Politecnico di Milano

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

Liquid Android: A Middleware for managing Android Intents in a Distributed Net over Wi.Fi

Relatore:

Prof. Luciano Baresi

Tesi di Laurea di:

Marco Molinaroli

Matricola:

837721

Anno accademico 2015–2016

Ringraziamenti

Ringraziamenti.

Milano, April 2017 Marco

Sommario

Parole chiave: PoliMi, Tesi, LaTeX, Scribd

Abstract

Text of the abstract in english. . .

Keywords: PoliMi, Master Thesis, LaTeX, Scribd

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Nowadays mobile devices have changed the way we approach to technologies, they are powerful enough to do most of the things we need in a fast and efficient way, without the need of use of 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 [23].
Many people have multiple mobile device for personal use, and for the reasons discussed above would be great if people can 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 do not need special developer licenses to build applications. So in this work 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 which 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 kind 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 limits of Android is that the system was designed to run on the top of a virtual machine and each application which can be ex-

ecuted starts a Linux process which has its own virtual machine (VM), so an application's code runs in isolation from other apps. This technique is called *"app sandboxing"* and it is used to guarantee an 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 communicate the various component of the applications and the operating system itself : 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 do 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 a devices to another one.
In a world where computers are everywhere and can do almost everything and can communicate among them in different but efficient ways, the fact that android devices are not able to easily exchange intents is such a major limitation to the android users. As we know our world is fast moving to a world of *"ubiquitous computing"* where there is no more 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 great if they can be used together in a smart way. The aim of this thesis work is to study enough 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 net connected in a LAN. 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 such as they were one single big device.
Each sentence or technology, that may appear not clearly explained here for the reader, is further discussed and clarified in next chapters.

1.2 Outline

The thesis is organized as follows:

In the second chapter the state of art is described: a full overview on current technologies, ideas and issues is provided. The chapter starts presenting the Android operating system with a brief history of versions. Then a deep presentation of Android's framework component is give to the reader, including security model and connectivity functionalities. The chapter continue describing what is a distributed system, listing its main challenges, properties and its working mechanism such as the communication models, and architectural patterns. The final section explains the term *Liquid* *computing*, presenting some existing technologies which can be useful to understand the problem an then the proposed solution and development.

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 are provided some restriction, considering only devices in which the developed system could be installed. The chapter continue with the full description of the problem, the main idea and also a working scheme of the component to be developed. Are then presented problematic scenarios to be studied, 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

In the fifth chapter

In the sixth chapter

Chapter 2

State of the Art

2.1 Android OS

As already mentioned in 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 kind of devices.
In this section i want to give to 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* launched in the United States [20].
At that moment the company of mountain view, Google, felt the need to create a new operating system which was able to 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 ease 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 version of the OS and with the help of the market growth Android has become a complete os. In the table below there is a brief description of the various distribution 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 an higher *API level* can not be executed in a device running lower versions of the operating system. This is a second major limitations for the *"Android ecosystem"*, moreover as mentioned before, the Android OS is released under an open source license, which is great for the developer, but which prevents Google to provide 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.

Table 2.2: Android OS versions fragmentation

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%

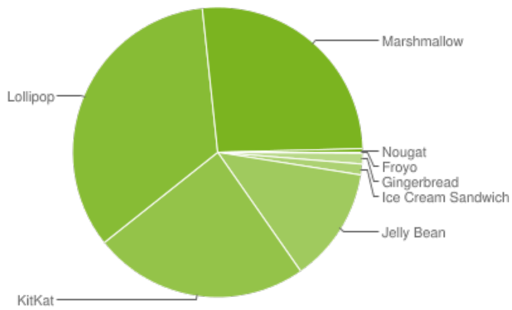

Figure 2.2: Android OS fragmentation chart

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 [11].

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.

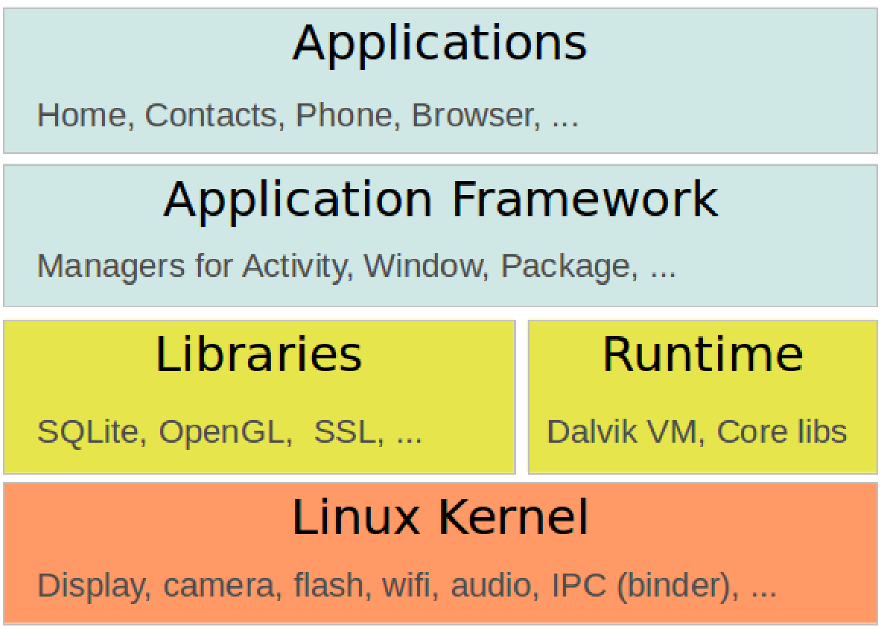

Figure 2.3: Android OS 4 layers

The OS can be divided into the four layers as depicted the Figure 2.3. An Android application developer typically works with the two layers on top to create new Android applications [24].

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 [17]. 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 version. It is stack based and it works by converting using a *just-in-time (JIT)*, 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, web browsing. [24]. 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 2.1.2 the Android application framework *("AppFramework")* is the core of the Android *development API*. It contains useful and needed components to build native apps.
The main components with which each application is composed are:

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)*. Acconrding 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 represents 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 deeper details that services are component which performs operations in background. As for the activities, services are initialized through intent and in the same way they describe the service to start and carries 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.

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 by 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 show a specified location on a map [9].

The 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, let 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. [10]. An activity is the

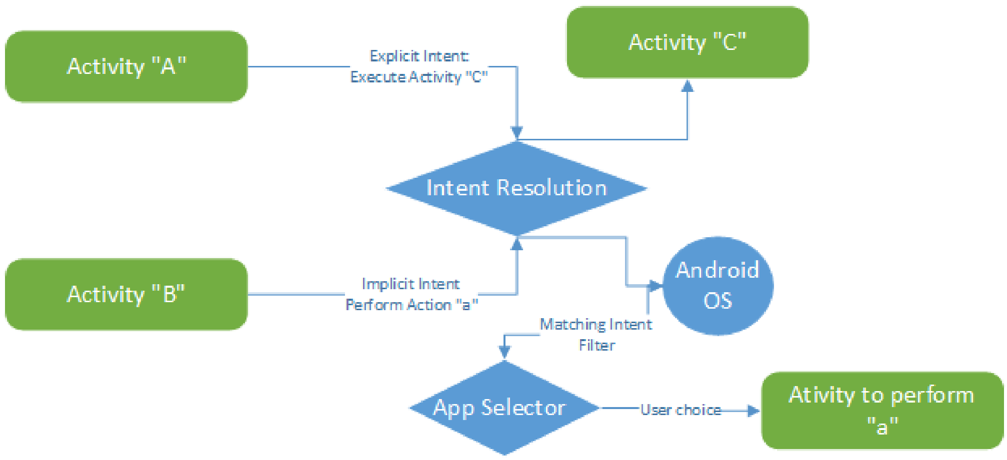

Figure 2.4: Intent resolution mechanism

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 aren't currently running [8].

2.1.4 Security

As described in 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 sensible data form malicious guys.
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 can not access another user's file unless explicitly granted permission. Android takes advantages of this user isolation, considering each application a different user provided with a dedicated *UID (User ID)* [14] 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:

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 application 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 *permission*. Applications can request permissions by listing them in a configuration file called *android manifest*. In Android 5.1 and earlier versions permission 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 permission are asked the first time that an application need them, and when are granted they can be revoked manually in the OS settings for that specific application.

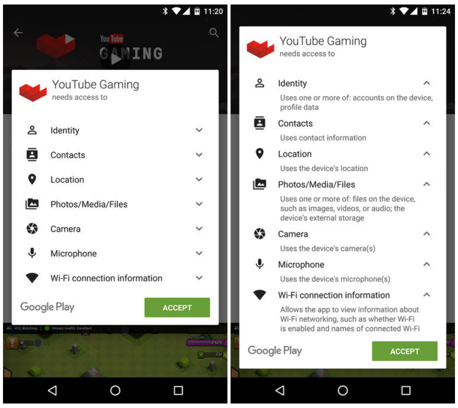

Figure 2.5: Android 5.1- permission example

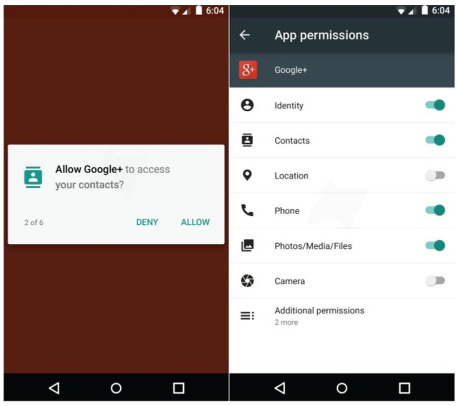

Figure 2.6: Android 6.0+ permission example

Figure 2.7: 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 to only those specified in the associated policy. In this way, the process cannot overwrite data and system settings outside of the specific raw block device [1].

2.1.5 Connectivity

As already amply 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 whit 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 to 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 one way or the other the autonomous components need to collaborate [21].
In Figure 2.8 it is possible to see how can be structured a distributed system: a the top we have the real distributed application, which is the final interface to be

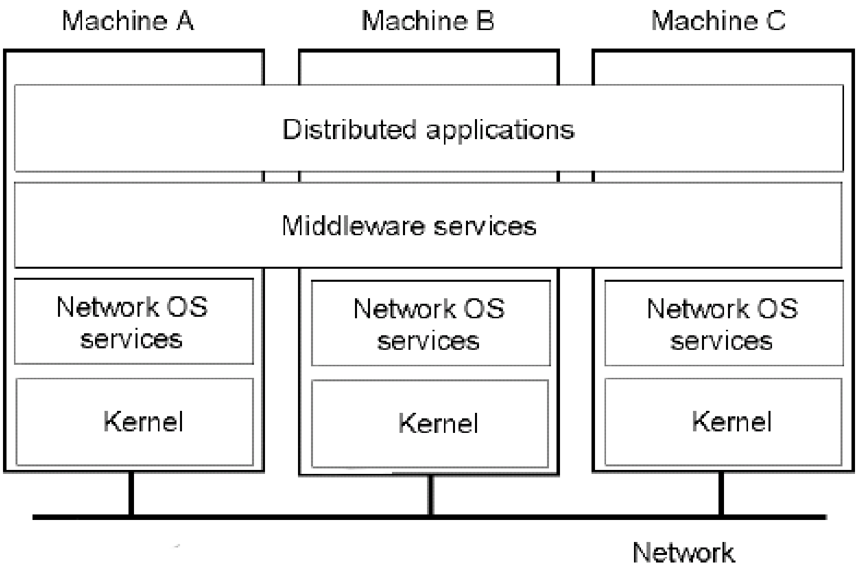

Figure 2.8: Distributed system structure

used, under which it is possible to have different combinations of services used to make communicate different machines that may use different operating systems. 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 act as intermediaries 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 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, is 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.

- Security, is crucial in many areas of computer science and specially in distributed systems, where data are exchanged by a several number of machines.

- Scalability, is the ability to easily increase the size of the system in terms of users/resources and geographic span.

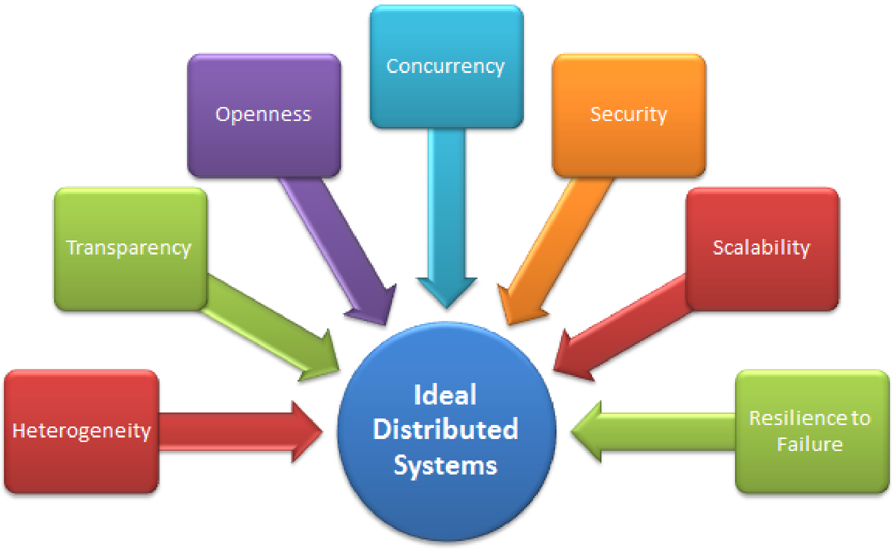

Figure 2.9: Distributed system challenges

- Failure handling, is important because having different components working together to a common goal means that distributed system can fail in many ways. This raises some issue: it would be nice id distributed systems can 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 level are listed in Table 2.3

Table 2.3: Transparency levels

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

2.2.3 Comunication Model

There are, in distributed system literature, some well known techniques to let communicate machines, programs and components. 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 [16]. As described in Figure 2.10 RPC provides the localization

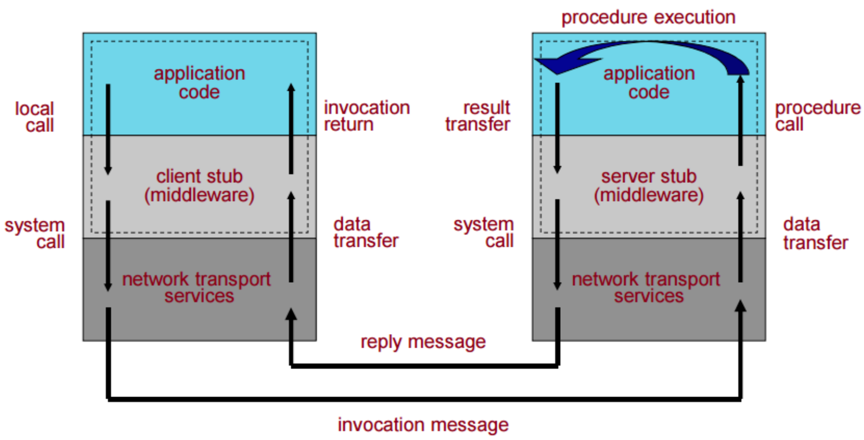

Figure 2.10: RPC in detail

of the code to be executed exploiting the network transport services, create a message which can be serialized and transferred over a standard network protocol and then provides methods to de-serialize the message and convert in 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 communicate object oriented (OO) programming languages. The Figure 2.11 shows in detail how RMI is supposed to work. Like RPC, RMI uses an IDL which is designed to support OO programming languages features such as inheritance and exception handling.

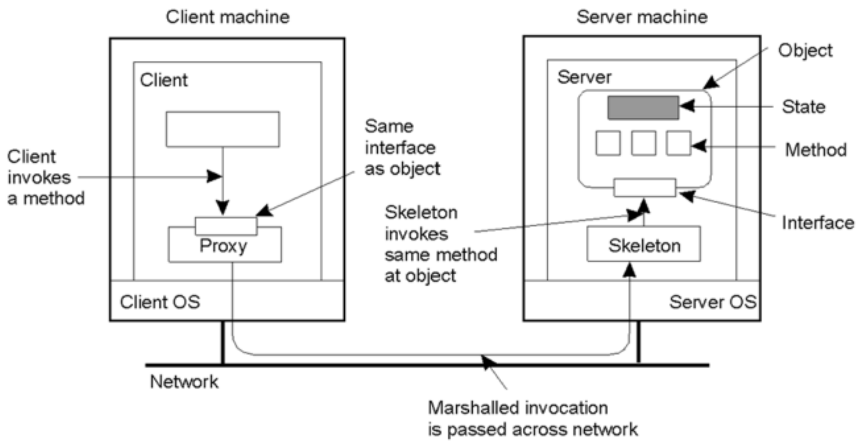

Figure 2.11: RMI in detail

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	• centered around the notion of message/event
• point to point communication	• multipoint support
• designed for synchronous communication	• usually asynchronous
• high coupling between the caller and the callee	• high level of decoupling

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

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

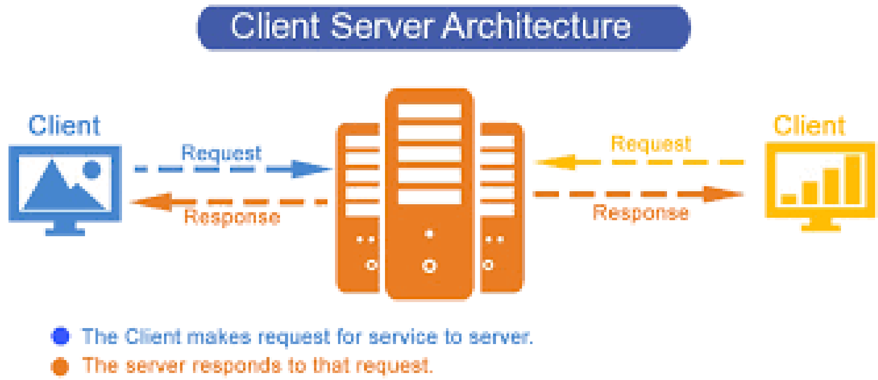

Figure 2.12: Client server architecture

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 clients 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 kind of systems is message oriented or through RPC.

Peer-to-Peer (P2P)

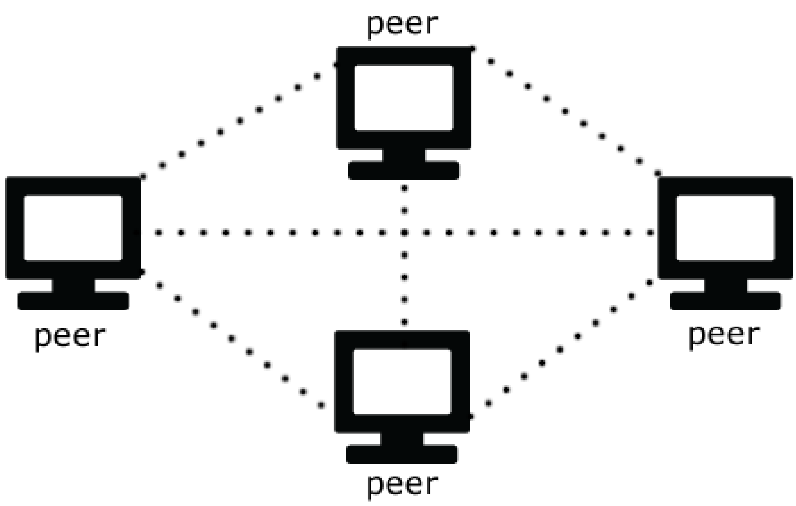

Figure 2.13: P2P architecture

P2P is a fully distributed architecture which in contrast to client-server has not a centralized service provider. Peers are both clients and servers themselves, P2P promotes sharing of resources and services trough 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.

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

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 they are interested to be notified about. 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.

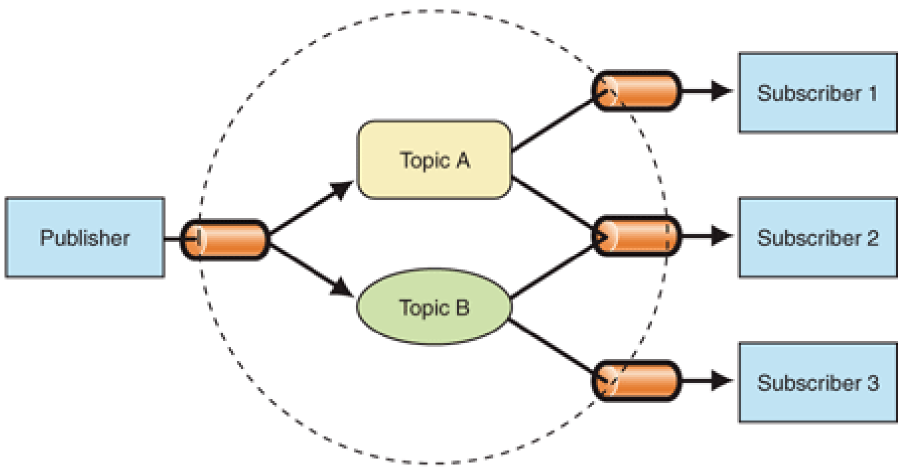

Figure 2.14: Publish-subscribe architecture

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, exactly the 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 e 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* such that they never change during the lifetime of an entity, are unique, and can not 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:

- *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 [21].

- *Structured naming* are composed from simple, human-readable names, not only file naming,but also host naming on the Internet follow this approach,in fact, flat names are good for machines, but are generally not very convenient for humans to use [21].

- *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 [21]. 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 in 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 here are not reported any detail of name resolution systems, but only basic naming notions to understand author's design choices in solving the thesis problem.

2.3 Technical Background

This section has the aim to give to 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 at the first device and offer to take over that task. In other terms liquid computation is a sort of what is called *ubiquitous computing* which is a model of man-machine interaction in which information elaboration is integrated in everyday objects.

Examples

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

- Apple continuity, is 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 email 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 automatically emails and for example is possible to start writing an email on a desktop pc and then completing and sending that email on a smartphone.
- Microsoft One Drive sync is a system, developed by microsoft to allow users to synchronize file 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 code.
Java is a known general-purpose computer programming language that is concurrent, class-based, object-oriented [22], and specifically designed to have as

few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA) [25], meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.
The *Java Development Kit (JDK)* includes many utility libraries, useful to develop any kind of application, I want to focus attention on network programming libraries to be used when developing distributed system using Java code. 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 [5]. Different types of sockets correspond to different underlying protocol suites and different stacks of protocols within a suite. In Figure 2.15 it is shown the

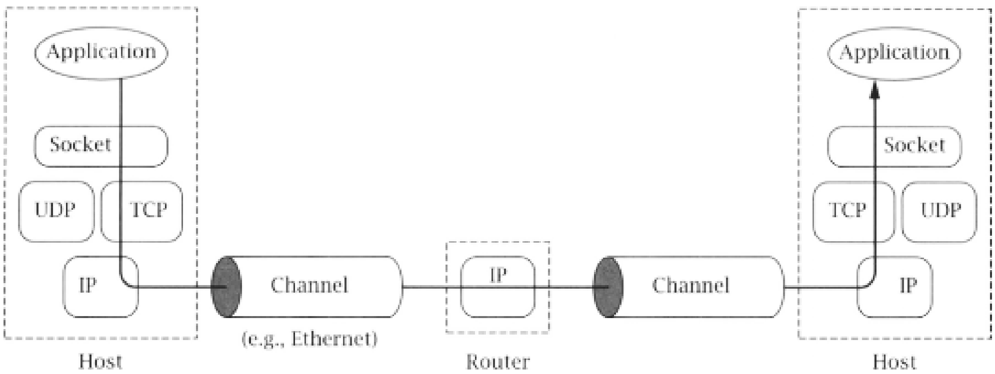

Figure 2.15: TCP/IP sockets

working mechanism of a *TCP/IP socket*: the application exploiting the socket communicate 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 is very easy to implement a client/server simple application using TCP/IP sockets. The following snippet of code are provided to give a concrete Java client/server application using sockets.

Listing 2.1: DateServer example

```
package ex.networking;

import java.io.IOException;
import java.io.PrintWriter;
```

```java
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;

/**
 * A TCP server that runs on port 9090.  When a client connects, it
 * sends the client the current date and time, then closes the
 * connection with that client.
 */
public class DateServer {

    public static void main(String[] args) throws IOException {
        ServerSocket listener = new ServerSocket(9090);
        try {
            while (true) {
                Socket socket = listener.accept();
                try {
                    PrintWriter out =
                        new PrintWriter(socket.getOutputStream(), true);
                    out.println(new Date().toString());
                } finally {
                    socket.close();
                }
            }
        }
        finally {
            listener.close();
        }
    }
}
```

Listing 2.2: DateClient example

```java
package ex.networking;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;

import javax.swing.JOptionPane;

public class DateClient {
```

```java

    /**
     * Runs the client as an application.  First it displays a dialog
     * box asking for the IP address or hostname of a host running
     * the date server, then connects to it and displays the date that
     * it serves.
     */
    public static void main(String[] args) throws IOException {
        String serverAddress = JOptionPane.showInputDialog(
            "Enter IP Address of a machine that is\n" +
            "running the date service on port 9090:");
        Socket s = new Socket(serverAddress, 9090);
        BufferedReader input =
            new BufferedReader(new InputStreamReader(s.getInputStream()));
        String answer = input.readLine();
        JOptionPane.showMessageDialog(null, answer);
        System.exit(0);
    }
}
```

Comments in the example gives the full explanation of how simple client/server Java application using TCP sockets works.

Java RMI

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

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.

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.

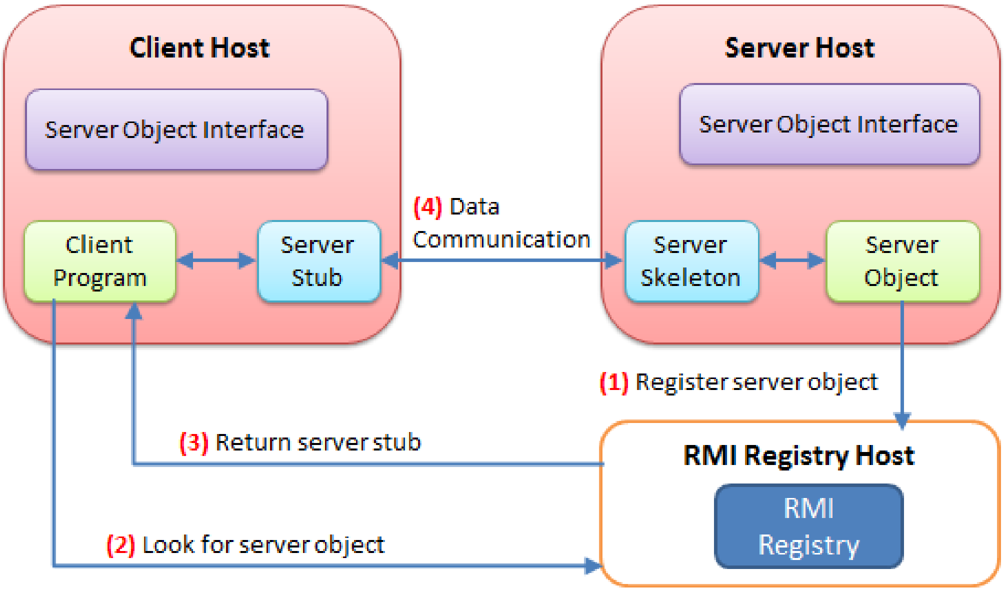

Figure 2.16: Java RMI structure

The aim of this technology is to let users easily connects 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 users to connect a printer or locate streamed music as it is for him to turn on light bulb [7].
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 [7]. 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 [4]. 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 of a connection the device that want to use the service, automatically, by finding it with Zerconf, knows connection variables , resolved by the protocol.

Implementations

Zeroconf is paradigm and its component 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 [13].

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 [12].

Chapter 3

Problem Analysis

In this chapter the specific problems of this work will be detailed and analyzed, explaining what are the limits 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 introducing this thesis work in 1.1, I studied in deep the Android operating system to find, and later implement a concrete solution, to the problem I will define and describe in dept in this section. All the work done by me is focused on Android because every mobile operating system is different to each other and has proprietary working mechanism 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 using for example 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. In the previous chapter, number 2, I have defined Android OS working mechanism and components, pointing out the main focus on intent generation and resolution mechanism. I have then defined in deep what a distributed system is and should be, explaining connection mechanism architectures and properties.
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's ideas 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 can not 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 task 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 themselves to cooperate. Solutions are often partial or proprietary and closed, even if some useful solutions exist.
The idea is let android devices collaborate and cooperate in a *Liquid environment* 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 Wi-Fi 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 am trying to define with more precision the problem I am going to face: which 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 2.1.5 Android devices are built to be connected to the Internet and most of them comes with a WiFi chip integrated. 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 version 4.4 (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 the Table 2.2, with this choice it is possible to cover the 84% of the active Android devices.
Having done these clarifications, now I am defining 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 work. As already said the Android OS is a pretty closed system itself, the intent resolution mechanism shows how it is difficult to let communicate various components inside a single device. On the other hand it is equally true that Android devices are real powerful modern computers and would be great 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 concrete, 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 send 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.
In Figure 3.1 is presented how such a system should work once the net is up. The distributed system in figure is a simplification of what the middleware for distributing intents will do. The architecture will communicate using standard android networking messages that are build on top standard protocols of the ISO/OSI stack.

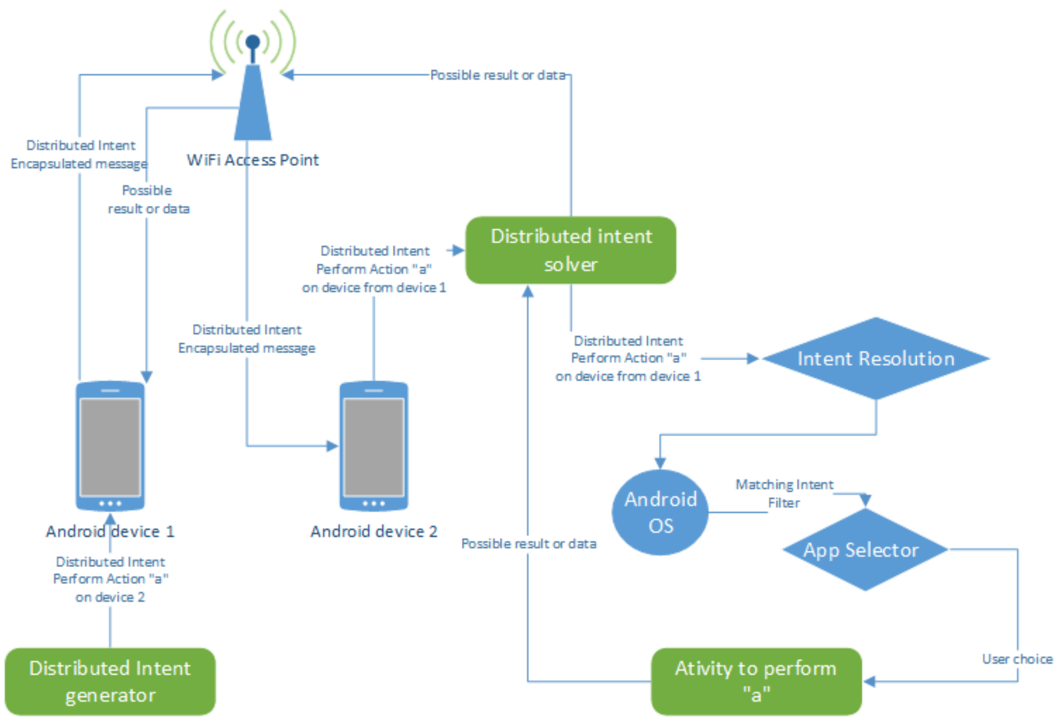

Figure 3.1: Distributed intent resolution

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, must be compliant with all the requests of *M2M (Machine-to-Machine) communication*. This type of communications is a constraint of my work and are explained in the next section 3.5.Another important point is 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 yet 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 resolution. What I'm doing is create a system to spread any kind of implicit intent and let the OS react as usual to perform the required action. It is not even marginal the choice of the type of network to be used in such a system. 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 this property dynamically react to continuous changes.
Next sections will properly define all the constraints of the given problem and propose a solution that fulfills them all.

3.4 Probelm scenarios

As already anticipated with the definition of the problem, the aim of this thesis 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 working middleware

In the best case, the result to be achieved would be a single Android APK, to be installed on devices as background bunch of services acting as a middleware. Liquid Android services which providing a communication interface can listen 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.
The 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, let the user of the system select on which available device execute the task, build and spread the the distributed intent message on the LAN, and when the message arrives at the other device the middleware transform 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 generate 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.

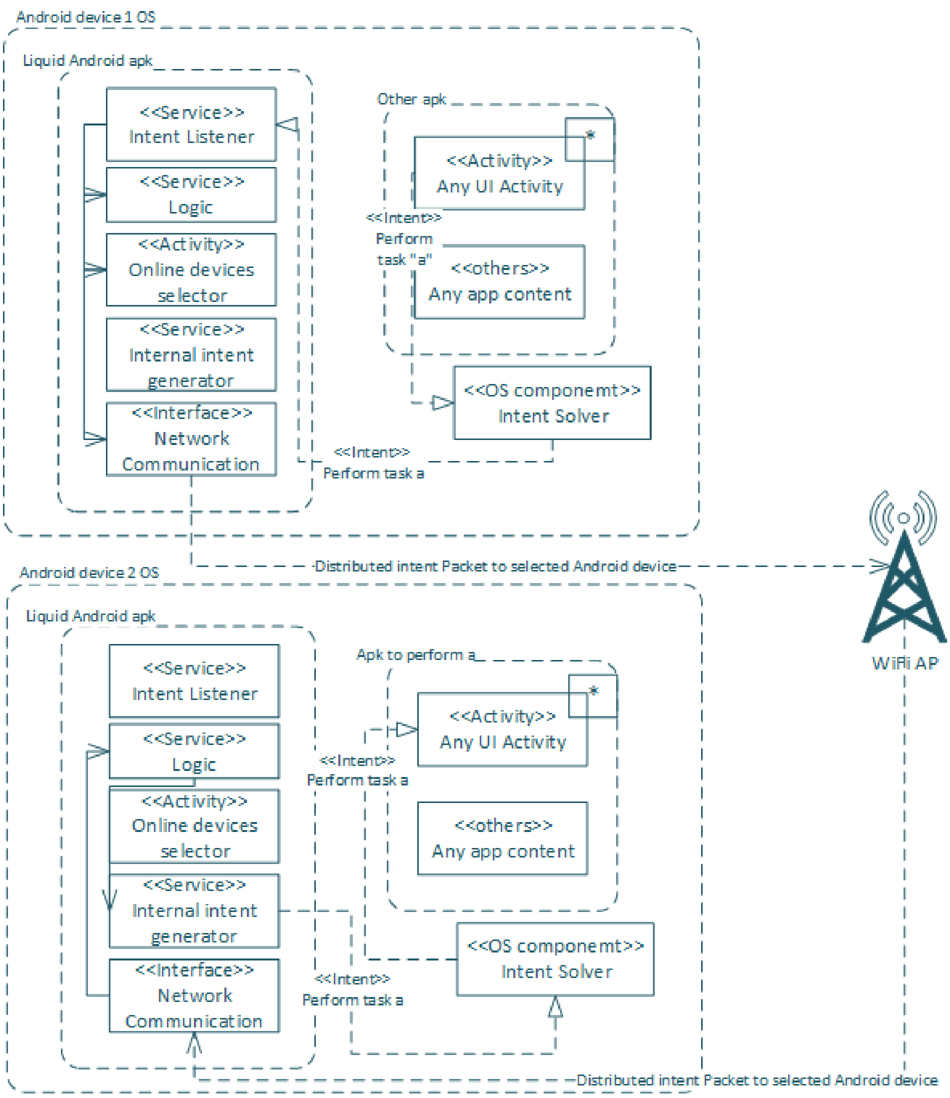

Figure 3.2: Liquid Android working as stand alone middleware APK

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 Android components. For these purposes it is necessary to produce accurate documentation for developer 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

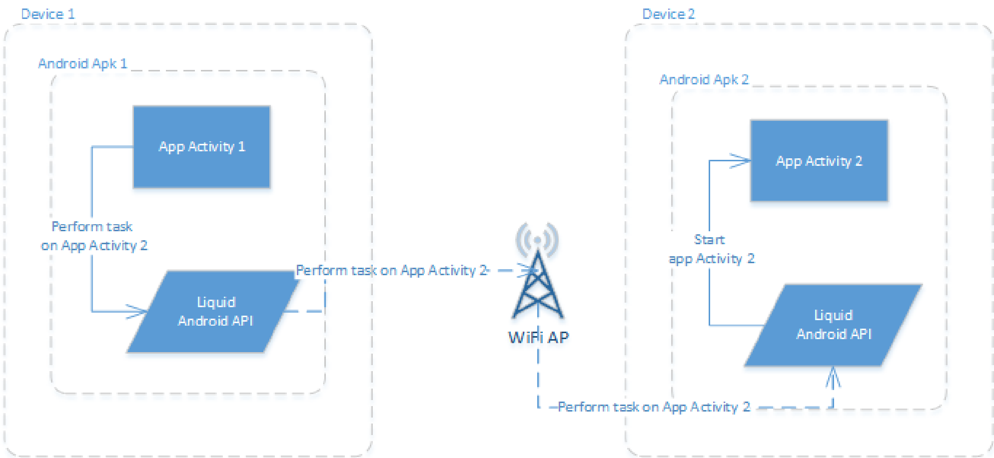

Figure 3.3: Liquid Android API working example

Android API which let 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

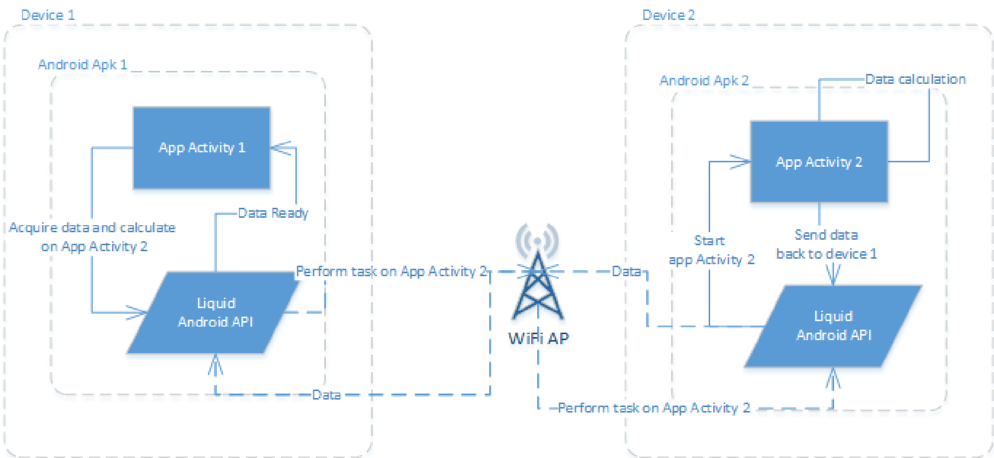

Figure 3.4: Liquid Android API data management example

Last interesting scenario tu 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 to be 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 ask to that device the results. In the Android environment there is not the concept of distributed file system, so data involved in such an application must be considered and efficiently exchanged between devices.

3.5 Constraints

In this section I would like to list 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.
Here is the list:

- *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 [6]. It can be considered one of the fundamental enabling technologies of this thesis work, it permits object to communicate without humans being involved. In This type of communication the reader of the content is a computer, in this case are 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 those which do the *magic*. 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 the chapter 2, to be more precise in Table 2.3, a distributed system must be transparent at many level, 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 in another one to be deserialized and analyzed to be executed. Messages must be as easy as they can 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 is but also to implement easily other kind of native Android distributed system application. Having a modular structure facilitates the specialization of its component and make 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 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 is possible to perform action as the administrator in Android environment. Rooting Android devices let application 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 sensible 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 issue. 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 has not a fixed number of devices involved in. The chosen network architecture must be able to react according to the changes. Android devices are free to join or leave the network 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 [3].

- *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 [18]. 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 task in more than one device simultaneously.

The listed requirements, as already told in some of them, 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 to 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 am presenting 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 the development of the solution will be reported step by step, with a full use case. The chapter starts with a list of similar solutions already developed, highlighting the differences between them and my thesis work. The remaining part is composed of tree mains 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 relation was taken into account when I made my choice.
The real implementation of the valid solution is left for the next chapter.

4.1 Existing Solutions

Android distributed systems already exists as specific purposes application to be installed on multiple devices, what it is different to the aim of this thesis work is that these application are closed source projects that can not be reused to build other purposes systems and there is not a coherent framework, library or API to be used to easily build such systems. I want to give some examples pointing out the nice features have this native distributed Android systems.

4.1.1 Boincoid and HTC Power to Give

Boincoid and HTC Power to Give are Android application which aim is to exploit Android devices computation power to contribute to scientific discoveries by doing some task. The common idea is to have an Android distributed supercomputer which can handle heavy task and compute tons of data for larger purposes.
BOINC is an open-source software platform for computing using volunteered resources [2]. 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 [15].

4.1.2 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 with which enjoy the contents. The Plex Media Server either running

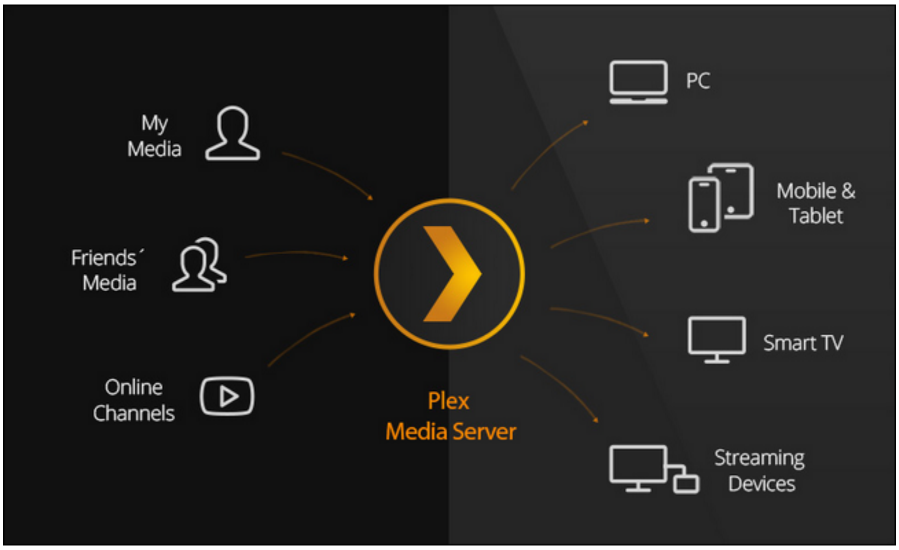

Figure 4.1: Plex Platform

on Windows, macOS, Linux, FreeBSD or 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 content and in addition it can search for Plex players in a LAN and send streamed content such as videos, movies or photo, to another player that can be also 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 foreground on the device.

4.1.3 Goolgle Home and Cast API

Google itself provides an application to control and send contents form an Android device, to some special devices in home network. *Google Home* is an Android application which can find, setup, manage and control, Google's home devices like the *Google Chromecast*. In this way is easy to setup and control and Android distributed system in which user can sent multimedia contents and command to the Google Home devices in the LAN. For these reason 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 limited number of special purposes devices.

4.1.4 DroidMote and Remote control systems

If we consider the possibility to control remote devices in a LAN, there are actually many different kind of applications that can do that also in an Android environment. DroidMote is probably the most complete application to control remotely an Android device from another one. It is composed by 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 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 kind of systems are capable to generate local intents in remote devices over a LAN but in a completely different way from 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 solution only to the problem of remote control, they can not exploit distributed Android devices computation power, in fact in an environment like this Android devices are not cooperating to perform task but one of them is only controlling another one.

4.2 General Idea

To better explain what I consider solution for my work it is important to understand the playground to my work. 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 of 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 having clear in mind the problem and its structure it is possible to find the best possible solution. In particular it is possible to decompose 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 the 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 of resolving all the three sub-problems described above. The server application has also 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 are asking to the so called servers to complete 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 defined the main actors of the problem I am facing, the next step is to understand how they can interact and communicate. As previously described defining the problematic scenarios in the chapter 3, the Liquid Android middleware in the

Figure 4.2: Liquid Android working example

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. The Figure 4.2 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, in the picture, *Device 1* 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 to 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 it can handle that implicit intent, find the other devices in the net, in this case the so called *Device 2*, le the user choose with which device complete the task, convert the intent in a network message and send it to that device. Once the message arrived to the *Device 2* the Liquid Android middleware server application is responsible to translate again the received message into the starting implicit intent sent by the *Device 1* and to start 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 be also, once implemented, a solution for the second problematic scenario proposed in 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 3.4.1. The implemented version of the general solution could be also 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 buy other Android developers.
I would like now 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 fast prototype. The developer should see my framework as help for his work. The idea is to provide a ready to use service, with which it is possible 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 of configure anything. Users, once installed the middleware 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, let me call it *general theoretical problem* by dividing it as discussed above and providing the solution for each of them, taking into account also 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.

4.3 Theoretical solution, extending the Android OS

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.

4.3.1 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* everytime, 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 2.2.4, in peer-ro-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.

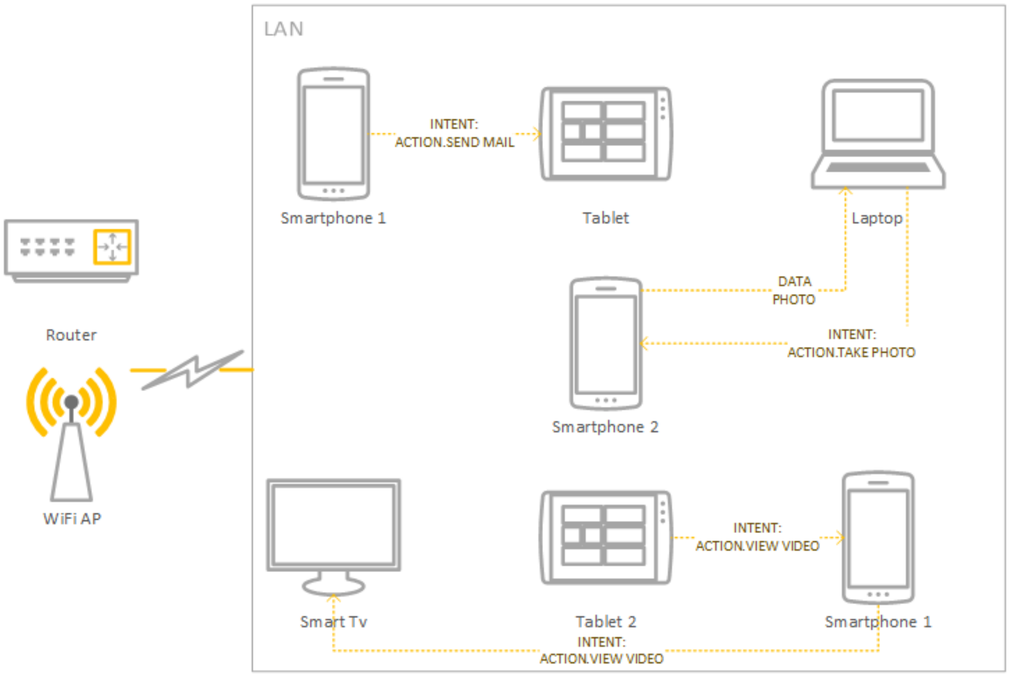

Figure 4.3: P2P Liquid Android network example

I think that an *unstructured P2P architecture* is the best choice for my system due to the fact that it do 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 4.3 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 2.3.3. Using a Zeroconf implementation to register and discover the service in the LAN will let the node to connect dynamically and in a easy way to the distributed system. The three main operation 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 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 communicate heterogeneous devices 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 changes frequently their connections variables, because of their nature to be 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 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, need only to know the service name and the chosen transport layer to find other nodes in the network.

Listing 4.1: Zerconf registration example

```java
public void registerService(int port) {
    // Create the NsdServiceInfo object, and populate it.
    NsdServiceInfo serviceInfo  = new NsdServiceInfo();

    // The name is subject to change based on conflicts
    // with other services advertised on the same network.
    serviceInfo.setServiceName("LiquidAndroid");
    serviceInfo.setServiceType("liquid._tcp");
    serviceInfo.setPort(port);
    ....
}
```

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*, I want that a node can decide whether 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 provide 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 completed the resolution process 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 requirements 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 model is highly influenced by the underlying physical network

4.3.2 Communication Model

4.3.3 Data management Model

Chapter 5

Case Study

Chapter 6

Conclusions and Future Works

Figures Copyright

Chapter 2: State of Art

Chapter 4: Solution

The images that are not explicitly listed are made by me and no copyright is needed.

Bibliography

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

[2] BOINC. *Open-source software for volunteer computing*. 2017. URL: http://boinc.berkeley.edu/ (cit. on p. 37).

[3] André B. Bondi. *Characteristics of scalability and their impact on performance*. WOSP '00, 2000. ISBN: 158113195X (cit. on p. 34).

[4] Marshall Brain and Stephanie Crawford. *How Domain Name Servers Work*. 2011. URL: http://computer.howstuffworks.com/dns.htm (cit. on p. 24).

[5] 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 p. 21).

[6] Inhyok Cha et al. "Trust in M2M communication". In: *IEEE Vehicular Technology Magazine* 4.3 (2009), pp. 69–75 (cit. on p. 33).

[7] 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. 24).

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

[9] Android developer. *Intents and Intent Filters*. 2017. URL: https://developer.android.com/guide/components/intents-filters.html#Types (cit. on p. 9).

[10] Android developer. *Intents and Intent Filters*. 2017. URL: https://developer.android.com/guide/components/activities/index.htmls (cit. on p. 9).

[11] Android developer. *Platform Versions*. 2017. URL: https://developer.android.com/about/dashboards/index.html#Screens (cit. on p. 7).

[12] Android developer. *Using Network Service Discovery*. 2017. URL: https://developer.android.com/training/connect-devices-wirelessly/nsd.html#teardown (cit. on p. 25).

[13] Apple developer. *Bonjour for Developers*. 2017. URL: https://developer.apple.com/bonjour/ (cit. on p. 25).

[14] 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).

[15] HTC. *PLUG IN. BE A PART OF THE FUTURE*. 2017. URL: http://www.htc.com/us/go/power-to-give/ (cit. on p. 38).

[16] M. Frans Kaashoek Jerome H. Saltzer. *Principles of Computer System Design*. Morgan Kaufmann, 2009. ISBN: 9780123749574 (cit. on p. 15).

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

[18] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: (1978) (cit. on p. 34).

[19] sourceforge. *jmDNS*. 2011. URL: http://jmdns.sourceforge.net/ (cit. on p. 25).

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

[21] 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. 12, 19).

[22] *The Java® Language Specification*. Addison-Wesley, 2005. ISBN: 0321246780 (cit. on p. 20).

[23] 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).

[24] 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 p. 8).

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