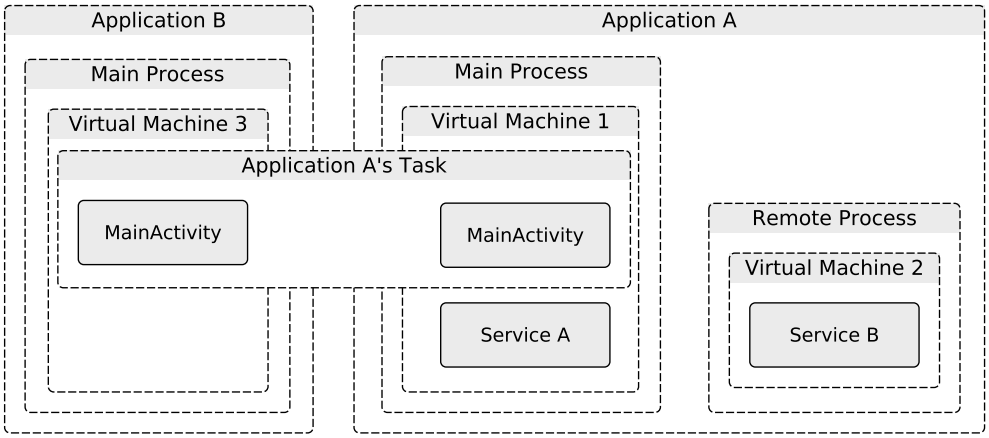


Figure 1.7: *Processes and tasks management in Android. It highlights the difference between* processes *and* tasks *in the Android terminology.*

Finally, the Hardware Abstraction Layer (HAL) completes the middleware layer. It provides standard interfaces for specific type of hardware components to the higher-level Java layer.

The two top layers consist of Java based software, i.e. the user-level applications and the underlying application framework. The latter provides a set of managers and interfaces that are exploited by the application developers to execute the code in the Android environment and get access to the OS features and the hardware capabilities of the device.
On top there is the application layer that contains all the Android applications. These can be classified according to their installation mode in two types: the *System Apps* and the third-party applications. The former are included with the platform and providing also various functionalities that can be accessed also by the other type applications.

Comparing the execution of Linux and Android applications, one main difference regards the control of their life cycle. In the Android environment in fact the applications must implement a set of callback methods representing a specific execution status, that the OS calls according to events due to the user interaction. Moreover, there are little differences on the processes, virtual machines and tasks management [12]. In particular on Android a Java virtual machine is instantiated for the execution of a process and every component of an application is associated with exactly one process, that is the default process for all components. Anyway an application can spawn some of its component to more processes or exploiting multi-threading in the same process. *Tasks* otherwise contain only a particular type of component (i.e., activities) but they can belong to different application processes. Tasks can be seen in the applications overview screen and

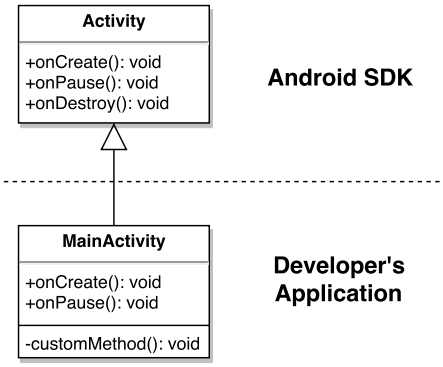


Figure 1.8: *To create a new activity a developer has to extend the Activity base class provided by the Android SDK.*

they are used basically as a logical history of user action or when an application uses the feature of another application creating an instance of them. For example in Figure 1.7 Application A has its default process and a remote process in which a background service is running. Application A's task contains its activity component. Furthermore the Application A uses a feature of a System Application B and so the OS create the default process and an instance of the activity of the Application B. Then this new activity is added to the Application A's task.

Finally, we mentioned the fact that an application is composed by different components. Such component can be distinguished mainly between two types: *Activity* and *Service*.
Briefly an Activity is a single screen with a user interface [13]. Applications usually are composed by different activities to allow user to perform different specific operation. Anyway these activities are all independent to each other and can be started multiple times by other applications. To create a new activity, developers have to extend the Activity base class overriding its lifecycle methods if needed as shown by the class diagram of Figure 1.8.
Conversely a Service has no GUI but they can interact with the user through notifications and they are designed to perform background operations for an indefinite amount of time. To avoid exceeding in details Services are deeply exposed in the Appendix A.3.

Android Power Management

Being Android a system targeting on mobile devices it comes that power management is one of the most features that the OS must provide. As shown by Figure 1.9 the Android power management strategy acts on a multi layer basis. At kernel level for instance the power management infrastructure embeds the Linux Power Management framework and adds some customization with the

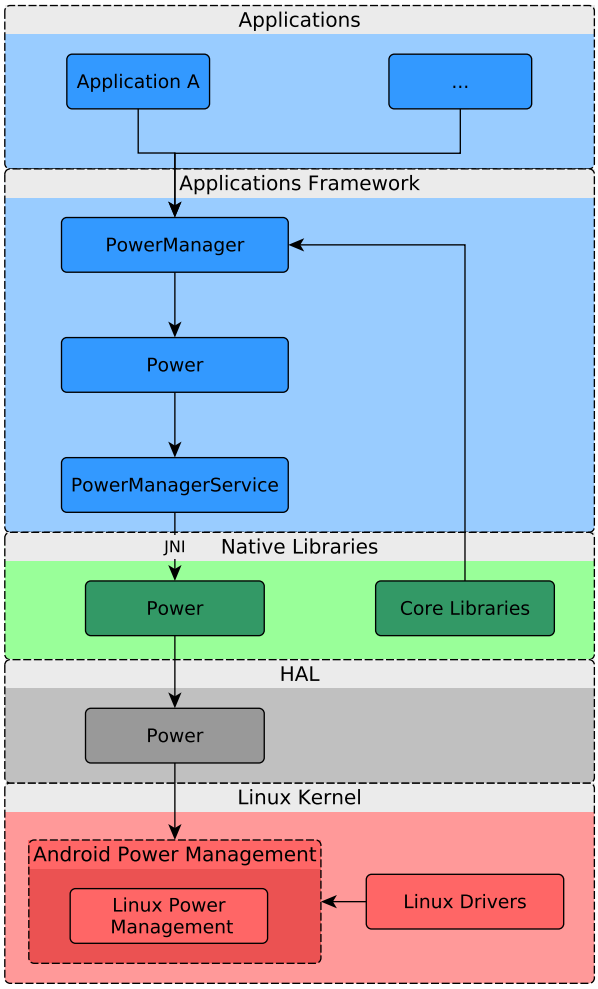


Figure 1.9: *The interaction between the power management modules through different layers.*

objective of minimizing the system power consumption by switching off unused components.

The entire management strategy is based on *wake locks*: applications must request CPU resource through the API framework and native Linux libraries in order to keep processor on, otherwise Android tries to take the CPU or peripherals into a suspend state. There are four type of wake locks that an application can request, the hardware components are then managed according to the type of locks still active:

- `PARTIAL_WAKE_LOCK`: it ensures that the CPU is on
- `SCREEN_DIM_WAKE_LOCK`: it ensures that the screen is on
- `SCREEN_BRIGHT_WAKE_LOCK`: it ensures that the screen brightness is

on at full

- `FULL_WAKE_LOCK`: it ensures that the full device is on, including backlight and screen.

To allow applications to interact with the power driver and wake lock features, Android exposes a driver Java class in its Application Framework layer called *Power Manager*. Further details on wake locks architecture can be found in [14–16].

We presented the power management of Android to give a completed treatment of the platform and to make the reader aware of the state-of-the-art system architecture related to power and energy topics. We are conscious about the power management issues related to thermal and performance that can be further investigate, but in this work we concentrated only on the energy point of view.

1.5 Thesis structure, novelty and contribution

This thesis aims to propose a change of paradigm to give an answer to some of the aforementioned challenges of a Pervasive Distributed Computing System. In particular the proposed paradigm is characterized by two elements:

1. The interconnection of mobile devices in a local network (domestic or corporate)
2. The availability of a resource manager software to manage the distributed system according to an energy-efficiency maximization objective.

In this sense the thesis presents an extension of the *Barbeque Run-Time Resource Manager* [17] to manage the execution of Android applications. The key aspect is the concept of *device Capability* that introduces a dependency between the device functionalities and its run-time status.

One of the main problem managing the execution of Android applications is that they are independent from our management system. Our solution ensures backward compatibility with pre-existing applications and allows applications developers not to change the way in which they make Android applications. To do this, we propose an *application profiler* that links the application information and required functionalities defined in the *manifest.xml* file and the *Capabilities* concept.

Finally, we enable the applications remote launching scenario across multiple Android devices by implementing a distributed version of the BarbequeRTRM that exploits an energy-efficiency oriented policy to select the device in which the application is executed. We have profiled two benchmark suites to characterize

the energy-efficiency of the single applications, proposing a classification model for non-benchmark applications.

To summarize, the novelty contributions are:

- Porting a distributed run-time resource manager on Android devices to manage local resource allocation and to monitor device status
- Enabling the management of the execution of Android applications through the novel concept of *Capability*
- Exploiting applications remote launching through an energy-efficiency oriented policy
- Proposing a classification of the Android application based on their energy efficiency profile
- Achieving a first step towards a *green* approach for computational tasks in an environment pervaded by personal devices, enabling users to reuse their available devices
- Enabling future more complex techniques of application's tasks offloading in a Pervasive Distributed Computing System of Android devices.

In Chapter 2 it is presented the state of the art regarding the mobile distributed computing approaches, the computation offloading for mobile systems and the resource management, highlighting the similitudes, limits and novelty with respect to our solution. The Chapter 3 exposes the system design, including the concept of Capability, the application profiler and the resource-aware abstract model to manage Capabilities, Android applications and resources. Furthermore, Chapter 4 presents the integration of our model with the BarbequeRTRM, highlighting the system architecture and the device selection policy. Moreover it presents an Application Programming Interface (API) that can be used by third-party application developers to interact with our system. In this sense the Chapter exposes some uses-cases enabled by our thesis. The experimental setup and results are exposed in the Chapter 5 focusing on the profiling of applications energy efficiency, the effects introduced by an energy-efficiency resource management approach and the testing and performance measurements of the policy. Finally, the Chapter 6 contains the thesis conclusions. It also discusses about challenges and improvements that can be covered by future works.

CHAPTER 2

State of the Art

This Chapter presents, with an historical and taxonomic approach, the state of the art related to the issues mentioned in the previous chapter. In particular the first part, covered by Section 2.1, concerns the main approach related to the mobile distributed computing: we present the main architectural solutions that enable the use of mobile devices as distributed system.

The second part, covered by Section 2.2, presents a survey of the main techniques of applications energy and byte-code profiling used by refactoring tools to enable the computation offloading. It presents also some real implemented offloading systems.

Finally, the Section 2.3, exposed briefly an overview of some works related to the resource management problem both in the mobile and distributed fields.

2.1 Mobile distributed computing approaches

In recent years early studies began to explore the idea of implementing the distributed computing paradigm in systems based on mobile devices [18]. The approaches and depicted scenarios are however quite different from each other.

The wireless network improvements led researchers to create distributed systems that cooperate in computational-intensive tasks and to coin some paradigms.

For example, in the *opportunistic computing* paradigm [19, 20], mobile devices are connected in an ad-hoc local wireless network to take advantage of computing resources of other devices. The most recent proposal in this direction is the AnyRun Computing (ARC) [21] system, which dynamically selects the best device for offloading the execution of tasks.

In what follows, we present some recent and noticeable solutions for mobile systems that exploit some well-known distributed infrastructures and approaches. They can be divided mainly in five categories, based on the computing paradigm they exploit:

- Transparent Computing
- Flexible Computing
- Voluntary Computing
- Enterprise Computing
- High-Performance Computing
- Mirroring.

It is also common to find service-oriented architectures as they enable an high level of transparency for the user, and they are easily extensible with new services and interfaces. An example of a service-oriented approach can be found in [22]. The authors proposed a Web Service Initiation Protocol (WIP) integrated in Android, making the device a web service SOA-based platform with real-time communication capabilities. This solution uses a proprietary application, the 2SAP, to perform the service discovery and registration. It does not allow to include developer-extension of the application, and the services can be managed only through the given application proxy. Moreover, the solution does not consider the availability of computing resources and energy budget, because the exposure of the service is not linked to the capabilities of the device.

2.1.1 Transparent Computing

A service-oriented approach that must be aware of the system resources is presented in [23], where the concept of transparent computing [24] is transposed to the mobile world. Its goal is to provide users with transparent services: users only concern whether they can get the service or not, but no need to know the underlying details. To do this, the solution requires a lightweight terminal without an operative system installed in advance. The software stack, including the operating system, the applications and the data is downloaded from a remote

server as a virtual machine, on the basis of the user requirements and the computing resources availability. In terms of security there is also the possibility of introducing different authority provisioning to different resources and services. Unfortunately, this approach requires the devices to be configured in an Internet connected and lightweight mode. Therefore stock devices are not suitable. Furthermore, there is no mention of energy-aware run-time management of the device.

The distributed aspect of this approach is given by the possibility of having the same services available for different terminals. However this is not properly what we intend as "distributed computing", because the computation is not distributed among devices but, more precisely, it is done locally once the service or the application is loaded on the single device.

2.1.2 Elastic Personal Computing

Based on the concept of *Flexible Computing* [25] this paradigm takes advantages of interconnected devices, basing on the fact that in many cases processing data in-place and exchanging them directly between devices can overcome bandwidth limitations, hence resulting in a more efficient approach with respect to offloading the entire job to a remote server.

Daniel Dìaz-Sàanchez et al. [26] proposed the Light Weight Map Reduce (LWMR) framework that enables the possibility of submitting a job by any device or group of devices, collecting the outcome and delegate tasks to other devices upon battery, network or location changes. This refines the Elastic Computing concept exposed in [27], providing a mobile version of the Hadoop MapReduce framework.

A previous work is the Hyrax system [28] that allows computing jobs distribution porting the Hadoop on an Android interconnected devices network. The centralized-architecture limitations of Hyrax are then overcame by MC^2 [29], which makes possible to setup of personal cloud computing systems made by nearby mobile devices. MC^2 provides the possibility to create private or public cloud service.

Elespuru [30] instead, investigates the feasibility of using smart mobile devices in a MapReduce system. The author implements a client-server MapReduce system for mobile devices which shows that devices are capable of performing at roughly an order of magnitude slower than the traditional clients, demonstrating that a large portion of processing can be moved to them, if enough exist at a given time to perform the necessary workload.

Finally, another worth to be mentioned work is GEMCloud [31], whose main

objective is to exploit mobile devices to execute computationally intensive and parallel tasks with a high degree of energy efficiency. The system is made by a central server and a database in charge of discovering available devices onto which deploying tasks. On the device side, a client application makes the devices visible or not, according to the device status, e.g. CPU and memory usage, battery level, and running applications. However what it is still missing in this solution is the possibility of implementing a task placement or scheduling policy, aiming at optimizing a metric, like performance maximization or energy consumption minimization.

The main difference of these works from our solution is that we do not want only to split and submit jobs to a device network and collect the outcome, but also we want to perform a complete launching of a task or application into another device that is more energy efficient and that has the capabilities to run it.

2.1.3 Volunteer Computing

The so-called *volunteer computing* [32] started in 1996 and the term had been introduced by Luis F. G. Sarmenta. In this approach users make their devices available for hosting external computational intensive tasks. It became more and more attractive for the users so that some projects received considerable media attention, such as SETI@home [33] and Folding@home [34].

The most representative framework enabling this paradigm is BOINC [35], started as a project for researchers to exploit processing power of personal computer around the world it has been extended by the NativeBOINC project [36] for Android devices. Similarly [37] links the BOINC middleware and the concept of volunteer computing to the mobile world.

In [38] the volunteer computing paradigm has been extended to mobile devices not connected to Internet, exploiting *WiFi Direct* to setup point-to-point connections. The device (node) can therefore become a distribution point or a simple proxy node towards Internet. The main goal of the solution is to extend the task distribution network, with an eye on device applications and resources management.

Another extension of the volunteer computing paradigm is REPC [39], a generic "randomized" task assignment framework that exploits mobile devices for participatory computing. REPC is another example of server-based distributed system. A centralized server in fact, hosts the execution of a Task Manager, in charge of assigning tasks to participatory devices. The overall goal is to guarantee the completion of a given minimal number of tasks, minimizing the number of tasks assigned per device. The central server is involved also in the

estimation of the run-time statistics regarding the tasks execution.

2.1.4 Enterprise Computation

The idea of using mobile devices for distributing computational load has found interest also in the enterprise world. Arslan et *al.* in [40, 41] proposed a distributed computing infrastructures using smartphones in enterprise context. The main idea is to use the enterprise devices of employees to perform computation while they are recharging, instead of the company servers, in order to reduce energy consumption and costs. Although the solution is quite complete and takes into account also device computation capability and power status, the application context is closely linked to enterprise workloads. Moreover, the client-server architecture represents a limitation in terms of scalability and flexibility. Another lack of this solution is the fact that the computation capability of the device is evaluated by the server by estimating the task completion time from previous tasks execution. It does not rely instead on a resource manager instance, running on the device that can expose capabilities and perform local optimizations, taking into account also the workload launched on the device by the user. Moreover, the recharging status is the only condition for which the mobile device is considered available.

2.1.5 High-Performance Computing (HPC)

The High Performance Computing (HPC) area had also considered mobile devices as computational nodes of a parallel system. In this regard, the DroidCluster proposal [42] shown a feasibility with collaborative Android systems and Message Passing Interface (MPI) used as reference programming paradigm. DroidCluster claims to be non-invasive, i.e., the framework does not interfere with the devices primary function. Obviously, in this case the target workload is made by parallel HPC applications, and no resource management is considered.

2.1.6 Mirroring strategies

The last approach we want to consider is the *Google cast* solution [43], because in the application sharing scenario it is similar to our proposal. It differs from our solution in two aspects. First, Google cast is a mirroring: the mobile device or laptop is the sender that controls the playback and the TV is the receiver which displays the content. All the user interactions and most of the user interfaces take place on the sender device rather than on the TV. Our solution performs a sort of migration instead: the mobile device is the launcher which discovers and launches a specific application and the TV or another device is the

computing element in which the entire application is launched and computed. The availability of a single application is subjected to the availability of some Capabilities which depends from device's resources, features and power conditions and availability. The second difference from Google Cast solution is that due to hardware and resource limitations, there are certain restrictions placed on applications supporting Google Cast. The Cast device is a low-power device with memory, CPU and GPU limitations, so the receiver application should be lightweight. All the interactions with the application must be done through a sender application. Google Cast supports a single concurrent media stream playback in the `audio` and `video` tags, while in our vision devices can be anything and also more power and resources availability, if they are convenient, are advisable and better accepted when migration target is selected. The interaction must be done through the computing device or through a custom-controller device (if designed). Our solution can support more concurrent stream, according to the available Capabilities.

2.1.7 Experimental results and limitations

Analyzing the previous works we noticed that they do not consider some aspects of the devices that could be interesting to study in deep. First of all they do not perform resource and energy management and optimization at single device level: [40], [39] and [31] are the only that consider device resource capabilities even if in the first work estimation is done remotely by the server basing on previous tasks completion time and in the second one it is delegated to a local "passive" application. Starting from our work it will be interesting to investigate how a local resource manager affects resources utilization and remote server decision.

A local resource manager could also act as a decision-maker for device election during tasks distribution, monitoring device resources and energy and performing optimal task distribution to other devices or servers. Moreover the most of solutions rely on a single device that acts as a "supernode" and make decision, while recently research is moving on investigate a multi-agent context in which nodes cooperate and make decision in a fully distributed-way.

Finally, no previous solutions consider the management and optimization of Android application but mainly distributed-designed or HPC applications. Future research is going to concentrate on studying in deep the possible interactions between HPC and device applications and their management strategies.

2.2 Computation offloading for mobile systems

The *offloading* is a practice studied from mid 1990s, consisting of executing part of an application (a specific task) on a secondary computing device or system.

Since the energy budget management is still a challenging activity due to limitations of battery technology as stated by [44], a possible exploitation of task offloading can be the pursuing of a device lifetime extension goal, by scheduling the task execution on a remote server providing higher performance.

Before going through the various solutions it is necessary to clarify the difference between offloading practice and other type of distributed computing approaches. As exposed by [45] offloading is not a classic client-server architecture because the device is not a thin client and it can decide whether delegate the computation to the server or do it locally. Moreover, it differs from load balancing techniques used in grid computing because with the former the offloaded tasks are migrated to servers that are not in the same computing environment. An exception can be found exploiting mobile distributed computing because in this case the offloaded tasks are migrated to other personal devices.

Looking at the research since nineties onward, we can group literature works in three different categories: feasibility studies, decision algorithms and framework proposals.

Regarding the first category, most of the feasibility studies have been proposed before the year 2000 [46–54].

Since 2000s onward, more works on policies and decision algorithms started to appear. The proposed algorithms aims at managing or adapting the execution of the mobile applications, on the basis of three different approaches: program partitioning [55–63], energy-aware execution [64, 65] and context-awareness [66–69].

A policy for task offloading decision is typically driven by two objectives

- *Performance improvement*: meeting response time requirements, meeting real-time constraints, enabling context-awareness
- *Energy saving*: extending battery life, achieving green-computing purpose.

Recently, some frameworks for mobile computing offloading have been proposed and gained attention. In what follow we discuss about some implemented tools those enable the computation offloading. Mainly offloading is performed on cloud-based systems, exploiting the computational power of high performance servers, although some recent works aim at exploit an interconnected mobile device as a cloud system: probably this will be the future research scenario. Between the various technological improvements that contributed to the

Year	Paper	Tool name	Partitioning	
			Level	Process
2002	[59]	J-Orchestra	VM	Manual
2004	[71]	Cloudlets	Class	–
2010	[72]	MAUI	Method	Manual
2011	[73]	μcloud	Class	Manual
2011	[74]	CloneCloud	Thread	Automatic
2012	[75]	MACS	Class	Manual
2012	[76]	COMET	Thread	–
2012	[62]	DPartner	Class	Automatic
2012	[77]	ThinkAir	Method/VM	Manual
2012	[78]	*Chen et al.*	Method/VM	Manual
2015	[79]	Jade	Class	Manual
2016	[21]	ARC	Method	Automatic
2016	[80]	COMPSs-Mobile	Method	Manual

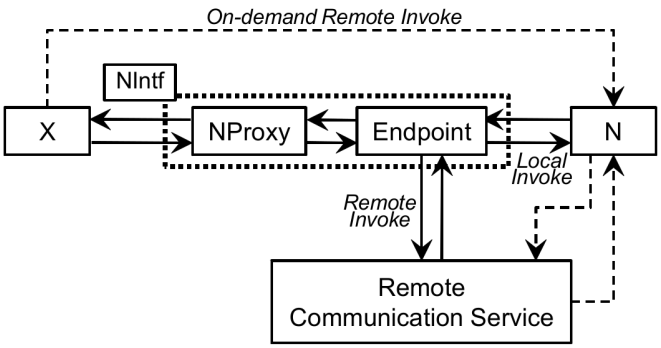
Table 2.1: *Offloading tools partitioning level and process*

grow of offloading interest there are wireless network bandwidth increasing, mobile agents development and the rebirth of virtualization [45].

Regarding the development of Mobile Cloud Computing-applications Orsini et al. [70] identified six main requirements that are involved in: *availability*, *portability*, *scalability*, *usability*, *maintainability* and *security*. Among these to reach scalability requirement one of the most used approach is to partition the application into tasks of different granularity level although it involves some major difficulties such as:

- *Correctness*: identifying which parts of the application have to run locally on the device
- *Effectiveness*: avoiding that network delay caused by the offloading is greater than the reduced execution time
- *Adaptability*: offloading adaptation to applications user requirements and run-time environment changes.

Application partitioning and offloading can be performed at class or method level to obtain a more fine-grained resource allocation. Table 2.1 shows further analyzed tools and their corresponding partitioning level and process. It can be noticed that works originally concentrated at VMs and class level, while in recent years they exploited the method-level partitioning both relying on programmers annotations or on automatic partitioning tools [81]. In what follow we will provide a taxonomy partition of some noticeable works that perform application partitioning and offloading.



centering

Figure 2.1: *DPartner: on-demand remote invocation design pattern [62].*

2.2.1 Server-based architecture

The first noticeable work that handles application partitioning we can find is J-Orchestra [59]. It is a tool that partitions any Java applications at bytecode level, providing a GUI to ask programmers to manually select the "offloadable" classes, but does not consider the mobile applications and devices.

In 2010 Cuervo et al. presented the MAUI system [72], a server-based system that enables fine-grained energy-aware offload of mobile code to a server infrastructure. It embraces the programmer-guided partitioning approaches in which the programmer can annotate the method that can be offloaded for remote execution. MAUI has a client-server architecture: the server acts as a coordinator for the offloading decision and local resources management for the incoming request. Four important goals are reached by this work: code portability exploiting partitioning for .NET applications, programming reflection, type safety and methods profiling through serialization to determine its network shipping costs. However, it is not considered the real scalability and usage of the cloud infrastructure.

In [62] authors proposed an automatic tool (DPartner) to refactor the Android application's bytecode to enable task offloading. The tool performs various analysis on the application bytecode: detecting which classes are movable, making these latter able to offload through a novel *proxy and endpoint* software design pattern (Figure 2.1). The refactored application package file is then deployed in form of a Java archive file for the device and movable Java bytecode classes for the remote server. The decision about offloading classes is based on a static analysis about performance and power consumption of the bytecode execution [82]. The endpoint then is in charge of monitoring the top n computational intensive classes, executing a prediction algorithm and handling the actual communication between classes. A major goal reached by this work is the fact that unlike

ThinkAir or MAUI it is transparent to the developer and it does not require any input or environmental conditions as CloneCloud does.

Eom et al. [83] presented MALMOS, a machine-learning task offloading scheduler. This framework enables a dynamic adaptation of the scheduling decisions based on the observation of correctness of the previous offloading decisions. In this work, authors integrated their system with the DPartner partitioning tool, improving the need of user-input and static decision rules previously required by DPartner.
The major drawback of these solutions is that, albeit some offloading tools retrieve dynamically the status of the device, however they depend on static predefined rules, application pre-processing or user-input to make decisions on whether and where to offload.

2.2.2 Cloud-based architecture

The main difference between cloud-based and server-based solutions is that the former uses remote cloud services that are not necessarily in the same environment of the device. Whereas cloud-based systems are highly scalable, server-based systems can perform local computations. This makes them less flexible, but it also minimizes network latencies.

Cloudlet model coined and presented by Satyanarayanan et al. in 2004 is one of the first work that exploits cloud-based architecture. Cloudlet is a mobile small-scale cloud datacenter that is usually located closer to the personal devices environment than common cloud infrastructures (i.e., in the same Local Network). The study shows the limitations of a WAN-based solutions from the perspective of the communication latency impacting on the usability and responsiveness of mobile applications. The solution uses a VM approach over target devices to encapsulate and separate the guest software environment from the permanent host software environment of the cloudlet infrastructures.

In [74] it is introduced the CloneCloud famework. It is a cloud-based tool extracting binary pieces of a given process to potentially execute on a virtual smartphone clone. The clone would run on a cloud to speed-up the overall process execution. Unlike MAUI, CloneCloud does not require developer's help or code annotation, because it performs an offline static analysis of different running conditions of the binary on both the target smartphone and the cloud. The outcome of this analysis populates a database composed by the pre-computed partitions of the binary that should be migrated.
The main lack of this approach is the need of input and environmental conditions, and consecutively their limitation, to perform the offline analysis for every

application built. At run-time, the profiler collects data from execution time and energy consumption both in the mobile device and server context. This in order to construct a cost model for the application, according to different scenarios. The offloading decision is based on the optimal solution calculated relying on both the static analysis and the dynamic profiling and it is performed by migrating a thread from the mobile device to the clone in the server.

In 2012 Kosta et al. proposed the ThinkAir framework [77] that is a cloud-based framework that improves the idea of MAUI and CloneCloud projects. In particular it addresses the MAUI's lack of scalability using VMs approach and eliminates the restrictions of offline static analysis of CloneCloud by adopting an online method-level offloading. The framework provides to the developers an API to sign which method they want to make offloadable and a specific compiler to translate the annotated code. The offloading is driven by an *Execution Controller* that takes decision basing on the current environment data and method's invocation history considering previous execution time and energy consumption. It is interesting the possibility for the user to set a policy among proposed ones. The framework provides also hardware, software and network profilers to collect various data and feed the energy estimation model used by the offloading policy.

2.2.3 Opportunistic computing paradigm

A first recent work that moves the approach from a centralized cloud architecture to a distributed mobile architecture is the AnyRun Computing system (ARC) [21,63]. It explores the *opportunistic computing* paradigm [19,20], where mobile devices are connected in an ad-hoc local wireless network to take advantage of computing resources of other devices. The major differences from the previous solutions are that cloud or server based systems suffer from a lack of flexibility, because of the need of a specific piece of computing infrastructure known a priori. Moreover they have hidden latencies and energy cost due to the networking communication and servers power consumption. Recent years improvements on hardware performance of devices made possible to get over those limitations, by using mobile devices instead of servers. In this direction, ARC provides a framework to refactor the code of the application to make classes and methods offloadable to any close device and a inference engine, based on Bayesian statistics, that decides whether and where to offload the method.
Recently the authors extend the refactoring process starting from the compiled application by extracting the portable offloadable code and embedding the task code in an Android Application Package.

Another recent work that exploits the *Mobile Cloud Computing* paradigm is

COMPSs-Mobile [80]. It is a framework that transposes the COMPSs programming model [84] to the mobile world. This model abstracts application developers from the parallelization and distribution details. COMPSs applications are composed by annotated methods, called Core Element (CE), those can run in parallel. The framework partitions the original Android applications during the building process, by replacing CE invocations from asynchronous tasks. This tasks are then coordinated at run-time by the toolkit basing on energy, economic and temporal cost prediction of hosting and offload a task execution. Moreover the system comprises a check-pointing and restore mechanism to avoid full re-execution of the application if some nodes fail.
This work marks the frontier of future offloading techniques that exploit interconnected mobile devices as a cloud system albeit it does not exploit all capabilities of devices, such as GPUs or dynamic resource provisioning with a local resource manager.

2.2.4 Overall results and limitations

Due to the heterogeneous benchmarks and devices used by the different works it is impossible to compare their results. Anyway it appears evident the benefit of using the offloading for mobile devices both for execution time and energy saving objectives so that it will be a promising research line for next years.

However, as stated by [70], while coexistence and deployment are reaching high results, adaptation scenarios and ease of use require further investigation. For this reason research can reasonably be focused on effective distributed resource management strategy, hosted on the mobile device. This should improve context-awareness and offloading policies. Certainly opportunistic offloading can get enormous advantage from this vision because of the possibility to enrich the instances of the resource manager towards an intelligent multi-agent system, that could cooperate to achieve energy-efficient performance speed-up.

2.3 Run-time resource management

Since the appearance of Multi/Many-Core architectures the research interest in Run-Time Resource Management increased to get the maximum in terms of performance, reliability, fault-tolerance and security, meeting the resource requests of the applications over more and more complex systems [85].
The resource management area faces the so-called *resource allocation* problem to provide an efficient assignment of resources to jobs [86]. The "runtime" adjective denotes the ability to perform such allocation dynamically during the execution of the system and applications.

As stated by [17] *virtualization* was the first attempt to resource management because it provides resource partitioning and isolation. However this approach was not really beneficial due to "hypervisor" overheads. Similar but with lower overhead was the solution proposed by the Linux Containers (LXC) [87], albeit it did not support reconfigurability and adaptivity of the partitions and did not provide a full-hardware virtualization.
From StarPU [88,89] on we can find frameworks that target heterogeneous multi-core architectures[1] [90] to obtain performance maximization and load balancing. Qilin et al. [91] went further introducing adaptive mapping for NVIDIA CUDA to react to runtime environments changes (i.e. hardware/software configurations).
The scheduling decisions can be taken accordingly to an offline profiling and static properties of the applications [92] or a run-time monitoring of application performance goal as implemented for homogeneous systems [93–95]. These works provide scheduling strategies basing on power consumption optimization, especially for real-time systems, checking the application performance requirements.

The topic of run-time management is still in the research spotlight, in fact, in recent year the BarbequeRTRM framework has been developed and continuously extended by the Barbeque Open Source Project (BOSP) [96] to meet application reconfigurability and resources allocation multi-objective scheduling for both homogeneous and heterogeneous architectures. This is the framework we intend to extend for our purpose and further details will be given in chapter 4.

[1] Unlike homogeneous platforms, heterogeneous architecture are composed by a CPU and an hardware accelerator.

2.3.1 Distributed resource management

In recent years the research on multi/many-core architecture resource management led to identify two main points of improvement to use the available cores efficiently. The first one is the principle of *malleable (or reconfigurable) applications* [97, 98] that can adapt their parallelism and resources configuration dynamically.
The second improvement regards the fact that a centralized management of large number of resources and cores is becoming unfeasible in term of allocation problem complexity and efficiency. In fact for these systems research is moving towards a more scalable distributed approach [99].

For high-throughput distributed computing systems, heterogeneity of resources and distributed ownership obstacle the formulation of an uniform allocation algorithm. In this scenario a proposed solution is to distribute the ownership of resources, so that the owner can define access and usage policies, for example limiting them to a specific user-group membership or in a time constrained period [86].

In the grid-computing environment there are higher latency and lower frequency utilization changes that allow more complex distributed procedures for the resource management as we can found in Arms [100] and in [101]. This type of approach anyway is not suitable for embedded on-chip many-core systems due to its complexity.

Kobbe *et al.* tried to solve this limitation proposing DistRM, a decentralized online resource manager based on a multi-agent system [102, 103]. The general idea is to deploy one intelligent agent per application that acts as a resource manager and that aims at speeding-up its application by reconfiguring it to use available cores on the chip. Among the improvements reached by this solution there is the distribution over the whole chip of the resource management communications overhead instead of concentrating only in one point.

Finally, distributed approaches are becoming exploited also in the energy-awareness field in order to minimize the management overhead and face the highly flexible computing infrastructure typical of energy-aware systems [104].

2.3.2 Resources Management for Mobile Systems

The energy-budget management limitations of mobile systems suggests us to further investigate the possibility of employing a run-time resource manager on each device in charge of performing task allocation and hardware configuration decisions. This in order to maximize an energy efficient exploitation of the distributed devices and increase user satisfaction.

Energy management on mobile devices has a recent research history, in particular we can find works that concentrate on estimating and modeling the applications energy consumption in order to identify energy-hungry tasks or piece of code. The eCalc profiler [105] achieves lightweight fine-grained estimates of application energy usage analyzing execution traces of the developed application without an expensive monitoring equipment. Basing on the execution trace it can also provide fine-grained feedback on the structure of the code to the developer. A more recent work [106] aims at relating hardware energy consumption to high-level application code inferring a fine-grained energy model based on the concept of "energy operations" identified directly from the source code. The novelty of this approach is that the authors identify a set of source-level operations (such as multiplications, method invocations...) that are easier to be interpreted by the developer and find their correlations to the energy cost by analyzing various of well-designed execution cases.

Moreover, there are some works that investigate the relation of energy consumption of smartphones with the operational status of the device and try to extrapolate an energy model of device components to aid creation of Green Software solution [107]. In fact, typically, in environments with power-limited sources, operating systems perform efficient power management strategies to reduce power consumption during idle time or low source level at the expense of latency and performance. This is done by shutting off unused components, setting their economy operation modes and performing system hibernation [108].

Our work aims at enabling an integration among all these approaches through an offline applications performance profiling that can be used by run-time resource management techniques that controls the hardware components of the device.

CHAPTER 3

System Design

Managing Android applications in a transparent way led us to introduce a novel concept of Capability. This chapter exposes the conceptual design of our system. First of all, Section 3.1 presents the Capability definition and related concepts; Section 3.2 is deputed to explain the importance of the applications profiler and how it works regarding to the Android environment. Finally, Section 3.3 argues about the Capabilities, applications and resources management model, that is how the system in a conceptual view takes into account Capabilities in order to perform operations on applications and their required resources.

3.1 Capability model

During the design phase of our work we faced the problem of encapsulating hardware requirements for a generic Android application that it would be transparent for users and developers. Our objective was to keep the back-compatibility with applications written for older Android version and at the same time enabling the possibility to manage in a fine-grained way the resources of the device. Currently, the Android system performs a soft-check on device hardware compatibility with applications. In fact it relies on a filtering mechanism based on the

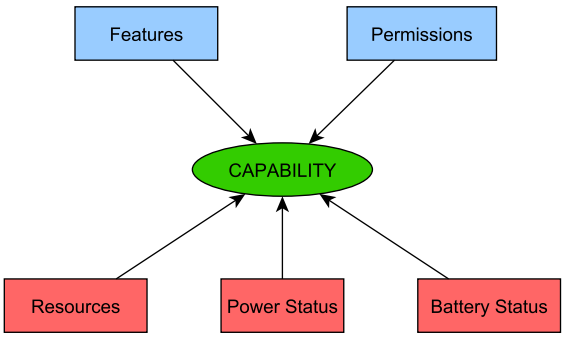


Figure 3.1: *Diagram showing what a Capability represents. The blue color refers to the Android framework layer; the red color regards the hardware-related layer; the green color refers to the new Capabilities layer.*

available hardware *features*[1] of the device. Anyway this type of features are statically embedded in the device but not information are taken into account about their utilization or run-time availability.
Moreover, the concept of feature alone is not sufficient to specify application requirements or constraint because it does not include information about, for example, the power needed by a specific feature to work properly.
For these reasons we define the novel concept of *Device Capability* that includes all the aforementioned information, as shown in Figure 3.1, and it is defined as follow:

Definition 3.1.1. *A **Device Capability** is an aggregated representation of device's specific resources status, available Android standard features and power conditions.*

The main idea is that if a specific Capability is available, then its subtended resources, features and power conditions are guaranteed.
A Capability can be *exclusive* or *shared*, depending if its use can be shared among different requesters (shared) or not (exclusive).
The idea is to think the entire model as a *Service-oriented architecture* in which the services are the *Capabilities* that a device can expose and that the resource manager can manage and take into account to schedule and to distribute the work among the different devices.

A key aspect is the connection between the Capabilities and the applications: this is initially made by mapping the Android *features* used by the single application with the Capabilities that subtend those features and then possibly adding user-defined Capabilities, so that a single application uses a set of required or

[1] see Appendix A.2 for more information about

optional Capabilities. The fact that a Capability is optional means that the application does not require it to run, but it could use it to perform for example some optional operations or to increase the accuracy of some outputs (e.g., using fine-grained location through wifi or 3G). Moreover, since a Capability can be defined also with some resources related, there is the possibility to have fine-grained resource requirements for an application.

A single Capability subtends one or more Android features, which can be required or optional: this mapping is shown in Table 3.1 and stored in a easily editable `.xml` file. This file is organized according to the example of Listing 3.1, each Capability as well as by features, can be defined by some resources with the minimum value required, declared as string representation which follows the pattern *sys.component.component* (from now on we refer to this representation with the name *Resource Path*), and a device status condition in which the Capability can be considered available (e.g., minimum battery charge value, maximum temperature value...).
At current state the resources and power status requirements related to the single Capability are calculate empirically.

Listing 3.1: *Sample Capabilities XML definition file*

```
<?xml version="1.0"?>
 ...
  <system name="NEXUS5_0"> <!-- STATIC CAPABILITIES DEFINITIONS -->
      < capabilities >
          <capability  type="SCREEN">
              <resources>
                <resource  id="sys.cpu.pe"  value_min="80" required="false"/>
                <resource  id="sys.gpu.pe"  value_min="60" required="false"/>
              </resources>
              <features>
                <feature  name="android.hardware.screen. portrait "  required="true"/>
                <feature  name="android.hardware.screen.landscape"  required="false"/>
              </features>
              <status>
                <battery  value_min="16"/>
              </status>
          </capability>
           ...
      </ capabilities >
  </system>
 ...
```

More formally, let be:

- **F** a set of *device features* $\{f_1, f_2, \ldots, f_n\}$

- $\mathbf{R}$ a set of *hardware resources* (e.g CPU, memory, etc..) represented as a pair $\langle$*Resource Path*$, value\rangle$
- $\mathbf{P}$ a set of *power status* (e.g. temperature, etc...) represented as a pair $\langle status, value\rangle$
- $\mathbf{B}$ a set of *battery status* (e.g. level, plugged, etc...) represented as a pair $\langle status, value\rangle$.

we can define the *Capability* concept as a sextuple

$$\mathbf{c} = \langle \mathbf{F_r^c}, \mathbf{F_o^c}, \mathbf{R_r^c}, \mathbf{R_o^c}, \mathbf{P^c}, \mathbf{B^c} \rangle \tag{3.1}$$

where $F_r^c, F_o^c \subseteq F$, $R_r^c, R_o^c \subseteq R$, $P^c \subseteq P$ and $B^c \subseteq B$. From now on the r subscript denotes required components while the o subscript denotes the optional ones.
For convenience we label as $C = \{c_1, c_2, \ldots, c_n\}$ the set of all defined capabilities.
Since a capability can be *exclusive* or *shared* we label as C_{excl} and C_{shr} the two set of capabilities respectively such that $C_{excl}, C_{shr} \subset C$ and $C_{excl} \cap C_{shr} = \emptyset$

Following the same formal notation an *application* can be defined as a tuple

$$\mathbf{App} = \langle \mathbf{C_r^{App}}, \mathbf{C_o^{App}} \rangle \tag{3.2}$$

where $C_r \subseteq C$ are the required capabilities and $C_o \subseteq C$ are the optional ones. We denote by A the set of all developed applications.

Finally, we can represent a *device* through two definitions. The first models a base device as a decuple

$$\mathbf{Dev} = \langle \mathbf{C_e^{Dev}}, \mathbf{R_d^{Dev}}, \mathbf{F_d^{Dev}}, \mathbf{A_d^{Dev}}, \mathbf{A_e^{Dev}}, \mathbf{R_a^{Dev}}, \mathbf{P_a^{Dev}}, \mathbf{B_a^{Dev}} \rangle. \tag{3.3}$$

The symbols with the d subscript denotes sets which contain elements that are fixed device hardware or software equipment (i.e., installed), the e subscript denotes sets which contain elements that are enabled by the user and the a subscript denotes sets which contain elements that are effectively available for use.
The second definition defines an extended device concept that includes also available capabilities and applications, i.e. computing by the resource manager, by adding the sets C_a, C_b (i.e., the set of booked capabilities) and A_a to the 3.3:

$$\begin{aligned} \mathbf{Dev'} = \langle & \mathbf{C_e^{Dev'}}, \mathbf{R_d^{Dev'}}, \mathbf{F_d^{Dev'}}, \mathbf{A_d^{Dev'}}, \mathbf{A_e^{Dev'}}, \mathbf{C_a^{Dev'}}, \mathbf{C_b^{Dev'}}, \mathbf{R_a^{Dev'}}, \\ & \mathbf{P_a^{Dev'}}, \mathbf{B_a^{Dev'}}, \mathbf{A_a^{Dev'}} \rangle. \end{aligned} \tag{3.4}$$

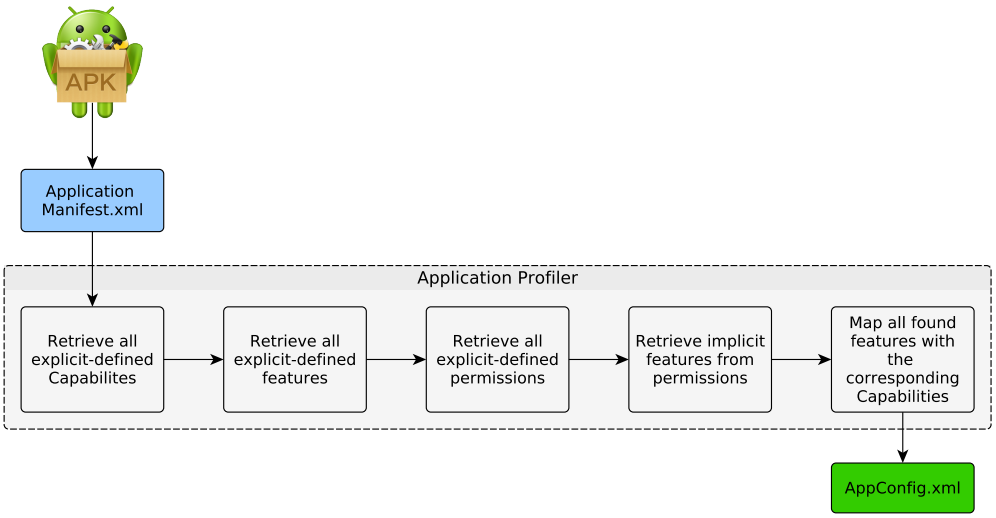


Figure 3.2: *The applications profiler steps to retrieve Capabilities requirements and to export the profile file from the application Manifest.*

In both definitions the following conditions must hold:

- $C_b, C_a \subseteq C_e \subseteq C$
- $R_a \subseteq R_d \subseteq R$
- $P_a \subseteq P$
- $B_a \subseteq B$
- $A_a \subseteq A_e \subseteq A_d \subseteq A$.

We denote by D the set of all base devices and D' the set of all extended devices. Moreover, we say that $Dev_i \cong Dev'_i$ if they differ only for the C_a, C_b and A_a sets.

3.2 The application profiler

To help Android developers to create compatible Android applications we propose an *application profiler* that parses the Android *Manifest.xml* file of the application (defined by the developer and explained in the Appendix A.1) to extract the initial set of information required to identify the Capabilities of the application. The manifest can be retrieved from any *Android Application Package (APK)* through the *ApkTool* [2] software that unpacks and decodes the binary resources and file structure of a given application.

[2] https://ibotpeaches.github.io/Apktool/

Capability	Explicitable Resources	Exclusive	Required Features	Optional Features
SPEAKER	Yes	No	.output	/
SCREEN	Yes	Yes	.screen.landscape	.screen.portrait
CONNECTIVITY_BT	Yes	No	.bluetooth	.bluetooth.le
CONNECTIVITY_WIFI	Yes	No	.wifi	.wifi.direct .location.network
SENSORS	No	Yes	.sensor	.sensor.accelerometer .sensor.barometer .sensor.compass .sensor.gyroscope .sensor.light .sensor.proximity .sensor.stepcounter .sensor.stepdetector
CAMERA	No	Yes	.camera	.camera.autofocus .camera.flash .camera.front
NFC	No	Yes	.nfc	.nfc.hce
LOCATION	No	No	.location	.location.gps
INFRARED	No	Yes	.consumerir	/
MICROPHONE	No	Yes	.microphone	/
TELEVISION	Yes	Yes	.type.television	/
TELEPHONY	No	Yes	.telephony	.telephony.cdma .telephony.gsm .location.network
TOUCHSCREEN	No	Yes	.touchscreen	.faketouch
MUSIC_HP	Yes	No	.audio.pro	.output
MUSIC_LL	Yes	No	.audio.low_latency	.output

Table 3.1: *Mapping between Capability and Android Features (`android.hardware`)*

The profiler performs four main operations summarized in Figure 3.2. First of all it retrieves all the Capabilities explicitly defined, both required or optional. These Capabilities can be further defined in detail by the developer overriding their standard resources or device's status requirements.
Secondly, the profiler retrieves all the explicited *features* (required or not). These information should be currently provided by the developer to allow the applications store to show device-compatible applications only.
Moreover, since some Android permissions (see Appendix A.2) imply the access to specific hardware features, the profiler is able to retrieve implicit features from declared permissions basing on the mapping done by Google [109]. In this case, the features are considered as required.
Finally, the profiler maps all the found features to the corresponding Capabilities through the mapping shown in Table 3.1.

The output of the profiler is an `.xml` file shown in Listing 3.2. The file contains the name and the package of the application, that will be used by the system to perform operation on it, and all the required or optional Capabilities. At this point it is possible for developers to manually add specific resources or power status constraints for each Capability (e.g., as done in lines 6–8 of Listing 3.2), otherwise the resource manager will consider the static definitions

mentioned in Section 3.1.

Listing 3.2: *Sample Application XML definition file*

```
<?xml version='1.0' encoding='utf-8'?>
 ...
  <application name="YouTube" package="com.google.android.youtube">
    < capabilities >
      <capability type="CONNECTIVITY_WIFI" required="true"/>
        <resources>
          <resource id="sys.net" value_min="70"/>
        </resources>
      <capability type="NFC" required="true" />
      <capability type="SCREEN" required="true"/>
      <capability type="CAMERA" required="false"/>
    </ capabilities >
  </application>
 ...
```

3.3 The proposed management model

After that the concept of Capability has been introduced, it is possible to explain the general management model implemented in the BarbequeRTRM. This model is shown in Figure 3.3.

3.3.1 Capabilities management model

At application level, the application profiler is in charge to identify the Capabilities of the application, whereas the user can decide which applications register under the control of the resource manager.
At device level, similarly, Capabilities are first enabled by the user, who can decide which Capabilities can be managed by the resource manager. How to perform this operation is explained in Section 4.2. Then the resource manager computes all the available Capabilities of the device checking for each Capability that its features, resources and power status requirements are available in the device.

The key aspect is that Capabilities availability is dynamic: Capabilities for instance are booked by the application scheduler. In this case exclusive Capabilities are assigned to one process only and made unavailable. Their availability is recomputed by the resource manager each time they are requested (e.g., during the applications filtering operation). Moreover, changes in device's power conditions or resources status also affect Capabilities availability.

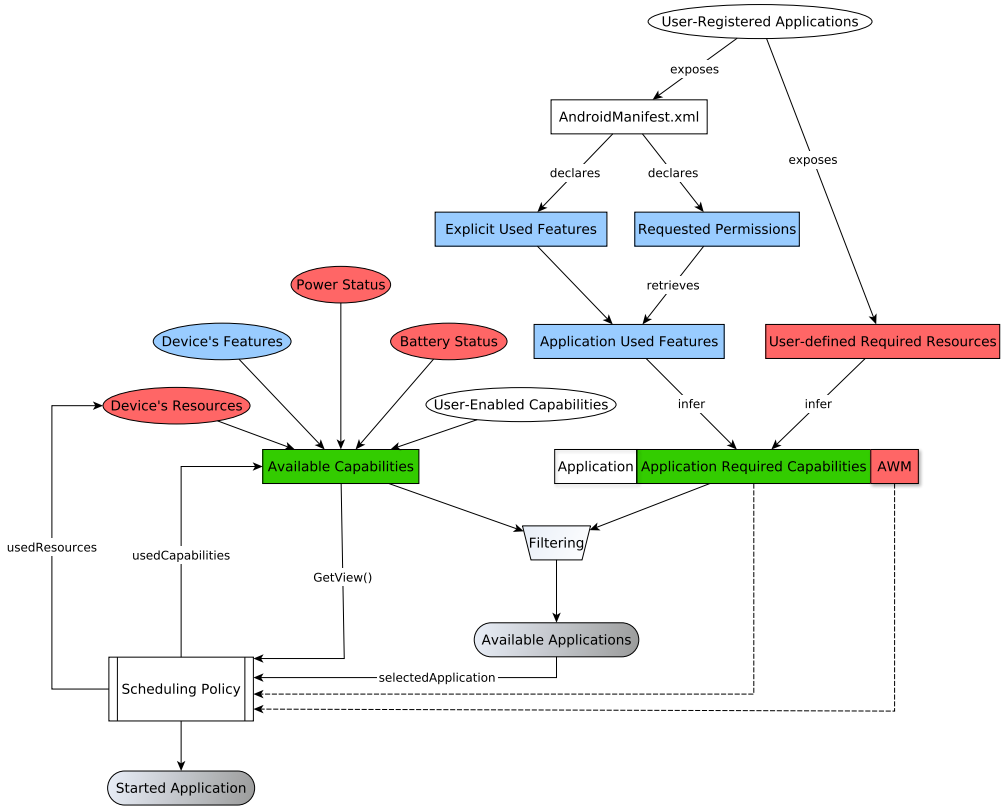


Figure 3.3: *The entire capabilities and application management model. The blue color refers to the Android features and permission layer; the red color regards the hardware-related layer; the green color refers to the novel Capabilities layer.*

More formally, the *capabilities filtering* operation can be modelled with the following function:

$$\begin{aligned} \text{CapFiltering}: D &\rightarrow D' \\ Dev &\mapsto Dev' \end{aligned} \tag{3.5}$$

Such that:

- $Dev \cong Dev'$
- $\forall c_i \in C_a^{Dev'}$
- $\exists c_j \in C_e^{Dev} \mid c_i = c_j$
- $\forall r_k \in R_r^{c_i}, \exists r_h \in R_a^{Dev'} \mid ResSat(r_k, r_h)$
- $\forall f_k \in F_r^{c_i}, \exists f_h \in F_d^{Dev'}$
- $\forall p_k \in P^{c_i}, \exists p_h \in P_a^{Dev'} \mid PowerSat(p_k, p_h)$

- $\forall b_k \in B^{c_i}, \exists b_h \in B_a^{Dev'} \mid BatterySat(b_k, b_h)$.

Where $ResSat(r_1, r_2)$ is a function that defines if the resource requirements of pair r_1 are satisfied by the requirements of the resource pair r_2.
The $PowerSat(p_1, p_2)$ is a function that defines if the power requirements of pair p_1 are satisfied by the power requirements of pair p_2.
The $BatterySat(b_1, b_2)$is a function that defines if the battery requirements of pair b_1 are satisfied by the battery requirements of pair b_2.

Given two pairs as $\langle status_i, value_i \rangle$ the *satisfaction relation* can be defined as:

$$\begin{aligned} &\forall status_1, \forall status_2, \forall value_1, \forall value_2 \\ &Sat(\langle status_1, value_1 \rangle, \langle status_2, value_2 \rangle) \Leftrightarrow \\ &status_1 = status_2 \wedge value_2 \geqslant value_1 \end{aligned} \tag{3.6}$$

Finally, the resource manager maintains different views of all enabled and available Capabilities, in this way they can be used to filter the runnable applications as explained below.

3.3.2 Android applications management model

While applications that are integrated with the resource manager usually follow an execution model, that in the case of the BarbequeRTRM is called *Abstract Execution Model*[3], one of the main problem managing Android application is that they are not developed according to this model so we cannot have the full control during execution and it is not possible to communicate with the application, for example, to adapt the execution to the set of resources actually assigned or to drive the resource manager with information about the current performance requirements. This limitation leads to design a new dedicated management model.

The main effort is to allow the management of Android application together with the management of integrated BarbequeRTRM applications. First of all, to distinguish between these two types of applications, it is introduced the attribute *reconfigurable* in the application definition: the standard Android applications are tagged as non-reconfigurable, while the BarbequeRTRM integrated applications can be tagged both reconfigurable or not. Moreover a lightweight implementation of the execution model, called *Execution Context* in the BarbequeRTRM jargon, is associated to the Android application, this allows to make the application managed by the resource manager core modules and fitted for the scheduling and execution flow.

[3] See Section 4.1 for further details

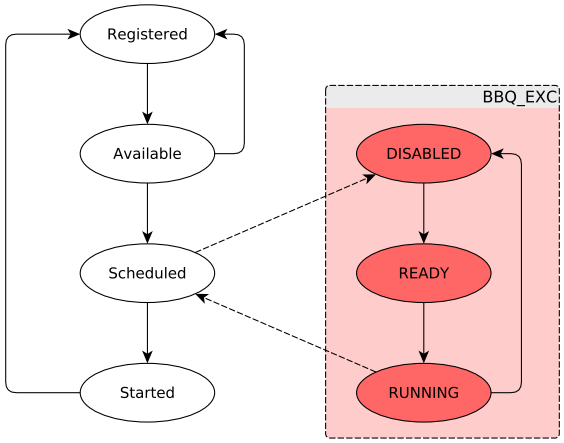


Figure 3.4: *State diagram of a managed Android Application. The white states are the application state itself as view by the manager, the red states are the application's Execution Context state*

Once an application is profiled by the *application profiler* (seen in Section 3.2) and enabled in the system, it is ready to be managed by the resource manager. During the start-up the latter takes all the `.xml` configuration files of the enabled applications and for each one it keeps track of all the declared Capabilities, as well as other information such as name and package. Then it creates on-the-fly a single static resources configuration, called *Application Working Mode (AWM)*. This will be used by the *Schedule Policy* to assign correctly the required resources and Capabilities. The static configuration is created according to these steps:

1. For each required Capability take the user-defined resource constraints
2. If some required Capability has not user-defined resource constraints, take the resource constraint of the static Capability definition (as exposed in Section 3.1)
3. For each specific resource type sum all the required values from different Capabilities.

Once the above steps are completed the application is *Registered* into the resource manager. It means that the application execution will be managed by the BarbequeRTRM. The other status are shown in Figure 3.4.

After the application and Capabilities initialization phase is completed a filter is in charge to check the available capabilities of the device and the required capabilities of each application to set an application in the *Available* state. Once an application is marked as available it can be exposed to other BarbequeRTRM instances on other devices and possibly scheduled.

Formally, the *application filtering* operation can be modelled with the following function:

$$\begin{aligned} \text{AppFiltering:} D &\rightarrow D' \\ Dev &\mapsto Dev'' \end{aligned} \tag{3.7}$$

Such that:

- $Dev \cong Dev''$
- $\exists Dev'$
- $Dev' = CapFiltering(Dev)$
- $\forall App_i \in A_a^{Dev''}, \exists App_j \in A_e^{Dev''} \mid App_i = App_j$
- $\forall App_i \in A_a^{Dev''}, \forall c_k \in C_r^{App_i}, \exists c_h \in C_a^{Dev'} \mid c_k = c_h.$

Before an application is actually launched it has to be scheduled by the *Schedule Policy*, so a new *Execution Context (EXC)* is created and its status set to *DISABLED*. Then the EXC is enabled and is set to *READY* state, which means that it has to be considered during the scheduling process. A new scheduling event is thrown by the system.
The *Schedule Policy* retrieves a view of all Capabilities and resources allocation and takes care of recomputating a new allocation considering the ready applications. When the scheduling is completed and committed, the EXC is set to *RUNNING* and the application is in the *Started* state.
At the end of the execution the EXC of the application is set to *DISABLED* and destroyed, while the application can return to the *Registered* state and its resources and Capabilities released. An application can switch between the *Registered* and *Available* states depending on the currently availability of device Capabilities.

3.3.3 Resources management model

In this model resources are associated to a Capability demand that can be required or optional. The difference is that, if a Capability requires a particular resource minimum value, the scheduler has to give to the application that minimum quota of resource, while if the resource is demanded as optional the scheduler may try to give it to the application in a best-effort way not guarantying that the demand is satisfied.
To effectively allocate resources to the applications we exploit Linux *Control Groups* even if the Android Linux kernel does not have a full support of all the

subsystems included in the framework. In particular at time of writing the only subsystems available by default are `cpu`, `cpuacct`, `freezer`. The former allows us to set the share of CPU time to assign to tasks in the same cgroup. The second subsystem is used to generate reports on CPU resources used by tasks in a cgroup. The last one enables the possibility of suspending tasks execution and resume it later (e.g., for migration or scheduling purposes).
In order to set the group of resources assigned to a task we need to retrieve its PID[4] and write it into the *cgroup_name/tasks* file.

Other than performing resource allocation, the Android extension of the BarbequeRTRM allows us to set the clock frequency of the CPU cores. Since Android is based on Linux, the run-time resource manager exploits the interfaces provided by the *cpufreq* framework exploiting a DVFS approach [110, 111].

[4]Process Identification Number

CHAPTER 4

Integration with the BarbequeRTRM

In this Chapter we explain how we implemented the aforementioned conceptual model with the BarbequeRTRM. After a brief overview of the actual resource manager architecture and features (Section 4.1), we expose the novel architectural modules we have introduced in our work. In particular Section 4.2 treats the different aspects of the integration: it starts with a brief overview of the management of applications that are integrated with the Abstract Execution Model (AEM) of the BarbequeRTRM; then it considers the extension to manage also non reconfigurable generic Android applications. Furthermore, it introduces the Barbeque daemon service module and how it cooperates with the native BarbequeRTRM. Finally, because this is one of the first work that exploits and enhances the distributed implementation of the BarbequeRTRM, the Subsection 4.2.4 concludes considering the distributed resource management architecture of the resource manager.
The developed device selection policy for application offloading is explained in Section 4.3, while Section 4.4 discusses about the API we made available for developers of third-party applications in order to exploit the integration with the resource manager. Finally, the Section 4.5 focuses on some application launching schemes that can take advantage of the integration with the BarbequeRTRM and explains in depth the distributed mechanism enabled and implemented by

our work providing also a possible use case scenario.

4.1 The BarbequeRTRM

The Barbeque Run-Time Resource Manager has been developed and continuously expanded by the Barbeque Open Source Project team [96] within *Politecnico di Milano*. The main goal of this project is to exploit run-time resource management taking care of both applications Quality-of-Service (QoS) requirements and dynamic resources availability. Moreover, the BarbequeRTRM is written in C/C++ language and is conceived to run in the user-space exploiting OS libraries: this allows the resource manager to abstract from the underlying platform leaving the Linux OS kernel in charge of communicate directly with the hardware [17, 112]. For this reason its application context spaces from energy-constrained embedded systems to High Performance Computing, making portability one of its main feature and supporting both homogeneous and heterogeneous platforms [113, 114]. At the time of writing the resource manager supports different hardware platforms [115, 116]. One of the main contribution of this work is to port a full version of the resource manager to the Android platform to support mobile oriented platform, like ARM big.LITTLE based SoC.

The BarbequeRTRM was initially born as a centralized approach that has been then refined to support a distributed hierarchical mode. This work goes in the direction of further developing distributed resource management strategies. The framework is highly modular and easily extensible thanks to a plugin-based architecture for adding policies and features. The *KConfig* configuration tool allows to build only selected modules basing on the user and platform requirements.

One of the peculiar ideas of the resource manager is to support applications that could reconfigure themselves at run-time adapting the configuration to the set of assigned resources. To enable this capability the applications supported by the BarbequeRTRM must be developed to be executed according to the so called "Abstract Execution Model", that will be better explained later. Applications can expose to the resource manager several resources requests configurations, called *Application Working Modes (AWMs)* in Barbeque jargon. Usually the AWMs are provided by the developer basing on a design-time optimal identification of a finite set of configurations exported in a file (a.k.a the *Recipe*) and are assigned to the application by the resource manager. The applications, moreover, has a set of tunable parameters called *Operating Points (OPs)* which are chosen by the application itself depending on the assigned AWM to match an expected QoS[1]

[1] Quality of Service

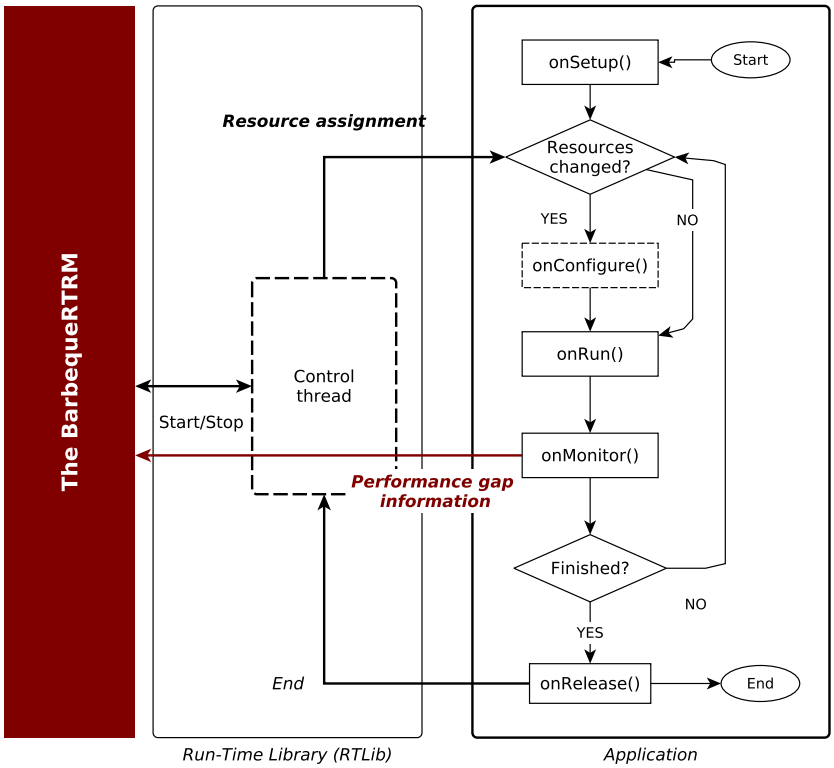


Figure 4.1: *The RTLib AEM implemented by an integrated application.*

for the end user. Generally a single AWM can support multiple OPs.
Since the applications have an active role on the self-adaptiveness of the system, the BarbequeRTRM framework provides a *Run-Time Library (RTLib)* that provides interfaces, features and application-transparent communication channel to allow interaction between the application and the resource manager.

Abstract Execution Model (AEM)

The reconfigurability property of an application required that its execution life-cycle can be controlled by the resource manager which must be able to start, stop, suspend or reconfigure the application and synchronize it with the dynamical resource allocation. Moreover, the application itself may need a step in its execution flow to verify if the current configuration match its requirements (QoS, performance goals...).
To meet these requirements the RTLib provides the *Abstract Execution Model (AEM)*. It exploits an event-based programming model and contains a set of callback that are required to be implemented in the applications and in which the application logic is split. A worth to be considered aspect is that the AEM is very similar to the Android application execution model and for this reason a

particular set of Android application can be easily adapted to be runtime manageable.
The AEM is described in Figure 4.1. It can be seen as a state machine in which each state (the white rectangles) correspond to a specific operational state of the application. Each schedulable application need to instantiate a so-called *Execution Context (EXC)* which is an implementation of the AEM. Since the basic implementation of the AEM is exported by the `bbque::rtlib::BbqueEXC` class, developers must implement and instantiate a class derived from `BbqueEXC`.

Briefly, the main callbacks methods that a developer need to fill with the application logic are:

- `onSetup`: since this method is called by the base class after the constructor it should contain the application initialization code such as memory allocation calls, variable initialization, opening input and output channels and so on
- `onConfigure`: this method is called when the BarbequeRTRM assigns a new AWM. It should contain all the code related to the applications parameters and components reconfiguration (e.g., threads, data structures...)
- `onRun`: this is one of the main method of the EXC and it is its entry point. It must host all the code to actually execute a cycle of the application computational task
- `onMonitor`: once a cycle of execution is performed, the application should check if the level of QoS or performance has been achieved. Developers can exploit this method to implement application-side run-time management policy to face any possible response of this check. Moreover, if there are no more data to be further processed the `onRelease` method is called, otherwise application should check if resources are changed to call the `onConfigure` method or the `onRun` one to perform another computational cycle
- `onRelase`: if the workload is finished then the application should clear its memory allocations, data structures or references. Developers has to put inside this method the code to perform these kind of stuff.

Projects involved

The BarbequeRTRM is portabile and highly customizable. For this reason, it was involved and extended in multiple European projects research (FP7 and

H2020) that are related to embedded systems [117, 118], heterogeneous systems [119] and High Performance Computing [120]. These projects tackle different computing-related challenges, in particular energy-efficiency and thermal management, as well as monitoring, modelling, prediction and decision support for disaster management and health care applications.

Moreover, the framework supports different programming languages, such as OpenCL, Java and OpenMPI, and different architectures as multi/many-cores CPUs (e.g., ARM Cortex processors), hardware accelerators and GPUs (e.g., nVidia CUDA).

4.2 The proposed architecture

The highly modular architecture and the extensibility of the BarbequeRTRM make it the ideal candidate for our purpose. To implement our conceptual model into the resource manager we introduced new specific modules that can be enabled only during the deployment on Android devices. In this section we explain the implemented modules and how they are involved in the management process.

As we can see in Figure 4.2, we can divide the system in three layers: the blue one is the *Application layer*, the green one is the *Resource manager layer* and the red one is the *Hardware layer*.

Regarding the *Application layer* we can notice that there are three different types of management depending on the fact that the application has been implemented according to the AEM or not. In particular we can have a *standard integrated applications* that are the actual supported applications for multi-core and HPC systems. They run in a standard EXC and are managed by the resource manager by the RTLib as described in the previous section.
Then we can also have *standard Android integrated applications*, that is Java Android applications written according to the AEM. They run into a special EXC split between Java and Native layers. In the former the application communicates with a service that through JNI calls communicates with the Native layer where there is the implementation of the AEM; also in this case the resource manager can manage the application through the RTLib. This type can be used by Android applications implementing a stream processing paradigm. This work re-enable the support for this kind of applications as briefly explained in the next section.
The last type is the novel part and it concerns *generic Android application* that does not follow the AEM and so they cannot be managed through RTLib. This type of applications can communicate with the resource manager through a *BarbequeService* daemon that they can bind (Figure 4.3) and whose API they can use

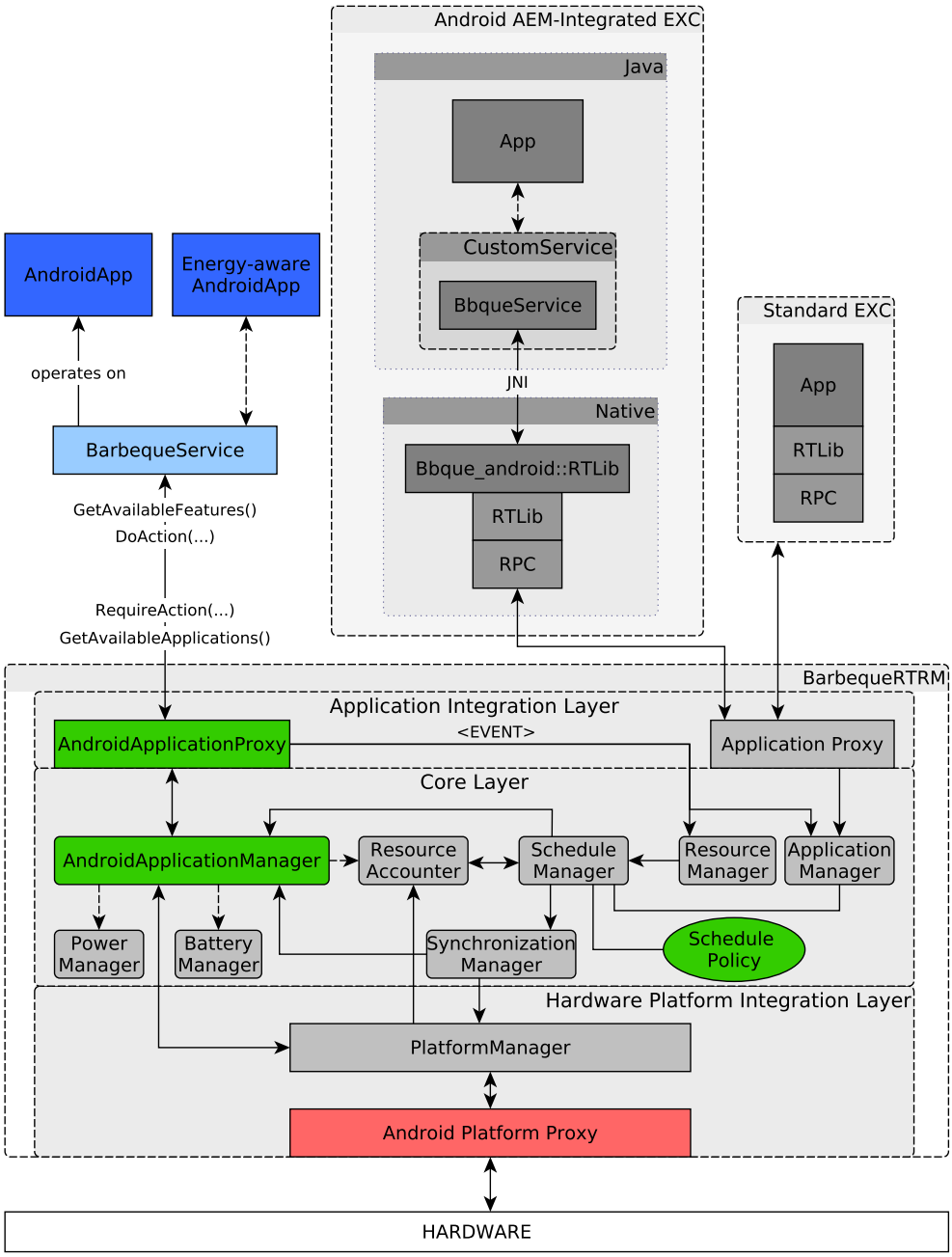


Figure 4.2: *The proposed architecture of the BarbequeRTRM modules. The grayed modules were already implemented, the colored modules are the new ones.*

to retrieve some information. The communication between the resource manager and the application through the *Barbeque Service* can be bidirectional (with the binding explained before) or monodirectional (from the manager to the application), the latter case is used to implement the management model of an application explained in Subsection 3.3.2.

The *Resource manager layer* shows part of the modules that compose the core of the BarbequeRTRM: the grayed modules were already implemented while the colored ones are introduced by this work. We can see that this layer is split in three sub-layers: the applications integration layer, the core layer and the hardware platform integration layer.
The applications integration layer is composed by two *proxies*, the *Application Proxy* is used for the communication with the EXCs and so the AEM integrated applications, while the new *Android Application Proxy* is used to communicate with the *Barbeque Service* and indirectly with generic Android applications. It receives and forwards requests from the daemon service to the core layer, dispatches application event notifications and forwards responses and commands to the service.
Both proxies communicate with the core layer where there are the manager modules in charge of take different decisions. For our model the *Android Application Manager* has a major relevance since it is in charge of performing the following operations:

- Initializes the set of enabled Capabilities provided by the systems
- Registers the applications
- Requests the available features, resources and check the power status of the device
- Computes all the available Capabilities
- Identifies the set of runnable applications filtering the available Capabilities of the device
- Maintains a current view of the usage of the Capabilities
- Dispatches managed application life-cycle commands.

To perform its duty the *Android Application Manager* needs to communicate with other BarbequeRTRM modules, in particular the *Resource Accounter* to retrieve the current resources available, the *Battery Manager* to retrieve the battery status and the *Android Application Proxy* to send commands to the *Barbeque Service*.
The hardware platform integration layer is implemented by the *PlatformManager* and other modules that are further explained in details. It exposes an interface of the systems resources to the core layer and is in charge to redirect the required operations to the proper layer's submodules.

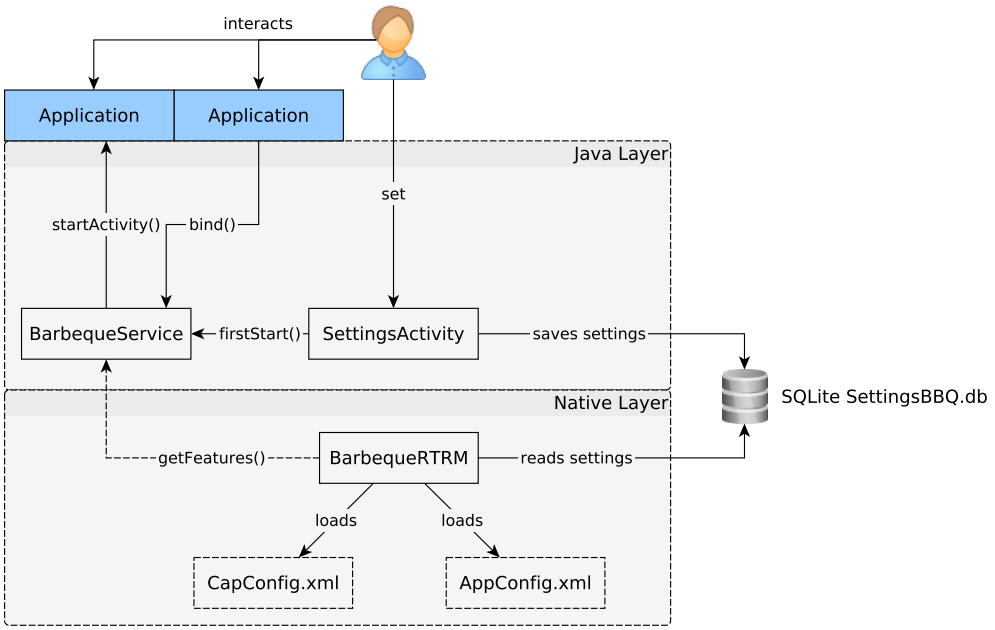


Figure 4.3: *How the users and the applications interact with the daemon* BarbequeService *and the* SettingsActivity. *Java and the Native modules are put in evidence.*

As well as the presented layers and modules there is also a *Settings Activity* that deserves a special explanation. This Activity allows the user to enable the Capabilities and the applications that have to be managed by the resource manager. It also starts for the first time the *Barbeque Service*, because it is required by Android security protocol to enable the automatic start of the daemon after the boot of the device. As shown in Figure 4.3, the settings and information are stored in a SQLite database, which is the main method to store local data for Android applications. The database is then opened and queried by the *Android Loader* module, that is in charge also to load and to parse the Capabilities and single application configuration (XML) files.

4.2.1 Integrated application management flow

In a work of 2012 [121] a lightweight version of the BarbequeRTRM was ported for the first time on Android. That work aimed at managing Android stream processing applications that are developed according to the AEM.
Unfortunately the Android support was not been carried out with the further development of BarbequeRTRM and latest version and features were not available for Android environment. Since with our work we implicitly re-enable the support also for this type of applications we briefly explain their management

architecture.

As standard native AEM-integrated applications they run into an EXC, however they are written in Java so they cannot directly exploit the native Barbeque RTLib framework. The proposed solution proposes a modified concept of EXC. The main idea is to expose the RTLib functionality to the Java layer through a service-based architecture as shown by the Figure 4.2. In the Java layer of the EXC the application binds a developer's custom *Service* which extends a basic implementation of the *Bbque Service*, the daemon service started at boot time. Its goal is to expose the RTLib methods to the Java layer by simply broadcasting an *Intent* when an EXC callback is called by the BarbequeRTRM. This is one of the event-based mechanism provided by the Android runtime framework to achieve inter-process communication towards multiple instances. For this reason, this class should be extended by the application developers to catch the broadcast intent and to effectively implement the callback methods of the application. Furthermore, the *Bbque Service* declares all the signatures corresponding to the standard RTLib native functions. The real implementation of such functions is in the standard RTLib of the *native* layer. The bridge between the Java and the native context is made by `Bbque_android::RTLib`, which exploits the JNI [2] library.

4.2.2 Non-integrated application management flow

In Chapter 3 we presented the lifecycle management model of a generic Android application and now we explain how this model is implemented in BarbequeRTRM.
First of all the *Android Application Manager* retrieves the set of the enabled applications from the *Android Loader* and registers them. The applications are enabled by the user through the *Setting Activity* presented before, that retrieves all the installed packages from the Android *Package Manager*. At this point the *Android Loader* has to read from the settings database the enabled applications. For each enabled application (Figure 4.4), it retrieves the application configuration *.xml* file in the application assets folder, and parses this file creating the application recipe and a static working mode, as explained in Subsection 3.3.2.
When all applications are registered, the *Android Application Manager* computes the available applications set filtering the required Capabilites of the applications and the available ones. At this point, the available applications can be scheduled and started.

[2] Java Native Interface

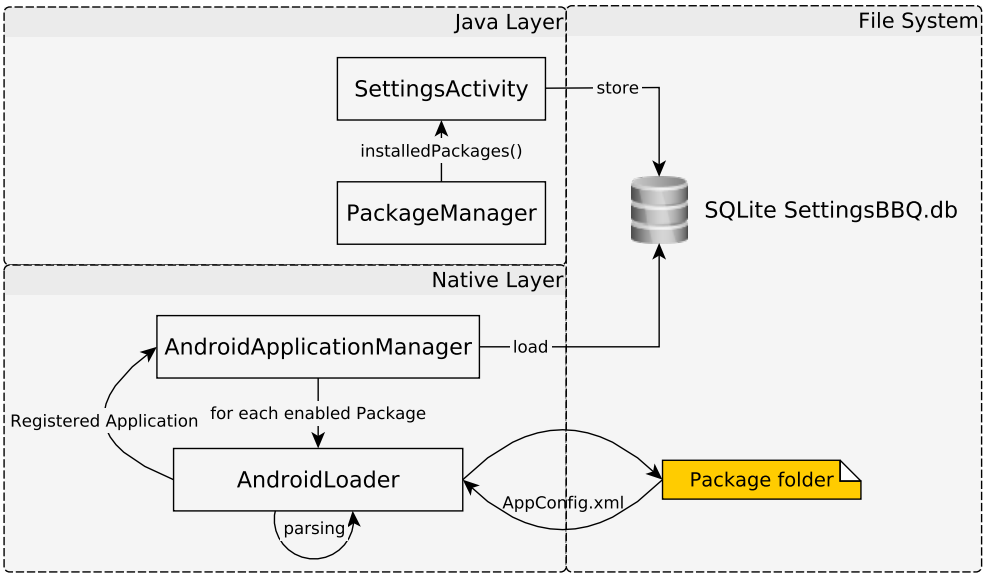


Figure 4.4: *The applications enabling and registering architecture.*

When an application is launched, the resource manager handles the request. In general the *Barbeque Service* is in charge of notifying to the *Android Application Proxy* that a new application has to be started attaching some application information such as name and package. The *Android Application Proxy* then performs the following steps:

1. It creates a new *Execution Context (EXC)* with the application information and its recipe
2. It enables the EXC
3. It notifies to the *Resource Manager* that a new application needs to be scheduled.

To realize the management of resources and Capabilities the *Resource Manager*, driven by the scheduling notification event, requests a rescheduling to the *Scheduler Manager* which in turn invokes the currently plugged *Schedule Policy*. This is in charge of retrieving Capabilities and resources system view[3] from the *Android Application Manager* and *Resource Accounter* respectively and computes the binding between the application constraints and available Capabilities and resources. When the scheduling is completed the policy leaves the control again to the *Resource Manager* that invokes the *Synchronization Manager*, which performs the commit phase: the scheduled views become effective in the system.

[3] This is an overall system view which contains information about the current resources allocation and availability

Because of the particularity of Android applications it may not be possible for the *Android Platform Proxy* to directly act on the resources. In fact we need to know the PID of the applications we want to assign resources to, but if the application is not already running we simply do not have this information. The solution to this problem is faced by the *Barbeque Service* as explained below.

When the *Synchronization Manager* commits the Capabilities schedule view, the *Android Application Manager* sends a start command for the scheduled application to the *Barbeque Service* to effectively launch the application (through the *Android Application Proxy)* and recomputes the available Capabilities and the set of runnable applications.
Finally, the *Barbeque Service* is in charge of collecting the command messages and if necessary starts the applications. At this point, it can wait for the PID of the started applications and sends it back to the *Android Application Manager*, that can invoke the *Platform Manager* to enforce the resource assignments.
As explained, this workflow is different from the Barbeque AEM, since the application cannot be reconfigured at run-time.

4.2.3 The Barbeque Service

In the Java layer we implemented a *daemon service* called *Barbeque Service* that acts as an interface towards Android. It allows the communication between the resource manager and the Android applications (non-integrated or energy-aware).

Goals

The *Barbeque Service* runs as a background Android service starting at boot time and communicating with the native BarbequeRTRM daemon:

- To retrieve features of the device during the *Android Application Manager* initialization through the *Android Package Manager* (see Appendix A.4.1)
- To expose a binding interface and API for other applications: with which it is possible to retrieve available Capabilities, applications or other information from the resource manager (see Section 4.4 for binding and API details)
- To perform the application launch and retrieve its PID through the Android *Activity Manager* (presented in Appendix A.4.2)
- To interact with the user via *Notifications*
- To set Android applications to "enabled" state.

Communication interface

The BarbequeRTRM and *Barbeque Service* intercommunicate through a RPC[4] mechanism. Because of the heterogeneous context of the two components (the resource manager is in the native layer while the service is in the Java one), we decided to implement this RPC mechanism through the Google *gRPC* framework. A practical tutorial about gRPC can be found in the Appendix B.
Google RPC exploits the Google Protocol buffers which allow to define a communication interface through a `.proto` file written in a Interface Definition Language (IDL). The general idea is to expose different communication services that contain the remote available functions. The parameters and objects are defined as `Message` type in the same file. The advantage of Protocol buffers is that messages and RPC definitions are language independent, so the interfaces are defined once in the IDL format, leaving to a compiler the burden of translating the definition into classes and methods language-specific. In addition the code devoted to the serialization of the messages data structure is also automatically generated.
Google RPC adds various plugins to manage the communication channels as well as the server and client implementations.

For the communication between the Barbeque Service and the Barbeque Android Application Proxy we defined the `.proto` interface file shown in Listing 4.1.
The interface includes the *Android Application Control* service which is implemented on the Barbeque Service side and the *Android Application Request* service that is implemented on the Android Application Proxy side.
The *Android Application Control* service exposes two remote functions:

- `GetAvailableFeatures`: to return the list of available features of the device
- `DoAction`: it accepts an `ApplicationControlMessage`, which contains the information about the application and the required action, and returns a `ReturnCode` message. This function is called by the BarbequeRTRM to send a command to the BarbequeService.

The *Android Application Request* service, otherwise, defines other two remote functions:

- `GetAvailableApplications`: to return a list of information of the available applications

[4] Remote Procedure Call

- `RequireAction`: to require to the resource manager the action set in the `ApplicationControlMessage`, passed as argument, and to return a `ReturnCode` message. This function is called by the BarbequeService to send an event request to the BarbequeRTRM.

Listing 4.1: *Protocol buffer communication interface definition between the* Barbeque Service *and the Barbeque* Android Application Proxy

```
service AndroidApplicationControl {
  rpc GetAvailableFeatures (EmptyParam) returns (stream Feature) {}

  rpc DoAction(ApplicationControlMessage) returns (ReturnCode) {}

}

service AndroidApplicationRequest {
  rpc GetAvailableApplications (EmptyParam) returns (stream AppInfo) {}

  rpc RequireAction(ApplicationControlMessage) returns (ReturnCode) {}
}

message ApplicationControlMessage {
  AppInfo info = 1;
  enum Action {
    START = 0;
    STOP = 1;
    REQUEST_SCHEDULE = 2;
    REQUEST_STOP = 3;
  }
  Action action = 2;
  int32 id = 3;
}
message AppInfo {
  string name = 1;
  string package = 2;
  int32 pid = 3;
}
message Feature {
  string feature_name = 1;
}
message ReturnCode {
  int32 code = 1;
}
message EmptyParam {}
}
```

The possible actions are defined through an `enum` type into the `ApplicationControlMessage`:

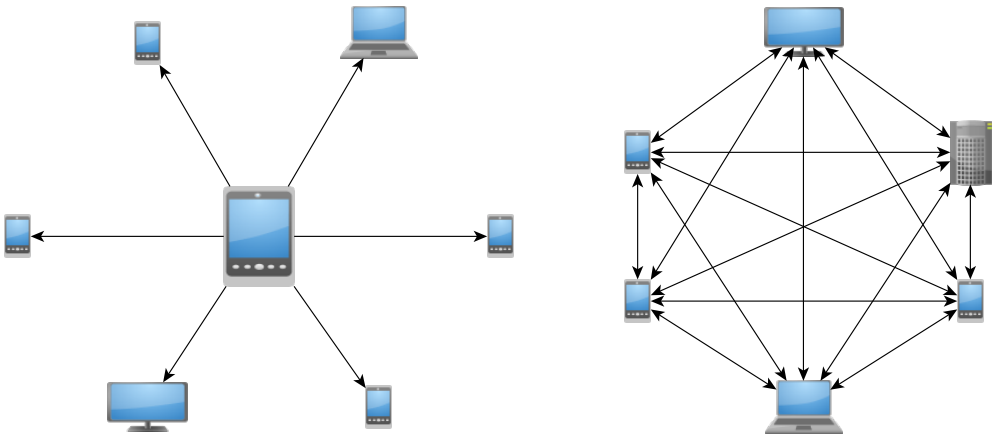


Figure 4.5: *The different distributed topologies available with BarbequeRTRM: a centralized version (figure a) and a fully-distributed one (figure b). Single-edge arrows indicate a master-slave relationship.*

- START: it is the application start command
- STOP: it is the application stop command
- REQUEST_SCHEDULE: it requires an application to be scheduled
- REQUEST_STOP: it requires an application to be stopped.

4.2.4 Distributed resource management

To enable our concept of a Pervasive Distributed Computing System, we had to port the applications management into a distributed context. For this reason the BarbequeRTRM has been extended to operate also according to a distributed configuration. In this section we provide an overview of this distributed resource management architecture.

In the distributed configuration, each device has a local running instance of the BarbequeRTRM. The resource management strategy can work through a *hierarchical* topology, in which one device acts as the "master" instance and coordinate the other "slave" instances, or in a *fully-distributed* topology, in which each device acts as autonomous agent and a coordination protocol is used to take decision (Figure 4.5). The development contribution of this work has introduced the possibility of adopting an *hybrid topology*. In this case, each device can act as a coordinator towards other devices relatively to the local user-selected application management, while the global resources management is a prerogative of each BarbequeRTRM instance.

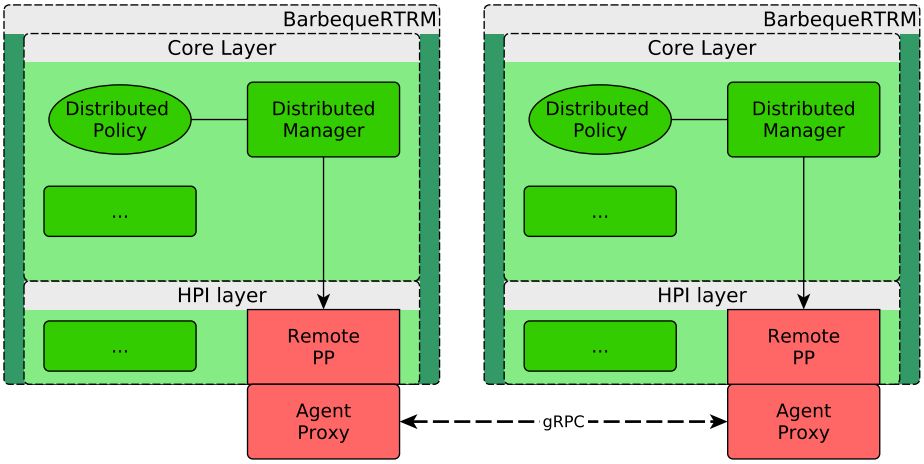


Figure 4.6: *The BarbequeRTRM distributed architecture*

BarbequeRTRM distributed systems extension

Figure 4.6 shows the general concept of the BarbequeRTRM distributed architecture. In particular the *Distributed Manager* module in the BarbequeRTRM core layer is in charge to manage the distributed group. In this sense, at time of writing, it is planned to:

- Manage the system topology, implementing a management of the group coordinator election algorithm
- Face changes in the available systems group (new systems discovery, nodes failure management), implementing a discovery and join algorithms, fault-tolerant mechanism and protocol
- Implement a coordinated decision protocol
- Implement a protocol to reach consistency and to collect the global status and runtime statistics of the BarbequeRTRM instances.

The conceptual design includes the possibility to plug a *Distributed Policy* which implements the aforementioned specific protocols so that they can be interchanged on demand. The *Distributed Manager* communicates with the *Remote Platform Proxy* which is a module of the Hardware Platform Integration Layer providing the functions needed to get information from other remote instances of the resource manager. The *Agent Proxy* plugin then implements the communication interfaces based on gRPC as previously explained.

In this work the distributed configuration follows an hybrid topology, because all the single instances of BarbequeRTRM are responsible for the global system

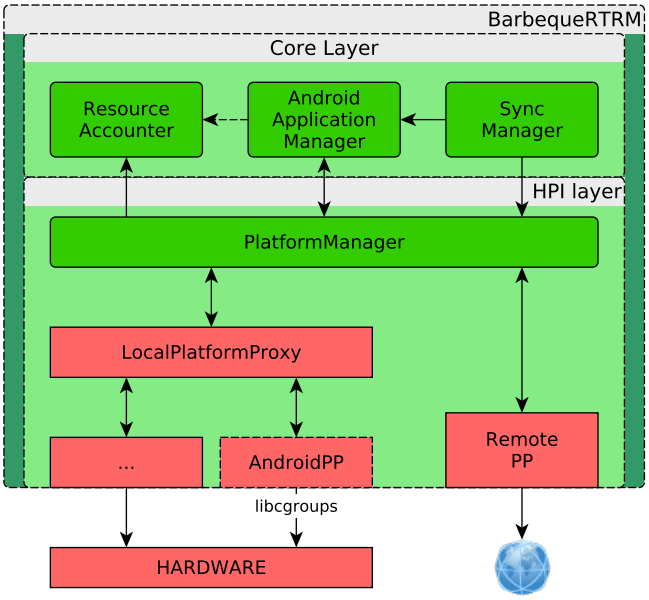


Figure 4.7: *The BarbequeRTRM Hardware Platform Integration Layer architecture*

resources and the Capabilities concerning only the application they scheduled. Moreover the set of interconnected devices is loaded by the *Distributed Manager* from a configuration file which contains all the available devices and their information. Currently, the only implemented functionalities of the *Distributed Manager* are related to have a consistent view of all available devices and their updated status and to manage remote device failure.

Hardware Platform Integration architecture

The conceptual diagram shown in Figure 4.7, describes the components of the Hardware Platform Integration Layer after the introduction of the support for the distributed systems.

The *Platform Manager* is the module that abstracts the underlying platform hierarchy.

In BarbequeRTRM the hardware platform resources are provided in a transparent way through the *Platform Proxy* interface that is effectively implemented by the *Local Platform Proxy* and the *Remote Platform Proxy*. The former provides an access to the local machine's resources (e.g., local CPU or OpenCL devices), dispatching the required operations to the appropriate specific proxy. These proxies can be easily added extending the *Platform Proxy* base class. For the use-case scenarios in which BarbequeRTRM is adopted three local prox-

ies are currently implemented: the *Linux Platform Proxy* which uses the Linux Control Groups (cgroup) framework for the cores and memory allocation; the *OpenCL Platform Proxy* devoted to assigning a specific GPU to an OpenCL application; and the *Android Platform Proxy*, which has been introduced with this work exploits cgroup management to assign CPU shared time, freeze processes and the cpufreq framework to control the CPU frequency governor parameters.

Communication interface

At time of writing the *Agent Proxy* communication interface is under development with the Google RPC framework, even if the plugin-based design allows the utilization of different communication protocols.

For the communication between the BarbequeRTRM instances we define a `.proto` interface shown in Listing C.1 of Appendix C.

It defines a single *Remote Agent* service implemented by each BarbequeRTRM *Agent Proxy* instance. This service exposes various remote functions:

- `GetSystemInfo`: this function returns a `SystemInfo` object that contains different information about the remote system (e.g., battery level, recharging status, system name) and model
- `GetAvailableApplication`: it returns a list of information about the available applications
- `SetApplicationManagementAction`: this function takes an `ApplicationManagementRequest` as argument which contain a specified action to be performed. This function is called by the coordinator BarbequeRTRM instance to perform the specified action for an application on the remote BarbequeRTRM instances. The possible actions are defined in the `enum` type of the `ApplicationManagementRequest` but since this part of the system is in the early design step it may change or be extended in further months. At this moment the message defines:
  - START: it start an application on the target device
  - STOP: it stop the execution of an application on the target.
- `SetCapabilitiesViewManagementAction`: it takes an `CapabilityManagementRequest` as argument with a specified action inside and a set of Capabilities. It is called by the coordinator BarbequeRTRM instance to perform operations on the remote Capabilities view. The available actions are defined in the `ViewAction` enum as:

  - GET_VIEW: it returns the specified view
  - BOOK: it books the given capabilities of the specified view
  - PUT_VIEW: it puts the given view
  - COMMIT_VIEW: it performs the commit on the specified view.

- `SetResourceManagementAction`: this function accepts a `Resource-ManagementRequest` which contains a set of `CpuConfig` object. This object specifies the core number and a clock frequency value for that core.

4.3 *BestWing* distributed device selection policy

The resource manager needs a policy to face the problem of choosing in which device launch a task. Since the goal of our work is to develop a "proof-of-concept" of the architecture, in our scenario the policy takes care of launching the entire application. Therefore, the policy is responsible of allocating the required resources and capabilities in an effective way. We developed a distributed device assignment policy which performs some steps to detect the different available devices and to identify the best device for the offloading of each scheduled application:

1. Retrieving the information about all interconnected devices from other BarbequeRTRM instances, like the battery level or plugged status

2. Computing the lowest *Energy Delay Product* (EDP) factor (explained in Subsection 5.2.1) and the corresponding CPU clock frequency for each available device according to an energy-efficiency model

3. Calculating the *Energy Consumption Index* (further explained in Subsection 4.3.1) considering also the battery level of the evaluated system.
   If the evaluated device is plugged and the temporary best does not, the former is marked as temporary the best. Otherwise the one with the lowest index is marked as temporary best

4. For the device marked as temporary the best, the policy verifies if the requested application is in the available applications set of the device

5. Once all systems have been considered, the policy takes the best device and books the capabilities and resources required by the application (*ResBooking* function further explained in Subsection 4.3.3)

6. Sending the launch request for the application to the selected device.

BestWing is a greedy policy since at each system-cycle it looks for the best device. Moreover, some steps are optimized to minimize the lost time due to communication latency or to avoid recomputation of information that are already available, as explained in Subsection 4.3.4. As stated, this is a sample policy that exploits the aforementioned Android BarbequeRTRM architecture. This enabling step allows to develop more sophisticated and efficient policies in the future.

In a formal way, from the 3.7, 4.2 and 4.6 definitions, the schedule operation can be modeled with the following function:

$$\begin{aligned} \text{AppSchedule:} A \times \wp(D) &\rightarrow D' \\ \langle App, DevSet \rangle &\mapsto Dev'' \end{aligned} \tag{4.1}$$

Such that:

- $\exists Dev \in DevSet, \exists Dev'$
- $e_{Dev,App} = \min_{Dev_i \in DevSet} \{e_{Dev_i,App}\}$
- $Dev' = AppFiltering(Dev)$
- $App \in A_a^{Dev'}$
- $Dev'' = ResBooking(Dev', App)$.

Where $e_{Dev,App}$ denotes the *Energy Consumption Index* as defined in the following.

4.3.1 Energy Consumption Index

For the policy implementation we used an index to characterize with a "score" the choice of a specific device. This in order to compare different devices and find the best one for the application execution. The index is based on an energy efficiency model that consider the *Energy Delay Product* (EDP) (whose measurement is explained in Subsection 5.2.1) and the battery level of devices. The lower is the index the higher is the probability that the policy will select the given device.
We therefore defined the *Energy Consumption Index* (e) of device i for application app as:

$$e_{i,app} = \alpha * B_i + \beta * min\{E_{i,app}(f)\ \forall f \in Freq_i\} \tag{4.2}$$

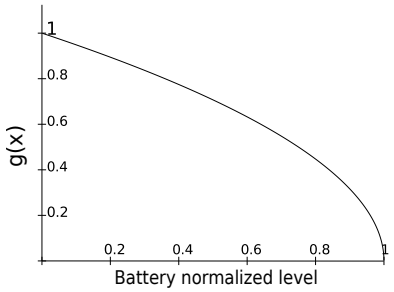


Figure 4.8: *The graph of the actual $g(x)$ Battery scaling function we selected for our implementation.*

As we can see, equation 4.2 is composed by two terms: the battery factor and the energy efficiency factor. In particular the B_i is a function defined as following:

$$B_i = g(x) \tag{4.3}$$

Where $g(x)$ is a function that scales the score to the actual normalized battery level. For our purpose we chose a function of type $g(x) = \sqrt{1 - ||x||}$ where x is the device battery percentage. In this way the 4.3, as shown by Figure 4.8, decreases slower for high battery levels despite the 4.4 term, giving a lower (i.e., better) value to device with low EDP values than high battery levels in the index calculation.

The second term of 4.2 represents the minimum EDP value – over all the ith device available frequencies set $Freq_i$ – of the $E_{i,app}(f)$ function defined as:

$$E_{i,app}(f) = EDP_{norm_{i,app}}(f) + EDP_{norm_{i,com}} \tag{4.4}$$

Where $EDP_{norm_{i,app}}(f)$ is the normalized EDP value of the specific application and device at frequency f, as defined in 5.4, while $EDP_{norm_{i,com}}$ is the normalized EDP value for the communication operations. This last term is defined as $||E_{com_i} * T_{com}||$, where E_{com_i} is the energy consumed for data transmission operations by device i and T_{com} is the communication time overhead to launch an application remotely.

The two terms are multiplied by the coefficients α and β which can be set according to different balancing strategies, depending on which is the policy goal: whether we want to give priority to the battery duration or the efficiency of the application execution. The Figure 4.9 shows the different values of the index with respect to the different normalized EDP and battery level in a balanced strategy ($\alpha = \beta = 0.5$) considering negligible the communication $EDP_{norm_{i,com}}$ term. It can be noticed that in the example configuration, until a certain threshold, the computed ECI tends to make the policy select devices with low EDP values

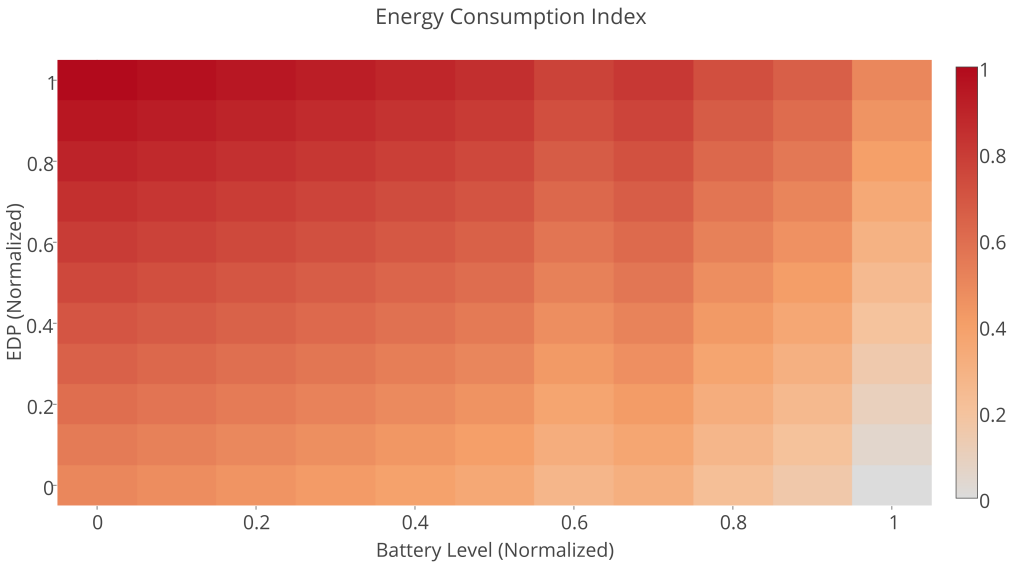


Figure 4.9: *The map of the different values taken by the Energy Consumption Index in a balanced strategy considering negligible the communication operations EDP.*

than an high battery level. For example considering a generic application, let be A a device with 80% battery level and 0.6 EDP value and B a device with 30% battery level and 0.2 EDP value: in this case the policy will select B as the best device due to its lower energy consumption index ($e_B = 0.518$ while $e_A = 0.523$), that is what we pursue. However the selected function of 4.3 preserves a reasonable trade-off between battery level and energy efficiency. In fact, continuing the previous example, a little less efficient ($E_C = 0.3$) but more charged ($B_C = 0.5$) device C has a lower index than the others ($e_C = 0.503$).

4.3.2 Capabilities booking

Once the best device is selected the policy takes care of booking the capabilities defined by the application. This step achieves two objectives: firstly it updates the availability of the target device capabilities set a new view, secondly it allows to allocate the resource subsumed by the booked capabilities.

Formally, we can model the *CapBooking* function as following:

$$\begin{aligned} \text{CapBooking}: & D' \times A \to D' \\ & \langle Dev, App \rangle \mapsto Dev' \end{aligned} \tag{4.5}$$

Such that:

1. $Dev \cong Dev'$

2. $\forall c_i \in C_r^{App}, \exists c_j \in C_b^{Dev'} \wedge (c_i \in C_{excl} \Rightarrow \neg \exists c_k \in C_a^{Dev'}(c_i = c_j = c_k))$

3. $\forall c_i \in C_o^{App}, \exists c_j \in C_b^{Dev'} \wedge (c_i \in C_{excl} \Rightarrow \neg \exists c_k \in C_a^{Dev'}(c_i = c_j = c_k))$.

The condition 3 defines the possibility to allocate the optional capabilities if they are available.

4.3.3 Resources booking

For each booked capabilities the policy is in charge to allocate their subsumed resources. We can model this operation through the following *ResBooking* function:

$$\begin{aligned} \text{ResBooking:} & D' \times A \rightarrow D' \\ & \langle Dev, App \rangle \mapsto Dev'' \end{aligned} \tag{4.6}$$

Such that:

1. $\exists Dev'$

2. $Dev' = CapBooking(Dev, App)$

3. $\forall c_i \in C_b^{Dev'}, \forall r_j \in R_r^{c_i}, \exists r_k \in R_a^{Dev'}, \exists r_h \in R_a^{Dev''} \mid$
   $r_j.ResPath = r_k.ResPath = r_h.ResPath \wedge r_h.value = r_k.value - r_j.value$

4. $\forall c_i \in C_b^{Dev'}, \forall r_j \in R_o^{c_i}, \exists r_k \in R_a^{Dev'}, \exists r_h \in R_a^{Dev''} \mid$
   $r_j.ResPath = r_k.ResPath = r_h.ResPath \wedge r_h.value = r_k.value - r_j.value$

5. Dev' and Dev'' differ only for the R_a set.

The dot symbol represents an access to the correspondent field of the resource pair as defined in 3.1.
The condition 3 refers to the allocation of all the resources required by the capability. Otherwise, the condition 4 defines the possibility to allocate the resources that are optional for the capability.

4.3.4 Policy optimizations

To avoid data recomputation and to minimize the communication latency lost time we performed some optimizations to the policy steps as further explained:

- The local system is the best system as default

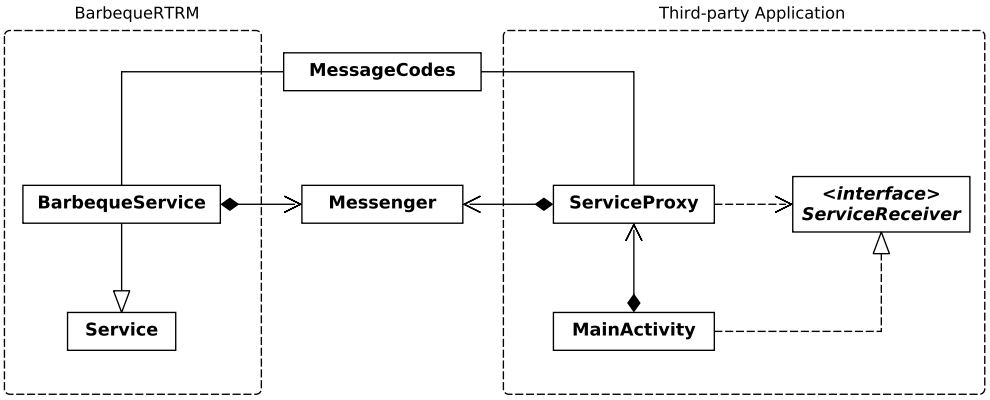


Figure 4.10: Barbeque Service *and class diagram of API provided to enable the support of third-party Android applications.*

- To avoid unuseful recomputation the EDP calculation step is skipped if the actual evaluated device is the same of the temporary best device
- If the temporary best device is plugged but the evaluated one is unplugged the unplugged device is skipped.

4.4 Android BarbequeRTRM API

We extended the BarbequeRTRM framework providing a set of API to allow third-party Android application to interact with the BarbequeRTRM. The API includes a main Java class, the `ServiceProxy`, and a callback interface, the `ServiceReceiver`. The former manages, in a transparent way to the developer, the binding operations and communications towards the `BarbequeService` daemon, the latter provides a set of callback methods that are called by the `ServiceProxy`.

The general idea, shown in Figure 4.10, is that the third-party application instantiates a `ServiceProxy` object in its activity. The activity has also to implement the `ServiceReceiver` interface callbacks to handler incoming messages from the daemon.

Listing C.2 of Appendix C shows the binding interface exposed by the *Barbeque Service* daemon, whereas Listing C.3 of Appendix C shows the binding operation in the proxy side (lines 29 – 41). With the `ServiceProxy` API, it is very simple for an external application to do the binding with the daemon service and, so, to use the resource manager API.

To maintain the external application completely unaware of the underlying binding operation, the `ServiceProxy` class implements some public methods

that can be invoked by the application:

- `doBindService`: to perform the daemon binding
- `doUnbindService`: to unbind from the daemon service.

The binding management can be summarized as follow. The *Barbeque Service* overrides the `onBind` method which is called when the binding operation is requested by a component of the third-party application and it returns a `IBinder` from its own `Messenger` member (line 9 of Listing C.2). Accordingly the `ServiceProxy` class instantiates the messenger reference of the *Barbeque Service* from the same `IBinder` passed to the `onServiceConnected` callback method (line 14 of Listing C.3). This is called once the binding operation is completed and the `onBind` function of the service returns the `IBinder` object. Finally, the `ServiceProxy` calls the `onServiceConnected` callback of the `ServiceReceiver` interface (line 15). In this way the adopted design approach makes all the above binding mechanism and protocol managed transparently to the developers. The latter is only in charge to implement the `onServiceConnected` callback to perform application-specific operations.

Once the binding is done, the third-party application and the daemon intercommunicate through the `ServiceProxy` object which exchanges messages directly with the daemon, exploiting the `Messenger` interface. A message contains a `RequestCode` value that is provided by the daemon API and can have attached a `Bundle` with different objects inside. At this step of development we implemented a set of `RequestCode` that can be exploited by a third-party launcher application:

- MSG_GET_APPLICATIONS: it is used to retrieve the available applications from the resource manager
- MSG_GET_CAPABILITIES: it is used to retrieve the available capabilities of the device
- MSG_START_APP: it requires a specified application to start
- MSG_STOP_APP: it requires a specified application to stop
- MSG_START_APP_ERROR: it is generally sent by the resource manager to notifies an error to the launcher during the starting of an application (e.g., if the activity is not found).

Listings C.4 and C.5 of Appendix C show sample sending and receiving message handling.

To keep the transparency of the communication the `ServiceProxy` class implements some public methods invokable by the external application:

- `getApplications`: to retrieve the available applications
- `startApplication`: to start a specific application.

The `ServiceReceiver` interface defines the following main callbacks invoked by the `ServiceProxy` once an operation is completed:

- `onApplicationsReceived`: called once the applications are provided by the resource manager. It accepts the retrieved list of information about the application as parameter
- `onStartAppError`: called if an application fails to start
- `onServiceConnected`: called once the `ServiceProxy` completed the binding with the *Barbeque Service*.

4.5 Application launching schemes

The Section presents two application launching schemes enabled by this work. The schemes are effectively implemented for our test purposes through two launchers that follow an energy-aware approach and exploit the BarbequeRTRM API. The first schema is presented in the Subsection 4.5.1 and concerns the launching of the application locally in the device. The second one, instead, is a remote launching schema. In this way the Subsection 4.5.2 explains in depth the distributed mechanism enabled and implemented by our work, providing also a possible use case scenario.

As aforementioned, the following sample applications are two customized Android launchers. As a launcher, it shows to the user a list or a grid of available applications and allows it to start a selected application.
In Android OS the launcher can be customized by installing a custom one chosen from a wide set from the marketplace. Anyway different smartphone manufacturers take the original Android OS image and develop it with own features, system applications and also launcher.
Our approach is similar, because we can encapsulate our launcher in a customized version of Android in which BarbequeRTRM is pre-installed as system resource manager.

4.5.1 Local launching

In this scenario the launcher shows the applications installed on the device. The novel approach is that the list will contain only applications made available by

the resource manager. In fact, this launcher is resource and energy aware and has some energy efficiency benefits because it does not show applications that cannot be actually run in the device due to the current workload or low power availability. It has also some user experience benefits, because being resource and energy aware prevents an application to run slow or not in an optimal mode, reducing the user experience and consuming uselessly power and resources of the device.

With reference to Figure 4.10 our launcher implements the `MainActivity` class. To perform its work the launcher, exploiting the BarbequeRTRM API, binds the *Barbeque Service* daemon and communicates with it through a `ServiceProxy` which uses a `Messenger` and a linked `IBinder` as exaplined in Section 4.4. Listing C.6 of Appendix C shows the binding code from the launcher side.

The activity of our application instances a `ServiceProxy` object and calls its `doBindService` method which performs transparently the binding operation (line 12). Once the binding is done, the application can use the `ServiceProxy` API methods to make request to the *Barbeque Service* daemon.

For example, Listing C.7 of Appendix C shows a sample request to obtain available applications from the BarbequeRTRM: our launcher activity needs only to call the `getApplications` method of the `ServiceProxy` object (line 14) and implements the `onApplicationsReceived` callback of the `ServiceReceiver` interface to perform the required operations with the returned information.

4.5.2 Remote launching

In this scenario the launcher shows the applications available also in other locally interconnected devices in which the BarbequeRTRM is running. The selected application can be launched also on another device because, for example, the user wants a video application to run on an Android TV or because the device has a low battery status and there are other unused close devices in which the application can be run. The choice of the target device can be manually or automatically set depending on the current policy activated in the BarbequeRTRM. In the first case the launcher should allow the users to choose in which device they want to launch the application (if necessary, passing also the data).
For our purpose we implemented the second case in an energy-efficient approach which details are provided in the following section.

To implement this scenario, we can imagine an Android *Custom Remote launcher* application. It uses the daemon API as the previous one, but now we

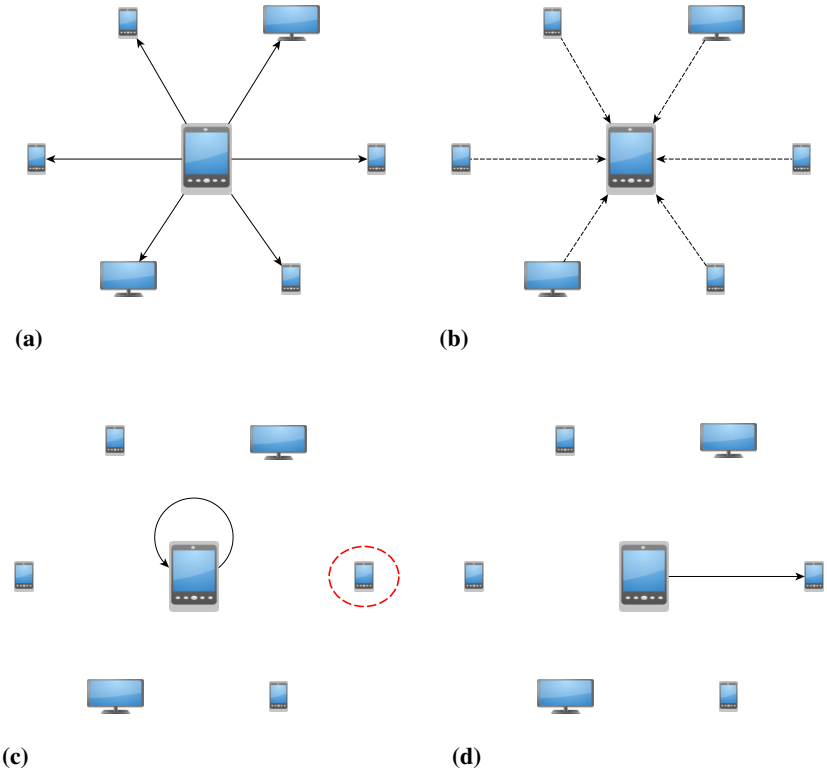


Figure 4.11: *The four main steps of an Android application remote launching. The red circled device is the best selected by the schedule policy.*

need to exploit the distributed features of the BarbequeRTRM. In this sense the Section 2.2 presented the mobile offloading as one of the major research topic for mobile systems in recent years. This work enables a first integration step of the offloading approach with the BarbequeRTRM. In fact, the main lack of the state-of-the-art approaches is the absence of the resource management perspective. Moreover we provide a first implementation of a Pervasive Distributed Computing Network System through common personal devices. In this sense we started to face the resource management problem also in service-based architectures of smart environments [122].

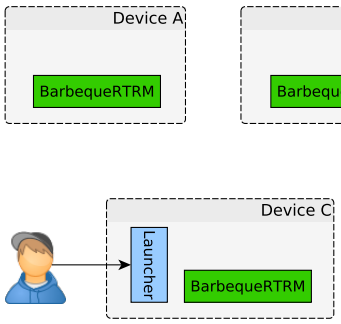
To exploit the distributed features of the BarbequeRTRM it is required that each device has an instance running and registered into one computing group. Thanks to the hybrid architecture of our resource management strategy each instance can take decision with respect to application which it has competence. In particular a device $D1$ becomes the coordinator of application $A1$ if the user interacts with the device $D1$ to start the application $A1$.

In the evaluated scenario the user selects the application through the *Barbeque Remote Launcher* which is very similar to the one presented in the previ-

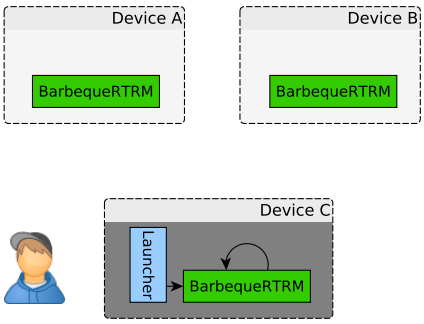
ous section.
Once the application is selected the coordinator BarbequeRTRM instance emits a rescheduling event which cause the device selection policy to perform a rescheduling considering also the new application. Figure 4.11 show the main step of the remote launching process. First of all the BarbequeRTRM coordinator retrieves the updated status of the connected devices (Figure 4.11a and 4.11b). Then the policy calculates the best device into which launch the application (Figure 4.11c). Finally, as shown by Figure 4.11d, the application is launched in the selected device (in this case a remote one).

To better understand a practical exploitation of our distributed system we can think to a sample application, like a game, which is deployed on a distributed architecture. We can assume the application is composed by a *Master* or *Server* package which performs the computing-intensive task and a *Client* package which implements the user interface and can performs lightweight computing tasks. The application can be launched either in the Server or in the Client operational mode by the user.

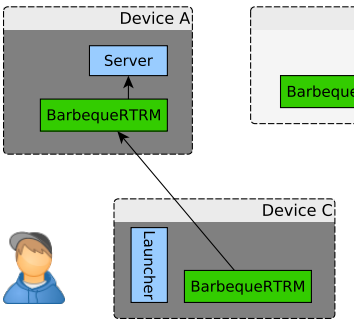
In such a scenario the BarbequeRTRM could be set to manage the Server package of the application so that the most energy-efficient device is selected to run the Server instance while the Client side can be launched on the users devices. The management process is shown in Figure 4.12. The Figure shows an environment composed by three devices, named A, B and C, which run a BarbequeRTRM instance each. Firstly, the user selects the game Server module from the Launcher of Device C (Figure 4.12a). The launcher then forwards the request to the BarbequeRTRM through the aforementioned API. When the BarbequeRTRM has been notified it performs the policy steps to select the best device (Figure 4.12b). At this point the coordinator BarbequeRTRM on Device C performs all the required operations to launch the Server module application on the Device A (Figure 4.12c). The last two steps concern the Client module of the game. In particular the user receives the remote launching notification and decides to launch the game Client module (Figure 4.12d). Once also the Client module is running the user interacts directly with the Client module of Device C which in turn can request computation and receive the outcome from the Server module running in the Device A (Figure 4.12e). To simplify the scenario we assume that the application instances are in charge to manage intercommunication and running mode. However one of the main research topic on our work should investigate the possibility to allow the BarbequeRTRM to interact with the application and set its operational mode basing on the resource and energy context changes.



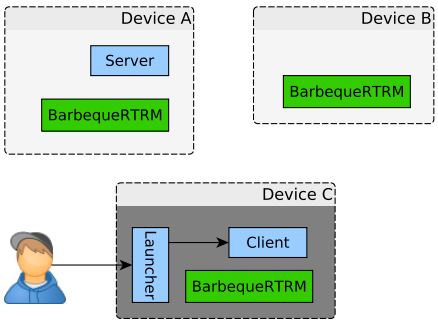
(a) *The user selects the Server module from the launcher of Device C.*



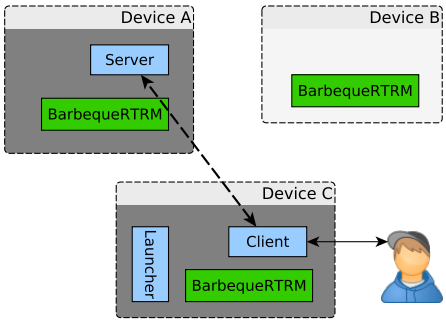
(b) *The BarbequeRTRM schedule the application and selects the best device.*



(c) *The BarbequeRTRM launches the Server module in the device A.*



(d) *The user launches the Client module in the Device C.*



(e) *The user interacts with the Client which exchanges messages with the Server on device A.*

Figure 4.12: *A simple game application distributed across the PDCN by the BarbequeRTRM.*

CHAPTER 5

Experimental Results

The main goal of this Chapter is to present the experimental phase of our work. The first part aim at providing an overview of the hardware and software setup and measurements methodology, whereas the second part introduces the experimental results about the benchmarks profiles, application classification, effects introduced by our energy-aware approach and, finally, the resource management overheads.

5.1 Introduction

During the experimental phase we were guided by four main questions that have been raised during the design and implementation of the framework extension:

1. How can we measure the application energy efficiency?
2. What considerations can we make on the obtained models?
   - Can we build a single general model?
   - Can non-benchmark applications be classified according to the obtained energy models?
3. What is the impact of adopting an energy efficiency approach in terms of device lifetime and application performance?

OS information	*NEXUS 5_5_0*	*NEXUS 5_6_0*	*ODROID-XU3*
Android version	5.0	6.0.1	4.4.4
Linux Kernel version	3.4.0-ElementalX-N5-6.00	3.4.0-g7717f76	3.10.9-gefae4c9
Build version	LRX21O	MMB29K	KTU84Q

Table 5.1: *The table shows the software information of the two Nexus 5 and the Odroid-XU3 used in our evaluation setup. The device name is used in our setup to distinguish them according to their software version.*

4. What are the overheads introduced by the resource management layers?

In the following pages we will provide detailed data and considerations to give answers to the aforementioned questions.

Since we evaluated the implemented solution on real devices, in Subsection 5.1.1 we provide some details about the used devices, the distributed system setup and the measurement methodology. Next, the Subsection 5.1.2 presents in details the two benchmark suites we used in our experiments.

As anticipated in Section 4.3 we followed an energy efficiency-aware approach to guide the device selection and implement the *BestWing* policy. In this regard Subsection 5.2.1 provides an overview of the energy efficiency objective with respect to our problem.
The first goal of this experimental phase has been to extract an energy-efficiency model from the different benchmark applications by profiling them in different hardware configurations. In this sense, we built application-specific models and a general one. In Subsection 5.2.2 we compare the two modelling approach and discuss about generalization.
Furthermore, we have analyzed the device lifetime benefits introduced by our energy-aware approach compared to the loss in terms of performance. The results are provided in Subsection 5.2.4.

Moreover, we profiled three non-benchmark applications to compare their energy models with the benchmark one. This led us to define an approach in which applications can be grouped into separate classes as explained in Subsection 5.2.3. In such a way the policy only needs to know the class of the applications to apply the correct model without the need of a new profiling activity.

The Subsection 5.2.5 concludes the Chapter and it regards the evaluation of the device allocation policy. In particular, we evaluated the execution outcomes over different scenarios and the overheads in terms of execution time and scalability.

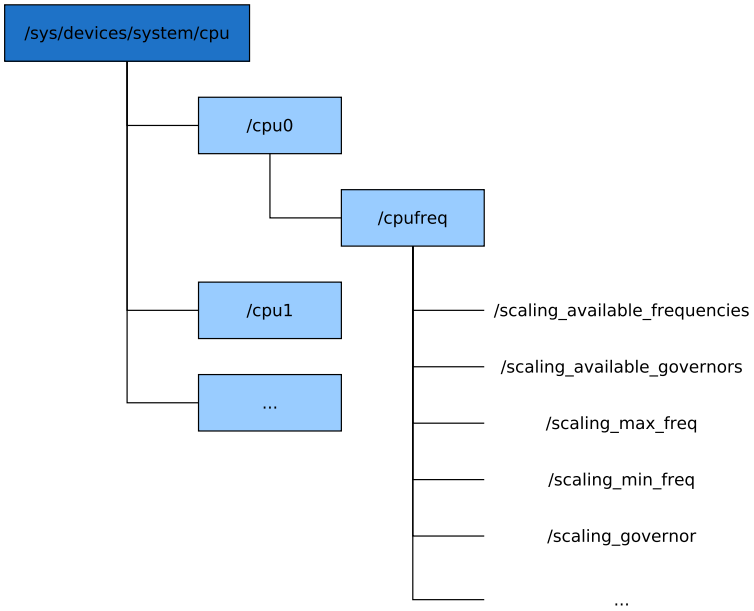


Figure 5.1: *Cpufreq framework file structure*

5.1.1 Experimental setup

For our testing purpose we have deployed the extended BarbequeRTRM on two Nexus 5 devices [123] and an Odroid-XU3 development board [124]. The Nexus 5 is a smartphone producted by LG Electronics and marketed by Google Inc from November 2013. It is powered by a 2.26 GHz quad-core Krait 400 CPU in a Snapdragon 800 System-on-Chip with 2 GB of RAM and a 3.8V 2300 mAh battery. It is also equipped with a 450 MHz Adreno 330 GPU.
The Odroid-XU3 is a development board powered by an ARM big.LITTLE technology on a Samsung Exynos5422 SoC with 2 GB of RAM. The "big" cluster is composed by a 2.0 GHz quad-core Cortex A15 CPU, while the "LITTLE" cluster is composed by a 1.4 GHz quad-core Cortex A7 CPU. It is also equipped with a Mali-T628 MP6 GPU.
To make sure that the system overhead did not affect our measurements we installed three different rooted versions of the Android OS. Table 5.1 summarizes the main software characteristics of the three devices.

In our implementation we exploited the Linux *cpufreq* framework to set the cores clock frequency of the device. The choice of the frequency is limited to a subset of them. Generally speaking the framework exposes a *sysfs*-based interface shown in Figure 5.1. It provides a directory-based structure with some files that contain the system information and setting parameters. In particular the *scaling_available_frequencies* and *scaling_available_governors* contain the

information about the available frequencies and governors[1].
For the Nexus 5 the available frequencies are (in MHz): 300, 422.4, 652.8, 729.6, 883.2, 960, 1036.8, 1190.4, 1267.2, 1497.6, 1574.4, 1728, 1958.4, 2265.6. Instead, for the Odroid-XU3 the available frequencies can be set in steps of 100 MHz between the minimum and the maximum frequencies depending on the considered cluster. In particular, for the "big" cluster the frequencies range from 1200 MHz to 2000 MHz and for the "LITTLE" cluster they range from 1000 MHz to 1400 MHz. For the profiling step of our experiments we used only a subset of them as shown in Tables 5.2 and 5.3. Finally, the available governors are [125]:

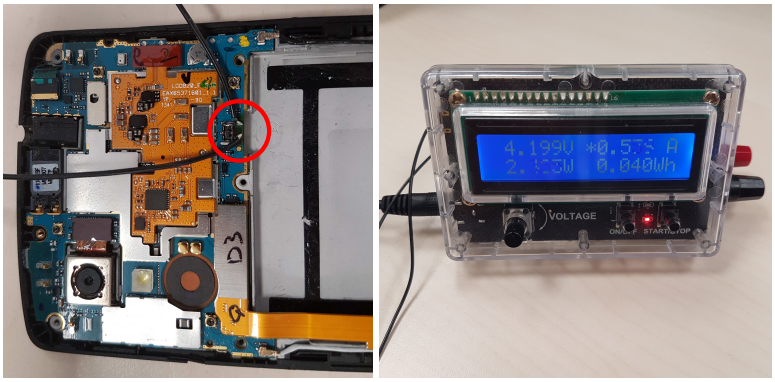
- *Ondemand*: it scales the frequency dynamically according to the current load jumping to the highest frequency and then possibly backing off as the idle time increases
- *Conservative*: it scales the frequency dynamically according to current load in a more gradually way than "ondemand"
- *Interactive*: it is designed for latency-sensitive, interactive workloads. It sets the CPU speed depending on usage like the "ondemand" and "conservative" governors. However, it is more aggressive about scaling the CPU speed up in response to CPU-intensive activity
- *Userspace*: it runs the CPU at user specified frequencies
- *Powersave*: it runs the CPU at the minimum frequency
- *Performance*: it runs the CPU at the maximum frequency.

The other files contains the current maximum, minimum frequencies and governor setting. These are the files we modify in order to force the desired frequency.

Distributed system setup

To setup our implementation of a Pervasive Distributed Computing Network System we interconnected the aforementioned devices through a wireless LAN. The Nexus 5 is equipped with an on-chip Wi-fi 802.11 a/b/g/n/ac network interface, while the Odroid-XU3 has needed to be extended with the optional Wifi Module 3 that is a Realtek RTL8188CUS-GR single-chip USB 2.0 network interface controller supporting the IEEE 802.11 b/g/n standards. Anyway the wireless access

[1] Governors are power schemes for the CPU.



(a) (b)

Figure 5.2: *Custom power supply and measurement setup for one of the Nexus 5 used in the energy consumption profiling. The red circle in Figure 5.2a highlights the power connectors attached to the Odroid Smart Power, shown in Figure 5.2b.*

point of our infrastructure has a 802.11 a/b/g interface so the maximum throughput of the network is limited to 54 Mbit/s using the 2.4 GHz bandwidth. The consequent low throughput allows us to evaluate the maximum communication overhead that would be reasonably caused by the hardware limitation.
The different devices are statically registered in each BarbequeRTRM instance. To evaluate the scalability of the policy with respect to the number of available devices, we deployed the BarbequeRTRM also on other Odroid-XU3 board and two smartphones.

Measurements setup

To measure the energy consumption of the application, in the profiling step we performed a custom setup for the Nexus 5. We removed the device battery and supplied the power with an Odroid Smart Power meter[2] through direct power lines connection as shown by Figure 5.2. It provided a fixed voltage to the device and collects at run-time the current, power and energy consumption values that can be logged in a file through an USB interface.
During the measurements we supplied the device with a fixed 4.2V voltage. The higher voltage value with respect to the battery nominal one is necessary to force the Android battery manager to interpret the current voltage as a about 50% of battery level. Moreover we switched off the wireless interface of the device, set to 50% the display brightness and deactivated the brightness auto-adjusting feature to avoid measurement bias due to external factors.
The Odroid-XU3 was powered directly through the aforementioned Smart Power

[2]http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137361754360

with a voltage set to 5V.
Since we measured the energy consumed by the benchmark in different core frequency configurations, we set the clock of the cores to a specific frequency taken from the *governor available frequency* pool, setting the scaling governor to `userspace` mode to grant us a complete control of it. Moreover, we disabled the `mpdecision` userspace daemon of Android, which is in charge to switch off the unused CPUs [110] and to avoid computation environment and power changes during our measurements. On the Odroid-XU3 board we measured each cluster separately, keeping online only their respectively cores.
Finally, due to thermal security mechanism of the SoC to test the device at the maximum clock speed we had to set up an artificial cooling system to avoid an automatic and unpredictable reduction of the clock frequency.

5.1.2 Benchmarks

Since our goal is to create an energy-efficiency model to manage common Android applications we exploited two Android device benchmark suites: the *MobileXPRT2015* [126, 127] and the *PCMark 2.0* [128]. They are composed by different *pseudo-applications* which simulate interactive or common tasks and test different hardware features and computing performance of the device. The benchmarks can be run by launching the entire set of part of it. In the first case the benchmark run all the pseudo-applications and computes a final overall score. The second case allows the user to select only a subset of the pseudo-applications to test. In what follow we presents in detail the two benchmark suites. Due to Android compatibility problem we cannot run the PCMark 2.0 suite on the Odroid-XU3 since it requires a 5.0 or higher OS version, while the official support for the Odroid-XU3 provided only OS images until version 4.4.4. Anyway, since our work aim at being an initial exploration of the profiling methodology we can make some comparisons and considerations only on the MobileXPRT2015 results.

The MobileXPRT2015 benchmarks suite

The MobileXPRT2015 is developed by the BenchmarkXPRT Development Community and released in 2015. It is composed by five pseudo-applications that mirror common mobile workloads:

- *Apply Photo Effects*: it applies four photo effects (Sepia, Vintage, Vignette, and Grayscale) to five 8-megapixel photos each
- *Create Photo Collages*: it creates five photo collages with four photos in

each one. Before including the photo in the collage it applies effects to each of them. The dataset used is the same of the previous pseudo-application

- *Create Slideshow*: it creates a slideshow video from a photo album and an audio file of 327 KB in a H.264 MPEG-4 compression format at a resolution of 720p. The photo used by this test are the same of the previous pseudo-applications

- *Encrypt Personal Content*: it encrypts with an 256 bit key AES[3] algorithm two sets of six files and then it decrypts them. The first set of files includes five photos and an MP3 audio file. The second set is composed by five photos and an MP4 video. The total size of the pre-encryption files is 165 MB

- *Detect Faces to Organize Photos*: it recognizes faces into a set of seven photos with a size range from 0.3 megapixels to 2.8 megapixels.

The benchmark calculates a performance score basing on a comparison to a calibration system[4] as follows:

1. For each pseudo-applications it calculates how many times the calibration system (T_c) is slower with respect to the evaluated one (T_d): $a_{App} = \frac{T_c}{T_d}$

2. It calculates the geometric mean of all the ratios, rounding it to two decimal places: $geomean = (\prod_{i=0}^{N} a_i)^{\frac{1}{N}}$ for $N = 5$

3. It multiplies the result by 100: $score = geomean * 100$.

The PCMark 2.0 benchmarks suite

The PCMark 2.0 benchmark suite is developed by Futuremark which creates industry standard benchmarks since 1997. Similarly to MobileXPRT2015 the suite provides an environment to test performance and battery life of Android devices simulating common tasks. Moreover it provides to the user some useful tools to analyze the results through an hardware monitoring chart which show the load, temperature and frequency of the CPU during the workload.
The Work 2.0 performance benchmark is a part of the PCMark 2.0 suite and comprises five common productivity tasks, for further details refer to the official technical guide:

[3] Advanced Encryption Standard

[4] The calibration system is a Motorola DROID RAZR M (Qualcomm MSM8960 Snapdragon S4Plus Dual Core at 1.5 GHz) running Android JB 4.1.2

- *Web Browsing*: it renders a pre-loaded web page performing scrolling, zooming and content searching through the Android WebView view. Finally it edits and adds an image re-rendering the page

- *Video Editing*: it plays, edits and saves a video using the OpenGL ES 2.0 MediaCodec API and the Google Exoplayer media player. All video files are encoded with the H.264 MPEG-4 AVC compression format and tested at different frame rates and resolutions (from 30 FPS at 1270x720 to 60 FPS at 3200x1800)

- *Writing*: it opens, edits and saves two documents using the Android native EditText view and PdfDocument API. Then it encrypts, decrypts and renders the PDF in a RecyclerView. The first document is a 2.5 MB ZIP file containing a 100 KB text file and two 1.2 MB images. The second document is a 3.5 MB ZIP containing a 90 KB text file, a 1.6 MB image and a 1.9 MB image

- *Photo Editing*: it opens, edits and saves a set of 4 megapixel JPEG images. The test uses four APIs (*android.media.effect*, *android.renderscript*,*android-jhlab* and *android.graphics*) to filter and manipulate the 4 megapixels images

- *Data Manipulation*: it parses data from four file formats (CSV, XML, JSON and Protocol Buffers). Then it presents the data with dynamic and animated charts using the open-source MPAndroidChart library interacting with them through animations and common simulated gestures.

The benchmark calculates a performance score starting from the pseudo-applications score. Higher score means better performance. The global score can be summarized by the following equation:

$$\begin{aligned} Work2.0score = geomean(&WebBrowsing2.0, VideoEditing, \\ &Writing2.0, PhotoEditing2.0, DataManipulation) \end{aligned} \tag{5.1}$$

Where $geomean$ refers to the standard geometrical mean formula. The pseudo-applications specific scores take into account time spent to perform the required tasks and frame rate measurements. More details can be found in the official manual.

5.2 Results

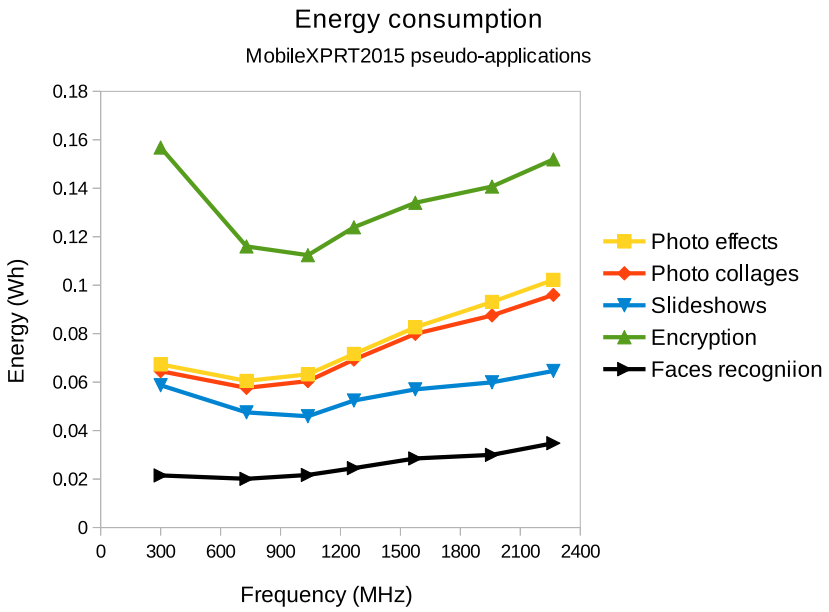
In this Section we present and comment the results obtained from our experiments to give answers to the questions raised at the beginning of the Chapter. The first question is covered by the Subsection 5.2.1, whereas Subsections 5.2.2 and 5.2.3 aiming at giving detailed results for the second question. Finally, the third and fourth questions are treated by Subsections 5.2.4 and 5.2.5.

5.2.1 Applications energy efficiency profiling

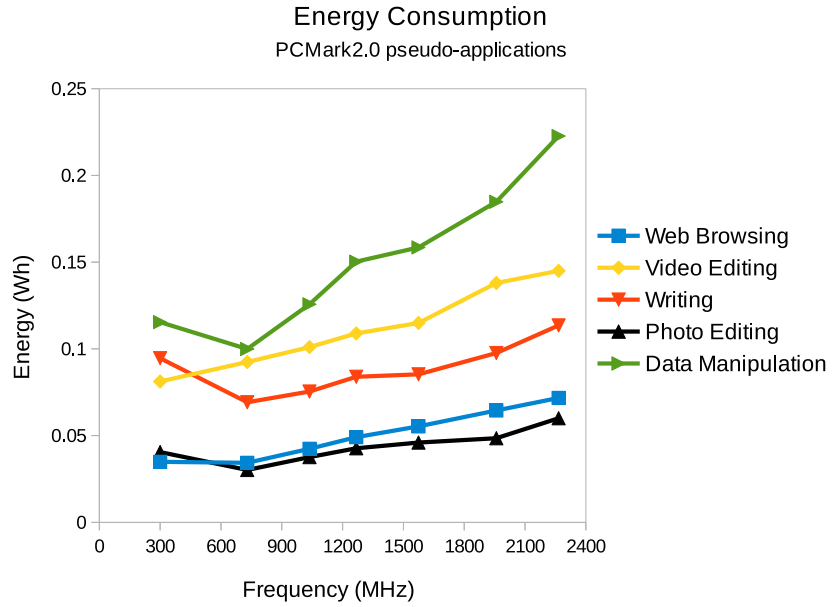
The goal of this Subsection is to provide a brief overview of the energy efficiency problem in the application execution. To let our *BestWing* policy to work properly, we profiled the energy efficiency of the aforementioned benchmarks applications according to different CPU frequency settings on different devices. In this regard, we used the Energy Delay Product (EDP) as a metric, originally proposed by Horowitz [129]. The efficiency is measured in terms of delay until the execution has been completed: it is defined as the product of the energy consumed in an application execution to its execution time. To weight performance versus power a parameter w has been introduced as a power of the time factor by Brooks and Cameron [130, 131]. In this model the parameter can be exploited dynamically to make the profile more application's use-case dependent, varying it according to the performance and Quality of Service (QoS) constraints of the application.
Generally speaking the EDP curve is represented by the following equation [132]:

$$EDP = E * T^w \text{ where } w = 1, 2 \text{ or } 3 \tag{5.2}$$

Observing Figure 5.3, which refers to the Nexus 5 measurements, we notice that according to our expectations the energy consumed in an application execution decreases as the frequency decreases until a certain frequency. For the MobileXPRT2015 benchmark suite this frequency varies between 729.6 MHz and 1036.8 MHz depending on the single pseudo-application. For the PCMark 2.0 suite is instead at 729.6 MHz, commonly to almost all the pseudo-applications. Beyond these thresholds the energy consumption inverts its trend and starts to increase.
In fact, despite the lower energy demand of the CPU, the computational time required to perform an execution cycle increased. This makes the background services and tasks, that are commonly performed, more influent in the energy consumption point of view, because they are monitored for a longer time. We can model this observation with the following equation, that shows the energy



(a)



(b)

Figure 5.3: *The figures shows the energy consumption of the single pseudo-applications of the two benchmarks suites measured on the Nexus 5.*

consumption of an monitored execution (E_{exe}) based on the application execution time and the set clock frequency f.

$$E_{exe}(f) = E_s(f) + E_{app}(f) \tag{5.3}$$

Where E_s is the *System Energy*, that is the energy consumed by the system services and background tasks during the monitored execution, while E_{app} is the effective energy consumed by the monitored application.
For our purpose, we assume E_s constant for all the experimental executions.

Once we determined the application energy consumption and the execution times for each available CPU frequency configuration, we took the mean values of the different observations. Then, starting from 5.2, we calculated the normalized EDP using the following equation:

$$EDP_{norm_{app}}(f) = ||\tilde{E}_{exe}(f) * T(f)^w|| \tag{5.4}$$

Where T is the application execution time and w is the weight parameter. For our purpose we decided not to focus on performance in particular but to give to energy and performance the same weight by setting the w parameter equal to 1. Anyway as previously mentioned, future research efforts could investigate the possibility to exploit this parameter in order to explore dynamic profiles to fit the application-specific performance and QoS requirements.

Finally, we interpolated the various EDP points to extract the model with its coefficients. These data are then added into a specific EDP models file that the resource manager will use for the policy (as explained in Subsection 5.2.3). The file is structured into different sections, one per application class, as shown by Listing 5.1.

Listing 5.1: *BarbequeRTRM EDP models configuration file*

```
<BarbequeRTRM version="1.0">
  <class name="Video_processing">
    <edp-model>
      <device model="NEXUS_5" type="POLYNOMIAL">
        < coefficient value="0.6862" exp="0" x="0"/>
          ...
      </device>
    </edp-model>
     ...
  </ class >
</BarbequeRTRM>
```

5.2.2 Benchmarks and applications EDP models

As previously said we run different execution tasks of the benchmarks in various frequency configurations to extrapolate their EDP models. In particular we performed 10 executions for each frequency configuration, for a total of 1100

Application	Frequencies (MHz)						
	300	729.6	1036.8	1267.2	1574.4	1958.4	2265.6*
Photo effects	1	0.53	0.47	0.50	0.54	0.59	0.61
Photo collages	1	0.55	0.50	0.56	0.59	0.61	0.67
Slideshows	1	0.43	0.33	0.36	0.33	0.34	0.35
Encryption	1	0.37	0.28	0.29	0.27	0.27	0.26
Face recognition	1	0.56	0.53	0.56	0.61	0.63	0.72
MobileXPRT2015	**1**	**0.45**	**0.37**	**0.39**	**0.39**	**0.41**	**0.42**
Web Browsing	0.77	0.56	0.66	0.74	0.82	0.95	1
Video Editing	0.72	0.70	0.72	0.76	0.80	0.95	1
Writing	1	0.40	0.33	0.34	0.31	0.33	0.37
Photo Editing	1	0.44	0.46	0.49	0.49	0.48	0.60
Data Manipulation	0.97	0.59	0.65	0.74	0.74	0.84	1
PCMark2.0	**1**	**0.58**	**0.59**	**0.63**	**0.64**	**0.71**	**0.81**

Table 5.2: *The normalized EDP values of the pseudo-applications of the MobileXPRT2015 and PCMark 2.0 benchmark suites for each evaluated frequency of the Nexus 5.*

Application	Frequencies (MHz)							
	"big" cluster					*"LITTE"* cluster		
	1200	1400	1600	1800	2000	1000	1200	1400
Photo effects	0.89	0.86	0.89	0.94	1	1	0.87	0.80
Photo collages	0.87	0.86	0.85	0.88	1	1	0.90	0.87
Slideshows	0.83	0.79	0.80	0.82	1	1	0.86	0.78
Encryption	1	0.91	0.90	0.93	1	1	0.81	0.69
Face recognition	0.84	0.85	0.86	0.89	1	1	0.88	0.79
MobileXPRT2015	**0.90**	**0.86**	**0.87**	**0.90**	**1**	**1**	**0.86**	**0.77**

Table 5.3: *The normalized EDP values of the pseudo-applications of the MobileXPRT2015 benchmark suite for each evaluated frequency of the Odroid-XU3 CPU clusters.*

pseudo-applications executions (i.e., 700 on the Nexus5 and 400 on the Odroid-XU3) plus 220 entire benchmarks executions (i.e, 140 on the Nexus 5 and 80 on the Odroid-XU3). Then, we performed the *Rank Sum Wilcoxon* test [133] between the various frequency repeated measurements population to verify if the shift for the population represented by one frequency versus the other is a result of the frequency scaling. Moreover, the outcomes allow us to attest that our results are significant at $p < 0.05$. Finally, we report the standard deviations in the Table 5.4. It is comprised between 0.316 and 7.103 for the execution time measurements, while they range between $3\text{x}10^{-4}$ and $6\text{x}10^{-3}$ for the energy consumption measurements. The obtained overall EDP values are presented in in Tables 5.2 and 5.3.

Benchmark	Energy consumption		Execution time	
	Min	Max	Min	Max
MobileXPRT2015	0.00032	0.0031	0.316	7.103
PCMark2.0	0.00052	0.0059	0.422	3.259

Table 5.4: *The minimum and maximum standard deviations of the execution time and energy consumption for all the applications, benchmarks and frequency configurations measurements.*

We can make some observations about the EDP profiles shown in Figures 5.4 and 5.5. In particular Figure 5.4 makes a comparison between the EDP model of the Nexus 5 and the two clusters of the Odroid-XU3 board regarding to the MobileXPRT2015 benchmark applications. First of all, we can notice that both the profiles has a similar trend, in fact for each lowest frequency the EDP value is always the worst due to the penalty introduced by the high execution times added to the energy consumption that, as explained in Subsection 5.2.1, is not a monotonic decreasing function of the frequency, but it re-increases below a frequency threshold.
Moreover, we can notice an high discrepancy between the two devices profiles in terms of efficiency. One of the possible reason of such outcome could be that since the Nexus 5 is a commercial product its hardware architecture is more optimized and the platform-specific software stack of the Android build has been implemented more accurately (e.g., with a more aggressive power management, more frequency scaling points availability, etc...) than in the Odroid-XU3, which is a development-purpose board. Anyway, these results justify the need to have a device-specific profile for each application because of architectures and drivers peculiarities of the various devices.
Finally, we can make a comparison between the two clusters of the Odroid-XU3 board. In fact, we profiled them has two distinct systems in order to provide a characterization of both architectures for our model. In this sense, in line with our expectations, the profiles of the "LITTLE" cluster has a better energy-efficiency for almost all the applications with respect to the "big" one. Conversely the two architectures are specular in term of efficiency, in fact the "LITTLE" cluster is more efficient at the highest frequency (1400 MHz) while the "big" cluster has is best efficiency in the middle-low range (1400–1600 MHz) following the general trend of the Nexus 5.

Our modeling methodology followed a bottom-up approach: we started to profile the energy consumption of the single pseudo-applications to calculate their normalized EDP values for the tested frequencies and to obtain their models. Then we considered the data related to the entire benchmark suites to build

a "global model".
From the pseudo-applications and the benchmarks EDP models shown in Figures 5.4, 5.5 and their comparison errors of Figures 5.6 and 5.7, we can observe that, for certain classes and architectures, the application specific model and the global model are comparable. In particular for the Nexus 5 the Slideshows and Encryption applications have a model close to the MobileXPRT2015 global model, with an error less than 10% on average. While for the PCMark 2.0 benchmark suite only the Data Manipulation pseudo-application model can be compared with a contained error ($< 10\%$). As shown by Figure 5.6 other models comparisons produce mean errors between 10% and 30%. These error percentages lead to a not negligible error of approximation such that we concluded that globalizing the model is not feasible.
Conversely, for the Odroid-XU3 board all the pseudo-applications have a model close to the MobileXPRT2015 global one with a restrained error ($< 10\%$). Anyway, as opposed to the Nexus 5 analysis, error peaks can be noticed for the Encryption and the Slideshows applications.
At time of writing, however, we do not have access to the performance counters of the hardware to determine which is the cause of such a difference in term of accuracy between the models and for this reason this aspect would need to be further investigate.

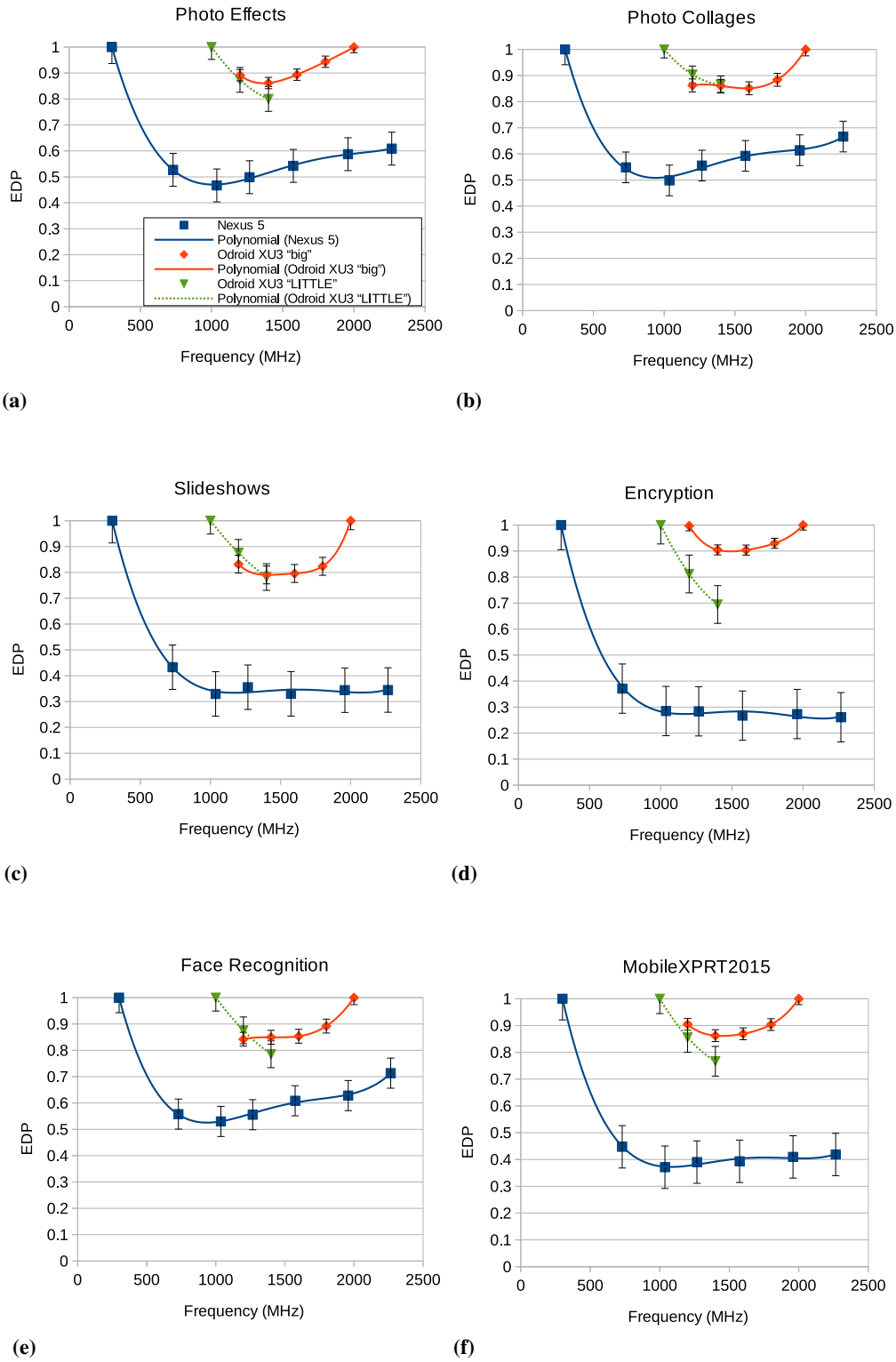


Figure 5.4: *The comparison between the Nexus5 and the Odroid-XU3 EDP models of the pseudo-applications (from 5.4a to 5.4e) of the MobileXPRT2015 benchmark obtained in our tests. Figure 5.4f is the global model of the benchmark suite.*

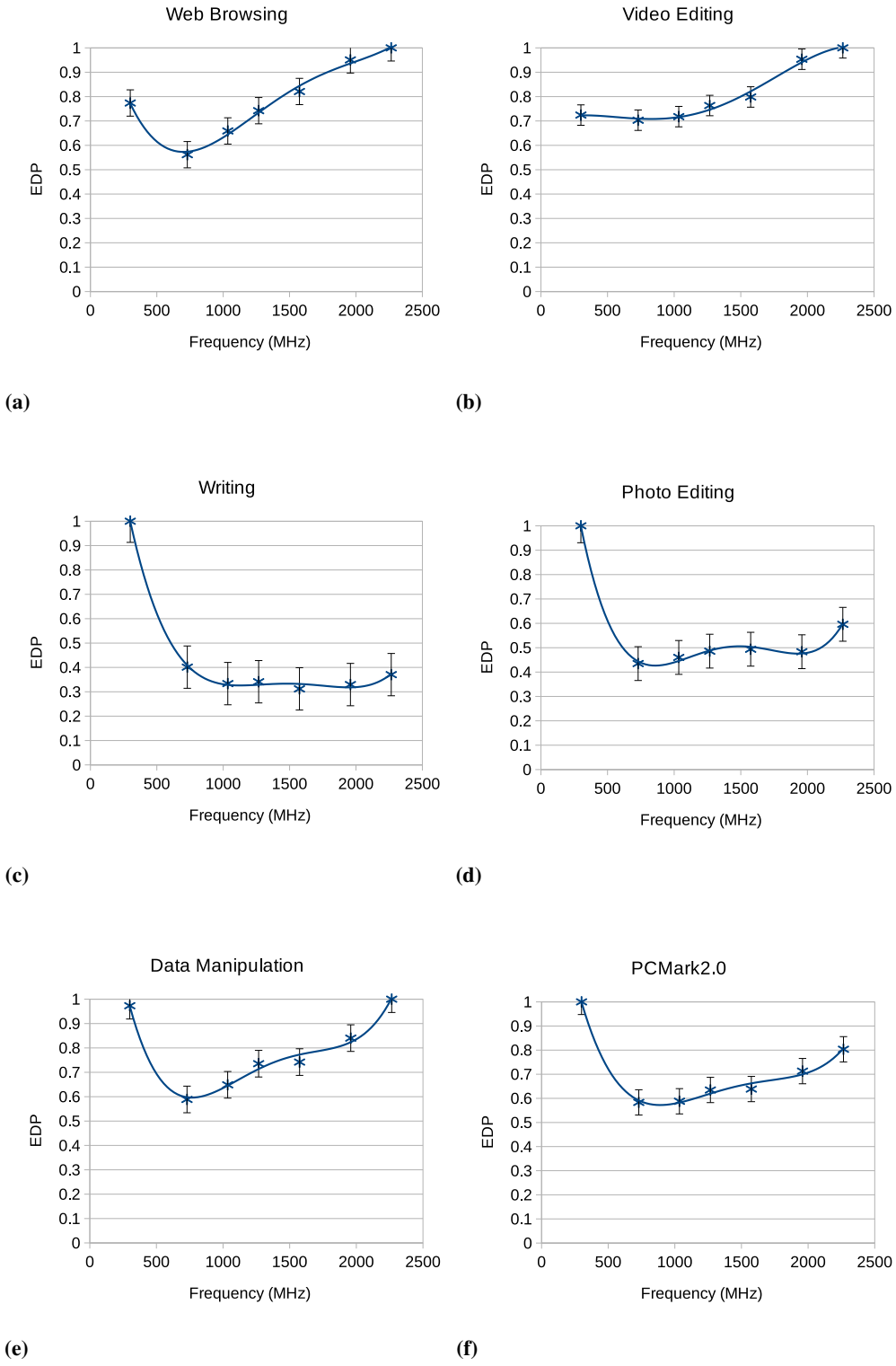
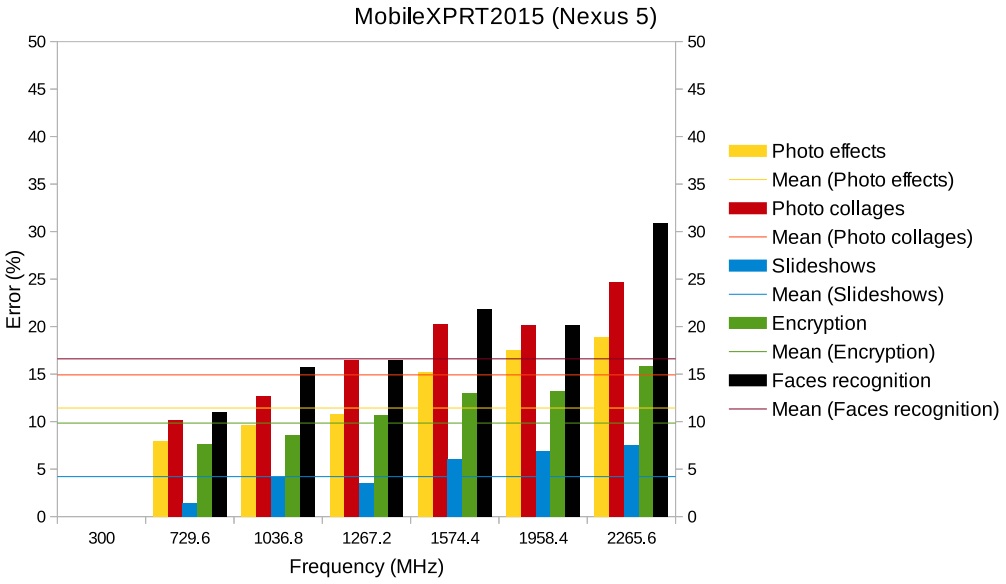
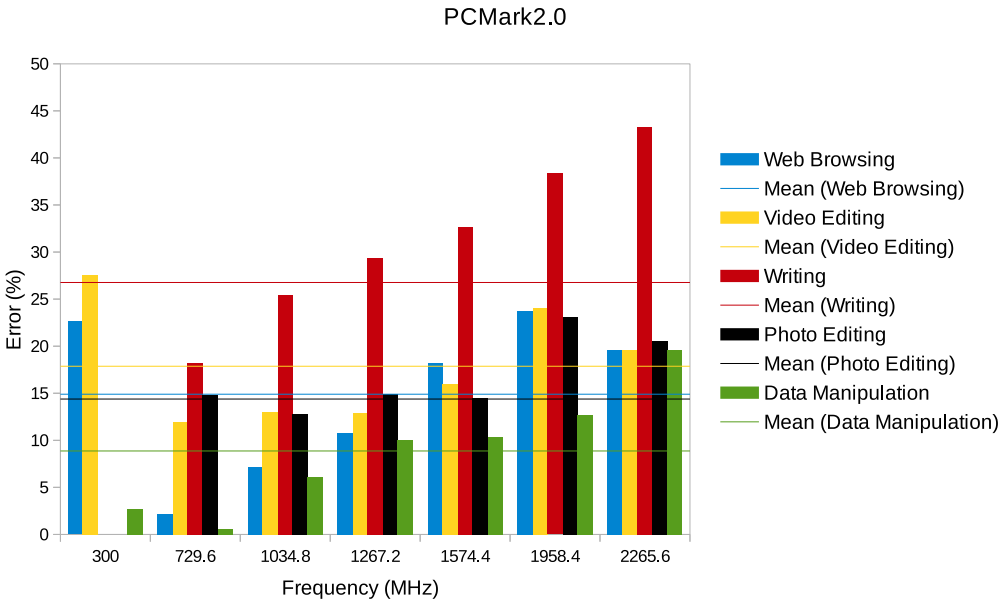


Figure 5.5: *The EDP models of the pseudo-applications (from 5.5a to 5.5e) of the PCMark 2.0 benchmark obtained in our tests. Figure 5.5f is the global model of the benchmark suite.*

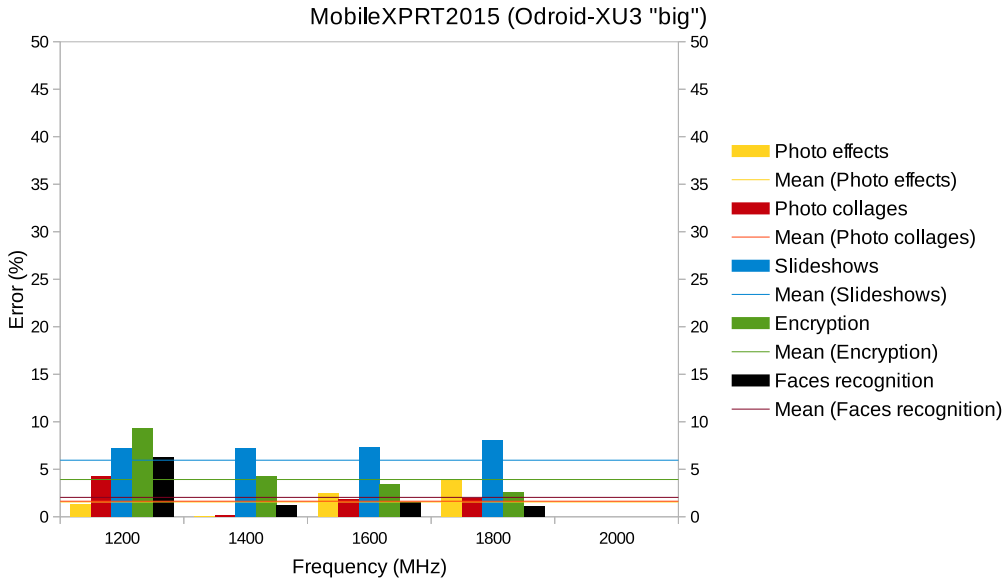


(a)

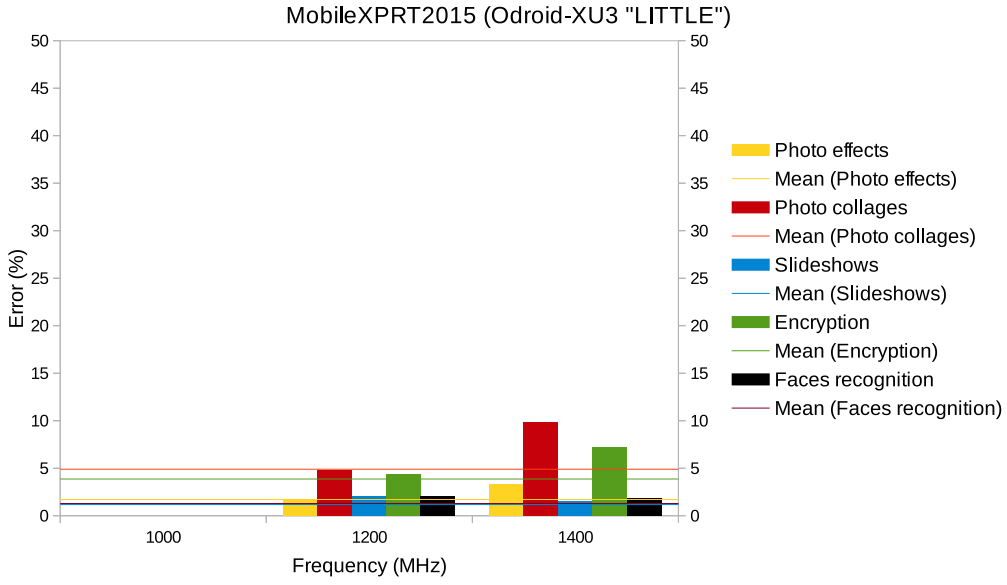


(b)

Figure 5.6: *The percentage errors between the pseudo-applications EDP models and the respective global benchmark one. All the graphs refers to the Nexus 5 device.*



(a)



(b)

Figure 5.7: *The percentage errors between the pseudo-applications EDP models and the respective global benchmark one. The first graph (Figure 5.7a) refers to the "big" cluster of the Odroid-XU3, while the second graph (Figure 5.7b) refers to the "LITTLE" cluster.*

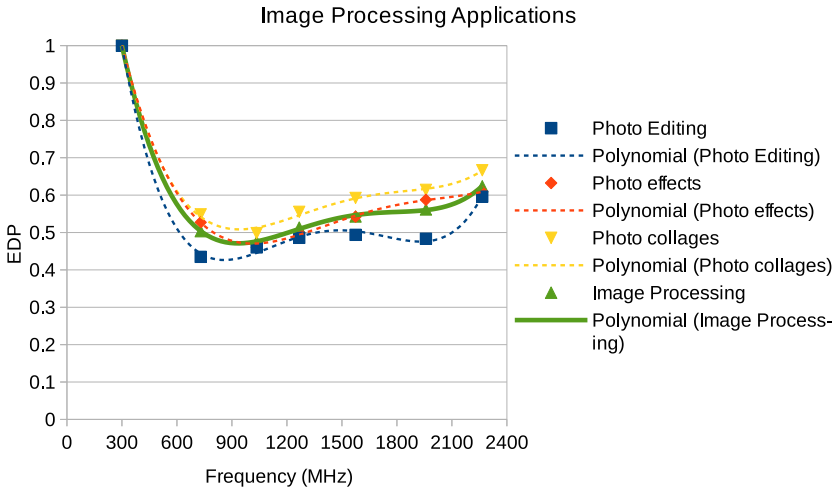


Figure 5.8: *The EDP model of the Image Processing application class extracted by averaging the three EDP models of the image processing pseudo-applications.*

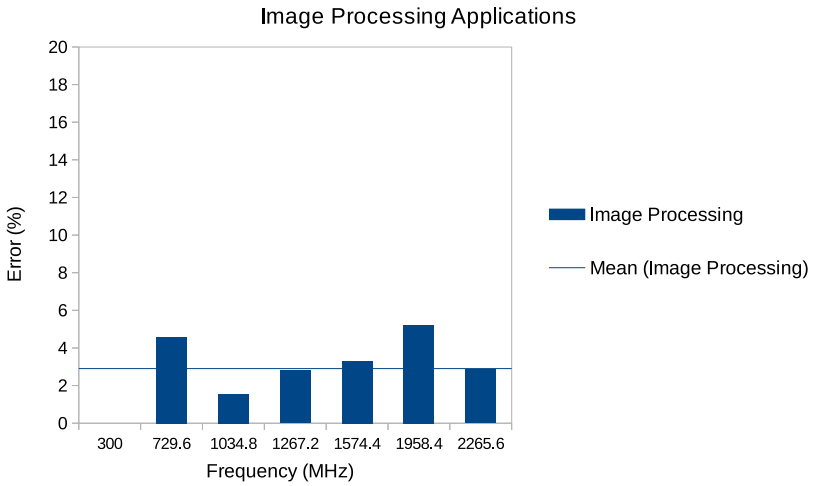


Figure 5.9: *The averaged percentage errors between the Image Processing application class model and the EDP models of the image processing pseudo-applications.*

5.2.3 Application classification

Looking at the applications models of the Nexus 5 of Figures 5.4 and 5.5, it can be noticed that the pseudo-applications that perform image processing has a similar model. From this observation this step of our experimental phase aimed at assessing if the models extracted from the benchmarks pseudo-application can be applied to other applications and organized in classes.

Initially we extracted a model from *Photo Editing*, *Photo Effects* and *Photo Collages* pseudo-applications to generalize the *Image Processing Applications* class as shown in Figure 5.8. Observing the Figure 5.9 it can be noticed that

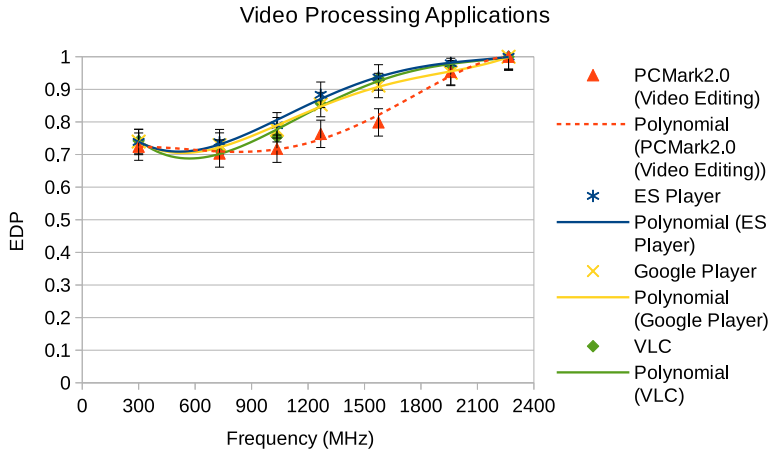


Figure 5.10: *The graph shows the EDP models of the video player applications extracted by our tests compared to the video editing pseudo-application EDP model.*

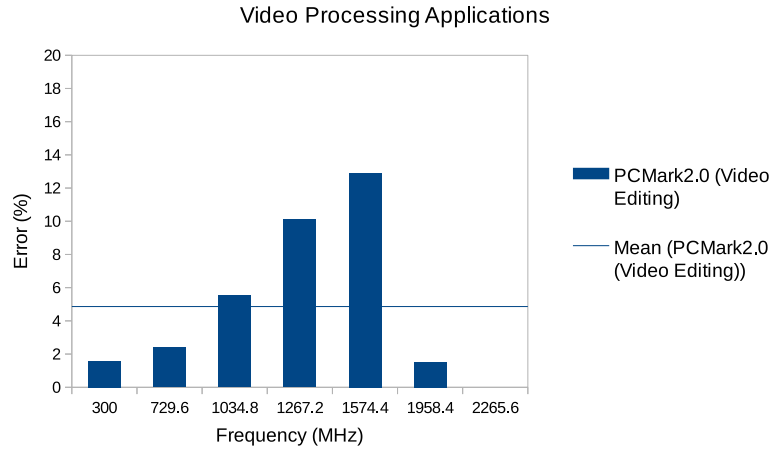


Figure 5.11: *The figures shows the percentage errors between the video players and the video editing pseudo-application EDP models.*

the average of the errors between the general model and the single ones is very restrained, in fact is less than 3%. Since there was a first confirmation about our intuition, we took these results as a first sight to further verify our approach.

In this sense we profiled three non-benchmark video player applications which has not been used for the EDP models characterization – the *ES Media Player*, the *Google Photo Player* and the *VLC Player* – to play a 210 seconds video encoded with the H.264 MPEG-4 AAC compression format at 30 FPS at 1920x1080 of resolution. The measurement of the time spent by the players to play the video comprises also the initial loading time which varies basing on the different clock frequency configurations.

Then, we compared the players and the PCMark 2.0 Video Editing pseudo ap-

plication models to identify a possible *Video Processing applications* class. The results of Figure 5.10 show a similarity between the four models with less than 6% mean error (Figure 5.11).
The error is due mainly to the fact that the video player applications perform only a reproduction of the video file, while the pseudo-application performs also time-varying and editing operations on the video file. Anyway, for all the applications the time composition is dominated by the reproduction task so that we can consider them comparable. Moreover, it can be noticed that from an energy-efficient frequency selection point of view the curves have their minimum points between $0.68 - 0.72$ EDP values and between $550 - 750$ MHz so the error is negligible ($< 3\%$).

All the above considerations led us to propose a *classification model* for the Android applications. In this sense an application can be classified according to its main task and managed with a class-specific model that we can provide to the resource manager (e.g., Image processing, Video processing).
Here below (Listing 5.2) we show how the application class is encoded into a specific attribute of the application *configuration file*.

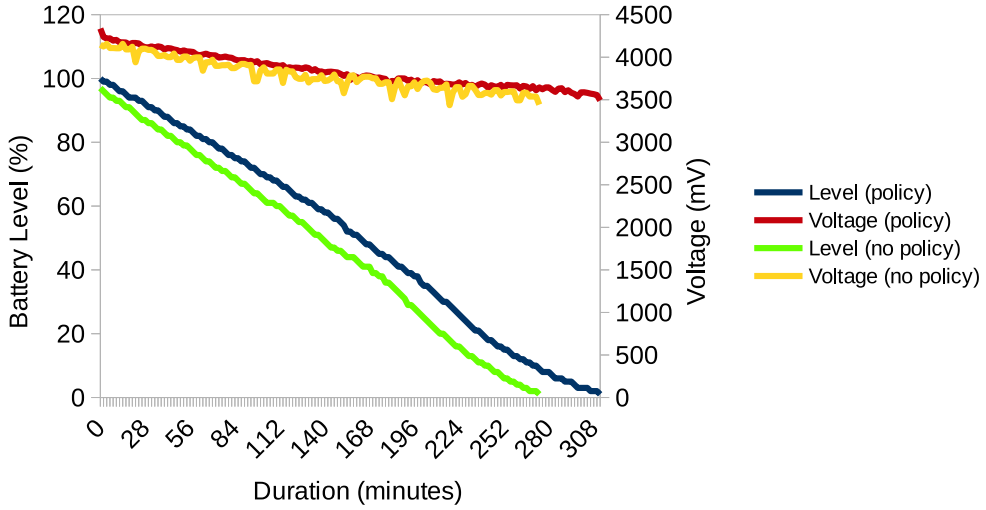
Listing 5.2: *Sample Application XML definition file focused on the class attribute.*

```
<?xml version='1.0' encoding='utf-8'?>
<BarbequeRTRM version="1.0">
  <application name="ES Media Player" package="..." class="Video_processing">
    ...
  </application>
</BarbequeRTRM>
```

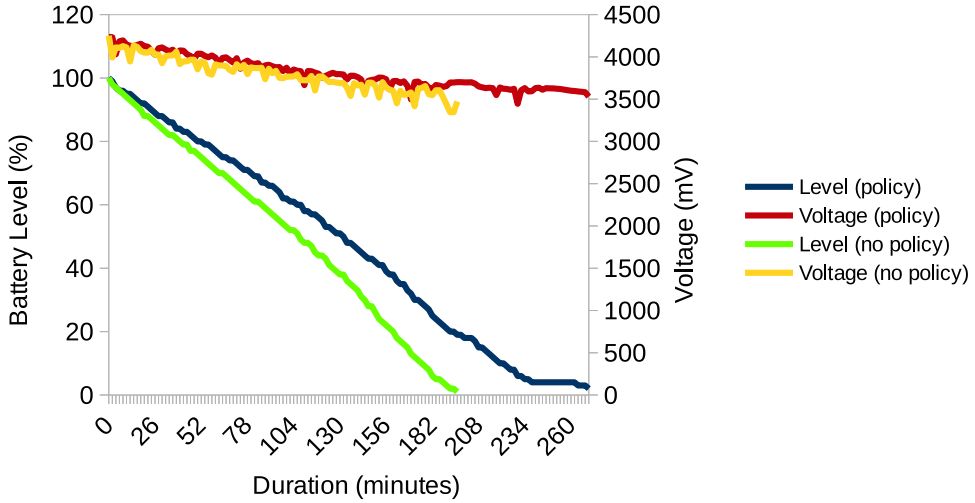
Finally, whereas the application cannot be classified, a special section of the *configuration file* can be added to specify its EDP model as shown in Listing 5.3.

Listing 5.3: *Sample Application XML definition file focused on the edp-model section.*

```
<?xml version='1.0' encoding='utf-8'?>
<BarbequeRTRM version="1.0">
  <application name="PCMark2.0" package="...">
    <edp-model>
      <device model="NEXUS_5" type="POLYNOMIAL">
        <coefficient value="1.8685" exp="0" x="0"/>
        ...
      </device>
    </edp-model>
    ...
  </application>
</BarbequeRTRM>
```



(a) *MobileXPRT2015*



(b) *PCMark 2.0*

Figure 5.12: *The device lifetime and voltage measurements during continuous runs of the benchmark suites on the Nexus 5. The Figures highlight the comparison between the case in which no CPU settings are applied and the case in which the* BestWing *CPU settings are adopted. Top Figure 5.12a refers to the MobileXPRT2015 suite, whereas Figure 5.12b refers to the PCMark 2.0 suite.*

5.2.4 Energy efficiency benefits and performance drawbacks

One of the step of our experimental phase was to evaluate how the adoption of an energy efficiency approach impacts on the lifetime and the performance of the device selected by the policy. In this sense we measured and compared the device lifetime and the score computed by the benchmark suites by running them in a continuous way on the Nexus 5. We tested two cases: first by using the standard

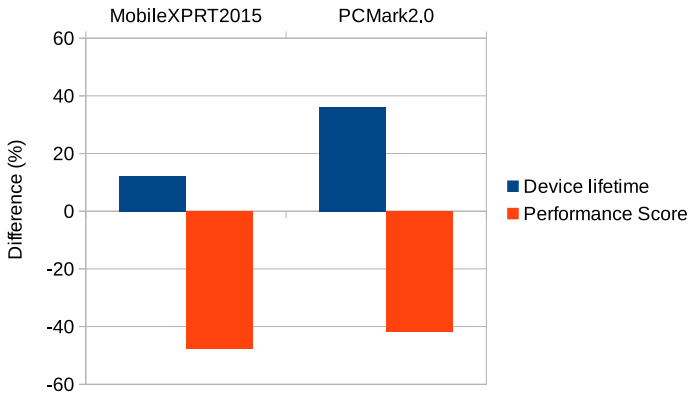


Figure 5.13: *The percentage of device lifetime increasing and application performance loss with respect to the standard and energy-efficiency approaches.*

Android resource and CPU management; then adopting the resources allocation and frequency scaling derived from the aforementioned energy efficient models and set by the *BestWing* policy.

From Figure 5.12 we can notice that the discharging rate and voltage trend are affected by the adoption of the policy. Firstly, the voltage is more stable due to fixed frequency setting, which reduces the fluctuations due to a frequent CPU frequency switching [134]. Secondly, despite the common constant trend, an energy-efficient approach led to a low discharging rate which results in an increased device lifetime. This result is highlighted by the Figure 5.13, which shows that from the battery point of view the device lifetime is increased between 12% and 36%, which confirms the main objective of the energy-efficiency approach. On the other hand, the same Figure shows also the impacts on the performance side. In this sense, as we expected, we obtained a loss in term of execution performance score. Specifically, in our tests it stands between the 42% and 48% for the PCMark 2.0 and MobileXPRT2015 benchmark suites respectively. This results in an increased execution time: from our observations the delay is limited between $111s$ and $117s$. Anyway, from an efficiency point of view these values represents the best trade-off between the energy consumed and the computing performance. In fact, any other configuration would affect negatively the energy or the performance side.

The results obtained in our experimental context are summarized in Table 5.5 and can be commented also according to the user experience point of view, making two considerations. First, the user can experience an absolute non-negligible increasing of the device lifetime, in the test case between about 30 and 72 minutes. Secondly, the performance loss cannot be considered in an absolute way, in fact the importance of the computational performance is application-dependent.

Metrics (average)	MobileXPRT2015			PCMark2.0		
	No-policy	Policy	Speed-up	No-policy	Policy	Speed-up
Device lifetime	$4h39'$	$5h13'$	$+12.2\%$	$3h19'$	$4h31'$	$+36.1\%$
Performance Score	241	126	-47.7%	3530.2	2059.3	-41.7%
Execution time	122.09s	238.61s	$+48.8\%$	513.34s	624.34s	$+17.8\%$

Table 5.5: *The measurements obtained from the two benchmark suites with respect to both the Android standard energy and resource management and the energy-efficiency approach.*

This is confirmed by comparing how the execution time increasing and the performance score loss are related in the two benchmarks. In fact for the MobileXPRT2015 they are directly related, while for the PCMark 2.0 they are very different due to the heterogeneity of the computing settings (e.g., different frame rates) that are tested and that are affected by the frequency scaling in different way. The above considerations lead to considerate the energy-efficiency approach impacts as globally positive. In particular, it is very suitable for applications that do not have real-time or response constraints to extend the lifetime of devices minimizing the performance loss.

5.2.5 Device selection policy overhead

The last step of our experimental phase aim at characterizing the policy in terms of its performance and overheads. Firstly we perform an evaluation on the device selection process over different test cases and scenarios. Then we treat about the time composition and performance of the policy steps. Finally we evaluate the scalability issues of the policy.

Policy execution

We evaluated the execution and the output of the *BestWing* policy over three test cases:

1. All devices are unplugged (i.e., battery powered)
2. Device 4 is plugged
3. Devices 1 and 4 are plugged.

Tables 5.6 and 5.7 show the devices and applications setup in the test cases. In each test case the policy has been executed over 24 scenarios to evaluate its behavior by permuting the different battery values to the devices as shown by Table 5.8. To compute the Energy Consumption Index of each device we con-

sidered the EDP profiles that were introduced in Subsection 5.2.1 and shown in Figure 5.4.

We can make some observations about the obtained outputs. First of all, in Table 5.9 we can notice how the policy selects the device according to its ECI when all devices are unplugged. In fact, for example in the scenario $S1$ the low ECI of device $D1$ due to its high battery level (95%) and low EDP values for all the applications made it the most suitable to be selected by the policy, as we expected. Moreover, we can also observe that both in scenario $S18$ and $S24$ the battery level has a lower weight than the EDP value in the computation of the ECI of devices. In fact, for the application $A3$ the policy selects the device $D1$ and $D2$ respectively despite their battery level (60%) is much lower than the device $D4$ battery (95%). Anyway those devices has a better ECI due to their low EDP optimal value (i.e., 0.31 at frequency 2265 MHz instead of 0.78 at 1400 MHz).

From Table 5.10 we can verify the trivial case in which the policy selected always the device $D4$ because it is the only one plugged, in line with our expectations.

Finally, Table 5.11 shows the behavior of the policy when there are two plugged devices. In this case, we can make the same observations of Table 5.9, considering that devices $D1$ and $D4$ have the priority over the others. In fact, for example in the scenario $S18$ the policy acts as expected selecting the device $D1$ for the application $A3$ due to its best EDP value (0.31 at 2265 MHz) despite the high battery level of device $D4$ (95%). Same considerations for the scenario $S16$, where, differently from the first table, the device $D2$ cannot be selected because is unplugged.

ID	Device
$D1$	Nexus 5
$D2$	Nexus 5
$D3$	Odroid XU3 "big"
$D4$	Odroid XU3 "LITTLE"

Table 5.6: *The devices setup used in the policy execution test cases.*

ID	Application
$A1$	Photo Effects
$A2$	Photo Collages
$A3$	Slideshows
$A4$	Encryption
$A5$	Face Recognition
$A6$	MobileXPRT2015

Table 5.7: *Applications used in the policy execution test cases.*

Scenario	Device battery level (%)			
	D1	D2	D3	D4
$S1$	95	75	60	35
$S2$	95	75	35	60
$S3$	95	60	75	35
$S4$	95	60	35	75
$S5$	95	35	75	60
$S6$	95	35	60	75
$S7$	75	95	60	35
$S8$	75	95	35	60
$S9$	75	60	95	35
$S10$	75	60	35	95
$S11$	75	35	95	60
$S12$	75	35	60	95
$S13$	60	95	75	35
$S14$	60	95	35	75
$S15$	60	75	95	35
$S16$	60	75	35	95
$S17$	60	35	95	75
$S18$	60	35	75	95
$S19$	35	95	75	60
$S20$	35	95	60	75
$S21$	35	75	95	60
$S22$	35	75	60	95
$S23$	35	60	95	75
$S24$	35	60	75	95

Table 5.8: *Battery level scenarios description for the policy execution test cases.*

Scenario	A1	A2	A3	A4	A5	A6
$S1$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S2$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S3$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S4$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S5$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S6$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S7$	$D2$	$D2$	$D2$	$D2$	$D2$	$D2$
$S8$	$D2$	$D2$	$D2$	$D2$	$D2$	$D2$
$S9$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S10$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S11$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S12$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S13$	$D2$	$D2$	$D2$	$D2$	$D2$	$D2$
$S14$	$D2$	$D2$	$D2$	$D2$	$D2$	$D2$
$S15$	$D2$	$D2$	$D2$	$D2$	$D2$	$D2$
$S16$	$D2$	$D2$	$D2$	$D2$	$D2$	$D2$
$S17$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S18$	$D4$	$D4$	$D1$	$D4$	$D4$	$D4$
$S19$	$D2$	$D2$	$D2$	$D2$	$D2$	$D2$
$S20$	$D2$	$D2$	$D2$	$D2$	$D2$	$D2$
$S21$	$D2$	$D2$	$D2$	$D2$	$D2$	$D2$
$S22$	$D2$	$D2$	$D2$	$D2$	$D2$	$D2$
$S23$	$D2$	$D2$	$D2$	$D2$	$D2$	$D2$
$S24$	$D4$	$D4$	$D2$	$D4$	$D4$	$D4$

Table 5.9: *Device selection for tested applications in the scenarios of the policy execution test case* 1.

Scenario	A1	A2	A3	A4	A5	A6
$S1$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S2$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S3$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S4$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S5$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S6$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S7$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S8$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S9$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S10$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S11$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S12$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S13$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S14$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S15$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S16$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S17$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S18$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S19$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S20$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S21$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S22$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S23$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S24$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$

Table 5.10: *Device selection for tested applications in the scenarios of the policy execution test case* 2*.*

Scenario	A1	A2	A3	A4	A5	A6
$S1$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S2$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S3$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S4$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S5$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S6$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S7$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S8$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S9$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S10$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S11$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S12$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S13$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S14$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S15$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S16$	$D4$	$D4$	$D1$	$D4$	$D4$	$D4$
$S17$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S18$	$D4$	$D4$	$D1$	$D4$	$D4$	$D4$
$S19$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S20$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S21$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S22$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$
$S23$	$D1$	$D1$	$D1$	$D1$	$D1$	$D1$
$S24$	$D4$	$D4$	$D4$	$D4$	$D4$	$D4$

Table 5.11: *Device selection for tested applications in the scenarios of the policy execution test case* 3*.*

Time overhead

We evaluated the *BestWing* policy performance by analyzing its time composition with respect to six different steps both in the local and remote launching scenarios:

- *System Information update*: the time required by the schedule to retrieve the updated information about the available devices
- *EDP computation (per-device)*: the time spent by the policy to calculate the minimum EDP value and the optimal frequency for a single device
- *Remote applications availability verification (per-device)*: the time taken to verify if the application is available in the current evaluated device
- *Application launching*: the time required to launch the application on the selected device

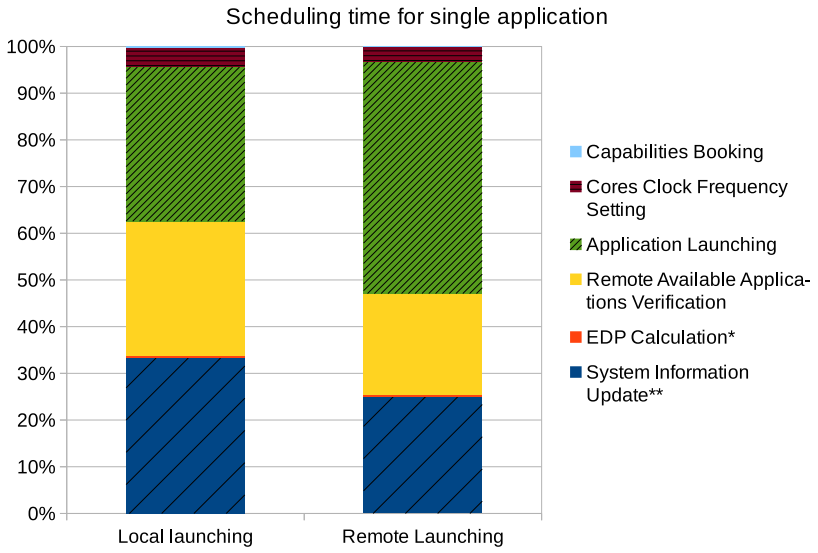


Figure 5.14: *Time composition of a single application scheduling respect to the launching mode. *The EDP calculation is considered per system. **The time to update System Information is considered per single remote system.*

- *Cores clock frequency setting*: the time required by the *Platform Manager* to set the clock frequency for the selected device
- *Capabilities booking*: the time spent by the policy to perform the synchronization phase to book the allocated Capabilities to the starting application.

As highlighted by the Figure 5.14 the time composition is dominated by the three steps (85% – 95% of the time) that require a network communication. In particular the *system information update* depends on the total number of available devices. Moreover, the *remote applications availability verification* step is performed only if the system index is the temporary best and in the worst case is performed for all the available devices. The *launching* step instead, strongly depends on whether the application is launched on the local device or on a remote one. The Figure 5.14 shows that in the remote scenario the launching takes about the 50% of the total time. The gap is due to the overhead introduced by network operations over the fixed Android system launching time loss.
All the discussed data refer to a test case, over 100 policy executions for each scenario, implemented with two available devices and a single scheduled application which results in a policy average execution time of 413ms for the local launching scenario and 591ms for the remote launching one. Since some steps depend strongly on the total number of available devices we will discuss about scalability in the following pages.

Finally, we investigated in detail the three aforementioned dominating steps

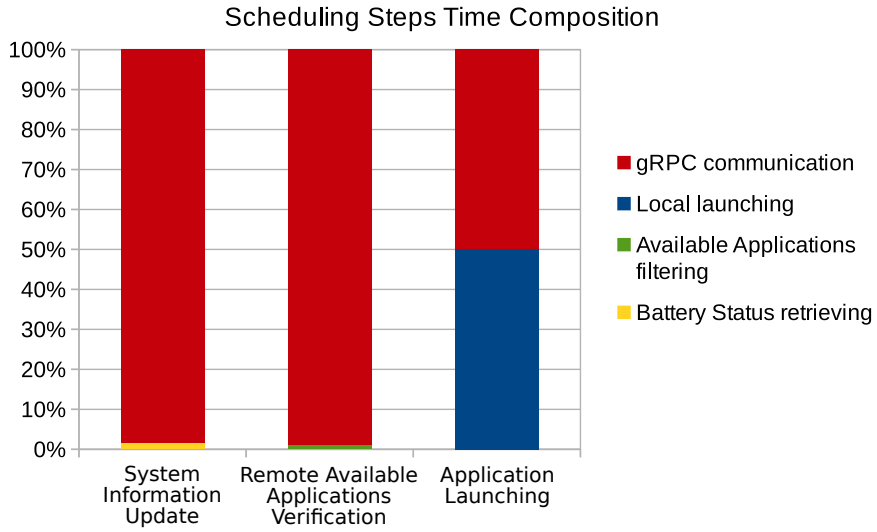


Figure 5.15: *The time composition of the policy steps highlighting the overhead due to the network communication.*

(*system information update*, *remote applications availability verification* and *application launching*) to identify the reason behind the high time consumption. As highlighted by Figure 5.15, which provides in detail their time composition, the overhead is actually introduced by the network operations (e.g., gRPC communications, etc...). Totally the network data transmission time loss is included between 69ms and 185ms compared to the actually computation time which is between 1ms and 3ms for the first two steps and about 100ms for the third one. Although during measurements we noticed that absolute network times can differ due to current workload status and clock frequency of the remote device during the RPC invocation, they still remain an order of magnitude rather than the absolute computation times. Obviously, making the above considerations we have to keep in mind that we are considering the worst case due to a low throughput of the wireless communication channel as explained in Subsection 5.1.1. Finally, we should consider the overhead introduced by the Android software stack which has not been originally thought for HPC use cases. First of all because all the calls from the applications and services side pass through the Java framework API, introducing a delay in the calling chain. Next, the overhead is caused by the virtualization of the Java system services in charge to manage the API requests. This may contribute in affecting the operations performance during gRPC calls.

Scalability evaluation

We evaluated the scalability of the *BestWing* policy by measuring the time to retrieve the remote devices status with respect to the number of the available

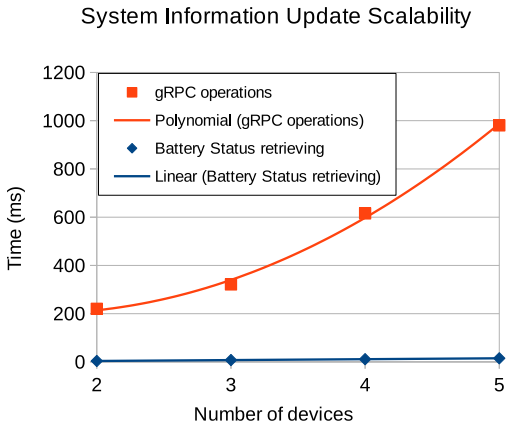


Figure 5.16: *The trend of the time overhead due to the update the system information by the resource manager with respect to the number of devices in the distributed system.*

devices in the Pervasive Distributed Computing Network System. We selected this particular policy step because it is the only one that is performed on all the available devices and it is required every time the policy is run. Moreover, it dominates the time composition of the selection process and so it impacts significantly in the policy execution time with compared to other steps as we have seen in Figure 5.14. Anyway similar considerations can be made for the worst cases of the other two steps. As shown by the Figure 5.16 the trend of the overhead increases polynomially with respect to the number of devices selected from 2 to 5 devices for *network operations*, while it increases linearly for the *battery status computation* so that the latter remaining negligible for our analysis. The results reveal that for a small network made of no more than 5 devices, such as a possible home setup, the delay introduced by the policy should be tolerated by the user, but in more complex and large computing network a more performing systems status update strategy has to be adopted in order to not introduce unacceptable overhead. In fact between 2 and 4 available devices the policy is delayed by a maximum of 600ms which maintains the overall device selection and application launching operations within $1 - 1.5$s. This time is quite similar to the second response time limit ($\sim$ 1s) described by Nielsen [135] which maintains the users flow of thought uninterrupted even they notice the delay. Furthermore, the time widely respects the users attention time threshold of 10s. With 5 or more available devices the delay introduced by the policy seriously affects the user experience. In fact, the fast growth of the delay due to the polynomial trend makes the application response time rapidly exceed the 10s Nielsen's limit so that, without a sort of feedback provided to the user, the latter experiences the delay and issues related to application responsiveness.

CHAPTER 6

Conclusions and Future Works

In this Chapter we discuss some final remarks and present some challenges that would be interesting to address in future developments of our work. The Section 6.1 presents a final summary of our work with some considerations for further developments in the actual research context.
Otherwise, the Section 6.2 presents some future challenges related to the Android applications management and optimization approaches. Finally, the Section 6.3 exposes some open issues that are not covered by our work and some improvements that can be covered by future works.

6.1 Conclusions

In this work we have considered the increased pervasiveness and computational capabilities of personal devices as an interesting chance to achieve a two-fold objective:

1. Extend the device lifetime by reusing old and abandoned devices to embrace *Green computing* purposes

2. Enable a new opportunity to deploy mobile applications on distributed devices.

In this scenario we performed a change of paradigm with a distributed rearrangement of mobile devices that can lead to improvements in terms of energy efficiency. In this regard, we introduced the *Pervasive Distributed Computing Network System* concept which enables task offloading and dynamic load balancing through a set of locally interconnected personal devices.

In this first step of the development we focused on the challenge of porting and extending a run-time distributed resource manager, the *BarbequeRTRM*, on different Android devices to run Android applications in an energy efficient way by scheduling them on the local or a remote device.
We started designing a model to manage the execution of Android applications from a resource utilization perspective. For this purpose we extended the notion of *Capabilities* which allows the encapsulation of the hardware requirements of the application in a transparent way with respect to users and developers. Moreover, the Capabilities are also used by the resource manager to have a global representation of the device status.

To face the main challenge of managing applications that do not support run-time reconfiguration or do not have API to interface with, we extended the *BarbequeRTRM* architecture with some modules. They allow the BarbequeRTRM to filter available applications basing on Capabilities availability and to perform some operations on them (e.g., making them runnable on a specific device, allocating specific resources, etc. . .).
Furthermore we provided a set of API for developers who want to implement energy-aware applications that can interact with the BarbequeRTRM.

The next step of our work regards the interaction of the different instances of the BarbequeRTRM that run on different devices to actually *enable a distributed resource management strategy*. At this regard our work is one of the first that introduces and exploits the distributed modules of the resource manager implementing a RPC communication interface and an hybrid distributed system management topology.

Eventually, we focused on developing a first device selection policy, called *BestWing*, to select the best device in term of energy efficiency for the execution of a specific application. The selection of the metrics to be considered in the best device choice was one of the most challenging step.
Since our objective has been the energy efficiency, we evaluated the energy efficiency of different hardware configurations for a specific set of applications. We started by profiling some benchmark applications that simulate typical mobile-oriented workloads to obtain specific energy efficiency models.
We have noticed a similarity between the models of the image processing bench-

marks so we profiled three non benchmark video player applications performing similar task to verify if we could obtain a workload classification to exploit in the policy. This has been obtained for the image processing and video processing applications with a restrained error.
Such models have been used by the *BestWing* policy to compute a custom-defined Energy Consumption Index to score the different devices taking into account also their battery and plugged status.
In this regard, the experimental phase of our work includes some measurements on the policy overhead and a discussion about the benefits and drawbacks in terms of device lifetime and application performance introduced by adopting an energy-efficiency model. In particular, as well as increasing the utilization of multiple mobile devices available to the user, the proposed approach increases the device lifetime between 12% to 36%, making it very suitable for applications that do not have real-time or response constraints. Moreover, we tested the policy in various scenarios with different device configurations and we investigated the time composition of the different steps of the selection process making some considerations about some critical issues, such as the scalability with respect to the number of the device registered in the system.

In conclusion, being the computation offloading problem one of the most active fields of study in the last years, we expect that the focus on this topic will move towards a more balanced approach between local systems and Cloud solution, which require high network bandwidth and have a cost. In this sense the possibility to manage the device through a resource manager according to an energy and performance-aware approach will be an additional key factor that would introduce further benefits, as we have shown.

6.2 Future works

At the current state, our work enables the possibility to manage monolithic Android applications through a resource manager over a network of interconnected devices. Overcoming the limitation of a single entire application launching would open up interesting challenges on different use cases scenarios. In this Section we provides some possible approaches for future works that exploits our management model and aim at optimizing the energy consumption and computation performance of every applications.

6.2.1 Application profiling

The first approach aims at improving the current application profiling methodology. In fact the models we extracted can be further refined in order to make the

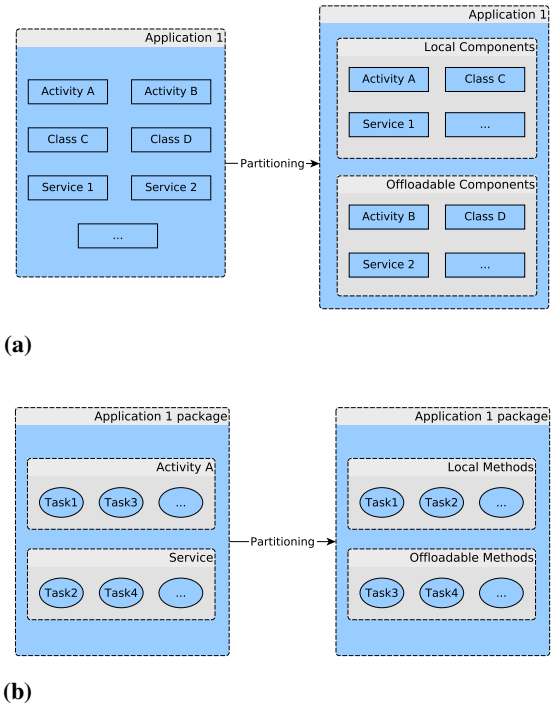


Figure 6.1: *The application partitioning process at two different levels. In the Figure 6.1a the application is partitioned at class levels, while in the Figure 6.1b it is partitioned at method (or task) level. In both cases the partitioned components are divided into two categories basing on the fact that they can be offloaded or not.*

classification more precise. We suggest one directions that can be exploited by research of next years. The general idea is to consider the application Capabilities as a discriminant to classify the applications with their purpose typology. In this sense a detailed Capabilities energy model has to be formulated. Then the applications can be compared according to their Capabilities specification and classified accordingly. In this way the energy model of the single class can be inferred by the composition of the single Capabilities ones.

Finally, another improvement regards the characterization of the time and percentage usage of the single capabilities by the application in order to allow a weighted dynamic capabilities allocation approach of the policy.

6.2.2 Application partitioning

This approach concerns the possibility of profiling and partitioning the Android applications at different level (e.g., class, methods etc...) to generate small blocks of code that can be run independently in different devices as shown in Figure 6.1. In Section 2.2 we exposed an overview of different state-of-the-art tools that allow this type of operations. One possible direction of our research

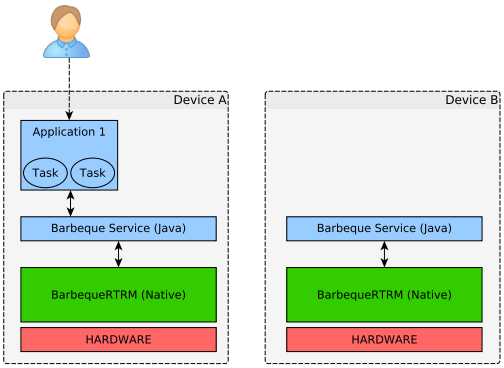
line could be an exploitation and evaluation of those tools to retrieve any critical issues. A further step would involve the development of a new distributed programming framework to make applications partitionable and reconfigurable. In this way the application would become resource-aware and it could be possible to exploit all the novel functionalities of the BarbequeRTRM: in particular the management of the offload of its high intensive computational tasks as explained below.

6.2.3 Application tasks offloading

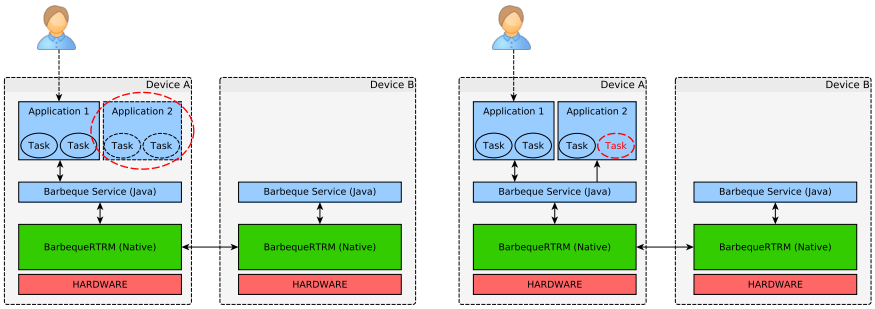
Once the application is successfully partitioned in blocks of code that perform different tasks it would be challenging to extend our model to give to the BarbequeRTRM the responsibility for managing the different blocks of the application. In such scenario the resource manager would select the best devices in which the different piece of codes can be launched and allocate specific resources on them basing on the required Capabilities related to the specific block.
The Figure 6.2 shows an example of the remote task offloading techniques through a *Pervasive Distributed Computing System* enabled by the BarbequeRTRM instances running on different personal devices [136].

6.2.4 BarbequeRTRM as Android resource manager

To allow the BarbequeRTRM to manage the entire device resources allocation in an effective way it would be interesting to embed it in a custom version of the Android Operating System. In this way it can act both as system resource manager and also as application manager.
To perform this merging it can be noticed that the lifecycle of a standard Android application is very similar to the BarbequeRTRM AEM. In this sense, it would be trivial to map some application callback with the AEM ones and to give to the resource manager the full control of the application schedule. Otherwise the big effort would consist in the modification of the Android Application framework.

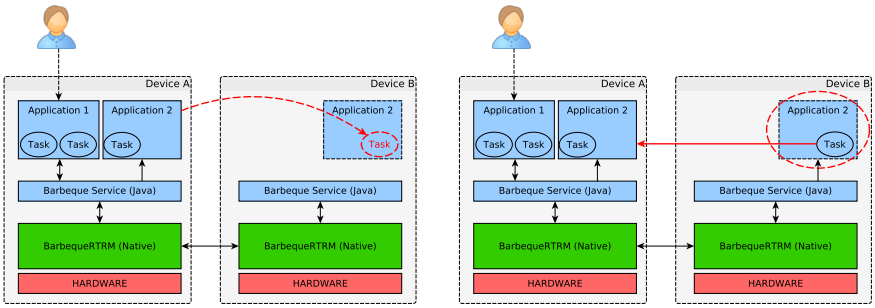


(a) *At the beginning the user selects the Application 2 in the Device A through the launcher.*



(b) *The BarbequeRTRM schedules the Application 2 main activity in the Device A after checking other devices status.*

(c) *The Application 2 and its tasks are started in the Device A*



(d) *the BarbequeRTRM decides to offload a task of the Application 2 to the Device B*

(e) *Finally the Device B creates the environment for the Application 2 and starts the remote task*

Figure 6.2: *The application task offloading process. At the end of the final step the remote task directly returns the results to the Application 2 main thread on Device A.*

6.3 Future improvements

Although our work is quite complete regarding the application remote launching, it is an initial step and some details have been deliberately omitted due to time and effort constraints. In what follows we present some critical issues that has to be addressed in future developments (and in which the author hope to be involved in).

6.3.1 Device profiling

To improve our Capabilities model and their resource specification it would be interesting to profile and inspect in depth single device features and hardware components. In this way a device profiler can stress and test resources constraints related to the single feature and components utilization. The result of this analysis can be further integrated into the Capabilities specifications according to their features and hardware requirements set.

6.3.2 Device selection policy

First of all the implemented device selection policy faces the device selection and resource allocation in a greedy way for a single application but it has some critical issues regarding multiple applications or tasks scheduling in the same device. In this sense an effective policy should be developed. Moreover the policy has some communication latencies that cause it to be not suitable with a large number of interconnected devices.
To face the first problem it would be required an allocation model that satisfy multiple constraints. This is a well-known problem in Cloud and high-performance distributed computing contexts so a deep study of the literature will give an acceptable research starting point.
The second problem is more difficult to solve because the communication latency does not directly depend on the policy but on the communication protocol used and the network infrastructure. Working on the policy a possible solution can go through a fully-distributed scheduling approach in which all devices cooperates with a lightweight message passing communication protocol to make scheduling decision and conflicts avoidance. In such a way it would be evaluated if the overhead produced by this mechanism is less than the one introduced by the RPC approach.
Finally, there are some main aspects about Capabilities that are not considered by our solution and that should be further investigate to implement other policies. Some topics can be the following: dynamic run-time Capabilities allocation,

considering device saturation and workload for multiple applications allocation, introducing an usage-based weight for Capabilities allocation, etc. . .).

6.3.3 Distributed system management

Regarding to the distributed system management some issues have not been addressed by this work but we consider them prioritary for future development. In particular there are two aspects that have to be covered: the distributed group management and the system security.

In our architecture the management of the distributed group is delegated to the *Distributed Manager* that was presented in Subsection 4.2.4. Since in the context of this work it was partially implemented, there are some details that has not been treated: the discovery mechanisms and fault tolerance protocol.
The *Distributed Manager*, at time of writing, does not support any automatic discovery mechanism and systems available in a group are provided statically with a configuration file. Enabling a dynamic management of the distributed group would be an interesting feature in a real use-case scenario. Moreover in a task offloading context it would be necessary a fault-tolerance protocol to maintain consistent the data involved in the computation. A first step could be an experimental implementation of well-know protocols that are already implemented in the literature of distributed systems.

Finally, since data are passed through various nodes a worth to be considered aspect is the group and communication security. Firstly it would be required a secure protocol to recognize allowed devices that can join the computational group. Secondly it would be exploited the security channel communication of Google gRPC which support different authentication and encryption mechanism such as SSL/TLS or token-based.

APPENDIX A

Android

A.1 Android manifest

Every Android application requires an *AndroidManifest.xml* [137] file in its root directory. It is used by the Android system to retrieve essential information about the application. Among other things the manifest:

- Contains the name of the Java package of the application, which is an unique identifier for the application
- Contains all the components of the application (activities, services, broadcast receivers, etc.) and the classes that implement a specific component
- Determines which processes will host application components
- Declares the permissions required by the application
- Declares the permissions that others are required to have in order to interact with the application's components
- Declares the minimum level of the Android API required by the application
- Can declare the device optional or required features by the application
- Lists the libraries that the application must be linked against.

The manifest is a `.xml` file structured as shown in Listing A.1:

Listing A.1: *AndroidManifest.xml structure*

```
<?xml version="1.0" encoding="utf-8"?>

<manifest>

    <uses-permission /> <!--PERMISSIONS DECLARATION-->
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature /> <!--FEATURES DECLARATION-->
    <supports-screens />
    <compatible-screens />
    <supports-gl-texture />

    <application>

        <activity> <!--ACTIVITY DECLARATION-->
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>

        <activity-alias>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </activity-alias>

        <service> <!--SERVICE DECLARATION-->
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </service>

        <receiver>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </receiver>

        <provider>
            <grant-uri-permission />
            <meta-data />
```

```
46              <path-permission />
47          </provider>
48
49          <uses-library  />
50
51      </ application >
52
53  </manifest>
```

The core components of an application are activated by *intents*: a bundle of information describing a desired action, data, a category of component that should perform the action and other pertinent instructions. Android finds an appropriate component to respond to the intent, launches the found component and passes it the Intent object. Components expose the kinds of intents they can respond to through *intent filters* in the manifest.

Regarding to *permissions*, they are restrictions that limit the access to a part of the code or data on the device. Each permission is identified by a unique label. If an application needs access to a feature protected by a permission, it must declare that it requires that permission. In some cases, the installer asks the user whether to grant the requested permission or not. It is possible to declare custom permissions.

A.2 Android Features model

A **feature** is a single hardware or software component of a device [109]. In the *AndroidManifest.xml* file it is possible to declare a feature that is used by the application through the tag `<uses-feature>`. The element offers a `required` attribute that lets you specify whether the application requires and cannot function without the declared feature or whether it prefers to have the feature but can function without it. If the attribute is omitted, it is assumed that the feature is required. This element has an important role in letting the application describe the device-variable features that it uses.

An important and debatable issue is that `<uses-feature>` elements are only informational: the Android system itself does not check for matching feature support on the device before installing an application. Only the Google Play performs this check when it presents to the user the installable applications. To determine an application feature compatibility with the device, it compares the features required by the application and the features available on the device.

Features can be *explicitly declared* or *implicit evaluated*. An explicitly declared feature is one that the application declares in a `<uses-feature>` element in the manifest. An implicit feature is one that an application requires

in order to function properly, but that is not declared in the manifest file. It is possible to discover an application implied feature requirements by examining the <uses-permission> elements declared in the manifest file. If an application requests some hardware-related permissions, it can be assumed that the application uses the underlying hardware features and therefore requires those features.

A full features reference can be found at [109] webpage.

A.3 Android Service

A **Service** is an application component that can perform long-running operations in the background and does not provide an user interface [138]. A service can be started by another application and continues to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and performing inter-process communication (IPC). The different between "started" and "bound" services is that the former are started by an application component calling `startService()` and they run in indefinitely even if the component that started them is destroyed. The second type are bound by an application calling `bindService()` and they run only as long as another application component is bound to them, unless they have been "started" before the binding.

To create a service it is necessary to create a sublcass of the `Service` class. Then, it should override some callback methods to handle service lifecycle and provide a mechanism for components to bind the service. These callbacks are:

- `onStartCommand()` called when another component calls `startService()` method. Using this method `stopSelf()` or `stopService()` methods are necessary to stop the service
- `onBind()` called when another component invokes the `bindService()` method. The implementation of this method must provide an `IBinder` interface that clients use to communicate with the service
- `onCreate()` called by the system when the service is created for the first time
- `onDestroy()` called by the system when the service is no longer used and is being destroyed.

A **bound service** is an implementation of the `Service` that allows other applications to bind it and interact with it [139]. To provide binding, it is necessary to implement the `onBind()` callback method. This method returns an

`IBinder` object that defines the programming interface that clients can use to interact with the service.
When a client wants to bind a service, it has to call the `bindService()` method and provides an implementation of the `ServiceConnection` interface. When the system creates the connection between the service and the application, the `onServiceConnected()` callback method is called to deliver the `IBinder` that the client can use to communicate with the service.
There are three ways to define the interface exposed by the service to communicate with the application:

1. Extending the `Binder` class for private service within the same application

2. Using a `Messenger` to work across different processes

3. Using a AIDL to handle multiple requests simultaneously.

Finally we provide in Figure A.1 the lifecyle of a started service that allows binding operations.

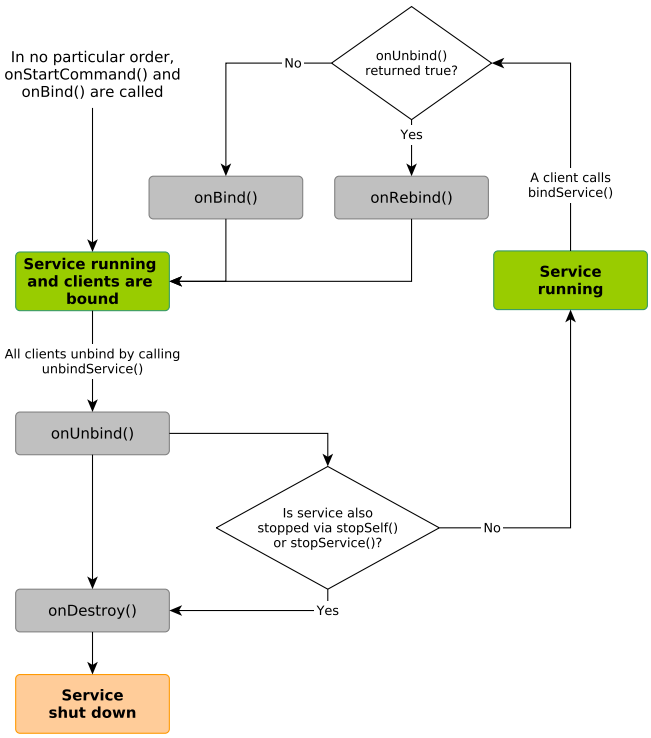


Figure A.1: *The lifecycle for a service that is started and also allows binding (derivative work from Free Electrons [140])*

A.4 Android system information and control

The Android API exposes some classes to retrieve system information and to control different aspects of the system.

A.4.1 Android *PackageManager*

This class provides various kind of information related to the application packages that are currently installed on the device [141]. The main methods of the `PackageManager` that can be invoked are:

- `queryIntentActivities(...)`: it retrieves all activities that can be performed for a given intent
- `getInstalledPackages(...)`: it returns a list of all packages that are installed on the device
- `getLaunchIntentForPackage(...)`: useful method to retrieve a ready intent to launch a front-door activity in a package. It looks first for a main activity in the category `CATEGORY_INFO` and next in the category `CATEGORY_LAUNCHER`
- `setApplicationEnabledSetting(...)`: this can be used to set the enabled setting for an application. If an application is disabled it cannot be launched by Android.

A.4.2 Android *ActivityManager*

This class provides methods to interact with the overall activities running in the system [142]:

- `getMemoryInfo()`: it returns information about the memory state of the system
- `getMyMemoryState()`: it returns global memory state information for the calling process
- `getProcessMemoryInfo(int[] pids)`: it retrieves information about the memory usage of one or more processes
- `getRunningAppProcesses()`: this can be used to retrieve a list of application processes that are running on the device containing also their package names and process PIDs.

A.4.3 Android.os package

The Android.os package provides basic operating system services, message passing and inter-process communication interfaces on the device. In this package we can find:

- The `IBinder` interface used when an application binds a Service to enable a lightweight *remote procedure call* through a `Binder`
- The `Messenger` class: a reference to a Handler which others can use to send message to it. This is used to allow message-based communication across processes and it uses a `Binder` to perform the communication
- The `PowerManager` class that gives the control of the power state of the device providing information about *interactive state* and *power-save mode state* of the device [143]
- The `BatteryManager` class: it provides a method to query battery and charge properties [144]
- The `Process` class, some tools for managing OS processes [145]. It is possible to kill or send a signal to a particular process knowing its PID and set the priority (Linux based) of a specific thread.

APPENDIX B

GoogleRPC

B.1 Protocol Buffers

Protocol buffers are language-neutral and platform-neutral automated mechanisms to serialize structured data for use in communication protocols, data storage and more [146]. Their main advantage is that the developer defines how the data has to be structured once, then it can easily use the source code automatically generated by the compiler to write and read the data to and from different streams through a variety of languages.
The information are defined in a protocol buffer message type in `.proto` files. Each message is a structured record of information with a series of name-value pairs as shown in Listing B.1.

Listing B.1: *Simple Protocol Buffers message definition*

```
1  message SystemInfo {
2    int32  battery_level  = 1;
3    bool  is_plugged  = 2;
4    string  system_name = 3;
5    string  system_model = 4;
6    repeated  int64  available_frequency  = 5;
7  }
```

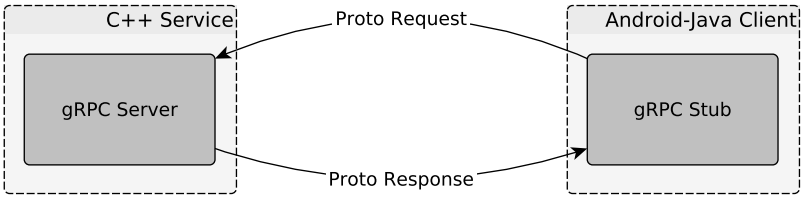


Figure B.1: *The gRPC architecture between two device. The server-side is implements a C++ gRPC server while in the client-side there is an Android-Java gRPC Stub. They exchange protocol buffer messages to each other.*

In this example, we want to define a `SystemInfo` message that contains some numbered fields specified by a name, a value type and other types of attributes (e.g. repeated, optional or required). Moreover, protocol buffers allow to structure the data hierarchically by specifying other message name in the value type of a field.
Once the message is defined the compiler generates the data access classes. These provide simple getters and setters for each field and some methods to serialize and parse the message structure to/from raw bytes. At this point the developer can use these classes in the application following the language-specific syntax as shown in Listing B.2.

Listing B.2: *C++ utilization of the generated Protocol Buffers class*

```
1  SystemInfo system_info;
2  system_info. set_battery_level (50);
3  system_info. set_is_plugged (0);
4  system_info.set_system_model("NEXUS_5");
5  // Writing the message to an output stream
6  fstream output("myfile", ios :: out | ios :: binary);
7  system_info. SerializeToOstream(%output);
```

Currently the version 3 of Protocol Buffers supports a lot of languages as C++, Java, Python, Ruby, JavaScript, C#.

B.2 GRPC

Like other type of RPC systems, in gRPC a client application can directly call methods on a server running on a different machine as if it was a local object [147]. In gRPC the main concept is the definition of a service which is made available by a server application and wraps the methods that can be invoked remotely. As shown in Figure B.1 the server implements the interface and runs a gRPC server to handle client requests, while the client instantiates a stub that provides the same methods provided by the server. To support cross-platform

implementations and a lot of variety of languages gRPC exploits the Protocol Buffers as IDL for the communication messages and services interface definition.

The gRPC services are defined in ordinary `.proto` files. Each service contains the available RPC methods with their parameters and return types. These last are specified as protocol buffer messages. Listing B.3 shows an example of a service and rpc definition.

Listing B.3: *Simple gRPC service interface definition*

```
1  service RemoteAgent {
2    rpc GetSystemInfo(GenericRequest) returns (SystemInfo) {}
3  }
4
5  message GenericRequest {
6    uint32 sender_id = 1;
7  }
8
9  ...
```

We defined the `RemoteAgent` service which exposes a `GetSystemInfo` rpc method. This last accepts a `GenericRequest` parameter which is a simple protocol buffer message defined in the same file. The method returns a `SystemInfo` message that it was defined in Section B.1.

At this point the `protoc` protocol buffer compiler, extended with a special gRPC plugin, generates the code from the proto file. The gRPC plugin lets the compiler to generate also the client and server code which is used by the developer to write the application code.

B.2.1 RPC typology

GRPC allows to define different type of service methods:

- *Unary RPCs*: the most simple rpc in which the client sends a single request to the server and gets a single response from it. This is the one implemented in the example of Listing B.3

- *Server streaming RPCs*: this type allows the client to send a single request to the server. This last responds with a sequence of messages through a read-only stream

- *Client streaming RPCs*: this type provides to the client a writable stream to send a sequence of messages to the server. The client waits until the server reads all the messages to gets its single response

- *Bidirectional streaming RPCs*: two independent read-write streams are provided to both the client and the server so that they can exchange sequences of messages in whatever order they like. In this case the order of messages in each stream is preserved.

Moreover, RPCs can be performed in two mode:

- *Synchronous*: the current thread of the client is blocked until a response arrives from the server
- *Asynchronous*: the client continues its work until a response interrupt arrives from the server.

B.2.2 Example tutorial

To better understand how the gRPC framework works we briefly present a simple implementation of a server and client applications which are simplify and extrapolated from our work. In this example we refer to the gRPC service defined in Listing B.3.

We decide to implement a C++ server and a Java client exploiting the generated protocol buffers and gRPC classes.

Server side implementation

Listing B.4: `server.h` *header file.* `RemoteAgentImpl` *class definition*

```
...
class RemoteAgentImpl final : public RemoteAgent::Service {
    ...
}
```

Listing B.5: `server.cc` *source file.* `GetSystemInfo` *method implementation*

```
#include <server.h>

RemoteAgent::Status RemoteAgentImpl::GetSystemInfo(ServiceContext* context,
      const GenericRequest* request,
      SystemInfo* system_info) override {
  system_info->set_battery_level(GetBatteryLevel());
  system_info->set_is_plugged(GetIsPlugged());
  system_info->set_system_model(GetSystemModel());
  return Status::OK;
}
...
```

The server side code is shown in Listings B.4 and B.5. We defined a `RemoteAgentImpl` class that implements the generated synchronous `Service` interface in the `server.h` header file. The service method `GetSystemInfo` is then implemented in the `server.cc` file overriding the generated one. It accepts a context object for the RPC, the `GenericRequest` protocol buffer request and a `SystemInfo` protocol buffer to fill in with the response information. The method then call different utility functions to populates the fields of the response object. It eventually returns an OK status code to communicate to the gRPC that it has finished and that the `SystemInfo` object can be returned to the client.

Once all the service methods are implemented, we can add some code in the `server.cc` to start up the gRPC server.

Listing B.6: *`server.cc` source file. gRPC server start up.*

```
1   ...
2   void RunServer(){
3     std :: string  server_address ("0.0.0.0:50051");
4     RemoteAgentImpl service();
5
6     ServerBuilder  builder;
7     builder.AddListeningPort(server_address, grpc::InsecureServerCredentials());
8     builder.RegisterService(&service);
9     std :: unique_ptr<Server> server(builder.BuildAndStart());
10    server->Wait();
11  }
```

As shown in Listing B.6 to build and start the server we used the `ServerBuilder` factory object. It requires an instance of the `RemoteAgentImpl` class to register the implemented service. Moreover, we have to specify through the `AddListeningPort` method the address and port we want to use to listen for client requests. Finally we build and start the server and block the thread until the process is killed.

Client side implementation

In the client side we just need to create the *stub* object, which represents the service locally, and a *gRPC channel* to communicate with the server.

Listing B.7: *`Client.java` source file. gRPC channel creation.*

```
1   public class RemoteAgentClient{
2     String host = "0.0.0.0";
3     int port = 50051;
4     // The gRPC channel
```

```
5       private final ManagedChannel channel;
6       // The gRPC stub
7       private final RemoteAgentBlockingStub blockingStub;
8
9       public RemoteAgentClient(){
10        channel = ManagedChannelBuilder.forAddress(host, port).build();
11        blockingStub = RemoteAgentGrpc.newBlockingStub(channel);
12      }
13      ...
14    }
```

As shown in Listing B.7, first of all we created an unsecure gRPC channel specifying the server address and port. Then we created the blocking[1] stub using the `newBlockingStub` method provided by the auto-generated `RemoteAgentGrpc` class.

Listing B.8: *`Client.java` source file. `GetSystemInfo` method invocation.*

```
1   public class RemoteAgentClient{
2     ...
3     public SystemInfo GetSystemInfo(){
4       GenericRequest request;
5       SystemInfo systemInfo;
6       request = GenericRequest.newBuilder().setSenderId(getMyId());
7       try{
8         systemInfo = blockingStub.getSystemInfo(request);
9       } catch (StatusRuntimeException e) {
10        // Log message
11        return;
12      }
13      return systemInfo;
14    }
15  }
```

At this point we can create and populate the request protocol buffer `GenericRequest` object through a factory object provided by the gRPC generated APIs. Finally we can invoke the `GetSystemInfo` method on the blocking stub as calling a local method, passing the request object. It in turn returns a `SystemInfo` object or throws an exception if any error occurs (Listing B.8).

[1] synchronous

APPENDIX C

Listings

C.1 BarbequeRTRM AgentProxy gRPC interface

Listing C.1: *Protocol buffer communication interface definition between the* BarbequeRTRM *instances*

```
service RemoteAgent {
    rpc GetSystemInfo(GenericRequest) returns (SystemInfo);
    rpc GetAvailableApplications (GenericRequest) returns (stream AppInfo);
    rpc SetApplicationManagementAction(ApplicationManagementRequest) returns
        (GenericReply);
    rpc SetCapabilitiesViewManagementAction(CapabilityManagementRequest) returns
        (GenericReply) {};
    rpc GetResourceStatus(ResourceStatusRequest) returns (ResourceStatusReply);
    rpc GetWorkloadStatus(GenericRequest) returns (WorkloadStatusReply);
    rpc SetResourceManagementAction(ResourceManagementRequest) returns (GenericReply) {};
     ...
}

message AppInfo {
    string name = 1;
    string package = 2;
    int32 pid = 3;
}
```

```
message SystemInfo {
    int32  battery_level  = 1;
    bool  is_plugged  = 2;
     string  system_name = 3;
     string  system_model = 4;
    repeated  int64  available_frequency  = 5;
    repeated  string  available_capability  = 6;
}

message ApplicationManagementRequest {
    uint32  sender_id  = 1;
    AppInfo info  = 2;
    enum Action {
        START = 0;
        STOP = 1;
    }
    Action  action  = 3;
    int32  awm_id  = 4;
}

message  Capability {
     string  name = 1;
    bool  set_available  = 2;
}

message CapabilityManagementRequest {
    uint32  sender_id  = 1;
    int32  view = 2;
    repeated  Capability  capability  = 3;
    enum ViewAction {
      GET_VIEW = 0;
      BOOK = 1;
      PUT_VIEW = 2;
      COMMIT_VIEW = 3;
    }
    ViewAction view_action  = 4;
}

message GenericRequest {
       uint32  sender_id  = 1;
}

message GenericReply {
    num Code {
      OK  = 0;
      AGENT_UNREACHABLE = 1;
      AGENT_DISCONNECTED = 2;
      REQUEST_REJECTED = 3;
   }
```

```
    Code value = 1;
    int32 extraValue = 2;
}

message ResourceStatusRequest {
    uint32 sender_id = 1;
    string path = 2;
    bool average = 3;
}

message ResourceStatusReply {
    uint64 total = 1;
    uint64 used = 2;
    uint32 degradation = 3;
    uint32 temperature = 4;
    uint32 power_mw = 5;
    uint64 frequency = 6;
}

message ResourceManagementRequest {
    repeated CpuConfig cpu_config = 1;
}

message CpuConfig {
    uint32 cpu_nr = 1;
    uint64 cpu_freq = 2;
}

...
}
```

C.2 Android BarbequeRTRM API

Listing C.2: *Binding interface exposed by the* Barbeque Service daemon

```
1  ...
2  public class BarbequeService extends Service {
3      ...
4      public final Messenger mMessenger =
5          new Messenger(new IncomingHandler());
6      ...
7      @Override
8      public IBinder onBind(Intent intent) {
9          return mMessenger.getBinder();
10     }
11     ...
12 }
```

Listing C.3: Barbeque Service *binding operation (proxy side)*

```
1  public class ServiceProxy {
2      private Context context;
3      private ServiceReceiver mApp;
4      private Messenger mService = null;
5      private boolean mIsBound;
6      private final Messenger mMessenger =
7          new Messenger(new IncomingHandler());
8      private ServiceConnection mConnection =
9          new ServiceConnection() {
10             @Override
11             public void onServiceConnected(
12                     ComponentName name,
13                     IBinder service) {
14                 mService = new Messenger(service);
15                 mApp.onServiceConnected();
16                 Log.i(TAG,"Service Attached!");
17
18                 try {
19                     Message msg = Message.obtain(
20                         null,
21                         MSG_REGISTER_CLIENT);
22                     msg.replyTo = mMessenger;
23                     mService.send(msg);
24                 } catch (RemoteException e) {
25                     // ...
26                 }
27             }
28     };
29     public void doBindService() {
30         Intent serviceIntent = new Intent();
31         serviceIntent.setComponent(
32             new ComponentName(
33                 "it.polimi.bosp.barbequedaemon",
34                 "it.polimi.bosp.barbequedaemon.BarbequeService"));
35         if (!mIsBound) {
36             if (context.bindService(
37                     serviceIntent,
38                     mConnection,
39                     Context.BIND_ADJUST_WITH_ACTIVITY)) {
40                 Log.i(TAG, "Binding success!");
41                 mIsBound = true;
42             } else {
43                 // ...
44             }
45         }
46     }
47 }
```

Listing C.4: *`getApplications` Message handling (`ServiceProxy` side)*

```
1   ...
2   public ServiceProxy {
3       ...
4       public void getApplications () {
5           if (mIsBound) {
6               try {
7                   Message msg = Message.obtain(
8                       null ,
9                       MSG_GET_APPLICATIONS);
10                  Log.i(TAG, "Requesting applications ... ");
11                  msg.replyTo = mMessenger;
12                  mService.send(msg);
13              } catch (RemoteException e) {
14                  // Nothing
15              }
16          }
17      }
18      class IncomingHandler extends Handler {
19          @Override
20          public void handleMessage(Message msg) {
21              switch (msg.what) {
22                  case MSG_GET_APPLICATIONS:
23                      Log.i(TAG, "Receiving applications ... ");
24                      Bundle bundle = (Bundle) msg.obj;
25                      mApp.onApplicationsReceived(bundle
26                          . getStringArrayList (APPLICATIONS_LIST)
27                          . toString () );
28                      break;
29                  // Other cases
30              }
31          }
32      }
33  }
```

Listing C.5: `getApplications` *Message handling (*Barbeque Service *side)*

```
1   ...
2   public class BarbequeService extends Service {
3       ...
4       private List<Messenger> mBoundApplicationsList =
5           new ArrayList<>();
6       class IncomingHandler extends Handler {
7           @Override
8           public void handleMessage(Message msg) {
9               Messenger app = msg.replyTo;
10              String appName;
11              Bundle bundle;
12              switch (msg.what) {
13              case MSG_GET_APPLICATIONS:
14                  // Client request
15                  // to obtain available applications
16                  // Service responds to the client application
17                  // the list of available applications
18
19                  Log.i(TAG, "Sending available applications ... ");
20                  // Sending the ArrayList to the requesting
21                  // application through a Bundle
22                  bundle = new Bundle();
23                  bundle.putStringArrayList(
24                      APPLICATIONS_LIST,
25                      getAvailableApplication());
26                  sendMessage(
27                      app,
28                      Message.obtain(
29                          null,
30                          MSG_GET_APPLICATIONS,
31                          bundle));
32                  break;
33              // Other cases
34          }
35      }
36      private void sendMessage(Messenger app,Message msg) {
37          try {
38              app.send(msg);
39          } catch (RemoteException e) {
40              // The client is dead. Remove it from the list.
41              mBoundApplicationsList.remove(app);
42          }
43      }
44  }
```

Listing C.6: Barbeque Service *binding operation (application side)*

```
...
public class MainActivity
    extends AppCompatActivity
    implements ServiceReceiver{
        ...
        private ServiceProxy mService;
        ...
        @Override
        protected void onResume() {
            super.onResume();
            // Binding BarbequeService
            mService.doBindService();
        }
        @Override
        public void onServiceConnected() {
            // Do stuff here
        }
        ...
```

Listing C.7: `getApplications` *Message handling application side)*

```
...
public class MainActivity
    extends AppCompatActivity
    implements ServiceReceiver {
    private Button mApplicationsButton;
    @Override
    protected void onCreate(Bundle savedInstanceState ) {
        mApplicationsButton =
            (Button) findViewById(R.id. applications_button );
        mApplicationsButton. setOnClickListener (
            new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    mService. getApplications ();
                }
            }
        );
    }
    @Override
    public void onApplicationsReceived( List<AppInfo> applicationsList ) {
        // Do stuff with the retrieved applications list
    }
}
```

Bibliography

[1] C. Brandolese and W. Fornaciari. *Sistemi Embedded*. Pearson, 2007.

[2] ARM. big.little technology: The future of mobile. *ARM*, 2013.

[3] M. R. Fernandez. Nodes, sockets, cores and flops, oh, my. `http://en.community.dell.com/techcenter/high-performance-computing/w/wiki/2329`. Accessed: 2017-03-03.

[4] A.S. Tanenbaum and M. Van Steen. *Distributed Systems*. Pearson Ed., 2007.

[5] B. Goddfrey. *A Primer on Distributed Computing*. 2008.

[6] B. Praveen and G. Matish. Security enhancement in distributed networking. *IJCSMC*, 2015.

[7] ISC. Internet systems consortium. `https://ftp.isc.org/www/survey/reports/current/`. Accessed: 2016-10-30.

[8] F. Berman, G. Fox, and A. J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[9] R. Grimm et al. System support for pervasive applications. *ACM Trans. Comp. Syst.*, 2004.

[10] Y. Gu et al. An empirical study of high availability in stream processing systems. *Middleware '09*, 2009.

[11] Z. Zhang. A hybrid approach to high availability in stream processing systems. *ICDCS '10*, 2010.

[12] B. Philips, C. Stewart, B. Hardy, and K. Marsicano. *Android Programming, The Big Nerd Ranch Guide*. Pearson, Atlanta, GA, USA, 2015.

[13] Google Inc. Activity. `https://developer.android.com/reference/android/app/Activity.html`. Accessed: 2017-03-01.

[14] J.S. George. Android power management. `http://www.slideshare.net/jerrinsg/android-power-management`. Accessed: 2017-03-01.

[15] M. Hsu and J. Huang. Power management from linux kernel to android.

[16] S. Brahler. Analysis of the android architecture. *Project report - Karlsruher Institu fur Technologie*, 2010.

[17] P. Bellasi et al. A rtrm proposal for multi/many-core platforms and reconfigurable applications. *ReCoSoC*, 2012.

[18] D. Datla et al. Wireless distributed computing: A survey of research challenges. *IEEE Commun. Mag. 50*, 2012.

[19] M. Conti et al. From opportunistic networks to opportunistic computing. *IEEE Commun. Mag. 48*, 2010.

[20] M. Conti and M. Kumar. Opportunities in opportunistic computing. *IEEE Computer*, 2010.

[21] A. Ferrari, S. Giordano, and D. Puccinelli. Reducing your local footprint with anyrun computing. *Computer Communications*, 2016.

[22] C. Wu and L. Li. Wipdroid – a two-way web services and real-time communication enabled mobile computing platform for distributed services computing. *Services Computing, 2008. SCC '08. IEEE International Conference on*, 2008.

[23] Y. Xiong, S. Huang, and M. Wu. Shared resource and service management for mobile transparent computing. *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, 2013.

[24] Y.X. Zhang. Transparent computing: concept, architecture and example. *Chinese Journal of Electronics*, 2004.

[25] D.F. Parkhill. *The challenge of the computer utility*. Addison-Wesley Pub. Co. Reading, MA, 1996.

[26] D. Dìaz-Sàanchez et al. Flexible computing for personal electronic devices. *13 IEEE International Conference on Consumer Electronics (ICCE)*, 2013.

[27] J. Wernsing. Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing. *SIGPLAN Not. 45*, 2010.

[28] E.E. Marinelli. Hyrax: Cloud computing on mobile devices using mapreduce. *Master's thesis, Carnegie Mellon University*, 2009.

[29] P. Jain et al. Mc2: On-the-fly mobile compute cloud for computational intensive task. *ICARE*, 2013.

[30] P.R. Elespuru, S. Shakya, and S. Mishra. Mapreduce system over heterogeneous mobile devices. In *7th IFIP WG 10.2 Int. Workshop Softw. Technol. Embedded Ubiquitous Syst.*, 2011.

[31] H. Ba et al. Mobile computing - a green computing resource. In *IEEE WCNC*, 2013.

[32] D.P. Anderson. Volunteer computing: The ultimate cloud. *Crossroads 16*, 16, 2010.

[33] D.P. Anderson. Seti@home: An experiment in public-resource computing. *Comm. ACM 45*, 2002.

[34] FoldingHome project. Foldinghome. `http://folding.stanford.edu/5`. Accessed: 2016-10-30.

[35] BOINC. BOINC. `http://boinc.berkeley.edu`. Accessed: 2016-10-30.

[36] M. Szpakowski. Native boinc for android. `http://nativeboinc.org`. Accessed: 2016-10-30.

[37] J.R. Eastlack. Extending volunteer computing to mobile devices. *Master's thesis, New Mexico State University*, 2011.

[38] C. Funai et al. Extending volunteer computing through mobile ad hoc networking. *2014 IEEE Global Communications Conference*, 2014.

[39] Z. Dong et al. Repc: Reliable and efficient participatory computing for mobile devices. *IEEE int. Conf. Sensing, Communication and Networking*, 2014.

[40] M.Y. Arslan et al. Computing while charging: Building a distributed computing infrastructure using smartphones. In *8th International Conference on Emerging Network Experiments Technology*, 2012.

[41] Y. Mustafa et al. Cwc: A distributed computing infrastructure using smartphones. *IEEE TRANSACTIONS ON MOBILE COMPUTING*, 2015.

[42] F. Busching, S. Schildt, and L. Wolf. Droidcluster: Towards smartphone cluster computing - the streets are paved with potential computer clusters. In *ICDCSW '12*, 2012.

[43] Google Inc. Chromecast. `https://developers.google.com/cast/`. Accessed: 2017-03-20.

[44] E. Eason. Smartphone battery inadequacy. `http://large.stanford.edu/courses/2010/ph240/eason1/`. Accessed: 2017-03-01.

[45] K. Kumar et al. A survey of computation offloading for mobile systems. *Mobile Netw. Applications*, 2013.

[46] G.H. Forman and J. Zahorjan. The challenges of mobile computing. *Computer 27*, 1994.

[47] A.D. Joseph et al. Rover: a toolkit for mobile information access. *ACM symposium on operating systems principles*, 1995.

[48] D. Kotz et al. Agent tcl: targeting the needs of mobile computers. *IEEE Internet Computing 1*, 1996.

[49] B.D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in odyssey. *Mobile Netw Appl 4*, 1999.

[50] C.E. Perkins. Handling multimedia data for mobile computers. *Computer software and applications conference*, 1996.

[51] J.E. White. Mobile agents. *Software Agents, MIT Press*, 1997.

[52] D. Wong. Java-based mobile agents. *Com ACM 42*, 1999.

[53] D. Wong et al. Concordia: an infrastructure for collaborating mobile agents. *International workshop on mobile agents*, 1997.

[54] P. Bellavista, A. Corradi, and C. Stefanelli. Mobile agent middleware for mobile computing. *Computer 34*, 2001.

[55] G. Chen et al. Study energy trade offs in offloading computation/compilation in java-enabled mobile devices. *IEEE Trans Parallel Distrb Sys 15*, 2004.

[56] Z. Li, C. Wang, and R. Xu. Energy impact of secure computation on a handheld device. *IEEE international workshop on workload characterization*, 2002.

[57] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: a partition scheme. *International conference on compilers, architecture, and synthesis for embedded systems*, 2001.

[58] Y. Nimmagadda et al. Realtime moving object recognition and tracking using computation offloading. *IEEE international conferenced on intelligent robots and systems*, 2010.

[59] E. Tilevich and Y. Smaragdakis. J-orchestra: automatic java application partitioning. *European conference on object-oriented programming*, 2006.

[60] C. Wang and Z. Li. Parametric analysis for adaptive computation offloading. *ACM SIGPLAN conference on programming language design and implementation*, 2004.

[61] C. Xian, Y.H. Lu, and Z. Li. Adaptive computation offloading for energy conservation on battery-powered systems. *International conference on parallel and distributed systems*, 2007.

[62] Y. Zhang. Refactoring android java code for on-demand computation offloading. *ACM*, 2012.

[63] A. Ferrari, D. Puccinelli, and S. Giordano. Code mobility for on-demand computational offloading. *IEEE*, 2016.

[64] P. Rong and M. Pedram. Extending the lifetime of a network of battery-powered mobile devices by remote processing: a markovian decision-based approach. *Conference on design automation*, 2003.

[65] N. Seshasayee et al. Energy aware mobile service overlays: cooperative dynamic power management in distributive systems. *International conference on automatic computing*, 2007.

[66] X. Gu et al. Adaptive offloading inference for delivering applications in pervasive computing environments. *IEEE International conference on pervasive computing and communications*, 2003.

[67] S. Gurun and C. Krintz. Addressing the energy crisis in mobile computing with developing power aware software. Technical report, Department of Computer Science, University of California, Santa Barbara, 2003.

[68] S. Gurun, C. Krintz, and R. Wolski. Nwslite: A light-weight prediction utility for mobile devices. *International conference on mobile systems, applications and services*, 2004.

[69] R. Wolski et al. Using bandwidth data to make computation offloading decisions. *IEEE international symposium on parallel and distributed processing*, 2008.

[70] G. Orsini, D. Bade, and W. Lamersdorf. Context-aware computation offloading for mobile cloud computing: Requirements analysis, survey and design guideline. *The 12th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2015)*, 2015.

[71] M. Satyanarayanan et al. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 2004.

[72] E. Cuervo et al. Maui: Making smartphones last longer with code offload. *MobiSys'10*, 2010.

[73] V. March et al. mcloud: towards a new paradigm of rich mobile applications. *Procedia Computer Science*, 2011.

[74] Chun B.G. et al. Clonecloud: elastic execution between mobile device and cloud. In *Sixth conference on Computer systems, ser. EuroSys '11*, 2011.

[75] D. Kovachev, T. Yu, and R. Klamma. Adaptive computation offloading from mobile devices into the cloud. *Parallel and Distributed Processing with Applications (ISPA), IEEE 10th International Symposium on*, 2012.

[76] M.S. Gordon et al. Comet: code offload by migrating execution transparently. In *10th USENIX conference on Operating Systems Design and Implementation*, 2012.

[77] S. Kosta et al. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. *INFOCOM, 2012 Proceedings IEEE*, 2012.

[78] E. Chen. Offloading android applications to the cloud without customizing android. *Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2012.

[79] H. Qian and D. Andersen. Jade: reducign energy consumption of android app. *Int. J. Network. Distrib. Comput (IJNDC) 3*, 2015.

[80] F. Lordan and R.M. Badia. Compss-mobile: parallel programming for mobile-cloud computing. *IEEE/ACM International Syumposium on Cluster, Cloud and Grid Computing*, 2016.

[81] J. Liu et al. Application partitioning algorithms in mobile cloud computing: Taxonomy, review and future directions. *Journal of Network and Computer Applications*, 2015.

[82] W. Binder et al. Using bytecode instruction counting as portable cpu consumption metric. *Elettronic Notes in Theoretical Computer Science*, 2006.

[83] H. Eom et al. Malmos: Machine learning-based mobile offloading scheduler with misc training. *3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, 2015.

[84] F. Lordan et al. Services: An interoperable programming framework for the cloud. *Journal of Grid Computing, vol. 12, n.1*, 2014.

[85] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164, 2002.

[86] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 140–146. IEEE, 1998.

[87] V. Damen. *Introducing Linux virtual containers with LXC*. 2010.

[88] C. Augonnet et al. A unified runtime system for heterogeneous multi-core architectures. *Euro-Par 2008 Workshops-Parallel Processing*, 2009.

[89] C. Augonnet et al. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 2011.

[90] A.P.D. Binotto et al. Effective dynamic scheduling on heterogeneous multi/manycore desktop platforms. *22nd International Symposium on Computer ARchitecture and High Performance Computing Workshops*, 2010.

[91] C.K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on hterogeneous multiprocessors with adaptive mapping. *Microarchitecture, MICRO-42*, 2009.

[92] A. Bahga and V.K. Madisetti. A dynamic resource management and scheduling environment for embedded multimedia and communication platforms. *Embedded Systems Letters, IEEE*, 2011.

[93] E. Bini et al. Resource management on multicore systems: the actors approach. *IEEE Micro*, 2011.

[94] X. Fu and X. Wang. Utilization-controlled task consolidation for power optimization in multi-core real-time systems. In *2011 IEEE 17th International Conference on Embedded and Real-time Computing Systems and Applications*, 2011.

[95] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. *ACN SIGOPS Operating Systems Review*, 2003.

[96] BOSP. The barbeque opensource project. `http://bosp.dei.polimi.it/`. Accessed: 2017-03-30.

[97] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pages 323–332, New York, NY, USA, 1992. ACM.

[98] D. G. Feitelson et al. Theory and practice in parallel job scheduling. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, IPPS '97, pages 1–34, London, UK, UK, 1997. Springer-Verlag.

[99] K. Wang, X. Zhou, H. Chen, M. Lang, and I. Raicu. Next generation job management systems for extreme-scale ensemble computing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 111–114. ACM, 2014.

[100] F. Berman et al. Adaptive computing on the grid using apples. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, April 2003.

[101] J. Cao, S. A. Jarvis, S. Saini, D. J. Kerbyson, and G. R. Nudd. Arms: An agent-based resource management system for grid computing. *Sci. Program.*, 10(2):135–148, April 2002.

[102] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, January 1998.

[103] S. Kobbe et al. Distrm: Distributed resource management for on-chip many-core systems. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, CODES+ISSS '11, pages 119–128, New York, NY, USA, 2011. ACM.

[104] A. Beloglazov, J. Abawajy, and R. Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, 28(5):755–768, 2012.

[105] S. Hao et al. Estimating android applications' cpu energy usage via bytecode profiling. *GREENS*, 2012.

[106] X. Li and J. P. Gallagher. Fine-grained energy modeling for the source code of a mobile application. *arXiv*, 2016.

[107] L. Corral, A. B. Georgiev, A. Silitti, and G. Succi. A method for characterizing energy consumption in android smartphones. *GREENS*, 2013.

[108] C. Margi, K. Obraczka, and R. Manduchi. Characterizing system level energy consumption in mobile computing platforms. *GREENS*, 2012.

[109] Google Inc. uses-feature. `https://developer.android.com/guide/topics/manifest/uses-feature-element.html`, 2016. Accessed: 2017-03-01.

[110] A. Carroll and G. Heiser. Unifying dvfs and offlining in mobile multicores. *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[111] M. Altieri et al. Coupled voltage and frequency control for dvfs management. *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[112] P. Bellasi, G. Massari, and W. Fornaciari. Effective runtime resource management using linux control groups with the barbequertrm framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 2015.

[113] G. Massari et al. Extending a run-time resource management framework to support opencl and heterogeneous systems. *PARMA-DITAM*, 2014.

[114] S. Libutti, G. Massari, and W. Fornaciari. Addressing task co-scheduling on multi-core heterogeneous systems: An energy-aware perspective. *HIPEAC Workshop on Energy Efficiency with Heterogeneous Computing (EEHCO)*, 2015.

[115] W. Fornaciari et al. Runtime resource management for embedded and hpc systems. In *7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*, 2016.

[116] R. Vavrik et al. Precision-aware application execution for energy-optimization in hpc node system. *High Performance Energy Efficient Embedded Systems (HIP3ES), HiPEAC*, 2015.

[117] 2Parma. Parallel paradigms and run-time management techniques for many-core architectures. `http://2parma.microlab.ntua.gr/`. Accessed: 2017-03-30.

[118] HARPA. Harnessing performance variability. `http://www.harpa-project.eu/`. Accessed: 2017-03-30.

[119] Contrex. Design of embedded mixed-criticality control systems under consideration of extra-functional properties. `http://www.harpa-project.eu/`. Accessed: 2017-03-30.

[120] MANGO. Mango: exploring manycore architectures for next-generation hpc systems. `http://www.mango-project.eu/administrativedata`. Accessed: 2017-03-01/.

[121] A. Troina. Android run-time resource management - jni based integration of the barbequertrm framework. *Maester's thesis - Politecnico di Milano*, 2012.

[122] A. Urbieta, G. Barrutieta, J. Parra, and A. Uribarren. A survey of dynamic service composition approaches for ambient systems. In *Proceedings of the 2008 Ambi-Sys Workshop on Software Organisation and MonIToring of Ambient Systems*, SOMITAS '08, pages 1:1–1:8, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[123] Lg and Google Inc. Nexus 5. `http://www.lg.com/us/cell-phones/lg-D820-Sprint-Black-nexus-5`. Accessed: 2017-03-20.

[124] ODROID. Odroid-xu3. `http://www.hardkernel.com/main/products/prdt_info.php?g_code=g140448267127`. Accessed: 2017-03-20.

[125] D. Brodowski and N. Golde. Linux cpufreq. `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`. Accessed: 2017-03-03.

[126] BenchmarkXPRT Development Community. Mobilexprt2015. `http://www.mobilexprt.com/`. Accessed: 2017-03-22.

[127] A. Morgan. Benchmark selection guide, vol. 1. `https://www.slideshare.net/k12blueprint/understanding-benchmarks`. Accessed: 2017-03-15.

[128] Futuremark. Pcmark2.0. `www.futuremark.com`. Accessed: 2017-03-22.

[129] M. Horowitz, T. Indermaur, , and R. Gonzalez. Low-power digital design. In *IEEE Symposium on Low Power Electronics*. Institute of Electrical and Electronics Engineers (IEEE), 1994.

[130] D. Brooks et al. Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *IEEE Micro 20*, pages 26–44, 2000.

[131] K.W. Cameron et al. Poster: high-performance, power-aware distributed computing framework. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC)*. ACM/IEEE, 2004.

[132] J. H. Laros III et al. *Energy Delay Product*, pages 51–55. Springer London, London, 2013.

[133] LM LaVange and G. Koch Gary. Rank score tests. *Circulation 114*, pages 528–2533, 2006.

[134] Y. Chen, E. Macii, and M. Poncino. A circuit-equivalent battery model accounting for the dependency on load frequency. In *Proceedings of the Design, Computation and Test in Europe*, 2017.

[135] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993.

[136] G. Massari, M. Zanella, and W. Fornaciari. Towards distributed mobile computing. *2nd Workshop on Mobile Systems Technologies*, 2016 (in publishing).

[137] Google Inc. App manifest. `https://developer.android.com/guide/topics/manifest/manifest-intro.html`, 2016. Accessed: 2017-03-01.

[138] Google Inc. Services. `https://developer.android.com/guide/components/services.html`, 2016. Accessed: 2017-03-01.

[139] Google Inc. Bound services. `https://developer.android.com/guide/components/bound-services.html`, 2016. Accessed: 2017-03-01.

[140] FreeElectrons. Freeelectrons. `http://free-electrons.com/`. Accessed: 2016-10-30.

[141] Google Inc. Packagemanager. `https://developer.android.com/reference/android/content/pm/PackageManager.html`, 2016. Accessed: 2017-03-01.

[142] Google Inc. Activitymanager. `https://developer.android.com/reference/android/app/ActivityManager.html`, 2016. Accessed: 2017-03-01.

[143] Google Inc. Powermanager. `https://developer.android.com/reference/android/os/PowerManager.html`, 2016. Accessed: 2017-03-01.

[144] Google Inc. Batterymanager. `https://developer.android.com/reference/android/os/BatteryManager.html`, 2016. Accessed: 2017-03-01.

[145] Google Inc. Process, 2016. Accessed: 2017-03-01.

[146] Google Inc. Protocol buffer. `https://developers.google.com/protocol-buffers/`. Accessed: 2017-03-01.

[147] Google Inc. Grpc. `http://www.grpc.io/`. Accessed: 2017-03-01.