POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica

POLITECNICO
MILANO 1863

Liquid Android: Towards Distributed Android

Relatore:

Prof. Luciano Baresi

Tesi di Laurea di:

Marco Molinaroli

Matricola:

837721

Anno accademico 2015–2016

Ringraziamenti

Ringrazio, innanzitutto, il Professor Luciano Baresi, relatore di questa tesi, per il continuo supporto al mio lavoro, la grande disponibilità a la professionalità dimostrata.

Ringrazio la mia famiglia, senza la quale tutto questo non sarebbe stato possibile, per la fiducia che hanno sempre riposto in me e il sostegno dimostratomi in questi anni di studi al Politecnico. Ringrazio i miei genitori, Marina e Maurizio, per tutti i sacrifici fatti per permettermi di raggiungere questo traguardo. Ringrazio i miei fratelli, Michele e Martina, per avermi, anche inconsapevolmente, supportato, e soprattutto sopportato, durante il mio percorso di studi.

Ringrazio Sabrina, per essere sempre stata al mio fianco durante il mio percorso, per aver condiviso con me momenti di gioia e di difficoltà, per avermi sempre sostenuto e aver creduto nelle mie capacità.

Grazie a tutti i compagni con cui ho affrontato questi anni di università: in particolare Matteo, Federico, Alessandro e Alessio, compagni di esami e di progetti, e Alberto, Marco, Jacopo, Roberto e Davide, con i quali ho condiviso la fatica, la gioia e momenti indimenticabili.

Voglio ringraziare anche tutti i miei compagni di squadra che in questi anni mi hanno aiutato a staccare un po' dalla vita universitaria permettendomi di fare parte di due grandi gruppi di "pallavolisti", la ASD Castellana Volley e la Canottieri Ongina Volley.

Voglio infine ringraziare anche tutti i miei amici del gruppo del "Corpus Domini", compagni di innumerevoli esperienze di vita.

Milano, April 2017 Marco

Sommario

Negli ultimi anni, i dispositivi mobili, come smartphone e tablet, sono diventati sempre più popolari, performanti e meno costosi. Ognuno di noi, di questi tempi, utilizza ogni giorno, questo tipo di dispositivi, per svolgere svariate attività. L'evoluzione degli smartphone, ha certamente influenzato le nostre vite, cambiando il modo in cui interagiamo nel mondo reale, ad esempio introducendo nuovi modi di entrare in contatto con altre persone, usando i social network, e offrendo la possibilità di accedere ad Internet, pressoché in ogni momento.

In questo contesto di continua evoluzione, i dispositivi mobili, che sono stati progettati per essere macchine di calcolo portatili per uso personale, stanno diventando sempre più simili ai normali personal computer, dal momento che hanno sistemi operativi completi e sufficienti capacità di calcolo. Per questo motivo, essi, stanno progressivamente sostituendo i normali computer nello svolgimento di parecchi compiti, poiché sono, appunto, abbastanza potenti per completare diversi incarichi. Ciò che manca al momento, tuttavia, è un modo concreto, per usare questo tipo di dispositivi contemporaneamente, in modo da collaborare assieme per raggiungere un obbiettivo comune, o eseguire applicazioni distribuite, appositamente progettare per essi.

Questa tesi ha lo scopo di trovare una soluzione che permetta ai dispositivi mobili di cooperare come se stessero eseguendo un sistema operativo distribuito. In questo lavoro, è mia intenzione, trovare un modo per estendere Android, che è un sistema operativo per questo tipo di dispositivi, trasformandolo in un sistema operativo distribuito, senza però dover influenzare i normali meccanismi, con i quali normalmente funziona. Un altro scopo del mio lavoro è, inoltre, quello di proporre, e realizzare un framework, che ho chiamato *Liquid Android*, che sia in grado di supportare gli sviluppatori, nella creazione di applicazioni distribuite per Android. Per raggiungere questi obbiettivi, questo lavoro comprende l'analisi di tutti i requisiti e lo sviluppo di alcuni prototipi.

A differenza di altre soluzioni, il mio framework, non necessita dei privilegi di *root*, ed è completamente estensibile e open source. Per funzionare, infatti, necessita solamente che i dispositivi coinvolti siano connessi alla stessa rete WiFi.

Abstract

In the last few years, mobile devices, such as smartphones and tablets, are becoming increasingly popular, powerful and cheap. By now, all of us, every day, use these types of devices to carry out all kind of different activities. The smartphone evolution, indeed, has affected our lives, changing real world interaction such as innovative ways to get in touch with other people using social networks, and providing Internet network connectivity almost all the time.

In this context of continuous evolution, mobile devices which have been designed to be portable computing machines for personal use, are becoming more and more complete personal computers, with fully functional operating systems and large computational capabilities. They are progressively, substituting standard notebooks and desktop in performing many tasks, thus they are powerful enough to perform multi purposes duties. Nevertheless, what is lacking at the moment is a concrete way to use these devices together, in cooperation to achieve a common goal, or run native distributed applications.

This dissertation aims to find a way to let mobile devices cooperate as if they were running a distributed OS. I want to find a way to extend a mobile operating system, Android, to make it similar to distributed OSs, though without affecting its main working mechanisms. Furthermore, I want to propose a framework, called *Liquid Android*, supporting Android developers to implement native Android distributed applications. My work analyzes all these requirements and develops some prototypes to reach these objectives.

A opposed to existing solutions, my framework does not need to force the system to have root privileges, and it is completely extensible and open source. It only needs devices connected to the same standard WiFi network to work.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Nowadays mobile devices have changed the way we approach technologies, they are powerful enough to do most of the things we need in a fast and efficient way, without the need to use a *regular computer* with a *standard desktop operating system*. Mobile operating systems *(mobile OSs)*combine features of a personal computer operating system with other features useful for mobile or handheld use; usually including, and most of the following considered essential in modern mobile systems; a touchscreen, cellular, Bluetooth, Wi-Fi Protected Access,Wi-Fi, Global Positioning System (GPS) mobile navigation, camera, video camera, speech recognition, voice recorder, music player, and so on. By the end of 2016, over 430 million smartphones were sold with 81.7 percent running Android, 17.9 percent running iOS, 0.3 percent running Windows Mobile and the other OSs cover 0.1 percent [1].
Many people have multiple mobile devices for personal use, and for the reasons stated above it would be great if they could use this wide variety of mobile devices together, equipping their operating systems with services to make them *distributed mobile OSs*
I stated that Android is the most common mobile operating system, it is open source and does not need special developer licenses to build applications. So in this dissertation, I will try to provide a solution to the problem of making mobile operating systems acting as distributed OSs, focusing my attention on Android devices. It is now clear which Android is not only a tiny operating system, but a full functional OS to be used for general purposes. One of the most peculiar characteristic of the Android OS is that it can be installed in a variety of devices such as *"handled"*, like smart-phones and tablets, *"wearable"*, like smart-watches, but also in other kinds of things like standard desktops and laptops, smart-tv and tv boxes, and so on.
The great variety of devices described above can run and benefit all the functions of the Android OS which is acknowledged for its ease of use, and the great abundance of applications, with which users can do almost everything.
However, one of the greatest limitations of Android is that the system was designed to run on the top of a virtual machine and each application which can be

executed starts a Linux process which has its own virtual machine (VM), so an application code runs in isolation from other apps. This technique is called *"app sandboxing"* and it is used to guarantee a high level of security, because different applications can not read write, or worse steal, data and sensible information from other applications.That is, each app, by default, has access only to the components that it requires to do its work and no more. This creates a very secure environment in which an app can not access parts of the system for which it is not given permission.
Under this limitations the Android OS provides a mechanism to make the various components of the applications and the operating system itself communicate : the so called *"intents"*. An intent is an abstract description of an operation to be performed,it provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities. However, even though the intents can be created and resolved within the same android running devices, there is not a mechanism that can send and resolve intents from one devices to another.
In a world where computers are everywhere and can do almost everything and can communicate among themselves in different but efficient ways, the fact that android devices are not able to easily exchange intents is a very major limitation to the android users. As we know, our world is fast moving to a world of *"ubiquitous computing"* where there is no longer a single *"fat calculator"* but a variety of multipurpose and specialized devices. In this world of pervasive computation, Android devices are widespread, cheap and powerful enough to do most of the things that we can imagine and would be wonderful if they could be used together in a smart way. The aim of this thesis work is to study the android framework to find a solution to this problem, and create a middleware to extend the Android OS, creating a distributed system in which intents can be generated from one device and resolved by others in a network environment, for example in *Local Area Network (LAN)*, or to devices in range using a *WiFi direct* protocol. This can help developers build distributed native Android application to exploit the power of any different device running the OS and let the users use their own devices as if they were one single big device.
The proposed solution is a *middleware*, which extends the Android OS, turning it into a distributed operating system, by providing the networking structure, the communication language and data management solution. The result is a standard Android application, which operates as a background working middleware which could open the way of native Android distributed applications. Furthermore, by using the same approach, and by following the same steps I used in the solution chapter of this dissertation, it would be possible to let different mobile OS to communicate in this way, and to easily build cross platform distributed systems.

1.2 Outline

The thesis is organized as follows:

The second chapter describes the state of art: it provides a full overview on

current technologies, ideas and issues. The chapter starts by presenting the Android operating system with a brief history of versions. Then a detailed presentation of Android's framework component is given to the reader, including security model and connectivity functionalities. The chapter continues describing what is a distributed system, listing its main challenges, properties and its working mechanism such as the communication models, and architectural patterns. The third section adds some technical background explanations, it presents the term *Liquid computing*, and other existing technologies which can be useful to understand the problem, the proposed solution and the development of the system. The last section contains a list of already existing solutions, divided in commercial and academical ones.

In the third chapter I have defined the faced problem, its constraints and its boundaries. The chapter starts with a contextualization of the given problem, giving a brief recap of the state of the art.Then some restrictions are provided, considering only devices in which the developed system could be installed. The chapter continues with the full description of the problem, the main idea and also a working scheme of the component to be developed. Then the problematic scenarios to be studied are presented, including detailed description of what the middleware to be implemented should work in these situations. There is, finally, a list of constraints that the system must meet to be considered a good solution to the given problem.

In the fourth chapter I have analyzed the solution in detail.It can be considered the core and the central point where the innovation is described. I split the chapter in three main sections. Firstly, I dedicated a few pages to what we can call "General idea" for the solution. Then the section proposed solution contains all the steps and the analysis I performed to solve the given problem. This is the part which introduces the innovation, and provides the theoretical solution. In the last section there is the description of the so-called *Liquid Android API*, which is an Android library I have produced, and used to implement the solution.

In the fifth chapter I focused the attention on the real case study. The chapter starts with an overview of the system developed, a list of non-functional requirements my application must meet and an explanation of the used technologies. Then the focus is shifted to the implementation. The second section of this chapter is fully dedicated to the presentation of my application, there is a complete analysis of my implemented Liquid Android app with a full working demo and live test cases. The last section presents a second application, which is a concrete case of use of my framework, again with a deep description, screenshot and a complete test case.

In the sixth chapter it is possible to find final considerations about what has been done. In the first section there is the an analysis of what I have achieved and what can be considered promising. Then there are some

considerations about what it would be possible to do with my system in future and how it can be extended to become complete.

Chapter 2

State of the Art

2.1 Android OS

As already mentioned, in Section 1.1, the Android operating system is an open source OS developed by Google based on Linux kernel, that can be installed on many different kinds of devices.
In this section I want to give the reader the basic knowledge of the Android framework to understand why and how the operating system works.

2.1.1 Brief History

The Android era officially began on October 22nd, 2008, when the *T-Mobile G1* was launched in the United States [2].
At that time a company in Mountain View, Google, felt the need to create a new operating system which could be installed on most modern mobile phones of the time. To meet this need the Google engineers created an OS that was based on the Linux kernel, lightweight enough and easy to be used with simple hand gestures by touching the screen of the phone.

Figure 2.1: The T-Mobile G1 and the Android 1.0 menù

The main characteristic of the OS were and are also now:

- The pull-down notification window.
- Home screen widgets.
- The Android Market.
- Google services integration (eg. Gmail).
- Wireless connection technologies (eg Wi-Fi and Bluetooth)

The success of the first version of the brand new mobile operating system and the open source philosophy guaranteed the fast spread of the Android devices all over the world. In few years, Google improved and released many versions of the OS and with the help of the market growth, Android has become a complete operating system. In the table below there is a brief description of the various distributions of the Android OS at the time of writing of this document.

As we can see in Table 2.1 there are, currently, 25 level of the Android *API*

Table 2.1: Android versions

Name	Version	Release Date	API Level
Alpha	1.0	September 23, 2008	1
Beta	1.1	February 9, 2009	2
Cupcake	1.5	April 27, 2009	3
Donut	1.6	September 15, 2009	4
Eclair	2.0 – 2.1	October 26, 2009	5–7
Froyo	2.2 – 2.2.3	May 20, 2010	8
Gingerbread	2.3 – 2.3.7	December 6, 2010	9–10
Honeycomb	3.0 – 3.2.6	February 22, 2011	11–13
Ice Cream Sandwich	4.0 – 4.0.4	October 18, 2011	14–15
Jelly Bean	4.1 – 4.3.1	July 9, 2012	16–18
KitKat	4.4 – 4.4.4	October 31, 2013	19
Lollipop	5.0 – 5.1.1	November 12, 2014	21–22
Marshmallow	6.0 – 6.0.1	October 5, 2015	23
Nougat	7.0 – 7.1.1	August 22, 2016	24–25

(Application programming interface) which developers can use to build Android applications. In particular, various API levels introduce innovations in the OS but, applications developed using a higher *API level* cannot be executed in a device running lower versions of the operating system. This is a second major limitation for the *"Android ecosystem"*, moreover as mentioned before, the Android OS is released under an open source license, which is great for the developers, but which prevents Google from providing updates, in a centralized way, to all devices. For this reason there are currently many active devices running different versions of the mobile OS, as we can check in Table 2.2, which shows, in percentage, the fragmentations of active machines running Android OS.

Data in Table 2.2 were collected during a 7-day period ending on December 5, 2016, by Google. Any versions with less than 0.1% distribution are not shown [3].

Table 2.2: Android OS versions fragmentation

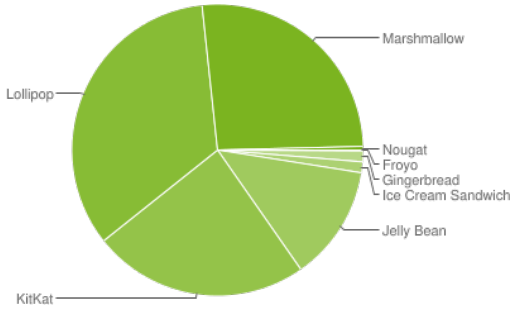


Figure 2.2: Android OS fragmentation chart

Version	API Level	Distribution
2.2	8	0.1%
2.3.3 - 2.3.7	10	1.2%
4.0.3 - 4.0.4	15	1.2%
4.1.x	16	4.5%
4.2.x	17	6.4%
4.3	18	1.9%
4.4	19	24.0%
5.0	21	10.8%
5.1	22	23.2%
6.0	23	26.3%
7.0	24	0.4%

2.1.2 Structure

Android is an operating system based on the Linux kernel. The project responsible for developing the Android system is called the *Android Open Source Project (AOSP)* and it lead by Google.

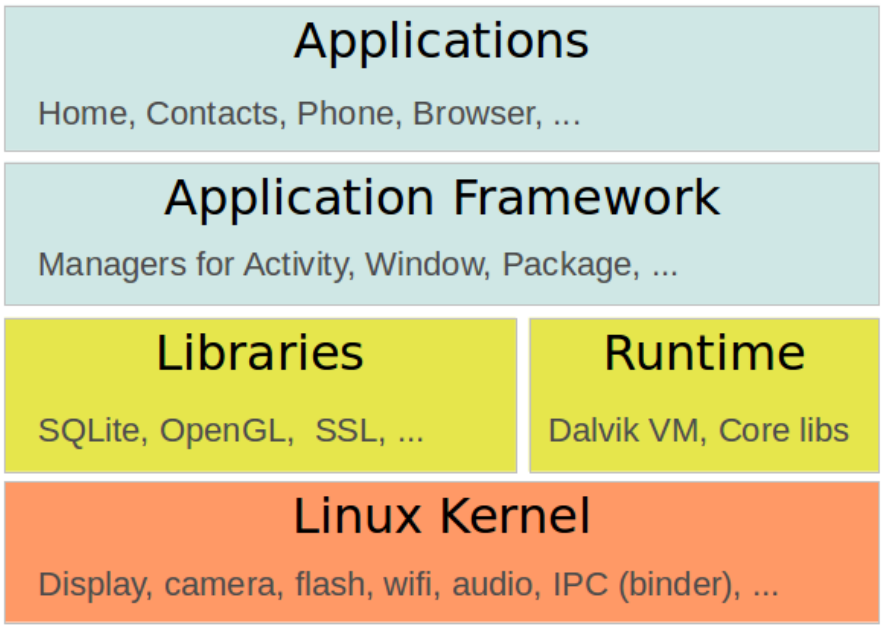


Figure 2.3: Android OS 4 layers

The OS can be divided into the four layers as depicted in Figure 2.3. An Android application developer typically works with the two layers on top to create new Android applications [4].

Linux Kernel

The Linux Kernel is the most flexible operating system that has ever been created. It can be tuned for a wide range of different systems, running on everything from a radio-controlled model helicopter, to a cell phone, to the majority of the largest supercomputers in the world [5]. This is in practice the communication layer for the underlying hardware.

Runtime and Libraries

Runtime is the term used in computer science to designate the software that provides the services necessary for the execution of a program.There are two different *"runtime systems"* which can work with the Android OS:

- *Dalvik VM* is an optimized version for low memory devices of the *Java Virtual Machine (JVM)* used in Android 4.4 and earlier versions. It is stack based and it works by converting, using a *just-in-time (JIT)* approach, each time an application is executed, Android's *bytecode* into machine code.

- *ART (Android Runtime)* introduced with Android 4.4 KitKat. This runtime uses an *AOT (Ahead-of-Time)* approach, with which code is compiled during the installation of an application and then is ready to be executed.

Standard Android libraries are for many common framework functions, like graphic rendering, data storage and web browsing. [4]. This layer contains also standard *java libraries*.

Application Framework

The application framework is the layer that contains the Android components for the application such as activities, fragments, services and so on.

Applications

Applications are pieces of software written in *java code* running on top the other layers.

2.1.3 Application Framework

In this section I want to give some details of the application composition and work flow to better understand the subsequent sections in which I will describe the given problem and the proposed solution.
As briefly described in Section 2.1.2, the Android application framework *("AppFramework")* is the core of the Android *development API*. It contains useful and necessary components to build native apps.
The main components with which each application is composed are the followings.

Intents

Intents are objects that initiate actions from other app components, either within the same program *(explicit intents)* or through another piece of software on the device *(implicit intents)*. According to the official Google's Android for developer documentation, an Intent is a sort of messaging object which can be used to request an action from another application component (eg. activities). There are three fundamental use cases:

- Starting an activity: we will see that activities represent a single screen in Android applications, intents allow to start activities by describing them and carrying any necessary data.
- Starting a service: I will explain later in greater detail that services are components which perform operations in the background. As for the activities, services are initialized through intents and in the same way they describe the service to start and carry any necessary data.
- Delivering a broadcast: broadcast is a message that any app can receive. The system delivers various broadcasts for system events, such as when the system boots up or the device starts charging.

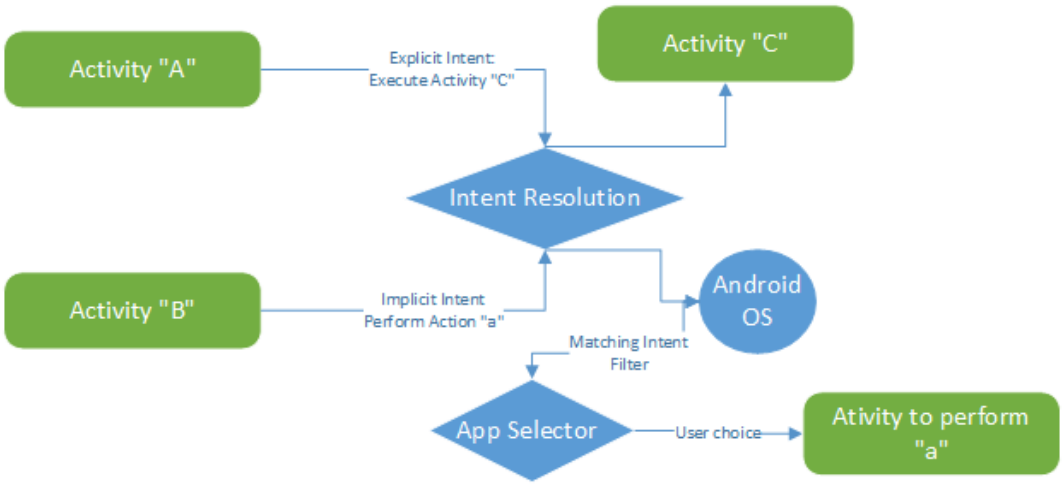


Figure 2.4: Intent resolution mechanism

As already mentioned there are mainly two categories of intents:

- explicit intents, used when it is needed to start component within the same application. As the name implies, explicit intents call components by using a name (the full *class object* name), for example, it is possible to start a new activity in response to a user action or start a service to download a file in the background.
- implicit intents do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it. For example, if you want to show the user a location on a map,

you can use an implicit intent to request that another capable app shows a specified location on a map [6].

Figure 2.4 explains well how an intent is resolved by the OS whether it is implicit or explicit. When an implicit intent needs to be resolved, the OS searches applications which can handle it by means of *intent filters*.A Intent filter specifies the types of intents that an activity, service, or broadcast receiver can respond to. The Android System searches all apps for an intent filter that matches the intent to be resolved. When a match is found, the system starts the matching component, or, if there are more than one, lets the user select the preferred action to be performed.

Activities

Activities are one of the fundamental building blocks of apps on the Android platform. They serve as the entry point for a user's interaction with an app, and are also central to how a user navigates within an app. [7]. An activity is the entry point for interacting with the user. It represents a single screen with a user interface *GUI*: in this way activities are containers for other Android's GUI elements (eg. buttons, textviews,...).

Services

A service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface [8].

Broadcast Receivers

Broadcast Receivers are components that enable the system to deliver events to the app outside of a regular user flow, allowing the app to respond to system-wide broadcast announcements. Because broadcast receivers are another well-defined entry into the app, the system can deliver broadcasts even to applications that are not currently running [8].

2.1.4 Security

As described in Section 2.1.1, Android was born to be a good mobile OS and it is mainly for this reason that the system is designed to protect personal and sensitive data from malicious people.
Like the rest of the system, Android's security model also takes advantages of the security features offered by the Linux kernel. Linux is a *multiuser OS* and its kernel can isolate user data from one another: one user cannot access another user's file unless permission has been explicitly granted. Android takes advantages of this user isolation, considering each application a different user provided with a dedicated *UID (User ID)* [9] Android in fact, is designed for

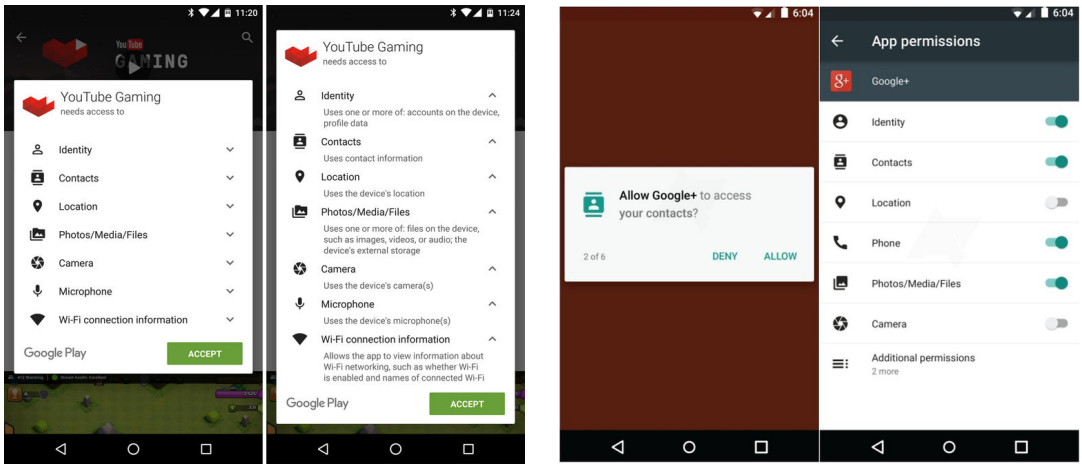
smartphones that are personal devices and do not need, usually, a multi physical user support. The most important security techniques adopted by Android are the followings.

Application Sandboxing

Android automatically assigns a unique *AppID* (Linux UID) when an application is installed and then executed that specific app in a dedicated process as that UID. This technique isolate all the applications at process level and additionally each app has permissions to read/write a specific and dedicated directory.

Permissions

Since applications are sandboxed and do not have the rights to read/write date outside them, it is possible to grant additional rights to android applications by explicitly asking them. Those access rights are called *permissions*. Applications can request permissions by listing them in a configuration file called *android manifest*. In Android 5.1 and earlier versions permissions are inspected and granted at installation time, when the user is alerted with a dialog box in which are listed permissions the application to be installed needs to work properly and when granted cannot be revoked. Starting from android 6.0 permissions are asked the first time that an application needs them, and when are granted, they can be revoked manually in the OS settings for that specific application. Figure 2.5 illustrates these behaviors by presenting some screenshots.



(a) Android 5.1- permission example **(b)** Android 6.0+ permission example

Figure 2.5: Android permission Examples

SeLinux

Security Enhanced Linux, is a *mandatory access control (MAC)* system for the Linux operating system. With a MAC the operating system constrains the ability of a subject or initiator to access or generally perform some sort of operation on an object or target. Starting in Android 4.3, SELinux provides a mandatory

access control (MAC) umbrella over traditional discretionary *access control (DAC)* environments. For instance, software must typically run as the root user account to write to raw block devices. In a traditional DAC-based Linux environment, if the root user becomes compromised, that user can write to every raw block device. However, SELinux can be used to label these devices so the process assigned the root privilege can write only to those specified in the associated policy. In this way, the process cannot overwrite data and system settings outside of the specific raw block device [10].

2.1.5 Connectivity

As already explained previously, many Android design choices are due to the fact that it was thought for mobile devices which must have connectivity to intercommunicate among them.
With the evolution of various wireless communication technologies, Android devices, nowadays, are equipped with different kinds of modulus, the most common are:

- Wi-Fi
- Bluetooth
- NFC
- Cellular Network

The Android Os provide a full library to operate with these technologies and it is possible to integrate in applications the possibility to communicate over these wireless modules. With the *Android connectivity API* data can be send and received in an efficient way.

I have only quickly listed some features and possible issues of my source; to have a complete idea it is possible to read all the official Android documentation in [8].

2.2 Distributed System

In this section I want to give the reader some basics about distributed systems, including technical details and examples to make the proposed solution easier to understand.

2.2.1 Definition

A distributed system is a collection of independent computers that appears to its users as a single coherent system.
This definition has several important aspects. The first one is that a distributed system consists of components (i.e., computers) that are autonomous. A second

aspect is that users (be they people or programs) think they are dealing with a single system. This means that, in one way or an other, the autonomous components need to collaborate [11].
In Figure 2.6 it is possible to see how a distributed system can be structured: at

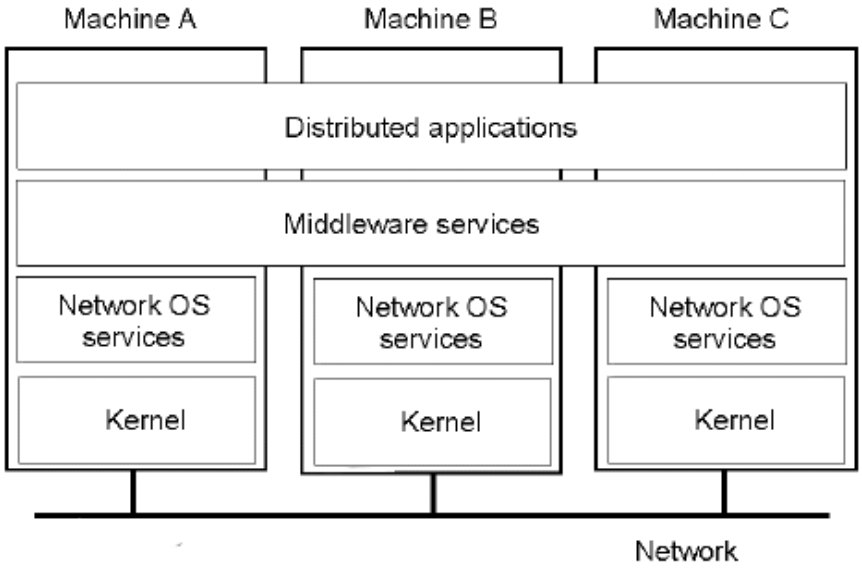


Figure 2.6: Distributed system structure

the top we have the real distributed application, which is the final interface to be used, under which it is possible to have different combinations of services used to make different machines that may use different operating systems communicate. The real magic is done by the layer called *middleware service* in the picture. A middleware in computer science is a set of software which acts as an intermediary between structures and computer programs, allowing them to communicate in spite of the diversity of protocols or running OSs.

2.2.2 Challenges

There are many challenges in the distributed systems field: distributed applications are often really complex and easily exposed to physical and technical failures because of their nature. Major challenges and property to be considered when developing a system of this kind are:

- Heterogeneity, a major challenge because there are many different component to be considered, distributed systems may be developed for example for different hardware, networks, operating systems and programming languages.

- Openness, determines whether a system can be extended and reimplemented in various ways, so distributed systems should use standards as much as possible. Developers should always choose the simplest ways during design and implementation phases.

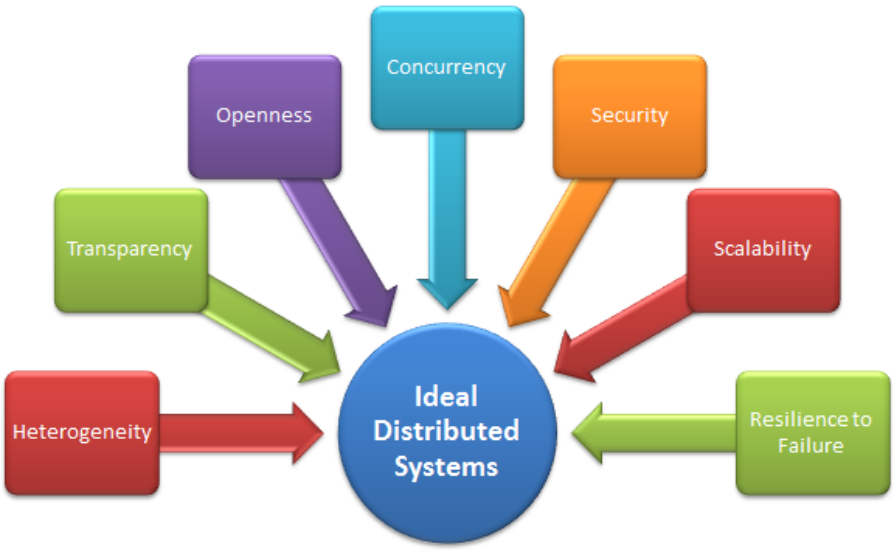


Figure 2.7: Distributed system challenges

- Security, crucial in many areas of computer science and especially in distributed systems, where data are exchanged by several numbers of machines.
- Scalability, the ability to easily increase the size of the system in terms of users/resources and geographic span.
- Failure handling, important because having different components working together to a common goal means that a distributed system can fail in many ways. This raises some issues: it would be wonderful if distributed systems could detect, mask and tolerate failures.
- Concurrency in distributed systems is a matter of fact, access to shared resources (information or services) must be carefully synchronized.
- Transparency levels are listed in Table 2.3

2.2.3 Comunication Model

There are, in distributed system literature, some well known techniques to allow machines, programs and components to communicate. Each of the methods described later exploits the network protocols by acting as a middleware: they use and mask lower layer protocols to provide ready-to-use communication services.

Remote procedure call (RPC)

RPC is a paradigm in which a client process invokes a remotely located procedure (a server process), the remote procedure executes and sends the response back to the client [12]. As described in Figure 2.8, RPC provides the localization

Table 2.3: Transparency levels

Transparency	Description
Access	Hides differences in data representation and how a resource is accessed
Location	Hides where a resource is located
Migration	Hides that a resource may move to another location
Relocation	Hides that a resource may be moved to another location while in use
Replication	Hides that a resource may be shared by several competitive users
Concurrency	Hides that a resource may be shared by several competitive users
Failure	Hides the failure and recovery of a resource
Persistence	Hides whether a (software) resource is in memory or on disk

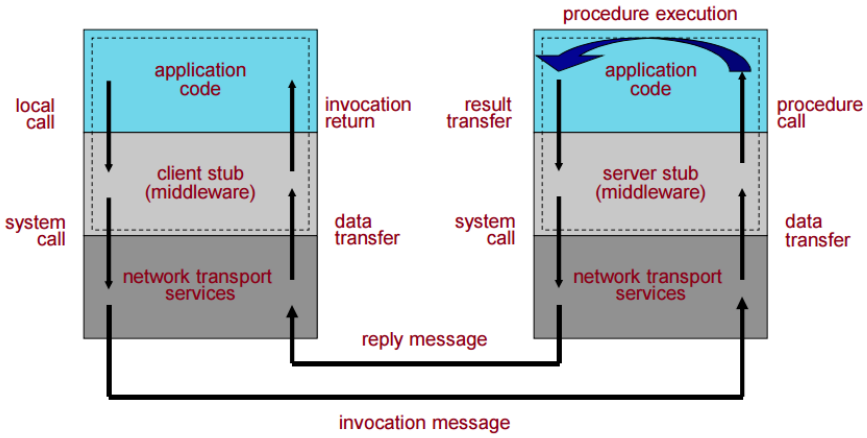


Figure 2.8: RPC in detail

of the code to be executed exploiting the network transport services, creates a message which can be serialized and transferred over a standard network protocol and then provides methods to de-serialize the message and convert it into a standard local procedure call in the receiver machine. Very important in this mechanism is the concept of *ILD* (Interface definition language) which raises the level of abstraction of the service by separating the interface from its implementation: in this way RPC can be language independent by generating automatic translations from IDL to target language.

Remote method invocation (RMI)

RMI exploits the same idea of RPC but with different programming constructs: it is designed to let object oriented (OO) programming languages communicate. Figure 2.9 shows in detail how RMI is supposed to work. Like RPC, RMI uses an IDL which is designed to support OO programming language features, such as

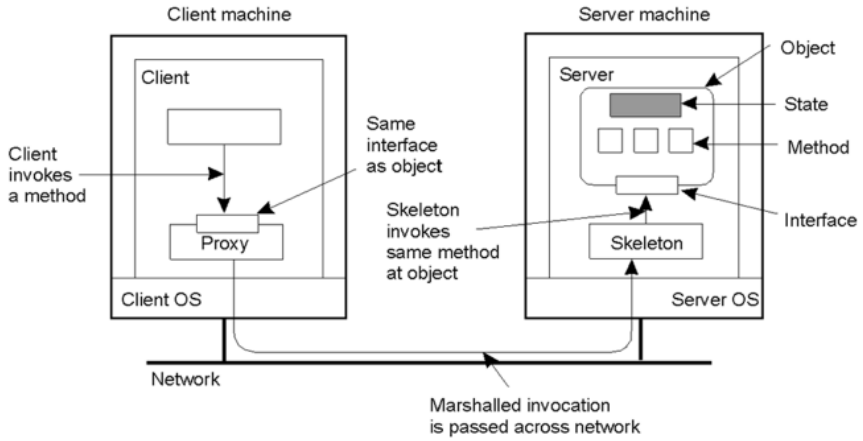


Figure 2.9: RMI in detail

inheritance and exception handling.

Message oriented

Message oriented communication is a style based and centered on the notion of simple messages and events.The most straightforward example of it is *message passing*. Typically message passing is implemented directly on the network sublayers (eg. sockets). Message passing differs from conventional programming where a process, subroutine, or function is directly invoked by name. In Table 2.4

Table 2.4: Comparison between communication models

RPC/RMI	Message Oriented
• natural programming abstractions • point to point communication • designed for synchronous communication • high coupling between the caller and the callee	• centered around the notion of message/event • multipoint support • usually asynchronous • high level of decoupling

are shown the most significant differences between RPC/RMI approach and message communication models. Moreover there are some implementations of message passing at middleware layer like *publish-subscribe* which is further explained in the following subsection.

2.2.4 Architectures

There are actually many different kinds of distributed systems which can be classified by means of their architecture composition.

Client-Server

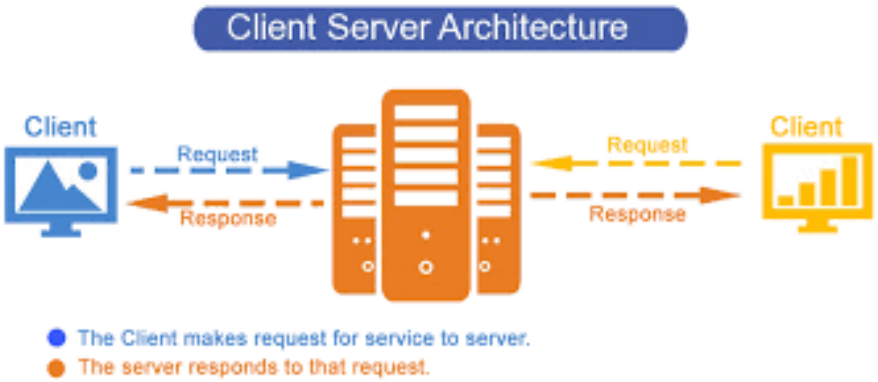


Figure 2.10: Client server architecture

Client-Server is the most common architecture in computer systems, there are many variants depending on the internal division of its components but it has a common separation of duties. Server side components are passive and wait for client invocations. Client computers provide an interface to allow a computer user to request services of the server and to display the results it returns. Servers wait for requests to arrive from clients and then respond to them. Ideally, a server provides a standardized transparent interface to clients so that clients need not be aware of the specifics of the system (i.e., the hardware and software) that is providing the service. The communication adopted by these kinds of systems is message oriented or through RPC.
Figure 2.10 shows the Client-Server architecture structure.

Peer-to-Peer (P2P)

P2P is a fully distributed architecture which in contrast to client-server does not have a centralized service provider. Peers are both clients and servers themselves, P2P promotes sharing of resources and services through direct exchange between peers. Compared to a centralized client-server architecture a P2P net scales better and typically does not have a single point of failure.
Figure 2.11 shows the P2P architecture structure.

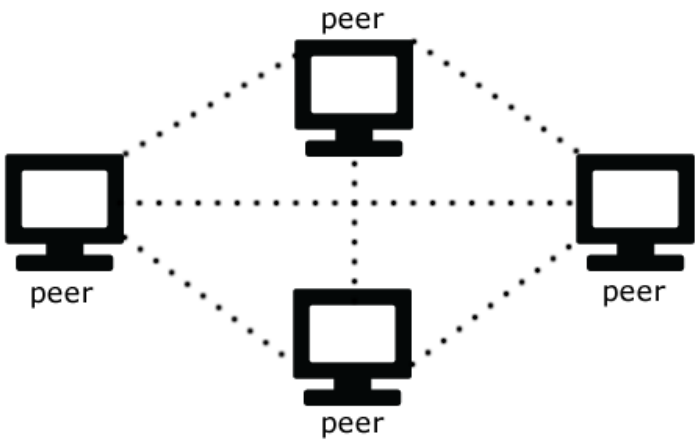


Figure 2.11: P2P architecture

REST style

Representational State Transfer (REST) is a style of architecture based on a set of principles that describe how networked resources are defined and addressed.An application or architecture considered RESTful or REST-style is characterized by:

- state and functionality are divided into distributed resources,
- every resource is uniquely addressable using a uniform and minimal set of commands (typically using HTTP commands of GET, POST, PUT, or DELETE over the Internet),
- the protocol is client/server, stateless, layered, and supports caching.

Event based

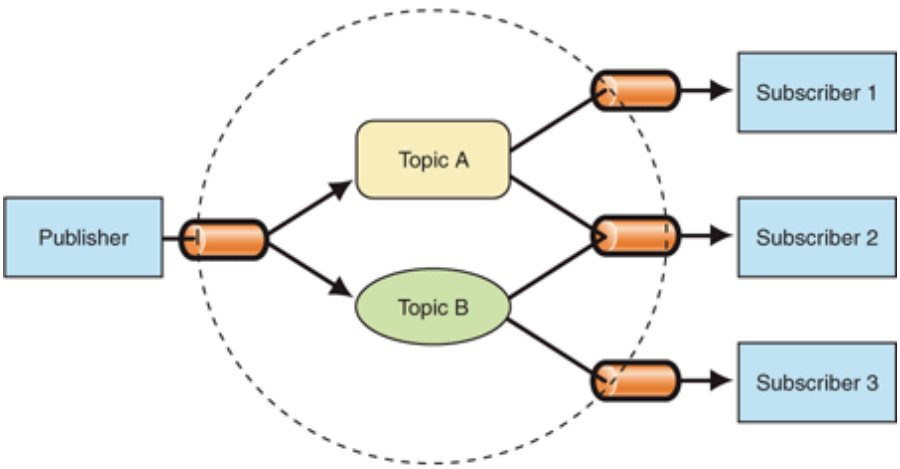


Figure 2.12: Publish-subscribe architecture

Event based is an architecture in which components collaborate by exchanging information about occurring events. In particular, components in the net can *publish* notifications about the events they observe or *subscribe* to events about which they wish to be notified. This architecture can be fully distributed with all the same nodes or can have some semi-centralized nodes which are specialized in computing events or routing messages. Communication is, in this case, purely message based asynchronous and multicast.
Figure 2.12 shows the publish-subscribe architecture structure.

2.2.5 Naming

Naming is one of the major issues when building distributed systems, in fact, it is often impossible to know a priori, the exact addresses and port services of all the components in a distributed network, especially when the system allows dynamic connections and disconnections. It is important therefore, to adopt a naming model or service, to automatize components discovery and connections, when running a distributed system. To understand how naming models and solvers work it is important to introduce some naming concepts in the distributed systems paradigm.
In distributed systems, names are used to identify a wide variety of resources such as computers, hosts, files, services as well as users. Names are usually accessed by an *access point* which is a special entity characterized by an *address*. Addresses are just special names which can be used by communications protocol to connect different machines. For this reason it is important to know access point addresses because otherwise it would be impossible to connect components. Dynamic systems let components change access points frequently, so having *location-independent* names is much more convenient than known static addresses which can change during system execution. *Identifiers* that never change during the lifetime of an entity, are unique, and cannot be exchanged between different entities. In this way, using identifiers, it is possible to split the naming problem in two: mapping a name to the entity and then locating the entity itself. Naming schemes are the solution to the first problem, and the most used ones are as follows:

- *Flat naming*, or unstructured, are simple identifiers represented by random strings of bits.An important property of such a name is that it does not contain any information whatsoever on how to locate the access point of its associated entity [11].

- *Structured naming* are composed from simple, human-readable names, not only file naming,but also host naming on the Internet follows this approach,in fact, flat names are good for machines, but are generally not very convenient for humans to use [11].

- *Attribute based naming* is a way to describe an entity in terms of *(attribute, value)* pairs. Flat and structured names generally provide a unique and location-independent way of referring to entities. Moreover, structured names have been partly designed to provide a human-friendly way to

name entities so that they can be conveniently accessed. In most cases, it is assumed that the name refers to only a single entity. However, location independence and human friendliness are not the only criterion for naming entities [11]. Using attribute based naming is possible to give more information about entities or services to be found.

The solution to the second problem is called *name resolution*. Name resolution is the process of obtaining the address of a valid access point of an entity having its name. Name resolution services highly depends of the naming model adopted by a system.
For sake of brevity, no details of any name resolution system have been reported here, but only basic naming notions to understand author's design choices in solving the subjects of this dissertation.

2.3 Technical Background

This section has the aim to give readers a useful technical background to understand the implemented and proposed solution in the following chapters.

2.3.1 Liquid Computing

The term was coined for Apple's liquid computing feature and refers to a style of work-flow interaction of applications and computing services across multiple devices, such as computers, smartphones, and tablets.
In a liquid computing approach, a person might work on a task on one device, then go to another device that detects the task in progress on the first device and offers to take over that task. In other terms liquid computation is what can be called *ubiquitous computing* which is a model of man-machine interaction in which information elaboration is integrated in everyday objects.

Examples

There are some implementations of this concept in mobile computer science, the most significant are:

- Apple continuity, a system, developed by Apple, with which a user can initiate a task on one device and end the task on another. For example it is possible to answer a call with a computer without using the phone.

- Google chrome and Gmail, developed by Google, allow users to surf the web and to write emails on every available device as if they were using a single device. By registering a Google account, chrome can save the navigation history of the user and show it on any logged device. In the same way Gmail saves emails automatically and for example, it is possible to start writing an email on a desktop pc and then complete and send that email on a smartphone.

- Microsoft One Drive sync is a system, developed by Microsoft to allow users to synchronize files and settings among their devices like desktops, notebooks smartphones and so on.

2.3.2 Java network programming

Since the entire Android development API is written in Java, the whole implemented solution will be in Java.
Java is a known general-purpose computer programming language that is concurrent, class-based, object-oriented [13], and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA) [14], meaning that compiled Java can run on all platforms that support Java without the need for recompilation.
The *Java Development Kit (JDK)* includes many utility libraries, useful for developing any kind of application, I want to focus attention on network programming libraries to be used when developing distributed system using Java. There are, currently many different possibilities among which to choose to let Java software components communicate on the network, the most simple and common are *Sockets* and *Java RMI*.

Sockets

Sockets are abstractions through which an application may send and receive data, in much the same way as an open file handle allows an application to read and write data to stable storage. A socket allows an application to plug in to the network and communicate with other applications that are plugged in to the same network. Information written to the socket by an application on one machine can be read by an application on a different machine and vice versa [15]. Different types of sockets correspond to different underlying protocol suites and different stacks of protocols within a suite. Figure 2.13 shows the working

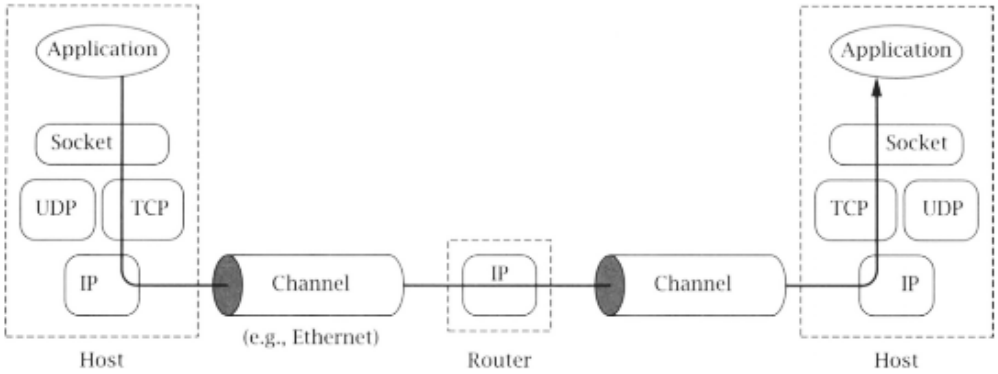


Figure 2.13: TCP/IP sockets

mechanism of a *TCP/IP socket*: the application exploiting the socket communicates using the TCP transport layer and the IP networking layer. In this way it is

possible to read/write packets knowing four variables: socket TCP port number and IP address for the sender and the receiver.
Java provide a network library with which it is very easy to implement a client/server simple application using TCP/IP sockets.

Java RMI

Java RMI is an implementation of *Remote Method Invocation* previously discussed in Section 2.2.3. It represents the best alternative to sockets when building network applications. Java RMI is a complete middleware in itself; in fact, it raises the level of abstraction of the network communication environment.

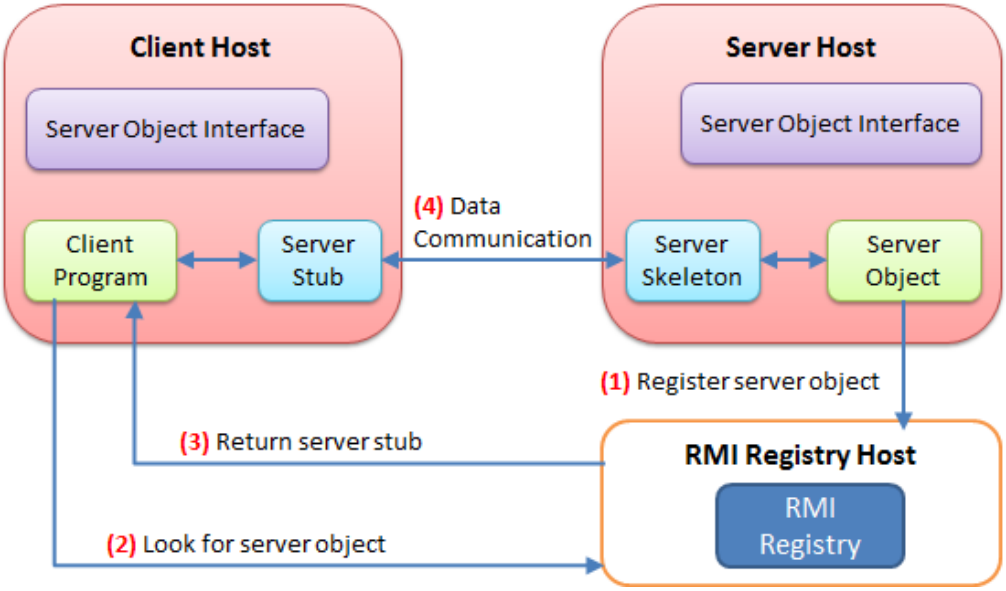


Figure 2.14: Java RMI structure

Remote method invocation allows applications to call object methods located remotely, sharing resources and processing load across systems. Unlike other systems for remote execution which require that only simple data types or defined structures be passed to and from methods, RMI allows any Java object type to be used - even if the client or server has never encountered it before. RMI allows both client and server to dynamically load new object types as required. Figure 2.14 shows the Java RMI complete structure, and in addition it explains its working mechanism, which can be understand by following the numbers in figure.

2.3.3 Zeroconf

Zero-configuration networking or *Zerconf* is a set of technologies that automatically creates a usable computer network based on the TCP/IP Internet paradigm, when computers or network peripherals are interconnected.
The aim of this technology is to let users easily connect to various local network services, without the need of configurations. The architecture of Zeroconf is built

around simplicity. It should be as easy for an end user to connect a printer or locate streamed music as it is for him to turn on light bulb [16].
It is built on three core technologies: automatic assignment of numeric network addresses for networked devices, automatic distribution and resolution of computer hostnames, and automatic location of network services.

Service Discovery and Name Resolution

To the end user, the most important facet of Zeroconf is the ability to easily browse for available services in the network. With Zeroconf you browse for services, not for hardware [16]. Internet protocols use IP addresses for communications, but these are not really human-readable; IPv6 in particular uses very long strings of digits that are not easily entered manually. To address this issue, the internet has long used the Domain Name System (DNS), which allows human-readable names to be associated with IP addresses, and includes code for looking up these names from a hierarchical database system. Users type in common names, like wikipedia.org, which the computer's DNS software looks up in the remote DNS databases, translates to the proper IP address, and then hands off that address to the networking software for further communications [17]. Zeroconf deals with record, find and resolve network services like DNS systems do. It associates the service itself, providing its name and description when registering it, with the machine that provides it, knowing the IP and the used port. When there is the need for a connection, the device that wants to use the service, automatically, by finding it with Zeroconf, knows connection variables , resolved by the protocol.

Implementations

Zeroconf is paradigm and its components can be implemented in many ways and using different technologies, therefore there are many different names to indicate services which provide Zeroconf functionalities.

Apple Bonjour is one of the first implementations of the Zeroconf technology, it is an Apple trademark and its registered name is Rendezvous. Bonjour, also known as zero-configuration networking, enables automatic discovery of devices and services on a local network using industry standard IP protocols. Bonjour makes it easy to discover, publish, and resolve network services with a sophisticated, yet easy-to-use, programming interface that is accessible from Cocoa, Ruby, Python, and other languages [18].

jmDNS is a Java implementation of multi-cast DNS and can be used for service registration and discovery in local area networks. JmDNS library is fully compatible with Apple's Bonjour.Java as a language is not appropriate for low-level network configuration, but it is very useful for service registration and discovery. JmDNS provides easy-to-use pure-Java mDNS implementation that runs on most JDK1.6 compatible VMs [19].

Android NSD implements the DNS-based Service Discovery (DNS-SD) mechanism, which allows your applications to request services by specifying a type of service and the name of a device instance that provides the desired type of service. DNS-SD is supported both on Android and on other mobile platforms. Adding NSD to your Android applications allows users to identify other devices on the local network that support the services the app requests. This is useful for a variety of peer-to-peer applications such as file sharing or multi-player gaming. Android's NSD APIs simplify the effort required for the implementation of such features [20].

2.4 Existing Solutions

In this section I want to analyze some existing solutions, in order to highlight the differences between them and the purpose of this thesis. I want to dwell on both academic and on commercial solutions, in order to have a complete background for the problems faced in this work.

2.4.1 Academic Solutions

From the academic point of view, there are no similar works, which have the purpose of extending a mobile operating systems to add to it distributed OS functionalities. Also in this case, I found some works in which developers, uses multiple mobile devices to reach a common goal but using specific purposes systems, with different solutions in terms of network composition, communication language and data management model.
This shows how, more and more developers, and anyway insiders, are interested in exploiting the growing computing power of these kinds of devices, to perform complex tasks; but at the same time, it is a demonstration of the lack of a framework, easy to use, to speed up the development phase of such systems.

DroidCluster

The most interesting work I analyzed is *DroidCluster* [21], a paper presented at the 32nd International Conference on Distributed Systems Workshop, in 2012. The authors of this work, have performed a complete analysis and a comparison between standard computer clusters and a cluster composed of Android mobile devices. They stated that, if yesterday's clustered workstations could compute climate models or simulate nuclear explosions, clusters of today's smartphones could do as well. Actually they were right, they performed an experiment using a cluster of six Android nodes connected at the same WiFi LAN communicating each other through MPI (message oriented communication), a technology I have already discussed in Section 2.2.3. To perform the test, they decided to run *LINPACK*, which is a software library for performing numerical linear algebra on digital computers. The parallel LINPACK benchmark implementation called HPL (High Performance Linpack) is one of the most used to benchmark and rank supercomputers [22]. However, to enable the installation of the testing tools, they

need the root access. Their results demonstrate that mobile computing platforms today surpass the computational power of workstations from a few years ago. So it is possible to integrate Android devices into a distributed cluster in a way that does not interfere the running Android system and applications.
The work I will develop and describe in the following chapter of this thesis, is supposed to work exactly in the same way of this experiment: I want to exploit the standard Android OS mechanisms and the WiFi connection to extend it and give distributed OS functionalities ready to use. In a sense this work is a feasibility study of what I am going to do in my thesis work. Moreover I do not want to root the system to create the distributed environment, in order to ensure a higher level of security.

2.4.2 Commercial Solutions

Android distributed systems already exist as specific purposes application to be installed on multiple devices, what is different to the aim of this dissertation is that these applications are closed source projects that cannot be reused to build other purpose systems and there is no a coherent framework, library or API to be used to easily build such systems. I want to give some examples pointing out the positive features that have these native distributed Android systems.

Boincoid and HTC Power to Give

Boincoid and HTC Power to Give are Android application whose aim is to exploit Android devices computation powers to contribute to scientific discoveries by doing some tasks. The common idea is to have an Android distributed supercomputer which can handle heavy tasks and compute tons of data for larger purposes.
BOINC is an open-source software platform for computing using volunteered resources [23]. It is a program that lets you donate your idle computer time to science projects. Boincoid is a port of the BOINC platform to the Android operating system. The result is an Android BOINC client that behaves exactly like the original one.
HTC Power To Give is very similar to Boincoid, it is a *CSR (Corporate Social Responsibility)* initiative from HTC that has been jointly developed with Dr. David Anderson at University of California, Berkeley. Using the HTC Power To Give, owners of Android OS smartphones can choose to 'give back' by supporting key research projects around the world. Scientific research often requires a vast amount of processing power for data modeling and analysis. HTC Power To Give, supported by the world's largest single distributed volunteer computing platform BOINC, lets users donate their unused smartphone computing power to science programmes across diverse fields as astronomy, environment, medicine and physics [24].

Plex for Android

Plex platform is a great, maybe the best, media content streaming distributed system platform. It is mainly composed of two components, the media server, and a client which enjoys the contents. Figure 2.15 illustrates the Plex Platform structure.

The Plex Media Server either runs on Windows, macOS, Linux, FreeBSD or

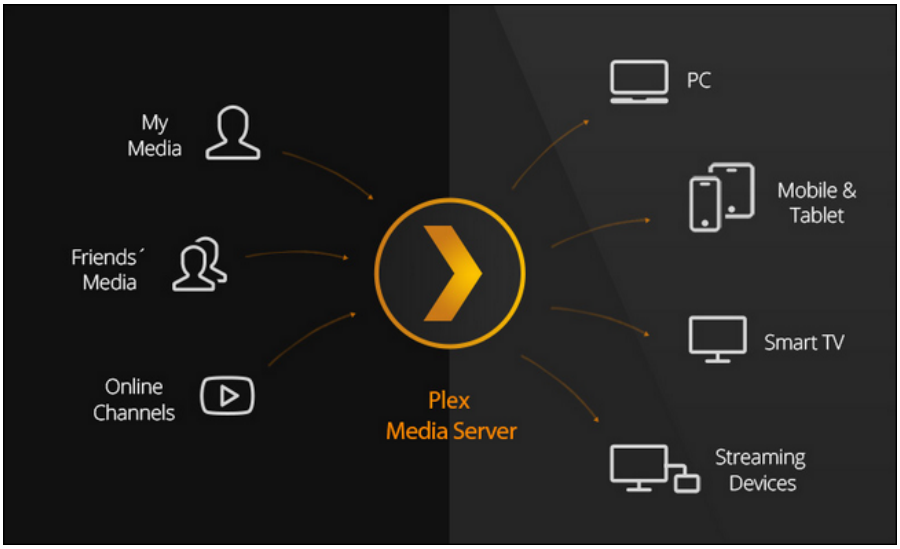


Figure 2.15: Plex Platform

a NAS which organizes audio (music) and visual (photos and videos) content from personal media libraries and streams it to their player counterparts. The players can either be the Plex Apps available for mobile devices, smart TVs, and streaming boxes, or the web UI of the Plex Media Server called Plex Web App, or the old Plex player called Plex Home Theater.

In particular Plex for Android application can connect to the media server to play its contents and in addition, it can search for Plex players in a LAN and send streamed content such as videos, movies or photos, to another player that can also be an Android device. In this way, the Android Plex application client can behave like the Liquid Android system I want to develop. It can send a sort of *Android intent*, to reproduce a media, from one device to another, and then it can send commands such as pause, rewind, forward and so on. The limits of such a system are that it is possible to send, and play, only multimedia contents, and only to devices which have the Plex app activity in the foreground on the device.

Google Home and Cast API

Google itself provides an application to control and send contents from an Android device, to some special devices in home network. *Google Home* is an Android application which can find, set up, manage and control, Google's home devices like the *Google Chromecast*. In this way, is easy to set up and control an Android distributed system in which user can sent multimedia contents and commands to the Google Home devices in the LAN. For these reasons, Google

provides a development library, included in the Android framework, called *Cast API* with which it is easy, for a developer, to build applications that can send multimedia streams to other Google devices specifically built for these purposes. Also in this case, the limitation is the kind of content: only multimedia, and also the type of devices involved which are a limited number of special purposes devices.

DroidMote and Remote control systems

If we consider the possibility of controling remote devices in a LAN, there are actually many different kind of applications that can do this also in an Android environment.
DroidMote is probably the most complete application to remotely control an Android device from another one. It is made up of two parts, the server, to be installed in the device to be controlled, and the client, to be installed in the one which controls. With this application it is possible to control entirely the device running the server component: is it possible to open applications, perform tasks, open system settings and so on.
These kinds of systems are capable of generating local intents in remote devices over a LAN but in a completely different way from how I want to develop the solution to the given problem. In this case, the *controller* is explicitly controlling the remote device as it is using only the *controlled* one. These systems are solutions only to the problem of remote control, they cannot exploit distributed Android devices computation power; in fact, in an environment like this, Android devices are not cooperating to perform tasks but one of them is only controlling another one.

Chapter 3

Problem Analysis

In this chapter I, will provide the in-depth analysis of the specific problems of this work, explaining what the limits are and the constraints the challenge has. The chapter starts with a brief recap, followed by the proper definition of what I faced, while in the last part there is a list of constraints my architecture will have fulfilled in order to have a universal and functional solution.

3.1 Contextualization

As already explained in the introduction, in Section 1.1, this thesis work proposes a solution to let Android applications execute in a distributed environment, created by federating nearby devices. This work focuses on Android because every mobile operating system is different from each other and has proprietary working mechanisms which have to be studied separately. Since there are many more Android devices than any other mobile OS, and Android is an open source software and there is no need to buy development licenses or proprietary hardware or software like for example, with Apple systems, I decided to work with it, even if by studying another mobile OS and implementing the same concepts of my solution it is certainly possible to achieve the same result I got by working only with Android. The Android OS is a centralized operating system designed for a single physical user, to be used on personal mobile devices such as smartphones and tablets. The result of this Google idea is that in contemporary society, there is a wide spread of Android devices, which now have computing capacity comparable to normal desktops and notebooks. Many people have multiple devices which they use separately: typically they use smartphones for calls and work emails and maybe tablets to easily surf the Internet and play games, but what they cannot do is use them together to perform a common task easily. Android, in fact, has not been thought to build a real distributed system, the networking functionalities are designed to exchange messages, and to replace standard personal computers in some tasks as, indeed, sending emails. The result is a non-collaborative, confused cloud of devices, which are connected to the net, but are not really connected among themselves to cooperate. Solutions are often partial or proprietary and closed, even if some useful solutions exist.
The idea is to let Android devices collaborate and cooperate in a *Liquid environ-*

ment like the one presented in Section 2.3.1. The fundamental requirement is the implementation of an android service, able to build and maintain a distributed net of android devices over a *LAN (Local Area Network)*, and then let one, or more devices in that net generate Android intents and distribute them one, or more, of the other devices involved. Thus in this chapter I am considering only Android devices that can be connected in a WiFi LAN.
After this brief recap of what has been said about the Android OS and distributed systems in the state of the art chapter, here I will try to define with more precision the problem I am going to face: what its constraints and its possible goals are.

3.2 Considered Devices

As anticipated above, I am going to take into account only devices that can be somehow connected to a LAN, but as described in Section 2.1.5, Android devices are made to be connected to the Internet and most of them come with an integrated WiFi chip. Another *"little relaxation"* I want to do is linked to the variety of Android OS versions. I want to take into account only devices updated to at least the 4.4 version of the Android OS (API level 19). This is due to the fact that starting from Android KitKat (4.4 version) Google brought some important improvements to the libraries of the framework and in addition, according to Table 2.2, with this choice it is possible to cover 84% of the active Android devices. Having made these clarifications, I will now define the problem.

3.3 Definition

How can we transform a standard mobile OS into a distributed version of it? This is the general question I want to give an answer in this dissertation. As already said the Android OS is a pretty closed system itself, the intent resolution mechanism shows how difficult it is to let various components inside a single device communicate. On the other hand, it is equally true that Android devices are really powerful modern computers and it would be a good thing if somehow it could be possible to have a device able to detect other devices in a LAN send data and task to perform in a transparent way and then get back, if necessary, result or data. Let me be more exact, often in a home environment there are several android devices, with a distributed intent resolution mechanism. It would be possible, for example, to take a photo from one device with the camera of another one, to generate an intent to open a file on a group of devices simultaneously, to play a video remotely and so on, only by generating intents and then sending them to the distributed net. *How can we let multiple android devices act as a single big distributed system?* This is the question that my thesis is trying to answer. My work is a concrete solution, it is about defining and creating a method to distribute android intents from one device to other in a LAN and then let the OS act as usual to manage and resolve them.
So I am trying to let different android devices talk by means of distributing intents using well known architecture: a master component, let me call it *distributed*

intent generator acting as client, and a slave component *distributed intent solver* acting as a server. The two components I will realize will be common Android background services registered on the WiFi LAN. Both of these components will result in android applications so that a single device could be used to control others or to be controlled.
Figure 3.1 presents how such a system should work once the net is up, the numbers near the arrows show the correct order of the flow of messages among the various components involved. The distributed system in the figure is a simplification of what the middleware for distributing intents will do. The architecture will communicate using standard android networking messages that are built on top standard protocols of the ISO/OSI stack.

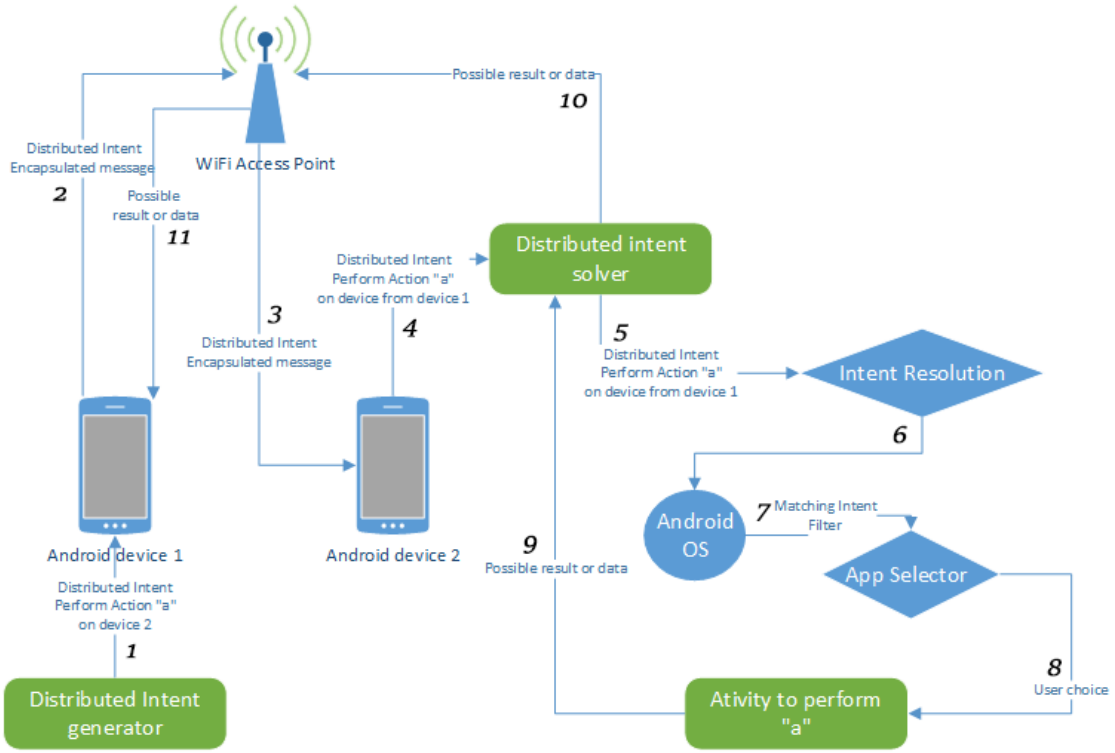


Figure 3.1: Distributed intent resolution

The important point is having a message with a well-defined content: it is what the two parts must write and read, so it has to be clear for machines and must be compliant with all the requests of *M2M (Machine-to-Machine) communication*. This type of communication is a constraint of my work and are explained in the next Section 3.5.Another important point is to let the Android OS work as it is designed for, the main aim of this thesis work is to build a middleware to let distribute native Android intents over the network. This is a new approach to this problem in fact, there are still some Android applications which let the user send stream or data to other devices in a LAN but, with specialized and ad hoc built messages within the same application context, using a mechanism really close to explicit intent resolution. My middleware is supposed to address the problem using a more general approach and a mechanism equal to implicit intent resolutions. What I am doing is creating a system to spread any kind of implicit

intent and letting the OS react as usual to perform the required action. The choice of the type of network to be used in such a system is not even marginal. Android devices are in fact, usually, mobile devices, and for this reason they can be easily moved from one place to another, so the network must take into account that, being able to dynamically react to continuous changes.
The next sections will properly define all the constraints of the given problem and propose a solution that fulfills them all.

3.4 Problem scenarios

As already anticipated with the definition of the problem, the aim of this dissertation is to give the feeling, to users, that they are working with multiple Android devices as if they were one single distributed operating system. I want to analyze some problematic scenarios an then in the next chapter of the thesis provide, if possible a solution to each specific case.

3.4.1 Background middleware

In the best case, the result to be achieved would be a single Android APK, to be installed on devices such as background bunch of services acting as a middleware. Liquid Android services which provide a communication interface can listen to distributed events invocations and react to them automatically. The middleware may intercept local intents, find online devices in the distributed network, let the user select on which of them execute the task and send the intent to the selected remote Android device.
Figure 3.2 shows a possible UML component diagram of the Liquid Android middleware, the scheme points out the interactions between the Liquid Android application (APK) and other possible applications installed on the device. The group of services intercepts the local intent, created by a different application in the same device, lets the user of the system select on which available device executes the task, builds and spreads the distributed intent message on the LAN, and when the message arrives at the other device, the middleware transforms the received message in a local intent to be resolved as usual by the operating system.
I want to provide a simple but concrete example to be more precise, every Android OS version comes with a web browser installed as an APK. When an application needs to open a URL with a browser, it generates an intent to perform such an action. With a middleware as described above it would be possible to click on the URL on a first device and to open the link in a second device, having only installed the Liquid Android APK on both devices.

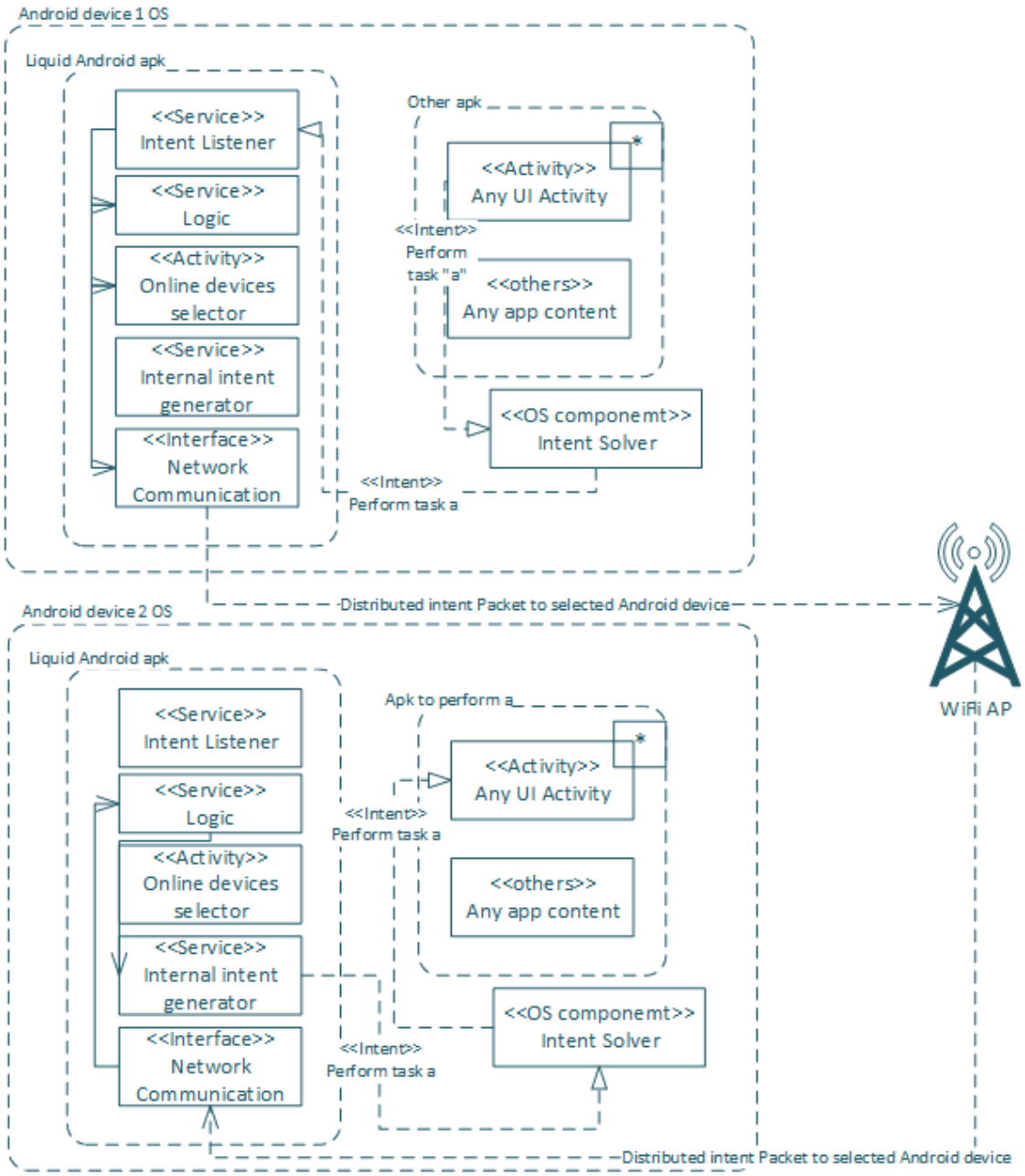


Figure 3.2: Liquid Android working as stand alone middleware APK

3.4.2 Development API

A second interesting scenario is one in which the Liquid Android middleware could become a ready to use *application programming interface (API)*. By abstracting the underlying implementation and only exposing objects or actions the developer needs, an API reduces the cognitive load on a programmer. By developing the middleware as an API is it possible to give to Android programmers a library to implement easily and faster, native Android distributed applications. The API implemented could be integrated during the development of such applications like other Android libraries to generate one single APK containing also Liquid

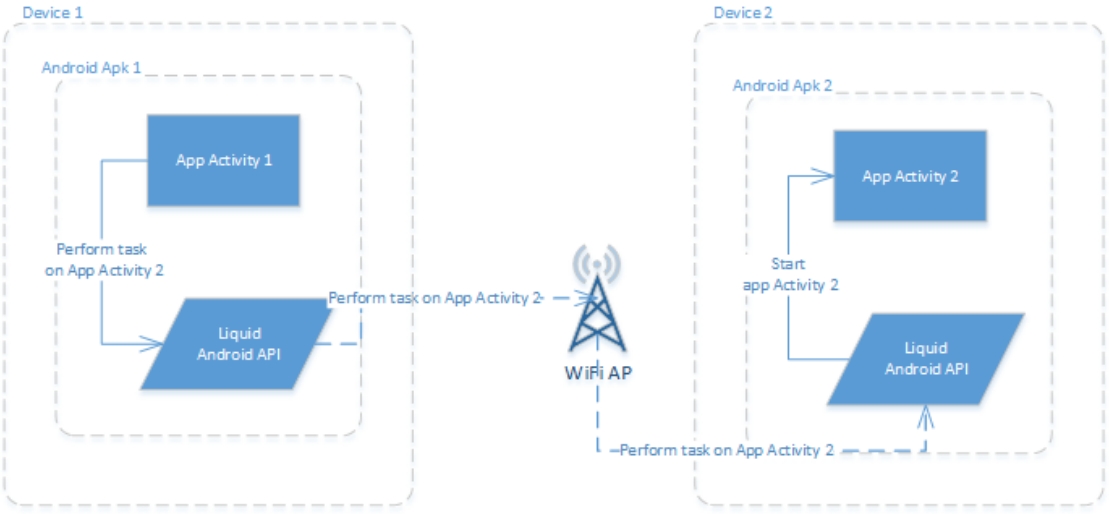


Figure 3.3: Liquid Android API working example

Android components. For these purposes, it is necessary to produce accurate documentation for developers who could use the Liquid Android API.
As already done for the previous case, let me make an example. In Figure 3.3 there is a scheme showing how the middleware could be used to build two different applications with two different packages (APK) including both the Liquid Android API, which lets them communicate by sending Android intent generated by *Android APK 1* and then received by *Android APK 2* installed respectively on two different devices.

3.4.3 Data management

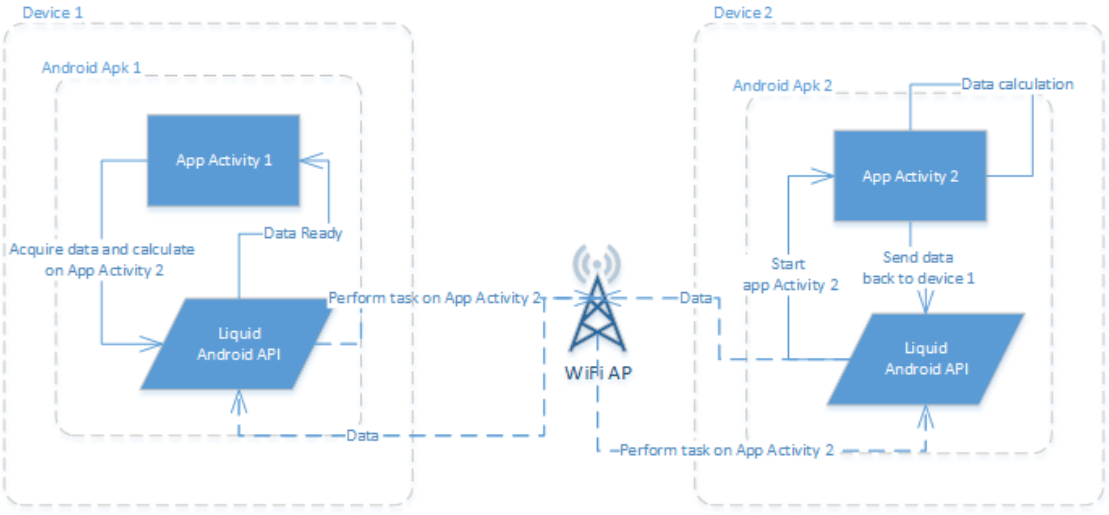


Figure 3.4: Liquid Android API data management example

The last interesting example to study is the data management problem in using such a system. This is a different type of scenario because it involves

both previous scenarios. Having a distributed system always raises the problem of distributed data and data consistency. The middleware to be implemented must consider also the possibility of being used to build distributed Android applications in which data are generated somewhere by one device and then they need to be processed for a result by another one. A simple, but not trivial, example could be the case of a distributed calculator. A device acquires data and sends them to another one to be processed and then asks that device for the results. In the Android environment there is no concept of distributed file system, so data involved in such an application must be considered and efficiently exchanged between devices. For this reason, it is necessary to find a reliable mechanism to transfer data: not only simple data types, such as numbers or strings, but also complex files, such as documents and pictures. Figure 3.4, shows a simple general example in which two devices exchanges data directly using the LAN connection, but to perform this operation, they need to serialize and deserialize any data they want to send through the network.

3.5 Constraints

This section lists a set of constraints for the defined problem, that become requirements that the solution must meet. The section should be divided into two parts, the first for the requirements of the network, the second for the ones of the Android distributed intent generator and solver. The two sections are actually closely related so here I preferred to keep the two parts together, analyzing the entire middleware structure.
The list is as follows:

- *M2M communication:* M2M communication is defined as a communication in which the two interlocutors are not humans. It is a communication completely handled by machines and computers [25]. It can be considered one of the fundamental enabling technologies of this dissertation, it permits an object to communicate without humans being involved. In this type of communication the reader of the content is a computer, in this case Android devices. The content of the messages must be well formed, the middleware must react properly to the event of receiving a distributed intent. So a clear, defined syntax with a well-fixed structure must be set in order to make everything understandable to a computer.

- *Transparent:* As already widely discussed a middleware is where the *magic* happens. The proposed solution is intended to be transparent to the Android OS and let it work as usual in resolving implicit intents whether they are distributed or not. Moreover as discussed in Chapter 2, to be more precise in Table 2.3, a distributed system must be transparent at many levels, in this case the middleware must act as resources manager and efficiently mask resources access and location.

- *Lightweight:* Another constraint to my system is the fact that whatever system I choose to be the solution, it must be lightweight. This is needed

because my system will work on a WiFi LAN. Messages must be encapsulated, serialized from one device and transferred to another one to be deserialized and analyzed to be executed. Messages must be as easy as possible because they are very frequent in such a system.

- *Modular:* The implementation of the solution must be modular; this is due to the fact that this middleware is intended to be used as it is but also to implement easily other kinds of native Android distributed system applications. Having a modular structure facilitates the specialization of its components and makes all the middleware more readable and easy to use. In this way, Liquid Android can be the substructure of other works.

- *Extensible:* the implemented solution must meet canonical programming principles. Extendability is one of the most important properties to take into account when building a computer system, especially when developing a middleware. Liquid Android modules have to be extensible to be improved or adapted to different purposes.

- *Secure:* Liquid Android middleware must meet standard Android security design principles as described in Section 2.1.4. The implementation must not exceed the limits imposed by the OS; I do not want to break the Android permission scheme and authorization model by *rooting* the operating system, a process with which it is possible to perform actions as the administrator in the Android environment. Rooting Android devices lets applications overcome the boundaries of standard applications, by letting them read and write data from all the OS. Moreover the middleware operates on mobile devices which usually contains and can manage many sensitive and personal data, communications between these devices must be as secure as possible to limit security threats.

- *Consistent:* Data and accessed resources involved in the system must meet consistency requirements. When developing distributed systems, consistency is one of the main issues. The implemented solution must take into account data produced during the use of the system and make them consistent according to a chosen consistency model.

- *Scalable:* the system to be implemented does not have a fixed number of devices involved. The chosen network architecture must be able to react according to the changes. Android devices are free to join or leave the network at any time, and the system should be able to detect and maintain a dynamic network. Scalability is, in fact, the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged in order to accommodate that growth [26].

- *Concurrent:* another important aspect of distributed systems is concurrency. Concurrency is the decomposability property of a program, algorithm, or problem into order-independent or partially-ordered components or units [27]. The implementation of the services must ensure this property

to the system. The middleware has to have the capability to handle different requests at the same time and execute tasks in more than one device simultaneously.

The listed requirements, as already told are, sometimes, general, in the sense that they have to be respected for the final product: a global and complete structure that starts from the construction of the network architecture arrives at the user's interaction activities on Android devices. This is because the problem I am facing is very big and complex, and it is transversal to the existing technologies, so the whole system must work properly. Keeping in mind what I have just stated, some of these constraints become fundamental requirements that my system must meet. My work has to be clear for developers to be used for further implementations of native Android distributed systems, but even if it can be less clear to an average user it, must be usable to those wishing to try distributed intents with their own devices in a home LAN.

In the next chapter I will present my idea, the *Liquid Android* middleware, the so-called solution to the given problem, explaining what I have done, my considerations about the situation here faced.

Chapter 4

Proposed Solution

In this chapter I will report the development of the solution step by step, with full use cases. I decided to divide the chapter in three main sections: the first explains the choice of the network, the architecture, the naming service, etc. It lays the foundations for the second part: the definition of the Liquid Android middleware, or better the structure of what will do the magic: intercept, encapsulate, spread, and generate distributed intents in the network so made. The two parts are closely related, therefore their relationship was taken into account when I made my choice.
The real implementation of the valid solution is left for the next chapter.

4.1 General Idea

To better explain what I consider the solution for my work it is important to understand the playground to it. As said in the previous Chapter, 3, I am trying to extend the Android operating system by adding some functionalities to make it similar to a distributed OS, without the need for rooting it or change its standard working mechanism and components, staying in the 7th layer of the ISO/OSI stack , the application layer. Using already developed and operating tools, and respecting all the above listed constraints, I am going to make mobile devices in a LAN network communicate and cooperate like they were using a single coherent distributed operating system. In order to understand what is needed and how it is possible to solve the problem, it is fundamental to understand the type of stack and the network structure we have to face, and standard Android working principle, in particular the intent resolution mechanism already described in Figure 2.4.
Only by a clear idea of the problem and its structure can the best possible solution be found. In particular it is possible to divide the main problem in some sub-problems, which can be understood as general steps in doing similar works of extending a mobile OS to become a distributed OS:

- Network architecture, that is the structure and the classification of the nodes involved in the distributed system. As previously said, it has to be as reliable as possible and allow dynamic connection due to the fact that

nodes are mobile devices and can be easily moved in and out the network range.

- Communication model, that is the way in which involved actors perform the communication. It has to be compliant to M2M, and possibly to H2M communication, and as lightweight as possible to allow fast exchange of messages and data between the network nodes.

- Data model, as discussed before in Chapter 2, when building distributed systems, it is also important to guarantee that data are managed correctly by adopting a consistency policy.

It is also necessary to identity the main actor involved in the problem, they are mainly two:

- Server application: it is the main actor of this thesis work, it must be an Android application which, once installed on a compatible Android device, can receive, resolve and forward Android intents. It contains the logic and the controllers needed to handle the network structure, find other devices in the network, send and receive messages. It is responsible for resolving all the three sub-problems described above. The server application also, has the double function of receiving a message from the network and translate it in a local intent to be resolved by the Android operating system, but also it can act as a client by forwarding a received intent from a third party client application to another server in the net, by encapsulating the intent in a network message.

- Clients: can be applications developed in several ways, they are those which ask the so-called servers to complete the task for them. In this case, clients could be any kind of third party Android application installed in the device, also running the server component, generating implicit Android intents that need to be resolved by the OS.

Once the main actors of the problem I am facing have been defined the next step is to understand how they can interact and communicate. As previously described, defining the problematic scenarios in Chapter 3, the Liquid Android middleware in the best case would be a system service which users can control to distribute intents in the local network by using the WiFi chip of the devices. Figure 4.1 shows exactly how the middleware is supposed to work. In the figure, the two devices are both supposed to be connected to the same local network *(they are under the same WiFi access point)* and they are executing any standard Android OS version starting from 4.4 KitKat *(API level 19)*. The so-called, *Device 1*, in the picture, is executing a third party application activity *(a client)* which contains a valid clickable URL link. By clicking a link, typically, Android applications generate an implicit intent asking the OS to open and show the page linked by the URL. Usually, in the absence of other applications capable of solving this kind of implicit intent, the process ends with the opening of a browser in the same device, which opens the URL in one of its activities. In this case, the Liquid Android application server, installed on both devices, should to tell the OS that

Figure 4.1: Liquid Android working example

it can handle that implicit intent, find the other devices in the net, in this case the so called *Device 2*, let the user choose with which device to complete the task, convert the intent in a network message and send it to that device. Once the message arrives at the *Device 2* the Liquid Android middleware server application is responsible for translating again the received message into the starting implicit intent sent by the *Device 1* and for starting the very same resolution mechanism for that intent by its own Android operating system, which should end by the opening of a browser activity to view the page.
It is clear that a solution for this scenario would also be, once implemented, a solution for the second problematic scenario proposed in Section 3.4.2. In fact we can consider the development of an API to build native Android distributed systems as a sub-problem of the first one, already described above and in Section 3.4.1. The implemented version of the general solution could also be used as a library to implement special purposes similar systems by simply extending my framework and including its implemented Java classes in other Android applications projects. For these reasons, in developing the solution I will try to make it as clear as possible, and to parametrize as far as possible the settings variable of my framework to make it easily extensible and ready to use by other Android developers.
I would now like to list the goals that my work has to match, in order to be a valid proposal for solving the given problem. These goals are not to be intended as set in stone, they are the general motivation that leads to construct a prototype of the proposed software architecture. According to my thoughts during the development, it is possible to identify the following goals:

- The middleware must work without any proprietary application: it has to interface itself to the upper layer without installing any other application of any vendor in the owned device. It must be completely neutral to the market, it must work with any version of the Android operating system

starting from the API level 19, also with Android customized versions developed by device maker like Samsung, LG, Huawei and any other brand. It is the fundamental requirement to create heterogeneous applications and to separate the various closed solutions of today and an open solution for everyone in the future.

- The middleware has to simplify the life of the developer, he should not have to worry too much about the substrates, he should be able to prototype fast. The developer should see my framework as a help for his work. The idea is to provide a ready-to-use service, with which it is possible to create new application by exploiting it.

- The middleware should offer the user the possibility to access directly to other devices in the network without the need to configure anything. Users, once the middleware is installed should use its functionalities of receiving/forwarding intents in a transparent way, in the same way they use other applications and with the same mechanism they learned by using the standard Android operating system.

The next sections contain all the steps necessary to have a full working system. Firstly I would solve the, what I would call the *general theoretical problem* by dividing it as discussed above and providing the solution for each of them, also taking into account the data management scenario. Then I would like to present the structure of the development API, while the actual implementation of the working Liquid Android application is left for the next chapter with some working tests and a deep component analysis.

4.2 Proposed Solution

In this section I will perform an in-depth analysis of the possible solution to the given problem: how I can extend the Android OS providing it distributed functionalities.

4.2.1 Network Architecture

The first step while creating a distributed system, is to define the networking architecture, in particular I must define the kind of nodes involved in the system and the way in which they interact, what they can do, which operation they can perform and in which way. As mentioned earlier the network architecture of the system must fulfill the following requirements:

- *dynamicity*, it must allow any device to perform the dynamic connection, and also disconnection, to the distributed system at any time, since the nodes of the network are mainly mobile devices and they can be moved easily. Nodes can *JOIN* and *LEAVE* every time, and the network must accommodate them automatically.

- *simplicity*, the network must be as simple as possible, it should not need any particular configuration on the nodes to *JOIN*. Any node should perform other nodes *DISCOVERY* in the network in a easy way without the need to know them a priori.
- *reliability and security*, are important non-functional requirements in such a system. I want to make the network reliable and secure as much as possible by the adoption of standard software engineering techniques.

The network architecture that fits better all the requirements listed above is certainly a P2P network. As seen in Section 2.2.4, in peer-to-peer networks there is not a clear distinction between clients and servers; in fact a peer-to-peer network is designed around the notion of equal peer nodes simultaneously functioning as both "clients" and "servers" to the other nodes on the network.

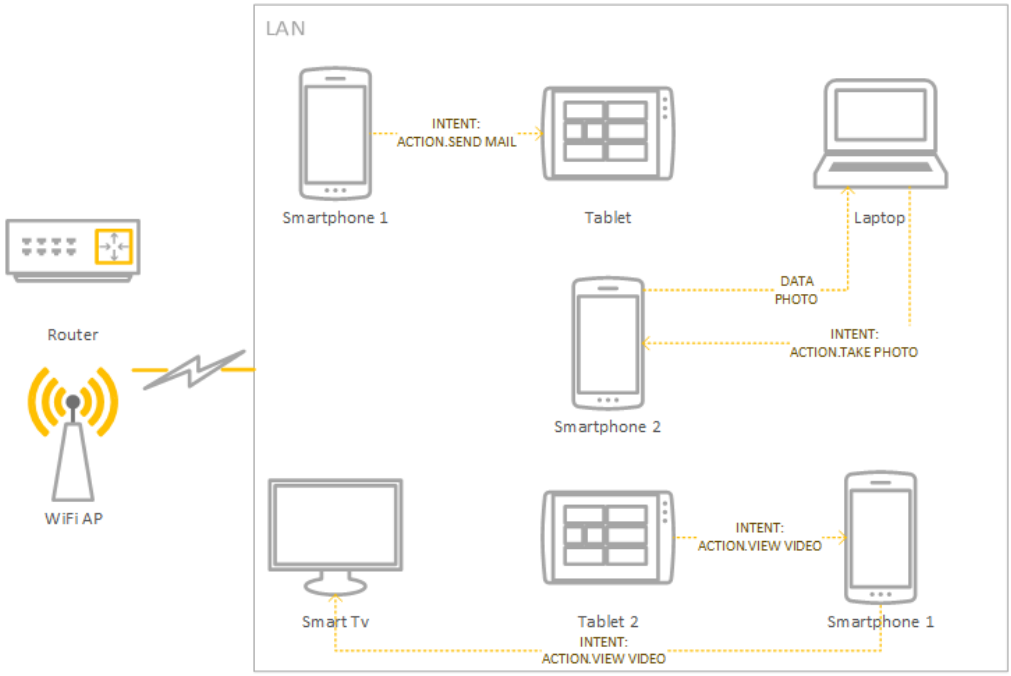


Figure 4.2: P2P Liquid Android network example

I think that an *unstructured P2P architecture* is the best choice for my system, due to the fact that it does not impose a particular structure on the overlay network by design and data is still exchanged directly over the underlying TCP/IP network, so at the application layer peers are able to communicate with each other directly, via the logical overlay links. In Figure 4.2 there is an example of my architecture in which different devices are in range under the same local network and they can communicate sending intents and data.
To obtain this kind of structure, and let it change dynamically depending on the devices in range, I need to equip the devices with a *network service*. In computer networking, a network service is an application running at the network application layer and above, that provides data storage, manipulation, presentation, communication or other capability which, in this case will be used in combination with a peer-to-peer architecture based on the application layer network

protocols. Different services use different packet transmission techniques. In general, packets that must get through in the correct order, without loss, use TCP, whereas real time services where later packets are more important than older packets use UDP. For example, file transfer requires complete accuracy and so is normally done using TCP, and audio conferencing is frequently done via UDP, where momentary glitches may not be noticed. In this case, I will adopt the TCP transport layer because I need to transfer packets in reliable way, as much as possible, and also avoid network congestions, the UDP protocol in fact lacks built-in network congestion avoidance while TCP has it.
To fulfill the above listed requirements of dynamicity and simplicity the right approach should be the Zeroconf one, already discussed in Section 2.3.3. Using a Zeroconf implementation to register and discover the service in the LAN will let the node connect dynamically and in a easy way to the distributed system. The three main operations a node can perform are:

- *JOIN*, to join the system a node must activate the network service, it must provide an internal endpoint for sending or receiving data in the computer network. The best abstraction to do that is to open a socket, as seen in Section 2.3.2. In particular, a TCP socket is characterized by two main parameters: the *IP* and the service *PORT*. In my system the TCP socket is a great choice because, as already stated, it is a network abstraction and it can be used to let heterogeneous devices communicate and can be implemented in several different ways using basically any development language. By knowing the couple of variable of the network service another device can send message, streams and so on, to it through the TCP socket. Since mobile devices frequently change their connection variables, because of their nature of being easily moved from one place to another, we need a system that can identify them dynamically. The name of the service and the chosen transport layer can be established once and for all and they can never change. In this environment Zeroconf provides service registration and discovery. By registering the service name, port, and transport layer, the node can be found in the LAN by other nodes looking for that kind of service. In this way, nodes do not need to know, a priori, the two variables, IP and PORT of any node in the network, to communicate with each other, they indeed, only need to know the service name and the chosen transport layer to find other nodes in the network.

Listing 4.1: Zerconf registration example

```
1 public void registerService(int port) {
2     // Create the NsdServiceInfo object, and
        populate it.
3     NsdServiceInfo serviceInfo  = new NsdServiceInfo
        ();
4
5     // The name is subject to change based on
        conflicts
6     // with other services advertised on the same
        network.
```

```
7       serviceInfo.setServiceName("LiquidAndroid");
8       serviceInfo.setServiceType("liquid._tcp");
9       serviceInfo.setPort(port);
10      ....
11  }
```

The snippet of code 4.1 is an example of how in Android it is possible to register a service using the Zeroconf approach. Once registered, the service Zeroconf provides name resolution functionalities to discover other nodes and then connect to them to start the communication.

- *LEAVE*, what I want is that a node can decide whether to join or leave the network at any time. To leave the network, a node should only unregister the service registered using Zeroconf and then close the socket to avoid accidental or malicious connections.

- *SEARCH*, since the network is dynamic, it is necessary to determine how the nodes, once the service has registered, can search for and find other nodes. As already mentioned, describing the *JOIN* operation Zeroconf also provides the network service discovery and the naming resolution mechanism. In this case Zeroconf uses a *Domain Name System (DNS) based Service Discovery*, the so called *DNS-DS*. DNS-SD allows clients to discover a named list of service instances, given a service type, and to resolve those services to *hostnames* using standard DNS queries. The specification is compatible with existing *unicast DNS* server and client software, but works equally well with *multicast DNS (mDNS)* in a zero-configuration environment. Each service instance is described using a *DNS SRV* and *DNS TXT* record. A client discovers the list of available instances for a given service type by querying the *DNS PTR* record of that service type's name; the server returns zero or more names of the form "<Service>.<Domain>", each corresponding to a SRV/TXT record pair. The SRV record resolves to the domain name providing the instance, while the TXT can contain service-specific configuration parameter. Once the resolution process is completed the node which started it to find other nodes in the network, knows any couple of IP/PORT of their open sockets and can connect to them to exchange messages or transfer data.

Given the network structure security is obviously a non functional requirement in building my system, but since my entire middleware, once connected, will allow to send and execute any kind of task, in the form of implicit intents, to each device involved I need to make some considerations about the security of such a system. The security of my system is highly influenced by the underlying physical network, if the LAN access is secured and protected, my system will indeed be, secure enough. The network should be protected using a firewall and the service ports used by the device involved in the system should not be accessible from the outside of the LAN. Moreover the WiFi should be protected using a strong password and a secure and updated access protocol like the *WPA2*. Furthermore, I want users of my system to be aware of the fact that it is running,

so once the service is activated users will be alerted by a notification in the appropriate Android notification area.

4.2.2 Communication Model

Once the network is up, the second step is ti let the device communicate to cooperate giving, to the final users, the feeling that they are working with a single big distributed operating system. To achieve this goal, as already explained several times, my system is supposed to intercept implicit intents, let the user select a target device, or a group of them, to perform the task, send it and once arrived, perform it in the selected device or group devices. With TCP sockets it very easy to exchange messages between networked devices, so it is equally easy to understand why I have chosen a message oriented structure as the communication model, briefly presented in Section 2.2.3, for my system. Furthermore, as already stated in Table 2.4 such a model gives me more freedom of choice regarding certain characteristics that the system must have. I want the communication is:

- *concurrent*, in the original Android OS, when an intent resolution mechanism is started by any other component of the system, for example an application activity, the OS stops its execution to perform, in foreground, the task the intent contains. I want to leave unchanged this kind of mechanism in my system, so when a, what i would call, *distributed intent* arrives from the socket, the operating system must treat it exactly as if it were a *local intent* and execute it in the same way it does with other intents. The arrival of a new distributed intent from another device is the event which triggers the standard OS resolution technique by suspending any other operation. I want the network to listen and accept messages with distributed intents to solve at any moment, in a fully concurrent way, so the last intent is triggered the first it is resolved and executed in a *LIFO*, last in first out, way like in a stack structure.

- *asynchronous*, the standard intent resolution mechanism, like many other in the Android operating system, is mainly asynchronous. When an implicit intent is triggered, it will be performed without knowing, a priori, what application will execute the task and how. To be more clear, for example when an application asks the OS to open a map of a place, the application which started the process does not need to know if the system completed the task and with which results. I want my system not to need to change this behavior, if an operation need to be performed the intent is triggered and the OS reacts as usual without the need to send back acknowledgment messages ACKs. When it is necessary to synchronize the communication among the various components involved when an implicit intent is triggered, Android provides a mechanism, using the system function *startActivityForResult(intent)*, to send the result back to the caller. I intend to keep the same mechanism also when dealing with distributed intents, but I want to maintain an asynchronous approach also in this case, the caller

could continue its execution without waiting for the results and then, once done, receive back the response, in the form of another distributed intent, from the called. I will give further details of this case when presenting the kind of messages that can be sent using my system and the data model solution part.

Now the difficult part is to establish how intents can be sent through the sockets and which kind of messages my system can send and then handle to perform various kinds of tasks.
The next step, therefore, is to find a way to send the intents from a device to another without losing information and once they have arrived to be executed like they were local intents. To do this job it is necessary to analyze in depth what an intent is in Android and what it can contain.
As showed in Chapter 2 in Section 2.1.3, intents are the way in which standard Android framework components communicate among each other, and they also represent the abstraction of actions to be performed by Android applications. In the Android developer framework, an intent is implemented as a Java Class containing the information and data to perform the task. An intent is an abstract description of an operation to be performed. It can be used in various ways: with *startActivity* to launch an Activity, with *broadcastIntent* to send it to any interested *BroadcastReceiver* components, and with *startService(Intent)* or *bindService(Intent, ServiceConnection, int)* to communicate with a background Service. However its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed [28].
Every implicit intent is characterized, mainly, by an *ACTION*, *DATA* and a bundle of *EXTRAS*. Table 4.1 explains in depth what an intent could contain, of course there are many types of action, data, categories and extras that are not reported here for reasons of brevity but that can be found in [28]. The real problem, for the purposes of my system, is that the Android intent Java Class cannot be serialized, automatically and sent through the socket as it is. In computer science, serialization is the process of translating data structures or object state into a format that can be stored, or transmitted across a network connection link, and reconstructed later in the same or another computer environment.
When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object [29]. In a Java environment, like the Android one, Java Classes that implements the *Serializable interface* can be automatically serialized, by the environment itself, and sent through a socket and then automatically deserialized to reconstruct the original sent object by the receiver, but as already told the intent Class does not implement this functionality. I need therefore to find a way to do this process in a fully functional and also efficient way. There are, actually, many alternatives to accomplish this objective. For example I could generate a new Java Object, containing all the intent attributes, that implements the Java Serializable interface, convert intents to this new Object and then let Java perform its automatic serialization/serialization. As an alternative I could generate some kind of semi-structured data, using a well know, what I would call, *container*

Table 4.1: Intent Structure

Attribute	Description	Examples
ACTION	The general action to be performed.	ACTION_VIEW ACTION_EDIT
DATA	The data to operate on, expressed as a Uri.	content://contacts/people/1 tel:123
CATEGORY	Gives additional information about the action to execute.	CATEGORY_LAUNCHER CATEGORY_ALTERNATIVE
TYPE	Specifies an explicit type (a MIME type) of the intent data. Normally the type is inferred from the data itself.	type */*
EXTRAS	This is a Bundle of any additional information. This can be used to provide extended information to the component.	EXTRA_TEXT EXTRA_TITLE

language, such as JSON or XML, which can be easily sent, as a string, over the socket and then parsed to rebuild the original Java Object.
Since the effort required to develop one of the two solutions, presented above as examples, is practically the same, I decided to opt for the second alternative, using a JSON structure, because it has substantial advantages.
JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, it is completely language independent. These properties make JSON an ideal data-interchange language [30].
It is based mainly on two types of structures:

- Key/Value pair set: it can be considered an object of an Object Oriented programming language,
- Collection of elements: it can be considered an array of Objects.

By converting any possible implicit intent in a *JSON-intent* the system complies with the following properties:

- *M2M communication*, and also *M2H communication*, as already stated JSON are really easy to read and understand by human and also by machines,
- *lightweight messages* and also *small overhead*, it is well known that Serialization in Java produces a big overhead and serialization/deserialization to be

more general lacks efficiency. JSON are merely *well formatted Strings* which can be sent unaltered on the socket without the need be further converted. The sender sends the JSON as a string and the receiver gets exactly that string.

- *freedom*, JSON is optimized to be used with a great variety of programming languages, the creation of a *JSON-intent* opens up various possibilities for further development: using the correct syntax, any device, even non-Android, could be a client of my system. In this way, it would be possible to send original Android intents to any Android device, with my service installed and active, from a Client running in different OS and using different programming languages. It is possible, in fact, to use Zeroconf to find the service in many other environments and to connect to the socket to send JSON-intents to the connected Android devices.

To obtain a general solution to convert any possible Android intent into a JSON-intent, and viceversa, I need to define the syntax of new object which must fit as much as possible to the structure of the Android intent Class. In fact, it would be easy to convert the major fields that characterize a given intent that are mostly correctly formatted strings, if it were not for the fact that the bundle of extras can contain different types of structured data.

JSON-Intent sintax

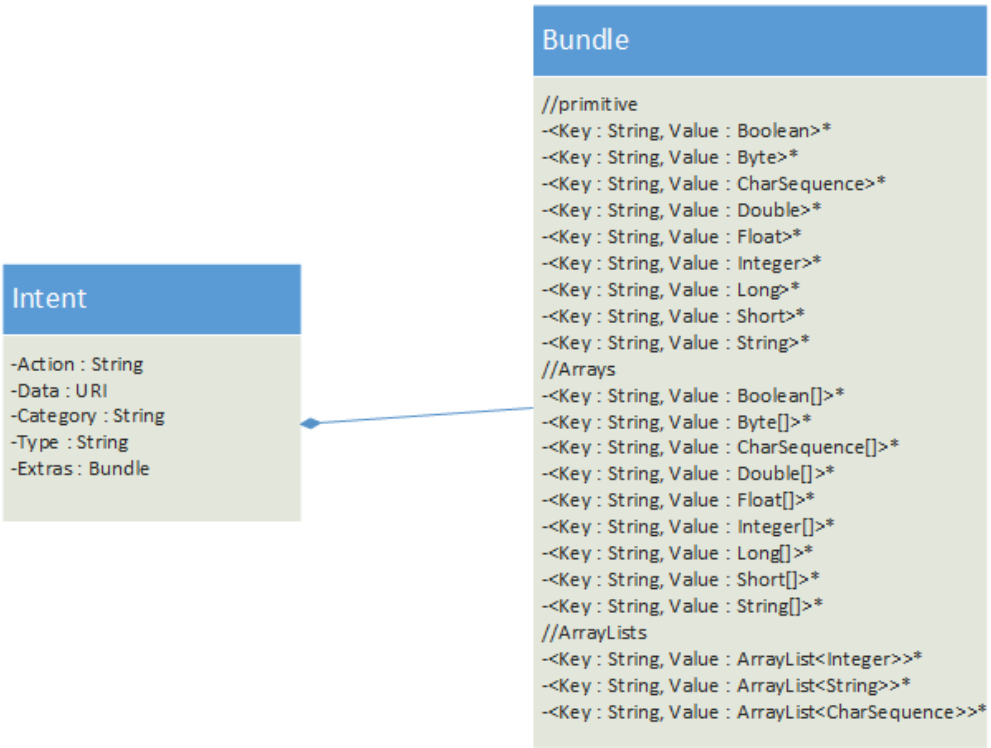


Figure 4.3: UML structure for an Intent

In this subsection, I am going to define step by step the syntax of my solution, explaining with listings and images all the elements. An important and

remarkable reflection is needed here: unfortunately it is not possible for me alone to consider every kind of intents and to convert properly any kind of data, but my system will work with the great majority of them. It is in fact unlikely that implicit intents will contain strange structured data types or *Parcelables and Serializable* data which are usually used with implicit intents, which, for construction of the operating system, my system can not handle. For these reason, I am not going to support, with this solution, that kind of data, since it would be very difficult, and highly inefficient, to send and reconstruct Parcelables and Serializable data types over my system. That said, I want to proceed with the definition of the syntax of the JSON-Intents.
In Figure 4.3 the complete structure of an intent is shown including the detailed description, using an UML model, of the bundle that is responsible for holding the Extras. The bundle is a sort of container of a non-predetermined number of couples <*Key,Value*> in which the Key is always an arbitrary String, while the Value can be any of the structured data types listed in the schema. The "*" notation at the end of any attribute in the bundle, means precisely that the number of <*Key,Value*> is variable, and optional, for any kind of structured Value data type.

I have already analyzed the structure of an Android intent object in Table 4.1,

Table 4.2: JSON-Intent fields

Name	Type	Description
action	String	It was a String also in the original Intent Object and does not need to be further converted. It is used also to identify, by using intent filters, which applications can execute the intent once arrived to the receiver.
[data][1]	String	It was a URI type in the Intent Object which can be easily stored as a string by using the URI.toString() method in Java.
[categories]	Array <Category> [2]	It was a Set of Strings in the Intent Object so it can be converted in an array of Category JSON Objects *(JSON Array)* containing all the category of the intent.
[type][1]	String	It was a String also in the original Intent Object and do not need to be further converted.
[extras][1]	Array <Bundle Object>[2]	It is a collection of all possible extras, it has to be converted in complex JSON Object which I will describe in the next tables.

now I am reproducing the same structure to convert it in a JSON-intent. Using some tables I would explain all the syntax I have created. In Table 4.2 the general structure for the JSON-Intent is shown, precisely in the first column, *Name*, there is in bold the String keyword used in JSON to identify the field. In the second column, *Type*, it is explained which kind of data type the field, identified by the keyword in the first column, contains. Since JSON allows to store only *String*, *Boolean*, *JSON Array*, *Number* and *JSON Object* data types in the third column, *description*, there is a short explanation of how the conversion is performed. Since *categories* and, as anticipated, *extras* are complex fields, containing array data types, introduced by me, I need to give further explanations of my choices in translating this objects form Java to JSON.

Table 4.3: Category fields

Name	Type	Description
category	String	It was a String also in the original Intent Object and does not need to be further converted.

In Table 4.3, using the same notation of the previous table, you can see how the *Category JSON object* is structured. It is, merely, a simple object containing a single field using the keyword *category* with a String value which is any possible Android intent's category. In the same way Table 4.4 shows the structure of the

Table 4.4: JSON-Bundle Object fields

Name	Type	Description
[key][1]	String	It contains the keyword, an arbitrary string, used to store the specific extra in the original Java intent object.
type	String	It is a string which describes the type of structured Java data type, it is used to correctly reconstruct the original Intent Object.
data	Variable	This filed is the one which contains the actual data, it can be a primitive data type such as a boolean a double and so and, or a supported array. The full description of this field is given in Table 4.5.

[1]The notation [Field] means that the field is optional
[2]The notation Array<Type> means that the Array contains Objects of that type

JSON-Bundle object. Every single JSON-Bundle object represent a possible *extra* of the original Java intent. The field *key* contains the string keyword used to identify the extra. The *type* field contains one of the keywords, in bold, defined in the next table, which is a String used to identify the structured Java data type, used to perform the so-called *deserialization*, by reconverting the JSON-Intent in a Android Java intent object. Depending on the type of data that it is necessary to store the *data* field can contain the actual data, if the original Java type was a primitive data type, or an array of JSON-Bundle objects without the field *key*, if the original data type was an Array or an ArrayList.

Java Object	**JSON Object**		
name	**type content**	**data type**	**description**
Primitive Data Types			
boolean	**boolean**	Boolean	The data filed contains the actual boolean value.
byte	**byte**	Number	The data filed contains the actual byte value.
CharSequence	**charsequence**	String	The data filed contains the actual CharSequence value converted in String.
double	**double**	Number	The data filed contains the actual double value.
float	**float**	Number	The data filed contains the actual float value.
int	**integer**	Number	The data filed contains the actual int value.
long	**long**	Number	The data filed contains the actual long value.
short	**short**	Number	The data filed contains the actual short value.
String	**string**	String	The data filed contains the actual String value.
Array Data Types			

boolean[]	**aboolean**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"boolean"* and in the data filed the actual boolean value.
byte[]	**abyte**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"byte"* and in the data filed the actual byte value.
CharSequence[]	**acharsequence**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"charsequence"* and in the data filed the actual charsequence value.
double[]	**adouble**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"double"* and in the data filed the actual double value.
float[]	**afloat**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"float"* and in the data filed the actual float value.
int[]	**ainteger**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"integer"* and in the data filed the actual int value.
long[]	**along**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"long"* and in the data filed the actual long value.
short[]	**ashort**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"short"* and in the data filed the actual short value.

String[]	**astring**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"string"* and in the data filed the actual string value.
ArrayList Data Types			
ArrayList <CharSequence>	**alcharsequence**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"charsequence"* and in the data filed the actual charsequence value.
ArrayList <Integer>	**alinteger**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"integer"* and in the data filed the actual integer value.
ArrayList <String>	**alstring**	Array <Bundle Object>[2]	The data filed contains an JSON Array of Bundle Objects having in the type field *"string"* and in the data filed the actual string value.

Table 4.5: Possible data types

In Table 4.5 there is the definition for any Java types of data my translation supports: for every Java type, listed in the first column of the table, the corresponding translation is provided, using the last three columns.
In this way I completed the translation of any generic Android implicit intent, which can, easily, be converted in a JSON-Intent by generating a JSON complying with the syntax defined above. The conversion process can be automatized developing an *Intent Converter*, which I will include in the *Liquid Android API*, which I will describe in the following sections.

JSON-Intent example

In this small subsection I would like to provide a full example of a JSON-Intent document which represents a theoretical proof of concept. Notice that I did not use all the fields, and obviously all the data types, I have defined in the previous pages: not all of them are mandatory to have a working system and maybe they can result redundant, and and make the example more difficult to understand.

The use case describes a working implicit intent used to ask the Android OS to send an email.
In the Listing 4.2 there is, in Java, a code example of a method which creates an Android implicit Intent to send an email and then asks the OS to resolve that Intent by using an Activity. In the Listing 4.3 there is exactly the same Intent converted using the syntax proposed above by me. The JSON-Intent thus created can be easily sent through the socket and then reconverted back to the original Java Intent object.

Listing 4.2: Implicit Intent example

```
public void composeEmail(String[] addresses, String
    subject) {
    Intent intent = new Intent(Intent.ACTION_SEND);
    intent.setType("*/*");
    intent.putExtra(Intent.EXTRA_EMAIL, addresses);
    intent.putExtra(Intent.EXTRA_SUBJECT, subject);
    if (intent.resolveActivity(getPackageManager()) !=
        null) {
        startActivity(intent);
    }
}
```

Listing 4.3: Conversion of the Intent in Listing 4.2 to JSON-Intent

```
{
  "action": "android.intent.action.SEND",
  "itype": "*\/*",
  "categories": [{
    "category": "android.intent.category.DEFAULT"
  }],
  "extras": [{
    "type": "string",
    "data": "example",
    "key": "android.intent.extra.SUBJECT"
  },
  {
    "type": "string",
    "data": "example@example.com",
    "key": "android.intent.extra.EMAIL"
  }]
}
```

Possible Messages

Having thus defined the communication model and language, I now need to define the possible kind of messages the system can handle: as mentioned earlier Android intents can be used with different methods to start different kinds of Android components, they are mainly 4 :

- *startActivity(Intent intent)*, ask the system to start an Activity to perform the operation contained in the intent passed as parameter.

- *startActivityForResult(Intent intent)*, it performs exactly the same operations of the previous method, what is more, it is used to receive a result from the activity when it finishes. The caller activity receives the result as a separate Intent object in the activity's onActivityResult() callback.

- *startService(Intent intent)* asks the system to start a Service to perform the operation contained in the intent passed as parameter.

- *sendBroadcast(Intent intent)* asks the system to send a broadcast message that any app can receive.

Since my system is supposed to work, mainly, by using implicit intents, the most common method it will use will be the first. I want, also, to support the use of the other three methods, so since it is impossible to understand what kind of method is needed only by analyzing any intent filed, I need to define standard attributes to let my system choose the right Android method to start the received intent. For these reason, my system will treat received intents with the first method unless otherwise specified. By including in the JSON-intent a standard extra field it is possible to support any method described above. Including in the extras array the extra with **key**:*"andorid.intent.extra.LIQUIDMETHOD"*, **type**:*"string"*, **data**:*"RESULT" or "SERVICE" or "BROADCAST"* my system will start the received intent with the appropriate method. Furthermore during the development of my system, I will analyze special cases of well-known intents which need a result back, for example intents asking the OS to take a photo are supposed to receive back the taken picture.

4.2.3 Data management Model

The last step is to establish how the system has to manage data created, or manipulated, by sending/resolving distributed intents. I found, basically, two solutions, with both different advantages and disadvantages.

Files over socket

Since the system exploits the LAN and creates a working P2P network, it is possible to send the generated data directly, using it by sending result intent or files over the socket. To continue with the previous example, in which one device of the network asks another one to take a photo, the device which performs the operation and holds the original picture can put it on the socket and send its file or the result intent,properly converted in a JSON-Intent, to the device that sent the request. In this way basically it is possible to realize any type of data management operation only by exploiting the underlying network.
This first solution has the advantage of not having to use external services to the local network, indeed it does not need to use the Internet connection. There is, however, the drawback of having to serialize/deserialize any data to be sent over the socket.

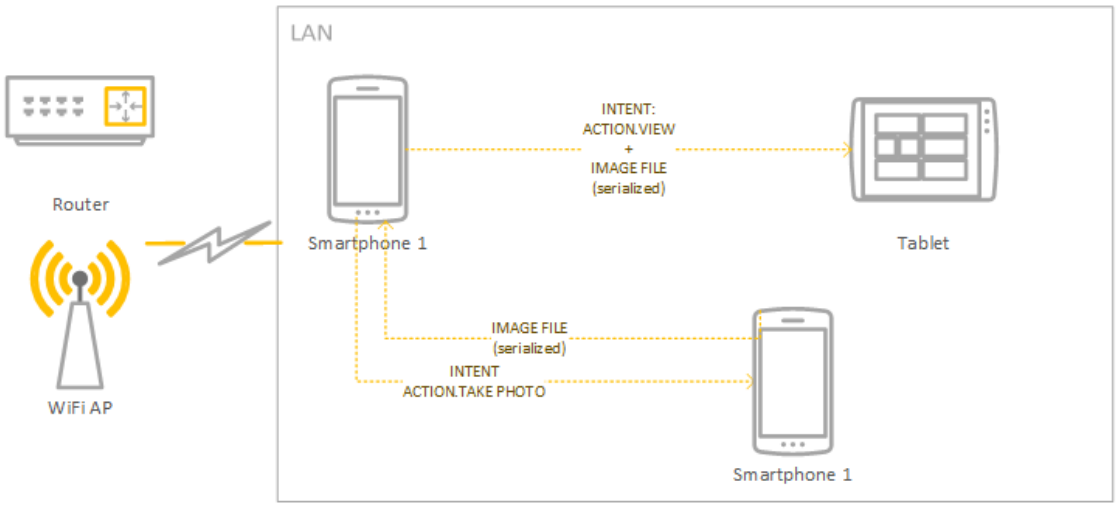


Figure 4.4: Files Over the Socket example

Cloud Group

It would be possible to easily implement a cloud group, with one of the well-known cloud services supported by Android, such as *Google drive*, where devices, while my system is running, can store and download data needed by the distributed intent and generated by executing them. In this way, the system should force to update in a cloud storage any data, such as files, text, numbers, results and so on. The cloud storage, indeed, must guarantee the devices involved to easily access right data. This means that before a user can run the system, it is necessary to set up the cloud group, in order to grant access to data only to authorized users. Once the group is up, and permissions are correctly managed,

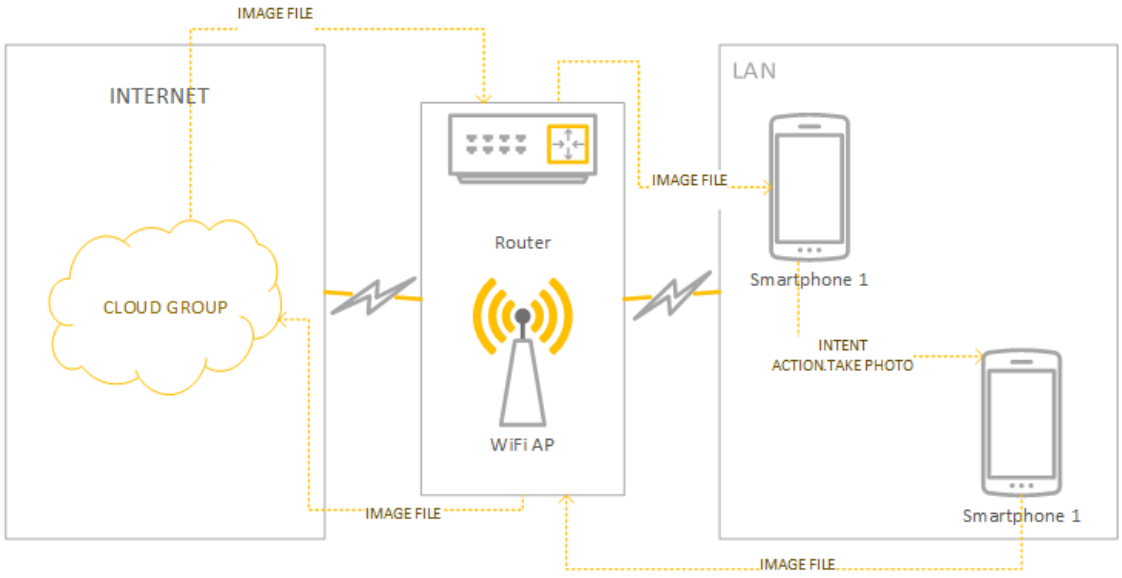


Figure 4.5: Cloud Group example

every time a data is produced and stored in the cloud, it would be possible

to notify devices in the group, interested in the data generated, and let them download, only by querying the cloud system.
This second solution has the advantage that can be easily implemented using existing cloud systems, but has as a drawback the fact that it needs an Internet connection and some security issues because data in that way can be accessed also outside my system.

4.3 Liquid Android API Library

Once the so-called theoretical solution is found, and described, which mainly solves the first scenario, presented in Section 3.4.1, I started the concrete development of the system, trying to solve the problem and also, to create a system which can be easily extended and used to build other purposes systems.
In this section I will describe the *Liquid Android Framework* as a development API, solving the second scenario, presented in Section 3.4.2, pointing out the main components, methods and configuration variables.

4.3.1 General Structure

I want to start giving an overview of the general structure of my system. I want to have few, but fully functional, components, and to accomplish the standard design pattern *Model View Controller (MVC)*.
In my API, the model is represented by the JSON-Intent structure described in the previous sections: intents are the way in which my system communicates, so they are the main kind of data to deal with. Then there are the controller components, which are responsible to manage the data, and to build and maintain the network. Finally there are, also, *User Interface (UI)* components, which are responsible for the interaction with the final user of the system.
In Figure 4.6 there is the complete structure, by using an *UML Class Diagram* of the framework, in which it is possible to identify the various components by looking at the legend under the picture: components in light-blue are Android activity used to interact with the user, so they indeed are UI components, while all the other classes are identifiable as controller components. The arrows in the figure explain the existing relations among the class components, pointing out the way in which they interact in the system.

4.3.2 Controller Components

First of all, I want to analyze the way in which the controller components build and maintain the network and then how they send/receive JSON-Intent objects and execute them.

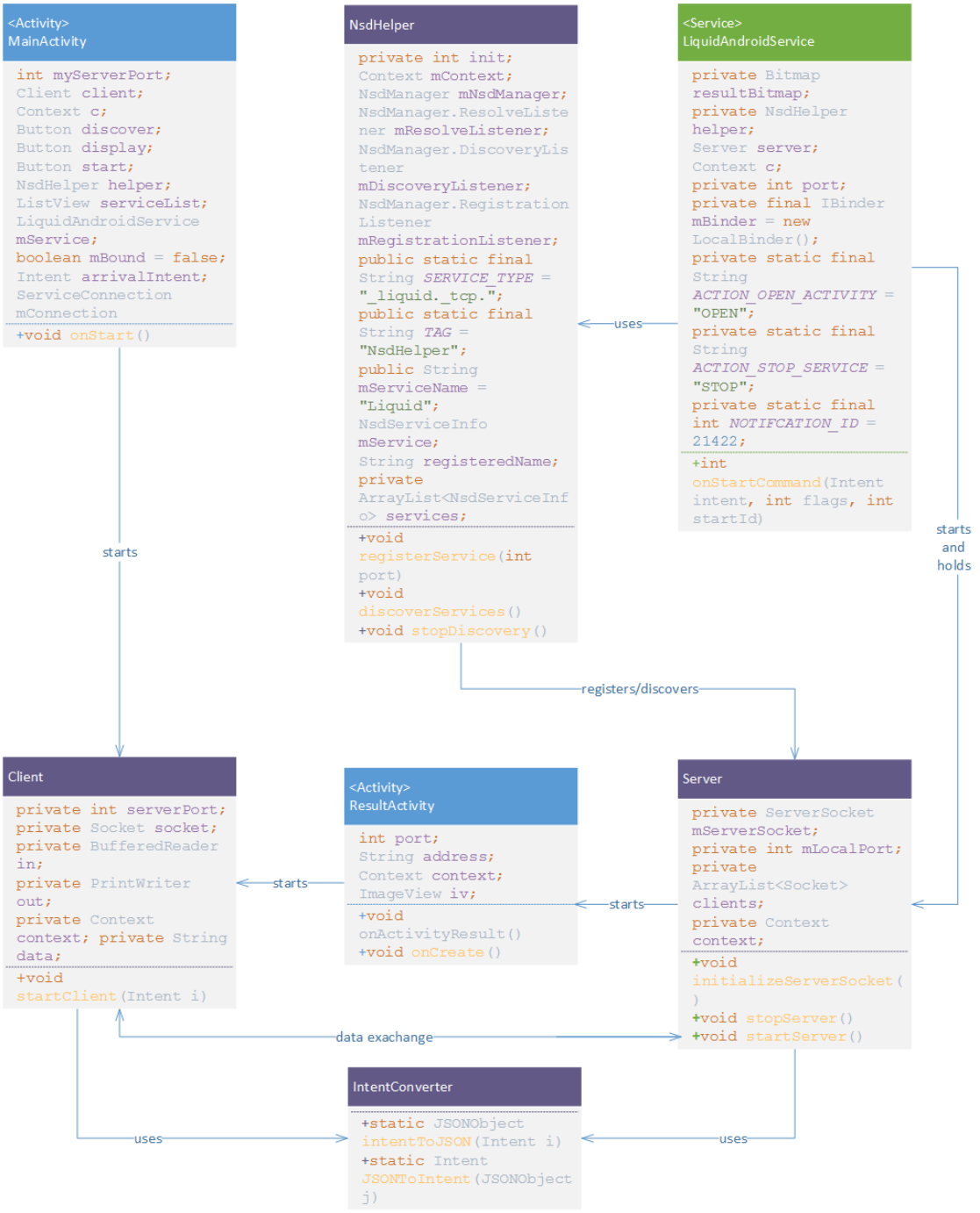


Figure 4.6: Liquid Android General UML
■ Android Service ■ Android Activity ■ Standard Java Class

NsdHelper

It is a standard Java Class component, which we can see as a *Helper Class*. It contains all the methods used to register the network service and to search and resolve other devices which are using the system, by finding other services in the LAN which are using the same name and the same transport layer. This class is an implementation of the *NSD Android API* already described in Section 2.3.3. Its

main configuration parameters are:

- *SERVICE_TYPE*, it is the string used to identify the network services in the LAN when the NsdHelper performs the register and discover operations. It uses the syntax: *"_<protocol>._<transportlayer>"*.
- *SERVICE_NAME*, it is the instance name: it is the visible name to other devices on the network. The name is visible to any device on the network that is using NSD to look for local services. The name, also, must be unique for any service on the network, and the NSD library automatically handles conflict resolution.

Since I want to give a flexible development API library, it is possible to change this parameter by using the proper setter method or by passing two strings to the NsdHelper constructor. For example is possible to use:

- *setServiceType("_<protocol>._<transportlayer>")*
- *setServiceName("name")*
- *new NsdHelper("_<protocol>._<transportlayer>","name")*

to change the configuration parameters when initializing the NsdHelper Class component.
Its main (public) methods are:

- *registerService(int port)*, this method performs the local network service registration, by using the variables described above and the service port passed as a parameter to this method.
- *discoverServices()*, this method performs the service discovery and resolution, by scanning the local network and saving found services in a list of resolved devices, containing the connection variables of the other compatible service in the network *IP* and *SERVICE_PORT*.
- *stopDiscovery()*, this is the method used to stop the service discovery mechanism.

Server

It is a standard Java Class component holding the server functionalities of the system. It is the component which opens the sockets and then waits and listens for invocations by other device clients. It is responsible for the reconstruction of the intent object by using the received JSON-Intent from the client, and it also has to allow the concurrency and let clients connect at any time. To perform this duty, my server component uses a multi-thread architecture: every time a client connects, it starts a separate *server-thread* to communicate with it. Once the message has been received the server translates it into a executable Android intent and then executes it by using one of the standard Android methods seen in the previous section when explaining the possible kind of messages.
Its main (public) methods are:

- *initializeServerSocket()*, this method initializes the *ServerSocket* Java component, by automatically selecting a free port in the LAN, which waits and listens for client connections.

- *startServer()*, this method is used to start the server-thread when a client performs an invocation, the started server-thread performs the operations contained in the received JSON-Intent message by reconstructing and passing it to the Android OS to be resolved.

- *stopServer()*, this method closes, and destroys, the server-thread when the client invocation is finished to avoid memory leaks when using the system.

Client

It is a standard Java Class component holding the client functionalities of the system. It is the component which connects to the server sockets of the other devices in the network and then sends the message containing the intent to be executed. It is responsible for the conversion of the intent object in a JSON-Intent to be sent on the socket. Once the message is created the client sends it to the selected server and then closes itself in an asynchronous way, without waiting for any ACK message or the result of the invocation.
Its main (public) method is:

- *startClient(Intent intent)*, this method send the intent, by translating it into a JSON-Intent, to the selected server. Then the client is automatically closed to avoid possible memory leaks when using the system.

IntentConverter

It is another *Helper Class*. It contains all the logic and methods used to perform the conversion of any Android Java Intent object in JSON-Intent objects, and viceversa, using the syntax proposed in the previous chapter. This standard Java Class does not have any variable or parameter, and does not need, also a special constructor. It has only two main Static public methods that can be invoked by using the Class name.
They are:

- *intentToJSON(Intent i)*, this is a Static method which performs the conversion from any Intent object, passed as parameter, to a JSON-Intent, returned by the method as a JSONObject, using the syntax created by me and explained in the previous chapter.

- *JSONToIntent(JSONObject j)*, this is a Static method which performs the reconstruction of the Java Intent object, by parsing a well-formed JSONObject. It returns a fully working original Android Intent object.

LiquidAndroidService

This is the Android background working service which really performs the extension of the Android OS giving it distributed functionalities. When this component is started, it is responsible for the initialization of the server, and the registration of the network service in the LAN, by calling the methods of the classes explained above, respectively the Server and the NsdHelper. This is a sort of container for the server side components of the middleware. It is implemented as a foreground working Android service, thus when it is in execution, the user is alerted by a notification. A foreground service is a service that the user is actively aware of and is not a candidate for the system to kill when low on memory. A foreground service must provide a notification for the status bar, which is placed under the Ongoing heading. This means that the notification cannot be dismissed unless the service is either stopped or removed from the foreground [31].

4.3.3 UI Components

I now want to explain how the users can control the distributed intent flow, by using standard Android UI components. Even if my system is supposed to run in background it is also necessary to have a few UI components to control the background working mechanisms. In this section I will not provide any screenshot of a real Android application, because in this part I only want to analyze the functionalities included in these components. I will include many real UI screenshots while presenting the Liquid Android application I have developed, using this API, in the next Chapter, 5.

MainActivity

This is the UI component responsible for controlling the entire system. It has four important functionalities:

- *Listen to implicit intents*, when an implicit intent needs to be resolved, the Android OS starts the resolution mechanism already described in Figure 2.4.

Listing 4.4: Intent filter example

```
<activity ...>
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <data android:type="*/*" />
        <category android:name="android.intent.category.
            DEFAULT" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.SENDTO" />
        <data android:scheme="mailto" />
        <category android:name="android.intent.category.
            DEFAULT" />
    </intent-filter>
</activity>
```

It searches for the best activity for the intent by comparing it to intent filters based on three aspects: *Action*, *Data (both URI and data type)* and *Category*. A match is only successful if the actions and categories in the Intent match against the filter. So, to let the Liquid Android API listen for any kind of implicit intent, it must include an activity declared with any possible intent filter to be called when an intent needs to be executed.
In Listing 4.4 there is an example of how an Activity needs to be declared in the manifest to be called by the OS when an implicit intent like the one in the JSON-Intent in the translation example, 4.2 is triggered. By extending the MainActivity component and adding any kind of intent filter to its declaration in the manifest, it is possible to listen practically to any type of existing implicit intents in the Android framework.

- *Find other devices in the network*, the MainActivity provides also this functionality by calling the methods of the NsdHelper class, and then it shows the resolved devices in the network in a ordered list.

- *Forward intents to selected devices*, once the other devices in the network are found, the MainActivity lets the user select on which device, or devices, forward the implicit intent it is managing at that moment. To perform this task, the MainActivity starts the Client component and exploits its methods.

- *Start/stop the background service*, since when the background service is working, the device is exposed to many threats, the user can control it by starting and stopping it every time it is needed by clicking on some special buttons in the MainActivity.

ResultActivity

This is the UI component which can be used when a received intent needs to send data or results back to the caller. This Activity implements the *onActivityResult()* callback method to manage the result or data produced by the received intent. In my API framework this problem is solved by using the files over the socket approach, described in Section 4.2.3, so when the callback is triggered, results are sent to the caller through the socket as JSON-Intent messages or serialized files, by using the Client class to send a message to the Server component of the caller. It is necessary to have this activity since it is impossible to call the *startActivityForResult(intent)* method from a background service in Android.

4.3.4 Use Cases

The full API code is available at github.com/mola15/LiquidAndroid, which is a public GitHub repository, so anyone can read, clone use and modify my code for different purposes.
As already said several times, I will use this API to fully develop the Liquid Android APK which will be a working solution for the first and third problematic scenario, but the Liquid Android API can be used as an Android library in the

development of native Android distributed applications. By including my API in any development project it is possible to speed up the process by exploiting my working mechanisms and my methods already developed. Since the API is in Java is just as easy to extend the classes and override some of my methods to accomplish special purposes distributed systems.
Exploiting my system would be easy to develop for example:

- Distributed Android Computing systems, in which Android devices can share hardware resources to reach a common goal.
- Android Computer Cluster, in which Android devices can be seen as node of clusters to perform the same task, controlled and scheduled by software.
- Distributed Android File system, by refining the data management model.

Furthermore, it is possible to adapt my work to accomplish many other purposes, at the end of the next chapter I will present a real, concrete small application, which exploits my system and represents a real use case for my framework.

Chapter 5

Case Study

In this chapter I will describe the real implementation of the system, which is the real solution of the problem faced by this dissertation: how it is possible to extend a mobile operating system, in this case Android, with distributed OS functionalities.
This chapter is mainly composed of three parts: the first one is a generic information section in which the proof of concept is explained in terms of technologies used, requirements to meet, goals and various technicalities. The second one is the report of the implementation and development of the application, with choices and descriptions of what has been done. The third part is a working demo of the system that has just been described, with live working test cases. It contains screenshots of the application while it is running and a complete description to explain each case step by step.

5.1 Conception

As already specified in the previous Chapter,4, my system has been implemented as a standard Android application, which can be installed on any Android device starting from the API level 19, Android 4.4 KitKat. The final APK package contains all the files needed for the system installation, and, once installed, the application performs the extension of the Android OS giving it distributed functionalities. I will use the complete API described in Section 4.3, to implement a background working middleware to distribute implicit intent in a LAN to any Android device with the service installed. In this way every time one of the devices, having the *Liquid Android APK* installed, triggers an implicit intent, my application could intercept and send it to any other device to be resolved and executed. The idea of this prototype is to prove that what I have stated, providing the theoretical solution, can work with a real configuration of Android devices in any LAN. Doing this, the thesis work is somehow *"proved"*: my communication language, defined with a JSON file, is concretely usable and working, not to worry the users about the kind of implicit intent they need to execute in one of the devices in the network. The translation process does not represent an issue for my application, because I have developed an automatic intent translator using the correct syntax proposed by me. I will not develop

clients for third party systems, even if I stated that it would be possible, especially in Java environments, but I will implement a simple Android application client, generating some standard implicit intent to perform some tests with my system.

5.1.1 Requirements

In this small subsection, I want to provide a full list of requirements my application must meet. In order to be considered a solution of the given problem, it must fulfill the constrains listed in Section 3.5 and also comply with functional and non-functional requirements.

Functional Requirements

Functional requirements are, indeed, the main functionalities the systems must have in order to properly work to perform a desired task.
The following lists summarize the main features of the system, so as to ensure a quick reference while reading this document:

- **FR1**: listen to implicit intents.
  My application should declare itself, in the android manifest, as a multi-purpose application which can be used to resolve, basically, any kind of implicit intents, in order to be selected by the Android OS whenever an intent resolution process is triggered.
- **FR2**: JSON to Intent, and Intent to JSON, conversion.
  My application must be able to perform the conversion using the JSON syntax I have explained in Section 4.2.2.
- **FR3**: forward implicit intents.
  My application must be able to forward any of the implicit intent it can listen, to other LAN connected devices with the *Liquid Android APK* installed.
- **FR4**: receive and execute intents.
  My application must be able to receive in any moment implicit intents, as JSON-Intent object, and then, let the OS resolve and execute them with its standard mechanisms.

Non-Functional Requirements

Non-functional requirements are important properties that my system must have in order to guarantee full functionalities. They are not specific for my problem but, they are general requirements a system needs to be considered complete. It is quite clear how a system can use my language but if it takes 15 minutes to perform a translation or to deliver a message it is completely useless. Non-functional requirements in this way complete my system, they are mainly:

- *Portability*: to have my application used by the largest number of users possible, so I have made the choice to use Android API level 19, to allow the installation of my system to, more or less, 84% of Android devices currently active.

- *Stability*: system must always be available, and able to offer all its services. For example, I should avoid possible system crashes during the delivery of a message from a device to another. In addition, data must be durable and not lost for any reasons.
- *Availability*: the services must always be accessible in time. In case of failure, it is possible for the user to manually restart it to be again usable.
- *Reliability*: since data are shared among devices, reliability is essential. Users can base their actions on other users' actions and on the status of the devices. Moreover, I assume that the memory where data are stored is stable.
- *Efficiency*: within software development framework, efficiency means using as few resources as possible. Thus, the system will provide data structures and algorithms aimed to maximize efficiency. I will also try to use well-known patterns, reusing as many pieces of code as possible, taking care to avoid any anti-patterns.
- *Extensibility*: my application must provide a design where future updates are possible. It will be developed in such a way that the addition of new functionalities will not require radical changes to the internal structure and data flow.
- *Maintainability*: also modifications to a code that already exists have to be taken into account. For this reason the code must be easily readable and fully commented.
- *Security*: Using a networking service security is always required. The fact that the system will be available only on LANs is the first step in this direction.

5.1.2 Used Technologies

This other subsection is an overview of the tools I have used to develop the Android application.
Android applications are written using Java code and any API level of the Android framework. The *Liquid Android APK* has been developed by using standard Android development tools and libraries without the need to rely on any third party API library. In particular I decided to make these choices.

- *Android API level 19*, as already explained several times I want my system to be installed on the largest number of devices, so it is a compromise between the great innovation introduced starting from this API level, and the number of devices which can execute API level 19 apps.
- *Android NSD library*, it is a standard Android library I am using to register, discover and resolve my network service. It is an implementations of Zeroconf and it is compatible with other implementation such as Apple's bonjour.

- *Standard Java libraries*, I decided to use only libraries included in standard Java development kit distribution, the most important are :
  - *org.json*, used to manage the JSON file, which are the messages exchanged by the devices using my application;
  - *java.net*, containing the classes which are useful to implement the socket network communication.

Technically, I used an IDE to help in development, in particular AndroidStrudio based on IntelliJ IDEA, a modern solution released by Jet Brains. It contains modern tools to check Java compile time errors flagging them, and tools to check run time errors with a complete and verbose stack trace.Finally, it supports various VCS (Versioning control systems), to push the code inside repositories and to have a complete overview of commits and forks. I chose Git, one of the most famous VCS, and GitHub as repository to save my code.

5.1.3 Implementation

I have developed a fully-working Android application called *Liquid Android*. I have built the application using the API library I have created and already presented in Section 4.3, so the structure of my code follows exactly the one presented with the *Liquid Android API* UML model, in Figure 4.6. The components and the methods I have used, to create the application, are exactly the ones presented in that figure with little modifications and adaptations.

Code organization

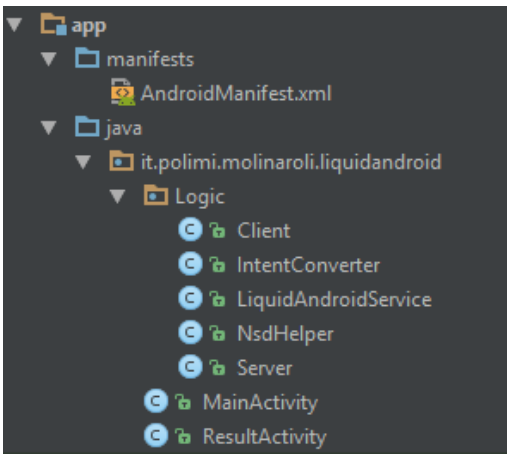


Figure 5.1: Code organization

As anticipated, the code is organized following the *MVC* design pattern, so the *controller components* are all contained in the *Logic* package, while the *UI components* are left inside the *Main* package of the application. Other components typical of the Android development framework are left in their standard locations, such as the XML file containing the *application manifest*.

Implicit Intents to listen

Listing 5.1: Liquid Android MainActivity Manifest example

```
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER"
            />
    </intent-filter>
  <!-- intent filters per ascoltare intenti impliciti -->
  <!-- BROWSER -->
  <intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT" />
    <data android:scheme="http" android:host="*" />
  </intent-filter>
  <!-- TAKE PHOTO WITH CAMERA -->
  <intent-filter>
    <action android:name="android.media.action.IMAGE_CAPTURE" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
  <!-- EMAIL -->
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="*/*"/>
  </intent-filter>
  <!-- MAP -->
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="geo" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
  ...
 </activity>
```

In Listing 5.1 there is part of the manifest, of my application, showing some common intent filters which the *Liquid Android* app can listen to. In the figure we can see that it is the MainActivity of the application which declares itself capable of managing intents to take a picture, send an email or open a map. By adding any intent filter to the manifest of the application, Liquid Android can listen and forward, automatically any kind of Android implicit intent. This snippet of code is the way in which the FR1 is practically implemented.

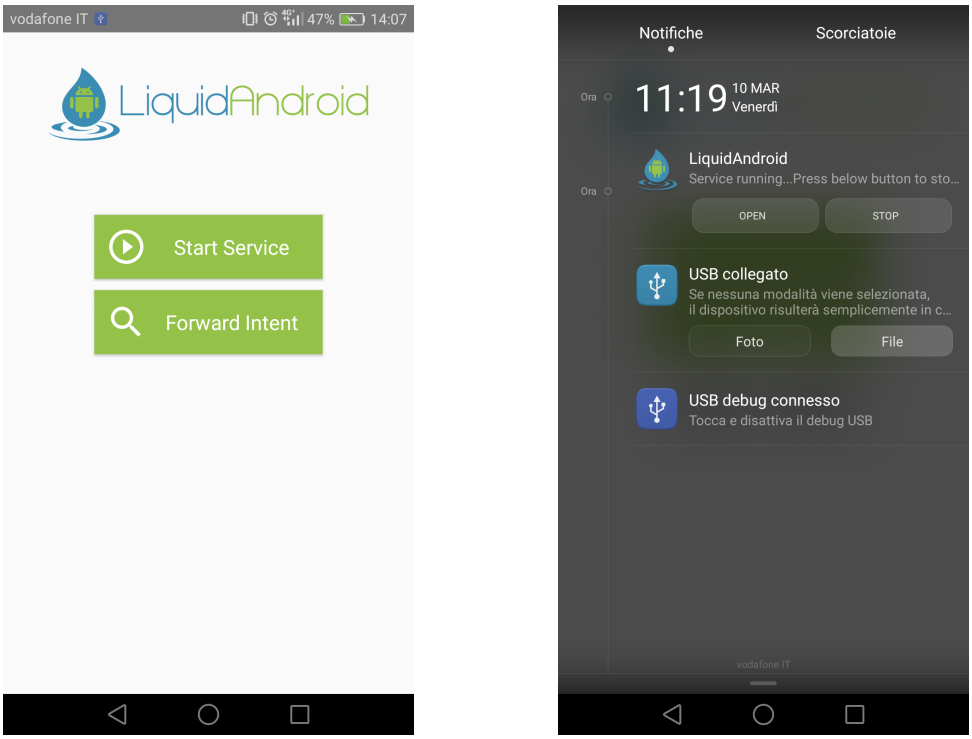
In the following section I want to describe my system in action, providing application's screenshots, UML diagrams, working tests and use cases.

5.2 Working Demo

This section is intended to show the reader the *Liquid Android Application* while it is working. After the structure and the implementation of the application

were explained, it is necessary to show the finished work. As anticipated, my application is simply a proof of concept of how it is possible to use a group of Android devices, as they were executing a single distributed operating system using well known Android mechanisms. Users should not worry about substrates, they can control everything with a single and simple standard Android UI.
I set up the demo by creating a LAN with a wireless router, and then I installed my application on three smartphones, connected both to Android Studio to debug the applications reading the consoles, and to the wireless LAN. This is only one possible environment configuration for my middleware application, but is simple and significant enough to provide a proof of my work.
I developed, also, for testing purposes, a simple client application able to generate standard Android implicit intents, which I will use to perform some live test cases. I called this Android app, *Intent Generator* which has the only feature of create intents and then ask the OS to resolve them. The same result can be obtained using any standard Android app which generates implicit intents and passes them to the OS to be solved. The full code of the *Intent Generator* application is accessible on GitHub at github.com/mola15/IntentGenerator.

5.2.1 Liquid Android UI



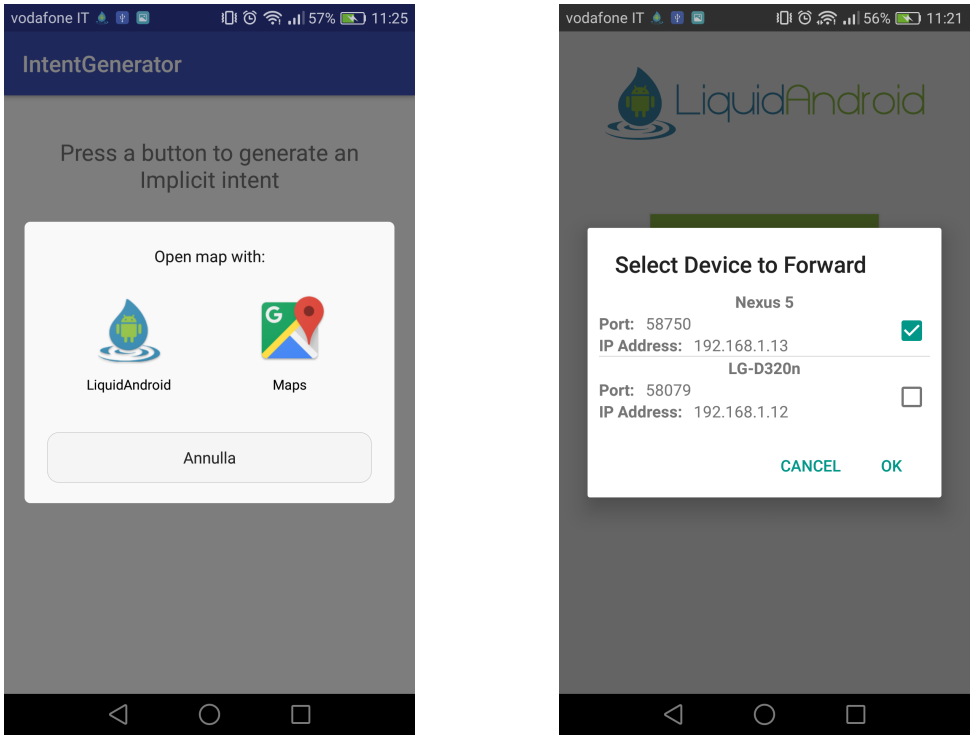
(a) Main Activity UI **(b)** Notification UI

Figure 5.2: Main Liquid Android UI components

Once the Liquid Android application has been installed on a compatible device, users can control it by its MainActivity and its *foreground service* notification, when the service is in execution. Figure 5.2 shows the Liquid Android applica-

tion's main UI components. By clicking the button *Start Service* the middleware executes and the extension of the Android OS is performed by the application. Once the button is clicked the server component of the application is up, and the network service is registered in the LAN, moreover the notification showed in sub-figure 5.2b appears in the Android notification area. From that notification, users can control the status of the service, because it cannot be removed from the Android notification area until the service is stopped by clicking the stop button, embedded in the notification. The other embedded open button, instead, starts the MainActivity, in sub-figure 5.2a, and puts it in foreground.

When any Android component asks the system to resolve an implicit intent,

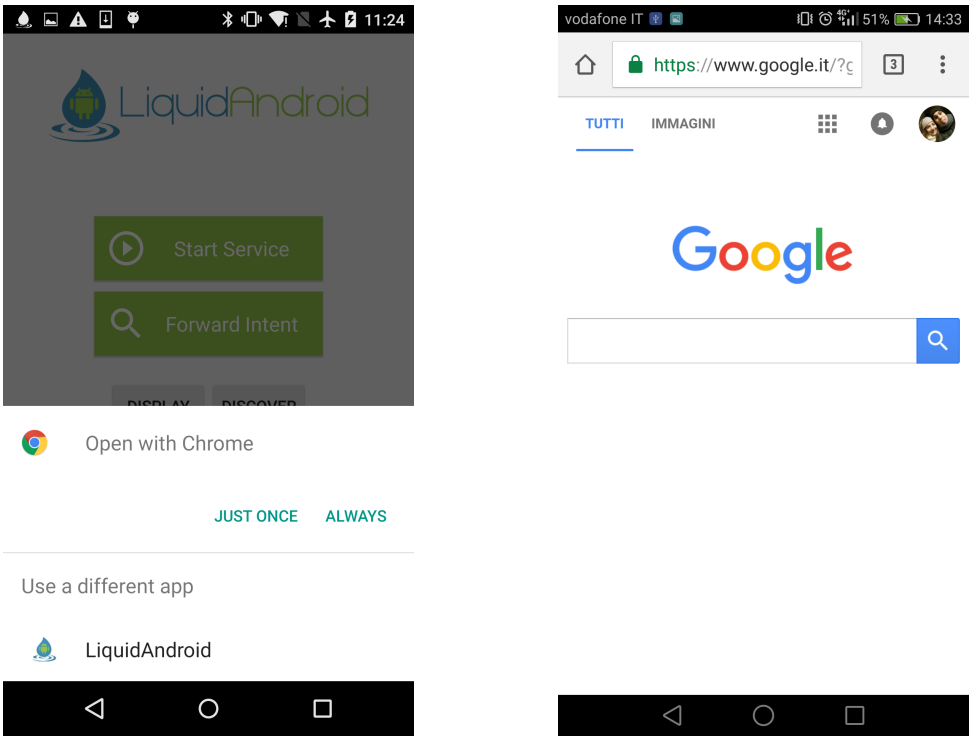


(a) App Chooser Dialog UI **(b)** Forward Dialog UI

Figure 5.3: Dialogs Components

which my application can handle, the Android OS opens the *App Chooser Dialog* and waits for a user choice. By selecting the Liquid Android application in the *dialog* showed in sub-figure 5.3a, the user ends on the MainActivity UI. Now, if the service is already in execution it can perform the forward action. The second button in the MainActivity, indeed, *Forward Intent*, can be used only while the service is running. When properly clicked, it opens the *Forward Dialog UI*, in sub-figure 5.3b, which my middleware automatically searches for devices in the LAN with the service installed and in execution, and lets the user select on which of them forward the previously intercepted implicit intent, by ticking the *check-box* as showed in Figure 5.3. Once selected on which device,or devices, the intent should be forwarded, by pressing the ok button, the application converts it in a JSON-Intent and sends it through the socket to them, generating Clients components which connects to the target Servers components. At this point, when

the message is received by the target devices, the JSON-Intent is automatically reconverted in the original Android intent, and a new intent resolution process is triggered by my application. Thus, the Android OS shows again, to the user, an *App Chooser Dialog*, and lets him select with which application, among the listed ones, perform the intent task. Figure 5.4 describes the working mechanism



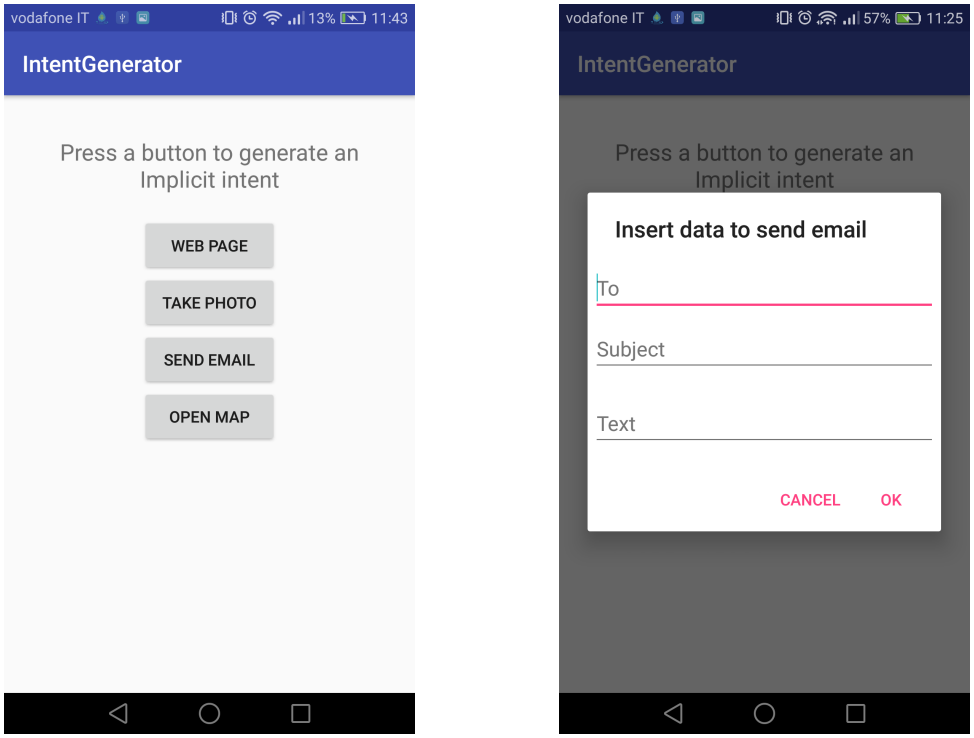
(a) App Chooser **(b)** Third party Browser Activity

Figure 5.4: JSON-Intent execution

above explained, in this particular case the devices received an intent to view the web-page *http://Google.it*. By selecting, in sub-figure 5.4a, *Open with Chrome*, the process terminates with the execution of the browser activity, in sub-figure 5.4b, showing exactly that page.

5.2.2 Live Test Cases

In this subsection, I want to present two live tests I performed to prove that my application respects all the constrains and fulfills all the functional, and also non-functional, requirements. As already explained, I created a second simple Android application working as a client for the Liquid Android middleware. This application *Intent Generator* is composed of a single activity in which there are some buttons to let the user create easily implicit intents to be resolved by the Android OS. In Figure 5.5 there are two screenshots of the *Intent Generator* application. In sub-figure 5.5a, there is the main activity, with which I generate some common implicit intents, to be forwarded using the Liquid Android middleware, by simply pressing the desired button in the UI. When the intent to be generated needs extra data the application shows a *Dialog Picker* to allow the



(a) Intent Generator MainActivity **(b)** Intent Generator data Picker

Figure 5.5: Intent Generator UI

insertion of additional data to the intent, as shown, in sub-figure 5.5b, when the *SEND EMAIL* button is pressed.
By using this environment I have performed the tests I am presenting in this section. Both tests cover all the functionalities the systems must have, taking into account, also the data management problem. The first test uses the system to forward an intent to send a text email from one device to another in the system. The second one is a bit more complicated, one device of the network asks two other devices to take a picture with their camera and then to have the taken photos back.

Send Email Live Test

In this live test scenario I assume that there are two Android devices, with the Liquid Android middleware application already installed and with the service in execution. Figure 5.6 shows the complete UML sequence diagram, which completely describes this live test.
The first device starts the process by using the Intent Generator MainActivity, precisely by pressing the *SEND EMAIL* button, already shown in sub-figure 5.5a. Then the user compiles the fields in the dialog and, once done, the implicit intent is created by the application and passed to the Android OS. The OS looks for activities capable of handling the intent thus generated, and asks the user, by showing the App Chooser, with which compatible application he wants to resolve the intent. In this case, the user selects the Liquid Android application, and ends on its MainActivity.

At this point the *Forward Intent button* is pressed and the intent is automatically converted and sent to the selected device/s. When the intent reaches to the target, the Liquid Android Service reconverts it into an intent object to be executed and passes it to the OS. The Android OS restarts a resolution process and at the end

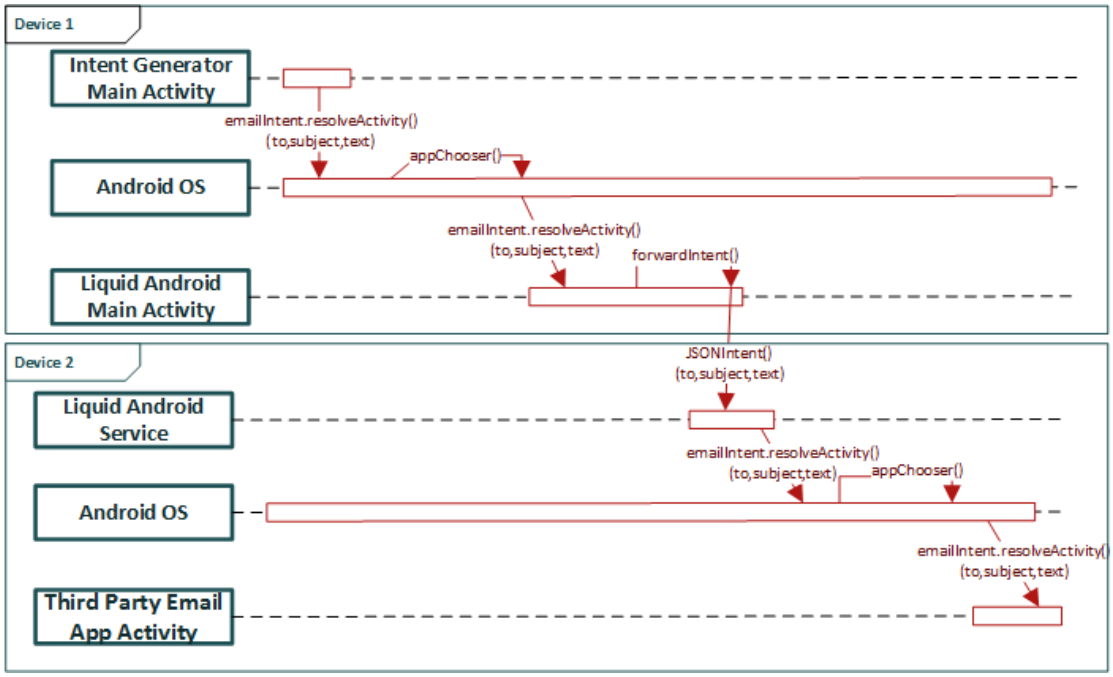
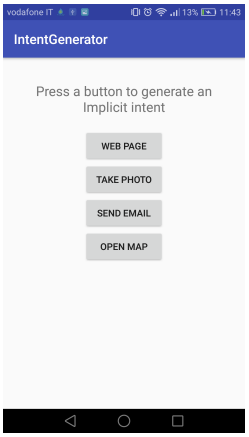
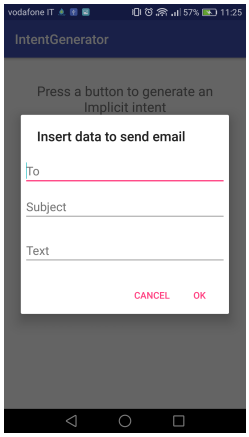


Figure 5.6: Live test 1 Sequence Diagram

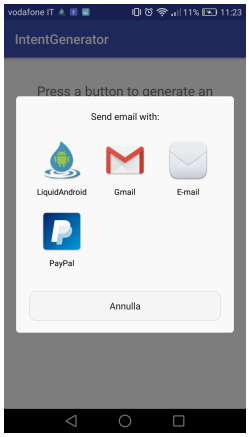
shows the App Chooser again. Now the user selects the Gmail application and completes the action by sending the email from the Gmail activity, already filled with the data the user inserted in the first device.
In Figure 5.7 there is the complete flow of actions presented with some real screenshots of the applications I have developed. In particular the screens from *a* to *d* are taken from the first device I have used, a Huawei P9, and screens *e* and *f* are taken from the target device, a LG L70.



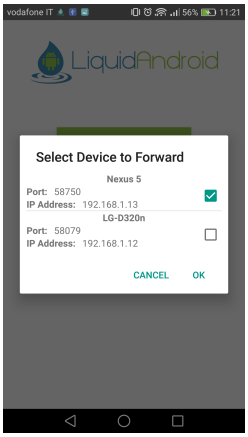
(a) Send Email



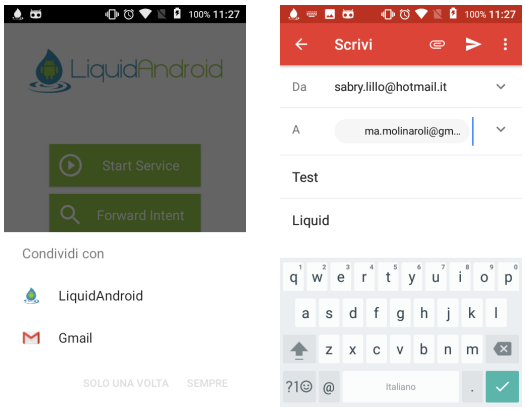
(b) Picker Dialog



(c) App Chooser



(d) Forward Dialog



(e) App Chooser **(f)** Gmail Activity

Figure 5.7: Complete email live test screenshots

Take Photo Live Test

The second test I want to report exploits the Android *startActivityforResult()* mechanism: it handles more complex data, and returns them to the caller. The environment I have used is exactly the one described in the previous test: three Android devices connected in the same LAN with the Liquid Android application already installed, and the background service already in execution. I want to prove that the middleware can work with more than two devices and how it manages concurrent requests. To do this, I want one device in the network to ask the other two devices to pick a photo at the same time, and once done, send the two different picture, taken with two different devices, to the original caller. I started the test by using the Intent Generator app, to generate the photo intent. When the intent is triggered, the Android OS opens the app Chooser, showing the Liquid Android application as an alternative to resolve the intent. I have forwarded the intent, using the forward button, and sent it to the other two devices in the network. Once the intent has arrived, the Liquid Android service triggers it, and passes it to the Liquid Android Result Activity, which starts the Android camera waiting for results. Once the user has taken the photo, with the camera, the Results Activity brings it, and send the picture back to the caller. When the picture arrives back to the caller, the Liquid Android application shows it in its Result activity. Since the intent was forwarded in two devices, when the two pictures are sent back to the caller, the two result are concurrent. My system allows that concurrency by accepting different calls using different threads and then it puts in foreground the last call arrived using a LIFO policy. This complete test scenario is described, as done with the other test, showing the interaction of the components, and their activation in the UML sequence diagram in Figure 5.8. In the diagram, only two devices are present, but the third device involved in the test has exactly the same behavior the *device 2* showed in the picture.

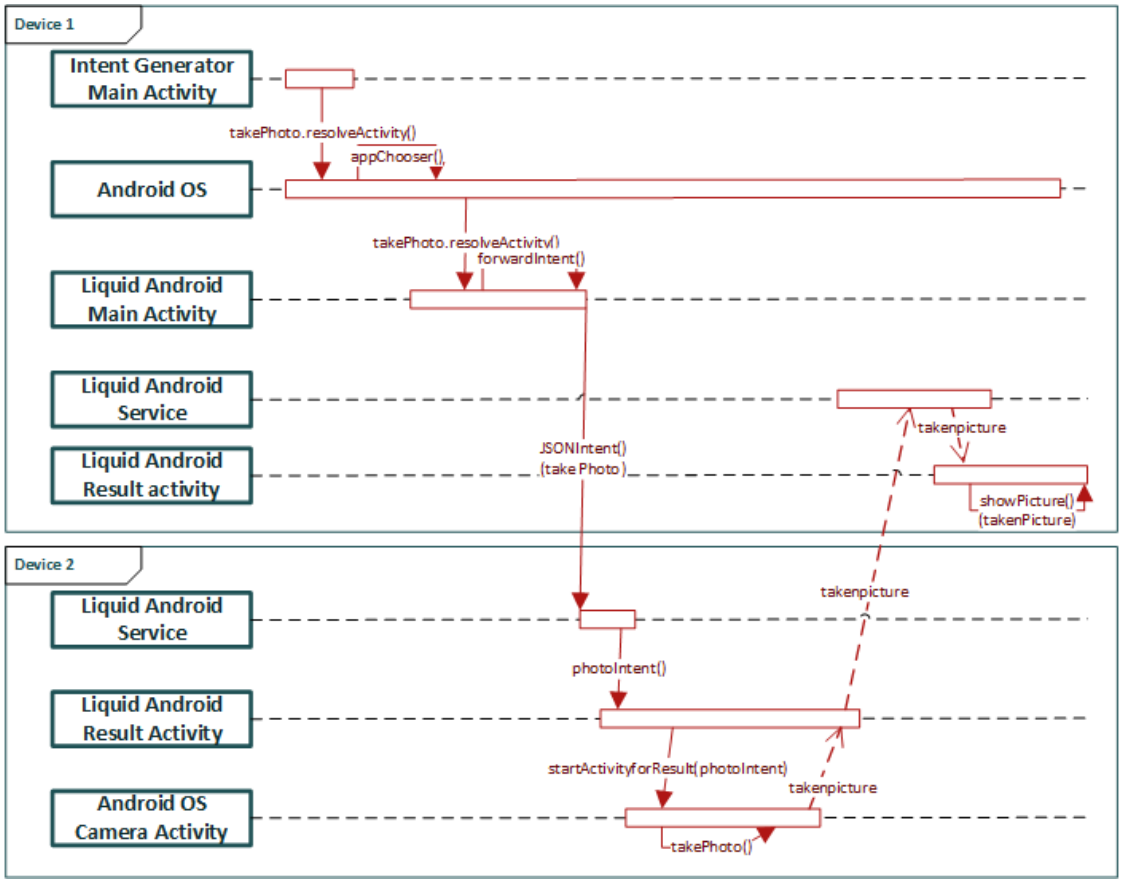
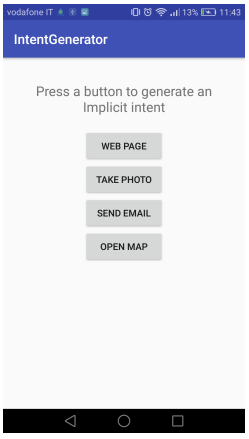
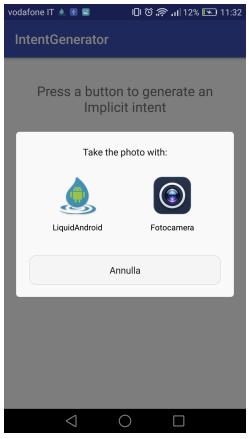


Figure 5.8: Live test 2 Sequence Diagram

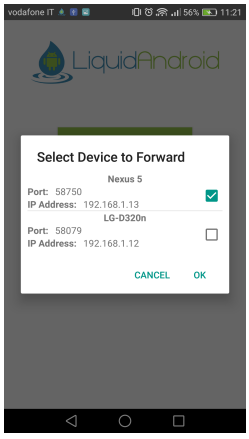
As already done with the previous live test case, I want to provide a set of screenshots showing the application behavior while running this example.
In Figure 5.9 there is the complete flow of actions presented with some real screenshots of the applications I have developed. In particular, the screens from *a* to *c* are taken from the first device I have used, a Huawei P9, to generate and forward the so-called photo intent.



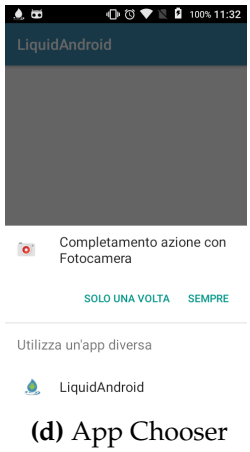
(a) Send Email



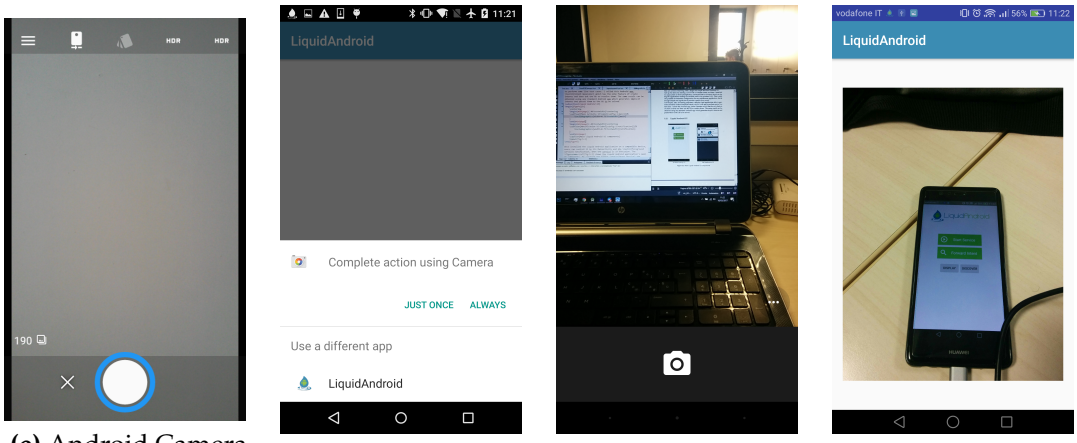
(b) App Chooser



(c) Forward Dialog



(d) App Chooser



(e) Android Camera **(f)** App Chooser **(g)** Android Camera **(h)** Result Activity

Figure 5.9: Complete photo Intent live test screenshots

Screens *d* and *e* are taken from the first target device, a LG L70, used to take the first picture. Screens *f* and *g* are taken instead, from the second target device, a Google Nexus 5, used to take the second picture. In the end screen *h* is taken again from the first device, showing the picture taken. All the pictures taken while performing this test, are saved locally on the local storage of the devices which have actually taken the picture, moreover they are saved in the local data storage of the caller device, and are accessible from its gallery application.

5.3 Liquid Museum

In this section I want to provide a real use of case for my framework, so I developed, for this purpose, two Android applications, using the Liquid Android API, which are a concrete example of how special purposes distributed applications can be built easily and fast exploiting my solutions and my system. The distributed application I have developed is composed of two APKs, one acting as a server app and one acting, indeed, as a client. These applications are supposed to be used in a real life use case: they aim to improve the experience of a guided tour in a museum. Both applications, are fully accessible on GitHub, the server app, *Liquid Museum* at github.com/mola15/LiquidMuseum, and the client side app, *Museum Client* at github.com/mola15/MuseumClient.

5.3.1 System Structure

I developed the two applications using the mechanisms described in the previous sections of this work. I embedded the Liquid Android API structure, and I used it to let the server application spread intents, exploiting JSON-Intents, to all the devices having the client application installed and with the service already started. Figure 5.10 describes the complete structure of this application. In this case I decided to solve the data management problem with a cloud solution. Data, in fact, are stored in the cloud using the *Firebase Platform*, by Google, which

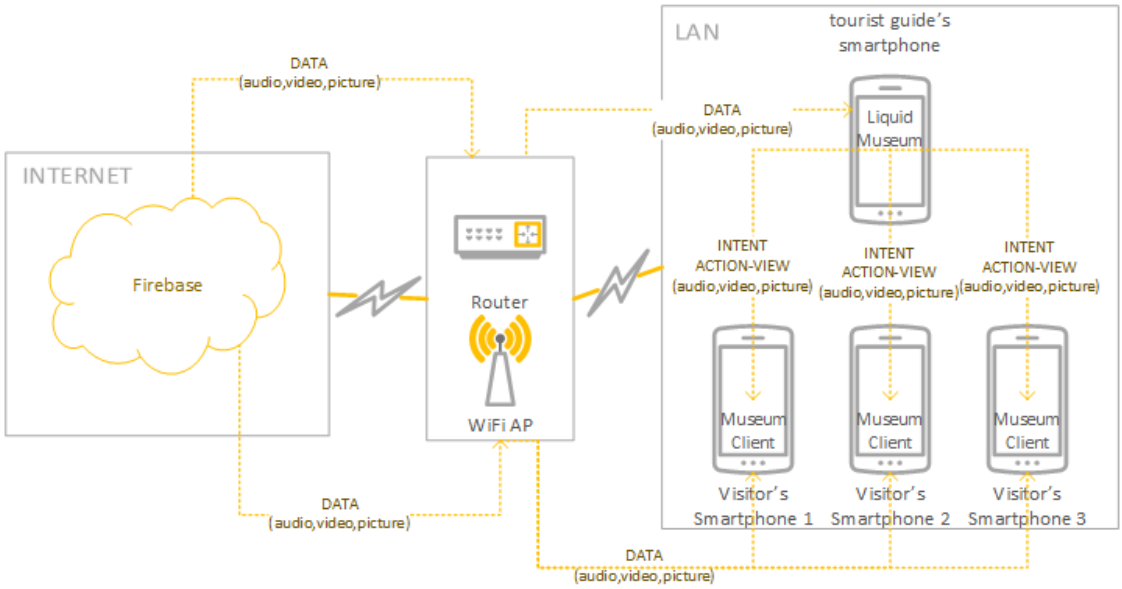


Figure 5.10: Liquid Museum distributed structure

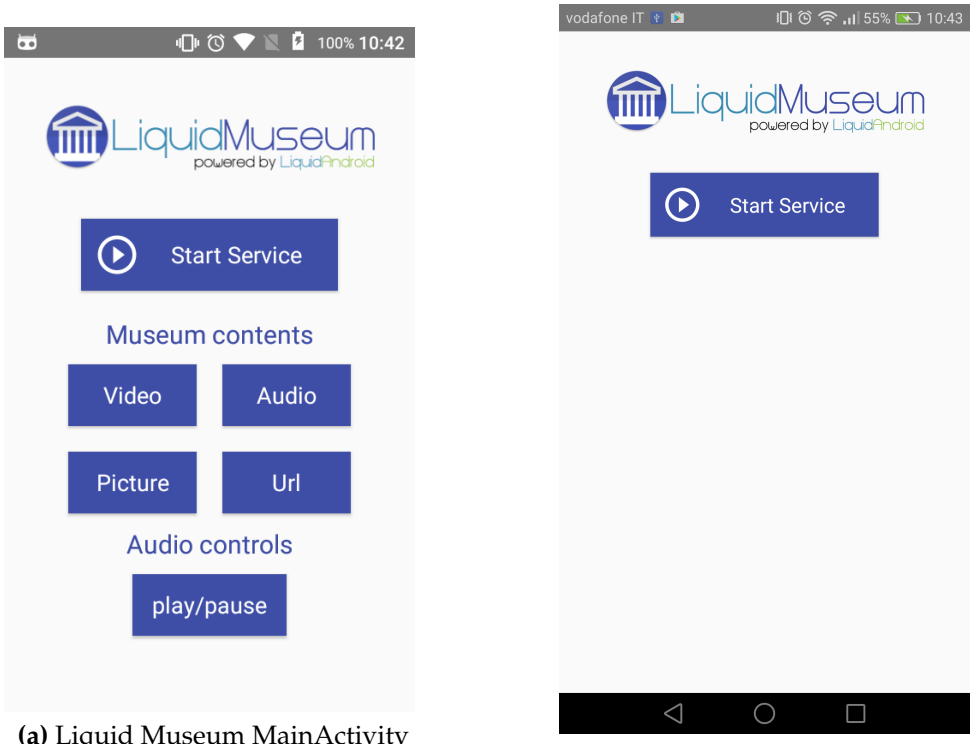
provides, among other things, a cloud storage, in which it is possible to easily save and manage any kind of file. The Liquid Museum application has been designed to be used by a tourist guide while he, or she, is doing his job, in a museum, during a guided tour with a group of tourists. The client app, Museum client, has been designed to be installed on the tourists' devices, to receive extra contents, in the form of distributed intents, while following their tourist guide in the museum guided tour. In this way the server application works as a controller for the client's one, and it can retrieve the right contents in the cloud and spread them to the clients, only by pressing some buttons in a control Activity. In this way the use of my distributed application enriches the guided tour providing insights, images, audios and video contents.

5.3.2 Functionalities

As I stated in the previous subsection, the client app can only connect to the server one, to receive JSON-Intents, reconstruct the original Android intent and execute it. It contains only a simple UI to start the service, which is exactly the same as the Liquid Android API: it works in the background and can be controlled with the very same notification I have described in Figure 5.2b.
To speed up the development and test phase of this application, the two applications register their service using the same name *Museum*, but in a real museum environment, in which different tourist guides perform different guided tours at the same time there is the need to let the user change this name, in order to connect only the desired devices to the tourist guide's Android device. To solve this problem, it would be enough to record the various services by adding to the name, the *ID* of the tourist guide.
The server is designed to spread contents to the clients app in range. It has, indeed, a control interface activity, with which it is possible to send contents to

all the clients in range only by pressing a button. In particular my application can:

- connect to the Firebase cloud to know which contents are available,
- send audios, videos, pictures or web pages to client, in the form of JSON-Intents,
- control the audio media player of each client in range by using the audio control button.



(a) Liquid Museum MainActivity

(b) Museum Client MainActivity

Figure 5.11: Liquid Museum and Museum Client UIs

Figure 5.11 shows the UIs for the two applications, in particular in sub-figure 5.11a there is the server application UI for *Liquid Museum*, while in sub-figure 5.11b there is the client application UI for *Museum Client*. In this specific application I used only few contents to give some examples, but this app can be easily extended to be used in the real world in a real museum guided tour. All the contents I put on the Firebase Cloud storage are available for free, and they are related to a real museum, the Louvre Museum.

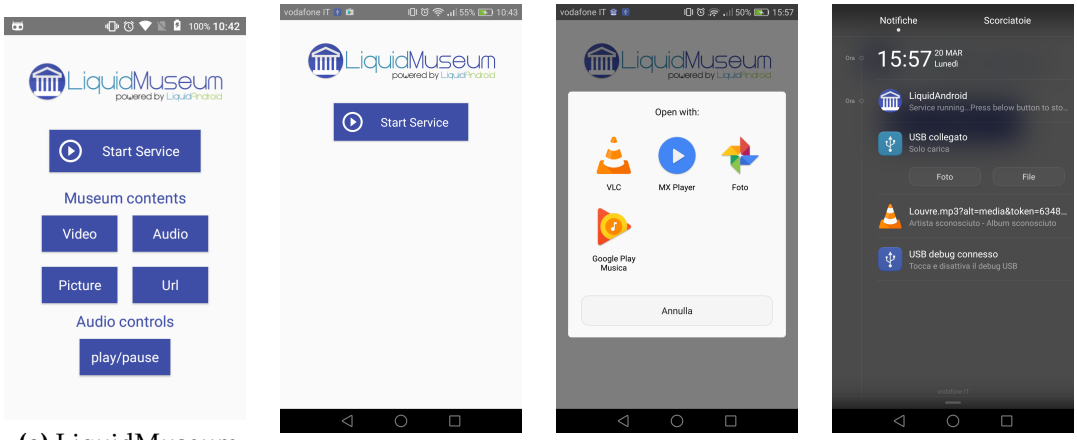
5.3.3 Liquid Museum Live tests

As I did for the Liquid Android application, I performed a live test with the Liquid Museum application. I installed the server component on a LG L70 device running Android Marshmallow (*API Level 23*), and the client component application on a Huawei P9, which also runs Android Marshmallow, and on a Google

Nexus 5, running Android Lollipop (*API Level 21*). The only requirements, for the system to work properly, are that all the devices involved must be connected to the same LAN and they must also have access to the Internet network to download contents from the Firebase cloud storage server. When the service is started in each device, the tourist guide can use its own machine to send contents to any device in range with the service already in execution. Furthermore, there is not the need to perform any set-up phase, in fact, if a new device becomes available at any time, the server application automatically detects it and sends commands, which are JSON-Intent, even to it.

Audio scenario

The most complete test scenario I performed, is the one in which the Liquid Museum server component sends to every client in range a JSON-Intent to play an audio file. In this case, the guide holding a smartphone and executing the server component of my distributed application, by pressing the *Audio* button, in sub-figure 5.11a, send a JSON-Intent containing the link to the file to be played in all the clients connected. When the button is pressed, the Liquid Museum application automatically finds clients in range, prepares a regular Android intent, converts it into a JSON-Intent, using my *Intent Converter*, and then sends the message to every client it finds. Devices running the client application, *Museum-Client* receive the JSON-Intent, reconvert it into an Android intent to be executed, and then trigger the usual standard Android intent resolution mechanism. At this point, tourists who are using the client application, can select with which application to execute the intent. For testing purposes, I used the *VLC media player* application for android, because it is free and I think is one of the most functional and complete multimedia player app which can be installed in Android systems. The process ends when the user selects the right application in the chooser and then the client device plays the audio file.
At this point, I want to focus the attention on the possibility to remote control the playback of the audio file from the server application. By pressing the *play/pause* button, the tourist guide can pause, and also resume, the playback in all the devices connected. When this button is pressed, the server application acts as a remote controller by simulating the pressure of a media key button. To do this, in Android it is possible to send a *broadcast intent* emulating the pressure of a physical button on the keyboard. For this reason, when the intent arrives to the client it is necessary to start it by calling the *sendBrpadcast(intent)* function, so the server must declare its intentions by adding the extra I have defined in Section 4.2.2 : having as fields **key**:*"andorid.intent.extra.LIQUIDMETHOD"*, **type**:*"string"* and **data**:*"BROADCAST"*. In this way, the client application, once the original Android intent object is rebuilt, can handle it in the right way, by calling the correct system method. Figure 5.12 shows some screenshots of the execution of this example, in particular, the sub-figure *a* is taken from the LG L70 running the server application component, instead all the other sub-figures are taken from the Huawei P9 running the client application component.



(a) LiquidMuseum **(b)** MuseumCLient **(c)** App chooser **(d)** VLC player

Figure 5.12: Liquid Museum running example screenshots

This section exhausts the topic, the next chapter deals with the conclusions which can be drawn from my dissertation and any future work which can be done to extend and make the work as complete as possible.

Chapter 6

Conclusions and Future Work

This chapter aims to analyze what has been done in order to give feedback and understand if the initial objectives have been reached.
The chapter is divided into two parts, the first is the conclusion of the work, with the analysis of what I have reached with my idea and my implementation; the second part is focused on future implementation.

6.1 Conclusions

Liquid Android middleware aims to enrich the Android OS functionalities by transforming it into a fully working distributed operating system. My work aims to differentiate itself from existing commercial solutions, seen at the end of the Chapter 2, in Section 2.4 offering both low-cost and easy-to-deploy solution using a distributed approach where mobile devices are actors and computing nodes.
Liquid Android middleware offers:

- automatic network construction and maintenance, when devices are connected in LAN,
- same intent resolution mechanism of the standard Android OS,
- intent conversion language and automatic tools to perform the conversions,
- same standard Android security mechanisms,
- development API, easy to use and to be extended to create other purposes distributed systems.

Theoretically speaking the goal was reached: having created the networking structure and having defined a new language suitable for the purpose to distribute intents over the network. In this way, the Android OS becomes a distributed OS over a LAN, because any task which can be done by creating implicit intent can be encapsulated in a message and sent, to be executed, on any device with the service active in the network.The research part was the study and the definition of mechanisms able to let different devices communicate using a standard

mobile operating system. I have defined the steps needed to perform this kind of extension with any existing mobile operating system. My steps, in Chapter 4 can indeed be used to create similar solutions for other kind of devices. The creation of the middleware and of reliable connections was more part of the implementation, where I did not introduce innovations in the used technologies. That is because reliable and powerful technologies assure the right behavior of the whole architecture.
Tests performed in this work aimed to stress the whole system simulating a common scenario with a small number of devices involved: they show good overall performance of the framework and its good behavior in terms of scalability and dynamicity. Real Android devices behaved as expected under test conditions and the test app built on purpose was always fully usable.
I think the work which has been done is positive, interesting and promising: the idea of having distributed mobile operating systems to let users use their devices in an environment as if they were a single big device, by sending tasks to any of them connected with my service, is certainly a good feature nowadays in which these kinds of devices are becoming more numerous and less expensive. Moreover it is possible to exploit such a system, to take advantage of the increasing computing capability of mobile devices, and let them cooperate respecting standard OS mechanisms.

6.2 Future Work

In this section I would like to briefly report and talk about the future developments which can be done on my work, in order to extend and make the framework as complete as possible. My thesis aims to find a unequivocal and uniform way to extend mobile operating systems to give them distributed system functionalities. In particular, I want to let any kind of device in a network cooperate with each other to perform tasks. Obviously, it was not possible for reason of time, and complexity to develop a system to be put on the market nowadays, compatible with all the devices. Thus I selected Android devices because they are the most common, cheap, and they use an open source OS. My work can be part of a more complex and complete architecture which can be widely adopted. Following the very same steps I have done, it would be possible to define a standard language definition to let any mobile devices communicate and cooperate under the same LAN. My JSON solution can be extended and easily used to adapt also IOS and Windows Phone similar intent mechanism, to reach the global goal.

Figures Copyright

Chapter 2: State of Art

Figure 2.2 is taken from [3] and is reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

Figure 2.3 is taken form [4] and reproduced under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Germany license.

Figure 2.6 is taken from [11], © 2007 Pearson Education. Inc.

Figure 2.13 is taken form [15], © 2002 by Academic Press.

Figure 2.14 is taken from a GitHub repository with an open source license.

The images that are not explicitly listed do not need any additional copyright references, or have been made by me and no copyright is needed.

Bibliography

[1] James Vincent. *99.6 percent of new smartphones run Android or iOS*. 2017. URL: http://www.theverge.com/2017/2/16/14634656/android-ios-market-share-blackberry-2016 (cit. on p. 1).

[2] Verge Staff. *Android: A visual history*. 2011. URL: http://www.theverge.com/2011/12/7/2585779/android-history (cit. on p. 5).

[3] Android developer. *Platform Versions*. 2017. URL: https://developer.android.com/about/dashboards/index.html#Screens (cit. on pp. 6, 85).

[4] Lars Vogel. *Android development with Android Studio - Tutorial*. 2016. URL: http://www.vogella.com/tutorials/Android/article.html (visited on 06/20/2016) (cit. on pp. 7, 8, 85).

[5] Greg Kroah-Hartman. *Linux Kernel in a Nutshell*. O'Reilly Media, 2006, p. 3 (cit. on p. 8).

[6] Android developer. *Intents and Intent Filters*. 2017. URL: https://developer.android.com/guide/components/intents-filters.html#Types (cit. on p. 10).

[7] Android developer. *Intents and Intent Filters*. 2017. URL: https://developer.android.com/guide/components/activities/index.htmls (cit. on p. 10).

[8] Android developer. *Application Fundamentals*. 2017. URL: https://developer.android.com/guide/components/fundamentals.html (cit. on pp. 10, 12).

[9] N. Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, 2014. ISBN: 9781593275815. URL: https://books.google.it/books?id=y11NBQAAQBAJ (cit. on p. 10).

[10] Android. *Security-Enhanced Linux in Android*. 2017. URL: https://source.android.com/security/selinux/ (cit. on p. 12).

[11] A.S. Tanenbaum and M. Van Steen. *Distributed Systems Principles And Paradigms 2Nd Ed.* Prentice-Hall Of India Pvt. Limited, 2010. ISBN: 9788120334984. URL: https://books.google.it/books?id=Fs4YrgEACAAJ (cit. on pp. 13, 19, 20, 85).

[12] M. Frans Kaashoek Jerome H. Saltzer. *Principles of Computer System Design*. Morgan Kaufmann, 2009. ISBN: 9780123749574 (cit. on p. 14).

[13] *The Java® Language Specification*. Addison-Wesley, 2005. ISBN: 0321246780 (cit. on p. 21).

[14] Computer Weekly. *Write once, run anywhere?* 2002. URL: http://www.computerweekly.com/feature/Write-once-run-anywhere (cit. on p. 21).

[15] K.L. Calvert and M.J. Donahoo. *TCP/IP Sockets in Java: Practical Guide for Programmers*. The Practical Guides. Elsevier Science, 2011. ISBN: 9780080568782. URL: https://books.google.it/books?id=lfHo7uMk7r4C (cit. on pp. 21, 85).

[16] S. Cheshire and D. Steinberg. *Zero Configuration Networking: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, 2006. ISBN: 9780596101008. URL: https://books.google.it/books?id=--R3jxPwQhUC (cit. on p. 23).

[17] Marshall Brain and Stephanie Crawford. *How Domain Name Servers Work*. 2011. URL: http://computer.howstuffworks.com/dns.htm (cit. on p. 23).

[18] Apple developer. *Bonjour for Developers*. 2017. URL: https://developer.apple.com/bonjour/ (cit. on p. 23).

[19] sourceforge. *jmDNS*. 2011. URL: http://jmdns.sourceforge.net/ (cit. on p. 23).

[20] Android developer. *Using Network Service Discovery*. 2017. URL: https://developer.android.com/training/connect-devices-wirelessly/nsd.html#teardown (cit. on p. 24).

[21] F. Büsching, S. Schildt, and L. Wolf. "DroidCluster: Towards Smartphone Cluster Computing – The Streets are Paved with Potential Computer Clusters". In: *2012 32nd International Conference on Distributed Computing Systems Workshops*. June 2012, pp. 114–117. DOI: 10.1109/ICDCSW.2012.59 (cit. on p. 24).

[22] Jan Matlis. *Sidebar: The Linpack Benchmark*. 2005. URL: http://www.computerworld.com/article/2556400/computer-hardware/sidebar--the-linpack-benchmark.html (cit. on p. 24).

[23] BOINC. *Open-source software for volunteer computing*. 2017. URL: http://boinc.berkeley.edu/ (cit. on p. 25).

[24] HTC. *PLUG IN. BE A PART OF THE FUTURE.* 2017. URL: http://www.htc.com/us/go/power-to-give/ (cit. on p. 25).

[25] Inhyok Cha et al. "Trust in M2M communication". In: *IEEE Vehicular Technology Magazine* 4.3 (2009), pp. 69–75 (cit. on p. 35).

[26] André B. Bondi. *Characteristics of scalability and their impact on performance*. WOSP '00, 2000. ISBN: 158113195X (cit. on p. 36).

[27] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: (1978) (cit. on p. 36).

[28] Android. *Intents*. 2017. URL: https://developer.android.com/reference/android/content/Intent.html (cit. on p. 47).

[29] Marshall Cline. *C++ FAQ: "What's this "serialization" thing all about?"* 2015. URL: https://web.archive.org/web/20150405013606/http://isocpp.org/wiki/faq/serialization (cit. on p. 47).

[30] W3C. *Introducing JSON*. 2007. URL: http://www.w3schools.com/xml/xml_soap.asp (cit. on p. 48).

[31] Android developer. *Running a service in the foreground*. 2017. URL: https://developer.android.com/guide/components/services.html#Foreground (cit. on p. 62).