IDEA:
Runtime Collection of Android Data

Advisor : Luciano BARESI

Master thesis by :
Kostandin CAUSHI matr. 898749

Academic Year 2017-2019
Ringraziamenti

Giunto al termine di questo percorso universitario vorrei ringraziare il mio relatore, il professor Luciano Baresi, per l’opportunità offertami e per la sua disponibilità e il suo supporto nella realizzazione e stesura della seguente tesi.

Vorrei, in particolar modo, ringraziare la mia famiglia che mi ha sempre sostenuto e incoraggiato. Dedico questo importante traguardo ai miei genitori, per i loro immensi sacrifici e per le difficoltà che hanno dovuto superare, tutto per potermi offrire una buona istruzione che mi permettesse di crescere non solo professionalmente ma anche come persona. A mia sorella dedico invece l’aperitivo di questa sera in memoria dei nostri festeggiamenti post-esame, del suo sostegno morale e della sua pazienza in questi anni di convivenza.

Un ringraziamento anche a tutti i parenti e amici che mi hanno rallegrato le molteplici buie giornate da polimi.

Grazie
Abstract

Nowadays we find ourselves in a world where the use of portable devices (smartphone, tablet, smartwatch, etc...) has become essential for our daily life, work and social life. In fact, we have a very huge range of applications you can use in order to take advantage of the various services that multiple companies have to offer. To use these apps however, we obviously also need the hardware part, which is also very diverse and distinct from one another.

This takes on a positive aspect, on the client side, for the vast selection available, but negative, on the app developer side, for the compatibility problem. Actually every new application, before being made available to the public, it should be tested in all devices in order to ascertain that everything is working as expected, but this is not materially feasible given the huge amount of products.

The purpose of the thesis is to build a tool that allows various companies, that are app owners, to be able to collect data, at runtime and not, from the many available systems, analyze them, process them and finally extrapolate any problems. To be more specific, we focused on the study of Android devices, given their big numbers in both sales and available options. As far as the nature of the data is concerned, we have aimed at extracting information on the model (for cataloguing), on the hw (for performance) and on the ui (for design).

At the beginning it was thought to develop a third-part app that would fulfil this task but, for the reasons given in the following chapters, it was not possible to do so. There so the final solution was to build IDEA (Imperceivable Data Extractor for Android), a library to be integrated into your apps, including a service that runs in background and that based on the configurations inserted by the customer collects the requested data.
Sommario

Oggi ci troviamo in un mondo in cui l’utilizzo di dispositivi portatili (smartphone, tablet, smartwatch, ecc...) è diventato fondamentale per la nostra vita quotidiana, lavorativa e sociale. Abbiamo infatti un’ampissima gamma di applicazioni da poter utilizzare al fine di usufruire dei vari servizi che le molteplici compagnie hanno da offrire. Per adoperare queste apps però abbiamo bisogno ovviamente anche della parte hardware, anch’essa molto variegata e distinta l’una dall’altra.

Questo assume un aspetto positivo, lato cliente, per la vasta selezione disponibile, ma negativo, lato sviluppatori di app, per il problema della compatibilità. Infatti ogni nuova applicazione, prima di essere resa disponibile al pubblico, andrebbe testata in tutti i dispositivi al fine di appurare che tutto funzioni come previsto, ciò però non è materialmente fattibile data la numerosità dei prodotti.

Lo scopo della tesi è costruire un tool che permetta a varie compagnie, proprietarie di app, di poter raccogliere dati, a runtime e non, dai molteplici sistemi, analizzarli, elaborarli e infine estrapolarne eventuali problemi. Nello specifico ci siamo focalizzati sullo studio dei dispositivi Android, dato il loro ampio numero sia nelle vendite che nelle opzioni disponibili. Per quanto riguarda la natura dei dati invece abbiamo mirato all’estrazione di informazioni sul modello (per la catalogazione), sull’hw (per le performance) e sulla ui (per il design).

Si pensava all’inizio di sviluppare un’app indipendente che adempisse a questo compito ma, per i motivi riportati nei successivi capitoli, non è stato possibile farlo. La soluzione finale quindi è stata quella di costruire IDEA (Imperceivable Data Extractor for Android), una libreria da integrare nella propria app, comprensiva di un service che runna in background e che, basandosi sulle configurazioni inserite dal cliente, raccoglie i dati richiesti.
Contents

Ringraziamenti I

1 Introduction 1
   1.1 Context 1
   1.2 Goal 2
   1.3 Achieved Results 2
   1.4 Document Structure 3

2 Android Background 5
   2.1 OS Architecture 5
   2.2 UI Components 7
      2.2.1 Activity 7
      2.2.2 Fragment 8
      2.2.3 Click Listener 9
   2.3 "Back-End" Components 9
      2.3.1 Service 9
      2.3.2 Context 11
      2.3.3 Intent 11
      2.3.4 Broadcast Receiver 12
      2.3.5 SharedPreferences 12

3 Analysis on Collectable Data 13
   3.1 Device Info 13
   3.2 Performance 14
      3.2.1 CPU 14
      3.2.2 Memory Usage 15
   3.3 Logs 16
   3.4 UI 17
      3.4.1 Screen Info 17
      3.4.2 View Info 18
      3.4.3 Called Method 18
   3.5 Others 19
      3.5.1 Intents 19
      3.5.2 SharedPreferences 20
      3.5.3 Object Value 20
      3.5.4 Time lapse 21
4 State of the Art and Design Solution
4.1 State of the Art .............................................. 23
   4.1.1 Monitoring Performance Resources ......................... 23
      4.1.1.1 Which app hoggs CPU on your Android ............... 23
      4.1.1.2 AnotherMonitor .................................. 25
   4.1.2 CatLog ............................................. 25
   4.1.3 Execution Time ....................................... 26
   4.1.4 Monitoring UI Resources ................................ 27
   4.1.5 Related Work ......................................... 28
4.2 Design Solution ........................................... 28

5 Implementation .............................................. 31
5.1 Runtime Service ........................................... 31
   5.1.1 Configuration ......................................... 32
   5.1.2 Lock Device Status .................................... 33
   5.1.3 REST Call ........................................... 34
5.2 New UI Components ......................................... 35
   5.2.1 RuntimeActivity ....................................... 35
   5.2.2 RuntimeFragment ....................................... 38
   5.2.3 Listeners ............................................. 38
5.3 Basic Information ........................................... 39
5.4 Logs ....................................................... 40
5.5 Intents .................................................... 41
5.6 SharedPreferences .......................................... 42
5.7 Others ..................................................... 42

6 Validation ................................................... 45
6.1 Setup ....................................................... 45
   6.1.1 Devices ................................................ 45
      6.1.1.1 Huawei p10 Lite ................................ 46
      6.1.1.2 Huawei p9 Lite ................................ 46
      6.1.1.3 Samsung s10+ (VM) .............................. 46
   6.1.2 Applications ........................................... 47
      6.1.2.1 RuntimeCollector ................................ 47
      6.1.2.2 EasyLib ........................................ 48
6.2 Tests ....................................................... 48
   6.2.1 Basic Information ....................................... 48
      6.2.1.1 Test Cases ....................................... 48
      6.2.1.2 Results .......................................... 49
      6.2.1.3 Discussion ....................................... 49
   6.2.2 View .................................................. 50
      6.2.2.1 Test Cases ....................................... 50
      6.2.2.2 Discussion ....................................... 50
      6.2.2.3 Problem Detection ............................... 51
      6.2.2.4 Results .......................................... 52
Chapter 1

Introduction

1.1 Context

There is a huge widespread of mobile applications nowadays, in 2017 we had 178 billion app downloads and in 2018 194 billion [26] [6]. Of them 76 billions were made from Google Play Store, 30 billions from App Store and the remaining ones from third-party sources. By watching further statistics we can notice that Android has the 85.9% share of the global mobile market with more than 4.000 different devices and 2.5 billion monthly active users [3]. Moreover Android allows developers to create and share freely their applications and as consequence of this we can see that only Google Play Store has more than 2.7 million of apps [4]. On one side it allows mobile developers to be more independent and more creative in experimenting and distributing their projects and ideas, on the other side however it brings out a huge problem cause if they want to share they products, they have actually to make it compatible with the thousands of available and used devices. Up to now there’s no tool that allows to test applications on all hardware, the only way is to deploy them, share them and wait for users feedback. Obviously this is not the best solution, given that customers can give bad reviews and ratings, making their product less appealing.

There are 2 important aspects to consider:

- **UI**: the most significant part in app development cause it has huge impact on user experience. So once the application design is completed it is very important and useful to be sure to preserve the same look on all devices. The problem is that having this high number of smartphones and tablets with different sizes and shapes, we have also lot of distinct displays with different resolutions and dpis. So starting from the same sketched app layout, it will be shown in dissimilar ways.

- **Performance**: most devices use also different hw components, such as cpu, ram, memory, etc... so it’s relevant to know if an application can run fluently or not given some specific configurations and if the specs meet the requirements. A very slow and non-performing app is pretty much unusable.
1.2 Goal

The aim of this thesis is to create a tool able to collect runtime values from different apps, allowing service suppliers to analyse them and find potential bugs or possible improvements.

We focus on Android applications given the widespread they have, as we reported in the previous section. So given the chosen system we started at first searching for of all the possible collectable runtime values that the OS allows us to gather. For example: device information, such as brand name, model, hw configuration, etc... that will allow all software analysts to classify the received data and group the similar ones; cpu and ram usage, that will allow them to study the performance level; ui real-time layout, that will allow to analyse the real look of the application in a specific device with a specific display; logs, in order to have information on potential bugs, events or tags; variable values, in certain moments, in order to examine some unique behaviours; sharedPreferences, to understand which are the most used configurations and settings.

Next step is to understand how to do that, so how to collect these data. Some of them are very easy to retrieve cause Android OS makes them directly available through some APIs, others are harder to get together and others are no more obtainable cause access authorizations has been restricted. Finally, given that we have all these information, we want to build some UI flows that will exhibit the user experience, obtaining in this way some relevant statistics about the most used features or most visited parts.

1.3 Achieved Results

The innovations of this thesis are:

- Deep analysis on Android collectable values, given the fact that, for most of them, there is no official documentation available, so we will provide some tests and proofs that will demonstrate the obtained results, even starting from some informal articles and news.

- IDEA, a library that each application can import inside, which offers:
  - A configuration file where every customer can define the required data to collect, when to do that and how. They can also change them dynamically at anytime.
  - A background service that executes the above specified operations and stores temporarily the cumulated information, allowing in this way also to put together the customer experience flows. It elaborates all the needed data and after checking that the device is locked, in order to not affect cpu performance, it sends them to the specified server address.
– UI components (Activities and Fragments) that have to be used instead of the basic ones that Android offers in order to automatically send information about the real-time UI look, that can differ from device to device.

– Click Listeners (onClick and onLongClick), also these ones have to be used instead of the default available ones, in order to automatically send information about the customer interactions with the application, so basically all the touch events.

– all customer preferences, by accessing all the sharedPreferences files and by retrieving all their <key, value> pairs.

• Tests on real applications in order to prove: the usefulness of the library, meant as evidences of improvement in the use of the new available means and any proof of its effectiveness, and also its low performance impact on foreground processes.

1.4 Document Structure

The thesis is divided in the following parts:

• Chapter 2: provides information on Android OS, how it works and the available components that can be used (focusing on the needed ones employed to reach the goal).

• Chapter 3: examines the possible collectable data, how they can be retrieved and what are the potential restrictions.

• Chapter 4: analyses the available applications and tools, that accomplish some of the wanted tasks, and their limitations. Given that it will show the final solution adopted.

• Chapter 5: explains in detail the provided features and their implementations, presenting also some code snippets and logical schemas.

• Chapter 6: provides some test cases on real applications, reporting also outputs and performance levels.

• Chapter 7: summarizes the conclusions and future work.
Chapter 2

Android Background

This section provides more detailed information on Android OS, starting from showing who is in charge of the development of the "Core" part of the system and what are the possible extensions up to the analysis of the multiple components that make up the Android architecture, such as: UI (Activities and Fragments); Intents and Broadcast Receiver; SharedPreferences; Listeners; Context. These elements will be analysed in particular as they are the ones that are used to build our tool, so it is very important to provide some background information that allows the reader to understand the made implementations.

We will also give some architectural details in order to outline the various processes and features made available after years and years of developments and updates (first release September 23, 2018).

2.1 OS Architecture

Android is a mobile operating system based on a unix-like and open source software. It is an embedded system designed primarily for smartphones and tablets and consists of: a free part, called "Android Open Source Project (AOSP)", which is Android’s core source and is implemented by Google Inc and a group of developers known as Open Handset Alliance; a proprietary part, such as Google apps (ex. GooglePlay, Gmail, etc...) or customized apps from devices manufacturers [23]. This means that we have multiple products that share a common base part but different extensions, due to different brands implementation, different hardware, technology, etc... It’s an important aspect to consider as it introduces in our analysis some limitations. In fact, for example, each company decides on its won how to manage the internal data, such as the device information or some particular apps data, for example they can be stored in unlike locations or mapped to different key values.

In general it is a very powerful and versatile operating system that adapts very well to different types of configuration and interfacing. It is possible to divide it entirely into 6 main sections and about as many layers, as shown in the image below [24].
Starting from the bottom we can see that the system is based on Linux kernel, to be more specific on ARM hardware components, that supports a wide range of drivers such as display, camera, wifi, etc... and also provides a lot of functionalities as power and memory management and so on. Next layer contains some high level standard interfaces that companies can implement and handle so in a custom way the features offered by the low level driver. Android Runtime is the most important part of all cause it incorporates inside Dalvik Virtual Machine (DVM), that is a sort of Java Virtual Machine, but optimized for Android, that allows each app to run its own process through exploiting Linux core features like memory management and multithreading, in this way we can run multiple applications at the same time. It converts .java files into .Dex format.

Moving on we can see that it includes lot of libraries starting from some open-source Web browser engine, to SQLite database, to libraries that allow to play and record audio and video, to the SSL ones responsible for Internet security. Application Framework provides many high-level services and APIs, inform of Java classes, that can be used by developers in order to exploits the available functionalities. We have for example the most used managers like Activity, Package, Window, View, Content, etc...

Finally at the top we can find the installed applications, both the standard

---

1https://miro.medium.com/max/1126/1*ZjA-aKUTuoYXIUEQhbfcw.jpeg
incorporated ones and the others that we can subsequently download.

On the next subsections we are going to analyse the most important components related to our project, which are mainly covered by the Application Framework layer.

2.2 UI Components

We will now introduce the first components that are graphical by nature or are strictly related to them. We will start from the most important and most used of all: the Activity, that is the manager used to handle screens (ui layouts and logics); then we will move on to Fragments, which are gradually becoming more important in recent Android developments as they allow the management of multiple views within the same logical unit. Finally we will talk about Listener, equally important as it is associated with graphical buttons or in general interactive elements and allows to trigger operations downstream of user interactions with the application.

2.2.1 Activity

An Activity is basically a window that can be filled with multiple graphic elements so as to allow the user to interact with the device. Various view-layouts can be associated with it, although, in case there is more than one, Fragments are usually used (they will be described in the following sub-subsection). It manages so the organization and functioning of UI objects, their display and communication with the back-end part [1], specifically we can dynamically make changes to these ones at runtime, we can associate them with logics that are triggered when the user makes particular predefined actions and finally we can communicate with other classes and with the system.

An application usually contains more than one activity in order to manage different screens corresponding to different functionalities. Each of these, however, must go through the multiple execution stages (also shown in the figure below): we start from the creation, where the activity is actually added to the top of the stack and where the first initializations and bindings to the services are made, to then pass into foreground. At that point, depending on the user’s commands, another screen can be opened, pausing the previous one in background, or we can go back, destroying the current one in foreground and resuming the activity suspended that is saved at the top of the stack.

To use this type of component you need to create a class that extends the standard android Activity and in addition you have to override the methods corresponding to the needed states of the lifecycle so as to insert the various required operations at each step. It is common to call the ”setContentView()” in the onCreate() in order to set the xml layout at the time of component formation. Finally we have to declare the new activity in the Android Manifest to define in this way the application structure and the main elements that compose it.
2.2.2 Fragment

It represents a portion of user interface, both logically and graphically, like a modular section of an activity, in fact it is not just a layout section or just a group of controls but it can be considered as a sub-activity. It generally identifies a specific functionality and also has its own life cycle very similar to that of the Activity but it has in addition a very important state called `onCreateView()` which allows to associate a certain xml layout to the component.

Fragment must be embedded in activities, it cannot run independently of them. It is widely used nowadays as it can be reused in several points of the app implementation and as it is easy to manage dynamically, actually within the same activity we can at runtime create, delete or modify fragments and switch between them [11].

In order to use it we need to have an activity and inside it we have to instantiate all the needed fragments plus a Fragment Manager that will allow

---

2https://developer.android.com/images/activity_lifecycle.png
us to switch between them and so to switch between different screens. We must also reserve a small or large space in the layout of the activity to accommodate the view of the fragment.

2.2.3 Click Listener

In general an event listener is an interface in the View class that contains a single callback method. This method will be called by the Android framework when the View to which the listener has been registered is triggered by the respective user interaction with the item in the UI. Android offers so a collection of interfaces with callbacks that the developer has to implement by overriding the method [12].

An example are the OnClickListenr and OnLongClickListener that actually, as the name suggests, responds to click and long click events that the user for example can make by interacting with a button. In order to use them you can, by code, set the wanted Listener directly to the view component or you can just declare a method and use the IDE, ex. Android Studio, to associate that function to the item by specifying also the action that activates the triggering.

2.3 ”Back-End” Components

In this collection we consider all the components that do not correspond to the UI description given above, that’s the reason why we called them ”back-end” components.

We will start by describing the most important item of all, the Service, which among other things is also the fundamental element on which our tool will be based, as it allows you to work in the background at every moment, not making yourself visible and without hindering foreground performances.

We will then move on to the Context which will also be very important, as we will see in the next chapter of analysis, as it will allow us to access some of the features offered by Android.

Later we will talk about the Intents and Broadcast Receiver who are in charge of sending and receiving various messages between different parts, including the system, and finally we will see SharedPreferences, mainly used to save data and configurations locally, which can also be retrieved in subsequent runs.

2.3.1 Service

A service is an application component that performs operations in background without having a user interface, for example think about push notifications.

There are 2 important aspects to consider: once another item has start service he will continue to run in the background even if the user switches to
another application; every other component can bind the service to interact with it and to exchange data and information.

There are 3 types of services: foreground (the results of the calculations are visible to the user, a sort of runtime controller), background (they are not visible) and bound (when application components binds to it, in that case it offers a sort of client-server interface to allow communication between the two parts) [18].

Just like Activity also Service has its own lifecycle, even if they are a bit different given also that service life is meant to be longer.

![Android Service Lifecycle](https://developer.android.com/images/service_lifecycle.png)

We have to consider also the fact that as other components also services runs in the main thread of their hosting process, this means that if inside it contains operations that need a lot of resources, you need to create a secondary thread that takes care of doing this job. In fact when the system runs low on resources it usually starts killing services, except the ones bound to an activity that has user focus.

In order to create a custom service you have to extend the basic available Service component and override the methods such as `onStartCommand()` and `onBind()`, this last one for example returns an IBinder that defines the interface for communication with the service, so the other elements can call `bindService()` and start the interaction. So multiple clients can bind simultaneously and unbind at the end when it’s no more needed. When all ones

---

unbind the service il destroyed. Finally, just like activities, also services need to be declared in the Android Manifest.

### 2.3.2 Context

It is basically the context of the current state of the application. It manages in some way the application environment by giving access to specific resources, classes and information. It is in fact the most important thing used in Android Development [9].

![Context Structure](image)

Figure 2.4: Context Structure

As you can see there are different types of context, we will focus on: Application Context, that is related to application lifecycle and allows developers to get access to some operations such as launching activities, broadcasting and receiving intents, retrieving preferences, etc...; Activity Context, that instead is related to the activity lifecycle and allows us to access to the realtime view components data, among other things.

### 2.3.3 Intent

It’s a data structure that contains an abstract description of the action that has to be performed [13], it’s basically a facilitator for the exchange of messages between different components within the same application or from one application to another.

There are 2 types of intents: Implicit Intents, where the name of the service to start is not declared, its just specified the action that will trigger some other applications to do something; Explicit Intent, it specifies the destination of the message, like a specific activity or service.
Considering all this, it can be used for multiple purposes:

- to start an activity specifying the name and any useful data;
- to communicate and send info to each class that includes the BroadcastReceiver component, which intercepts the intents sent in broadcast;
- to communicate with a background service;
- to send an action to the OS in order to start an app that can accomplish it.

### 2.3.4 Broadcast Receiver

Is a component used to intercept messages [intents] that are sent from other parts of the same app, from another app or from the system itself. Each app is registered to receive some specific types of intents and so, in order to function properly, this Broadcast Receiver needs the ”intent-filters” to be defined, cause everytime an intent is created, it has an intent filter inside with an action string, this means that just the ones that matches are collected [7]. Moreover it will always run regardless of the status of the application (on foreground or background).

### 2.3.5 SharedPreferences

Is an interface that allows access and modification of files that contain preference data. The main goal is to be able to save information/configurations that are not linked to the single run and that needs to be retrieved from different points inside the application. Specifically, multiple files can be created, distinguishing the various areas of use, which contains <key, value> pairs [17].
Chapter 3

Analysis on Collectable Data

In this chapter we will report the analysis carried out to understand what information can be extrapolated from the Android OS, how to do it and eventually if there are restrictions due to the type of device or usage.

We will start from some general data on the device, to some information on performance (cpu and memory), on ui, configurations, intents, etc ...

The difficulties encountered in this part are due to the absence of official documentation from Android developers. In fact over the years and with the new versions released some actions and features have become unusable as access authorizations have been restricted to prevent anyone from accessing sensitive information.

3.1 Device Info

First we need to provide to the analysts, on the server side, some basic information about the device so that every time they receive some data collection they can catalogue and group them based on, for example, the smartphone brand, the model number, etc...

We can notice that they are actually all visible inside the device settings, this means there must be a file or interface that makes some way them available and to which we can maybe access and extract useful data.

Starting from this assumption, we started first of all looking for a component that could perform this task, and then, later, we will decide which of these data to save, in order to highlight the uniqueness of the device.

Just by doing some few researches we immediately found the "Build" class [8], in the official Android Developers site. I contains inside a lot of information about the single product: starting from the model to the build version to the hardware configuration, exactly the type of data that we need. They are also easily reachable from every application, activity or service you are using, cause they are exposed at system level.

Among all of them we have selected the following fields [with examples], that will identify a single device:

- Brand ["Brand : HUAWEI"]
- Model ["Model : WAS-LX1A”]
- Build n° ["Build num : WAS-LX1A 8.0.0.376(C02)”]
Furthermore, since we will later collect performance data, we have also chosen to have information on the processor:

- **CPU model**
- **Processors n°**

To do this you need to use a very important directory called "/proc" where inside are not contained "real" files but runtime system information (ex. system memory, processes info, hw configurations, etc...) [14] [15]. However, there are important considerations to take into account:

- **Differences between Devices**: each brand that extends and implements its own version of Android manages these files as it is more appropriate and useful to them, in fact by doing tests with different devices we noticed that the same type data were present in different points or were not present at all.

- **Restrictions on Recent Android Releases**: gradually, with the new versions, Android has restricted the authorizations to access these files in order to not allow third-party apps to obtain sensitive information on the others. However, it is still possible to get from the application some of its own info. The main problem with these updates is the absence of official documentation, but there are news as widespread as "Android Processes & Security" [33], which shows step by step the occurred changes, as for example the fact that starting from Android 7.0(Nougat) the access to "/proc/[PID]" directories is denied, without providing in this way info on the running processes. Later on, on the next chapter, we will show some examples and consequences of this.

### 3.2 Performance

Performance is a key point in the application execution analysis as over time these apps require an increasing allocation of resources. Understanding how they behave in general and in particular associated with certain devices with certain configurations is fundamental, especially if one does not have the possibility to carry out tests in all possible cases. For this reason we are also focused on the study of the amount of average CPU requested and the amount of memory allocated.

#### 3.2.1 CPU

During the creation of an application is important to keep an eye on the CPU load necessary for correct execution and therefore to understand which are the categories of devices that can use the product. In particular it is useful to have both an average %CPU and the instant one.

The initial idea was to use the resources mentioned in the previous section, the "/proc" directory, to access the info of each app and to report the load
used, but as we said before this is no longer feasible, as those files can no longer be read. But we found a back-door: by calling the "ps" [Process Status] command in a separate process, this goes to read the files in "/proc/[currentPID]" and returns all the information of the calling process\(^1\).

In particular:

- **%CPU**: cpu utilization of the process in "##.#" format. Currently, it is the CPU time used divided by the time the process has been running (cputime/realtime ratio), expressed as a percentage.

- **VSZ (Virtual Memory Size)**: it includes all memory that the process can access, including the memory that is swapped out, the one that is allocated, but not used, and the one that is used by shared libraries.

- **RSS (Resident Set Size)**: is used to show how much memory is allocated to that process and is in RAM. It does include all stack and heap memory and the one from shared libraries, as long as the pages from those libraries are actually in memory. It does not include the one that is swapped out.

- etc...

We think that these kind of information can be very useful, not just from the app point of view, but also for the device manufactory that can find eventual problem in resources management by analysing this content.

### 3.2.2 Memory Usage

This is also an important aspect to consider, as the allocation of resources can also limit the performance of the device. We are not talking about storage but RAM with Heap and Stack. Especially considering that in the last few years RAM memories have grown exponentially in order to meet the highest demands.

We found a very useful component such as Activity Manager [2] that gives lot of informations about activities, services and processes, in particular it contains a sub-class, MemoryInfo that allows us to extrapolate:

- **Total RAM**

- **Available RAM**

- **LowMemory RAM** (it is a very important boolean that is used a lot, not just by most applications but also by the system itself cause when it is set to "true" it means that the available RAM memory is below a certain predefined threshold, so someone can use this information in order to do something, for example the OS uses it to activate background services killing.

\(^1\)list of all available information: http://linuxcommand.org/lc3_man_pages/ps1.html
We also further investigated the use of HEAP, that is responsible for saving dynamic allocations, also considering that it is usually very limited in size. In this case we exploit another important component, Runtime [16], that interfaces the applications with the environment, so it has access to runtime memory allocation data, we are able to retrieve:

- **Available HEAP**
- **Used HEAP**
- **MaxAvailable HEAP**

In this way it is easier to understand if the developed application is facing some performance problems, not because of hw configuration, but for example for some memory allocation problems, such as the situation in which the user is running a lot of applications and Memory Management is slowing down all processes cause it has lot of pages to handle.

### 3.3 Logs

They are very very important because they provide a complete overview of all the steps that are carried out gradually, in addition of course to the various errors to which you go against.

It is possible in fact to use them in multiple ways, for example to:

- understand the logical flow of our app;
- debug the code, the most used feature;
- identify any errors, printing the various messages in order to understand the causes;
- debug the runtime values of some parameters;
- print custom messages inserted by developers, in order to identify the flow path;
- etc...

To derive this data we took inspiration from the IDE that is used in PCs for Android apps development, namely Android Studio, which, when it compiles and runs an application, shows the *Logcat* functionality that prints all the logs. From this we started our research and we found the possibility of using the "*logcat*” command to access the log files and extrapolate all the lines.

We have also added a further feature that allows, in addition to obtaining the entire content, also to filter out specific TAG within that enormous size.

In all this, however, there is an important limitation to take into account:
starting from Android Jelly Bean 4.1 permissions are no more granted to access other apps logs. Also in this case there is no official documentation but making some researches and tests it was easy to verify that. We will provide some proofs of that in the following chapter, showing also some application that are no more useful due to this restriction.

3.4 UI

Moving on to the graphic part, it is necessary to first identify the display that you are going to analyse by gathering some information that will help the analyst to uniquely identify the product. Subsequently we find it useful to be able to know exactly how the standard layouts developed for the application are actually shown by the device, so as to be able to verify any discrepancies or problems. It would also be useful to have evidence of touch events raised in order to understand the interactions between the user and the app, thus also being able to reconstruct a sort of flow of the user experience.

3.4.1 Screen Info

We started collecting information on the resolution and density of the screen. In general this part of analysis is disconnected from the type of device that is used (Brand & Model n°), since different companies though having different products, however, use standard displays, except for particular cases. In this way, on the server side, anyone who analyses the sent data can also catalog them by type of screen, instead of by device info. In particular, in order to identify the single display, we decide to collect this data:

- **Density** [Ex. "3.0 - xxhdpi"]
- **Density Dpi** [Ex. "480"]
- **Width** [Ex. "1080px - 360dp”]
- **Height** [Ex. "1920px - 640dp”]

As you can see we have highlighted two important parameters, among the others: the name of the layout and the width in dp, as they are the ones that uniquely identify the layout resource used by the device for a given activity or fragment.

An important thing to consider is the fact that, to get these values, the context of the running foreground app is used, which means that the data may be wrong because if the app is not fullscreen it will consider only the Activity frame, ignoring the height of the Navigation Bar. We therefore added a check on its presence or not and in case there is we added the amount in the total.
3.4.2 View Info

This is another very important aspect of our research as in the development phase multiple layouts are created for various density or min width, each of these categories however covers within it different screens with different resolutions and dpi, so in the same static layout they match multiple runtime layouts.

It is therefore interesting to be able to collect data on the realtime graphic aspect of the app, in particular on where the various components are positioned and their real dimensions.

These are the info transmitted for each item:

- **Id** [Ex. "id:scrollView2"]
- **Class** [Ex. "android.widget.ScrollView"]
- **Position** [Ex. "x:24.0 y:48.0"]
- **Width** [Ex. "1032"]
- **Height** [Ex. "750"]

We have inserted 2 indexes in order to identify the various components, in fact if it is an activity or fragment, you can just use **Class**, otherwise if the element is a Button and there can be more than one inside of the view, you need to insert an **Id** in order to identify it uniquely.

To do this we need to have a reference to the context and so it is important that the screen we want to capture is running in foreground. This entails the fact that to have all activities data it is necessary to open them all, at least once.

3.4.3 Called Method

Another point of the research consists in being able, in some way, to record the flow of methods called, so as to have a general overview of the path followed and the logic applied.

Analyzing this aspect we have divided the calls mainly into 2 categories:

- **Custom Calls**: that are the custom methods inserted in the various classes and there is no way to be able to automate their collection past some data, unless you take advantage of particular TAGs inside the logs or anything else that would make all the things very complex and low performing. So for this option the only solution is the manual collection of data (by manual we mean that everytime we are inside a custom call, an exposed specific method has to be called in order to record the event).

- **OnClick Calls**: which can be click or longClick. Being Listeners, so interfaces, it is possible to implement them and make available new classes that can automatically execute the logic that one wants, leaving
the rest to the developer. This category is very important, perhaps the
most important of the two, since it is the one that responds to touch-
events linked directly to the graphic components and therefore the one
that best reflects the high level flow of the customer.

So we came to the conclusion of being able to have the data of all the calls
(some unfortunately in manual, others even more important in automatic)
allowing us to understand the main functionalities and features used, which
are very important for analysis purposes and company marketing.

3.5 Others

There are many other elements we need to collect to get a complete picture
of how the app works remotely.
We need to understand first of all what are the interactions between the various
internal and external parts of the app, in other words we are talking about
Intents.
It would also be nice to be able to know the preferences that individual users
have chosen, so that they can also carry out both functional and marketing
analyses.
Finally, there is also the possibility of being able to know the valorization of
certain parameters at certain times and the possibility of knowing the timing
that the execution of certain functions requires.

3.5.1 Intents

They are essential means of communication, not only between components
of the same app [for example to switch from one activity to another], but also
between different apps or between apps and the system, so being able to save
these messages is important.
These are mainly of 2 types:

- Explicit Intents: addressed to a specific component or class, so it is
  not possible to intercept them, unless you are one of the endpoints that
  created or received it, cause they are not sent in broadcast but is systems
task to direct them to the appropriate recipient.

- Implicit Intents: do not have a destination assigned but they have
  enough information to let the system to sort them to the right compo-
nents. They are usually sent in Broadcast, so easily interceptable.

Not being able to collect the first type, also because it is not possible to access
the "Intent Manager" at system level, nor to extrapolate them in some way
from the logs, it was decided to intercept at least the implicit ones.
The main solution we found is so to use a Broadcast Receiver that intercepts
all broadcast messages and allows the user to handle them by reading their
content or by triggering some logic. Note that in order to be able to read
and filter intents it is very important to specify all the required Intent Filters, cause the broadcast component is going to collect only the one that matches the predefined types.

### 3.5.2 SharedPreferences

In every application there is the need to save data locally that are not linked to the single run, but obtainable every time the need is found. Here comes into play the files of SharedPreferences that exposes APIs that allow you to do so.

The purpose of our analysis in this case was to find a way to access all those resources in order to extract all the configurations and settings saved by the customer.

At first we tried using the SharedPreferences component retrieved by the Context, but we encountered a problem since in order to be able to retrieve the files you have to pass some keys, that are actually the files names, and we don’t have them. We had so to find another approach to resolve it.

By making some researches we identified the directory that contains them all, that is “shared_prefs”, located inside the app’s folder, so by doing a scan inside it you can get all the resources and then for each one extract all the pairs key, value stored.

It’s important to remember that also in this case, in order to access this external directory, it is useful to use the Context cause it can identify the package and the right path to look for.

Given this considerations we have to take into account that it is not possible to use a third-party app to do it, but you must use the same one.

### 3.5.3 Object Value

In many fields of computer science it is very important to know which values a certain object/variable takes at a given moment, because depending on the result you can make certain considerations rather than others.

This indeed has applications in the world of development (debugging to analyze the flow), security (to analyze the runtime values and find any breaches in the system), etc... We therefore focused on finding a solution to provide this information in a way that was as automated as possible.

We tried by creating an abstract class to be extend that contains a Map<String, Object>, in order to save various parameters with a generic type in different keys, and it has also, inside the set method of the map, an automatic call to the tool so to pass the new values of the object. With this solution we could have at each moment the values associated to that element but the implementation and usage by developer user would be more tricky and complex than it is usually.

We decide so to discard that solution and so as a result of our analysis we kept the ”manual” way, it means that the application have to call the tool through a method to pass the information. Given this, we wanted provide different
ways to do that, by leaving to the user the choice (for further details in the implementation we refer to the following chapters).

3.5.4 Time lapse

Last but not least important is the ability to measure the time interval between a call and another or between a request and a response to external services.
Very important feature as it allows you to understand if and where there are slowdowns and performance problems in the app or outside it, for example measuring the time elapsed between sending the request until the response is received.
We found already existing components such as TimingLogger, that will be shown in the next chapter, that is made by Android Developers and allows you to accomplish this task. There are also others already built solutions.
Given these results we decided to skip this part, also because, given the already provided features, they can be exploited to obtain the needed data.
Chapter 4

State of the Art
and Design Solution

This chapter contains some existing solutions/tools that satisfy some of our goals. In particular we will analyse the pros and cons but above all the limitations they face.

We are going to report the main solutions found about: the monitoring of performance resources, where we found some interesting news and applications with huge limitations; logs, where we are going to show one of the most used applications in the past but not too much nowadays, CatLog; execution time and ui resources.

Finally, based on the results obtained and on the considerations we have made, we will propose our architectural solution and in the next chapter we will go into the details of the implementation.

4.1 State of the Art

4.1.1 Monitoring Performance Resources

Let’s start with the analysis of resources, specifically cpu and ram.

We found a lot of interesting articles and tools, some of which were once paid but which over time and with the new limitations introduced have become almost unusable. In particular we have selected an article that addresses both the problem of the new releases, and cites within it two important and widely used applications whose functionalities, however, have been reduced, since they can no longer be used.

Finally we will also talk about one of the most popular apps, AnotherMonitor, that once it was paid. The developer has also described on his page the various restrictions introduced step by step, which we will report here.

4.1.1.1 How to find out which app hoggs CPU on your Android

This article actually focuses much more on the CPU than on the RAM, but reports the problem of access to resources [31].

In fact we read:

"UPDATE: Google doesn’t allow third-party apps to access the hardware status anymore, due to security reasons. So, there’s no way of checking the CPU usage without rooting your phone."

23
The article in fact explains how it was once very common to have applications that monitor resources and other applications, while today with the new releases these features are mainly delegated to the Android system itself, which already integrates tools that make reports on consumption and utilization. Next it goes on by explaining how to consult this data on the device without having to install anything new. However, to have more advanced readings and graphs depicting use over time, 2 applications are suggested:

- **TinyCore** [20]: it is a very well designed and small size application that is able to provide information about cpu usage, cpu frequency, ram usage and battery level. Except for the last one, they are all data that can be collected from the previously described ”proc” folder and from Activity Manager Component.

Watching the playstore page of the app we noticed this news:

"Android 8.0 users: Due to security changes in Oreo, apps can no longer access CPU load information. However CPU frequency information is still accessible and I made it a non-pro feature Oreo devices."

- **CPU-Z** [10]: it is like the previous one, but it gives some additional information such as system name and architecture, model name and brand, and some sensors information. But also in this case they are all general information, they can’t access specific process data.

---

2TinyCore: https://lh3.googleusercontent.com/AiBvpSthaUoiEc2p0vR1HvBPhhKPDUbVoHxC4QoO6i08qQG8ZQU9e2Fuopu4e8wA=w958-h933-rw
2CPU_Z: https://lh3.googleusercontent.com/9EK-1qqRYVFHlzs6dSNdhBQRGT_RoP3y7N1ADrsEj-cRelsecyRvUxKfRPB0oFU=w958-h933-rw
4.1.1.2 AnotherMonitor

It is actually an application that records real-time CPU and memory usage [32]. It is able to collect the list of all available apps that are running and for each of them accesses the previously described ”/proc” directory to get all the information needed. Furthermore, once these data are obtained, it is also able to provide information on the overall management of the CPU and memory. However, due to the restrictions introduced with recent Android releases, as also reported on the available git page of the application, these resources are no longer accessible, in fact:

- **starting from Android Lollipop 5.1 (API 22, March 2015) it is no longer possible to access the list of processes obtainable through the Activity-Manager, however they have solved this problems by using an unofficial library, which however does not work for all processes.**

- **starting from Android Nougat 7.0 (API 24, August 2016) the accesses to the ”proc” system files have been restricted, so from that moment on the app could not collect information on the single process but could only report a general overview of the CPU.**

- **starting from Android Oreo 8.0 (API 26, August 2017) the ”proc” authorizations have been further restricted, so that now the app is no longer able to show any data relating to the CPU either in total or in the single process, the only analysis it is able to do is related to memory consumption (storage).**

The main problem encountered was the lack of official documentation that led the developers to long analysis and subsequent fixes, clashing with blind difficulties and solutions.

The app is no longer usable because as mentioned before it is no longer possible to access information that is not strictly related to the application itself and even in that case it would face restrictions.

Also in our case, therefore, it is not possible to use an external tool, or a third party app, but in some way it must be incorporated into the application itself (ex. as a library).

4.1.2 CatLog

It was a very popular application used by developers in order to see the logs of all running apps [27].

Besides the fact of recording all the logs printed on console that appear gradually also has other features such as:

- classify these according to their level, also providing a further option to be able to filter only the desired ones.
– view the various types of logs with different colors in order to distinguish
the errors from the info ones, etc... with the option, also here, of being
able to filter out the type.

– search within the multiple lines specific TAGs or words.

– and much more...

Unfortunately, as also written on the page reported previously, from the An-
droid 4.1 Jelly Bean version, although having the permission "READ_LOGS",
which allows you to read the logs of other applications, it is no longer possible
to do so, unless you are a system app or you have root privileges.

Thanks to this further discovery, in our case, the solution to create a library
that can hook up to the apps developed by the various companies and that
allows them to receive the data useful for analysis is increasingly emerging.
Even wanting to use a third app we will not be able to integrate it into the
system, unless we put our hands to the OS and then have it installed to all
users, which is not feasible and convenient, nor to guarantee root privileges
without being able to use the device.

4.1.3 Execution Time

We found many tools that allow the analysis of execution times, we report
only some that are in particular integrated and mentioned in the documenta-
tion of Android Developers.

Figure 4.2: Catlog

---

3Figure 1: https://lh3.googleusercontent.com/fH-3Oq7jwPrNY5XCdZh2Z56Et6iy0b0cbb
0uFUuSpdVuFe4Qe7j2LpSisZl3ZT_PjW1w=w958-h933-rw
Figure 2: https://lh3.googleusercontent.com/WNRUHXiVRKX0gu4jOuMEDLz4JwtaEiHQpacSApIrs
2nNPROS8FGGuaF0nmlbHUmQPEM+w958-h933-rw

26
A first example is TimingLogger [19], a utility class to help log timings splits throughout a method call, for example:

```java
TimingLogger timings = new TimingLogger(TAG, "methodA");
// ... do some work A ...
timings.addSplit("work A");
// ... do some work B ...
timings.addSplit("work B");
// ... do some work C ...
timings.addSplit("work C");
timings.dumpToLog();
```

obtaining this output:

```
D/TAG ( 3459): methodA: begin
D/TAG ( 3459): methodA: 9 ms, work A
D/TAG ( 3459): methodA: 1 ms, work B
D/TAG ( 3459): methodA: 6 ms, work C
D/TAG ( 3459): methodA: end, 16 ms
```

Another way is to use the Traces made available by the Android OS and exploit them by using various tools that can read them and offer different graphical or printed results.

Since the available solutions are well taken care of and sufficient to satisfy the requests of the customers, we have decided not to focus on this area, which in the future can be integrated into the library if desired, perhaps adding further configurations.

### 4.1.4 Monitoring UI Resources

Another very important field in which we have addressed our research are the UI components. Specifically, as previously mentioned, it would be nice first to know the real structure of the views in the various devices, since when the layouts are implemented, only a certain amount are developed which are the default ones.

Another aspect to consider is being able to keep track of the various interactions that the user performs with the application, so as to then be able to carry out analyses on user experiences.

However, we have not found any relevant project or article that could help us or give us a starting point in building our final solution. We only found a lot of tools that focused on the visual (video), non-structural (xml, json or other) screen recording and others that allowed graphics debugging but only locally, not remotely.
4.1.5 Related Work

Not having the official documentation available, it is very important to start with the analysis of the evolution of the Android operating system [5] and with it the changes that have been made and any compatibility problems. In fact, as reported in the study carried out by McDonnell [29] in 2013, Android evolves with a rating of 155 APIs update per month on average and about 28% of the API references in the application are outdated. We have seen how in the previous applications some of these are no longer available because they still use old features that are no longer accessible. There is a great deal of research in this area, especially as FILO [30] that takes care of analysing the various applications, identifying within them any pieces of code that are no longer backwards compatible due to the updates made. However, the accesses to the various resources are not analysed, thus verifying the authorization levels offered.

As far as these are concerned we find studies that are particularly interesting and similar to our purpose, but outdated, such as DELTA [22], that is a multi-purpose logging tool that allows researches to analyse data, such as information about the status of the operating system (memory, network, etc...), and detect anomalies.

Another strand based on data extraction is Android Forensics, like the article by Grover [25] which reports the development of a monitoring system that allows to constantly collect data in order to allow analysts to do research on user security.

Finally there is a last strand that deals with analysing some data to build a unique profile of the user that can somehow outline his uses and customs, in particular his preferences and needs, in order to be able to provide them with applications for marketing purposes. An example is TYDR (Track Your Daily Routine) [21].

Moving towards the user experience we find analysis on latency, energy and UI usage information that is critical to the definition of user QoE (Quality of Experience), such as that of Cha and Lee [28] that through their QX-probe tool, allow to obtain these information.

In general we have a considerable number of researches, but they focus more on security and performance than on the analysis of subsequent low-level Android updates, giving these for known, but this is not the case and in fact some of these solutions do not work anymore.

4.2 Design Solution

To sum up we have seen that starting from the 4/5 version of Android we have great limitations at system level that does not allow access to specific files useful for performing various analyses, in particular the accesses by third-party apps have been restricted.

In fact, our initial idea was to build an independent app that could collect this
realtime data from all the running apps, analyse them, process them, catalog
them and then send them to a central server that would then deal with the
sorting. In this way we would have had a centralized and packaged core that
would have carried out the task for which it was designed, without the need
for any external entity to get their hands on it. However, for the reasons given above, this was not possible, otherwise most of
our goals, not to say any, would not have been met.
The final solution was therefore to produce a library that includes all these
features inside and, in addition, to expose a configuration file so that the
customer can decide which functionalities to activate.

![Functional Solution Diagram](image)

Figure 4.3: Functional Solution

We will therefore have a centralized service that runs in the background
and will collect application and system information based on the developer’s
indications. Once these data have been collected, they will be filtered and
processed in order to best build a payload to be sent to the indicated servers.
The new classes implemented must be included in the new application that
needs to be developed and must be used as indicated in order to take ad-
vantage of the various features. In addition, very important, you must also
declare the Runtime Service in the Android Manifest, otherwise it will not be
initialized.
The service is created in order to use as few resources as possible and to do this we made the decision to perform most of the actions when the device is locked or in standby. Obviously some operations must unfortunately be done at runtime, such as collecting information on real views, since the layout must actually be visible at that time to do it, or the interception of broadcast intents, etc...

Furthermore, based on the configurations set, the service activates data collection every few minutes or hours, specified by the developer.

Moreover, all these configurations must also be able to be set and changed dynamically, so it is not possible to use an interface to do it and in addition it is necessary to save the various modifications so that they will also be available in subsequent runs.

We will deal in detail with the implementations and target solutions in the next chapter.
Chapter 5

Implementation

Starting from the previous conclusions we will proceed, in this chapter, with the analysis in detail of the various phases of the implementation process and the different components used to offer the above examined functionalities.

We started with the creation of a simple app, with 2/3 activity and some fragments, and within it we integrated a service that was automatically started when the app was launched and remained in the background until the app was killed. We have gradually integrated the various features and updates one after the other, using this simple application also as a test case.

An orderly and logical explanation of all the subsequent steps that led to the final solution will follow, showing in particular how we were able to achieve our goals and also briefly how to use the new created components, cause it will be further shown in the next chapter.

5.1 Runtime Service

As previously mentioned in our final solution we have a centralized service that encompasses the heart of our logic, since it is the one who receives all the needed data, some of which it extracts alone, organize them, process them and finally send them to the outside.

Since it is a service it extends the base class of the Android OS, Service, so that it can be defined as such and can also run in the background. In addition it was built as a Bound Service, so all the components can bind to it in order to interact and exchange data. However, this is not enough, as the service would run in the main thread of the activity, and since it requires lot of resources, it would greatly affect the foreground processes. For this reason we decided to create, at its start, a secondary thread, in order to move all the logic inside there, thus separating the two processes.

This thread is first activated only if the device is locked or in standby mode and at that point periodically, based on the waiting time established by the developer, first checks the features that have been activated, then collects all the required information, process them and finally creates a REST call to the server defined by the company, to which it associates all the obtained and useful data in a payload.

In order to use this RuntimeService there is the need on users side, first of all, to have to integrate in their project the various useful components, created by us, and then having to declare it in the AndroidManifest, as shown below.
The service must then be called and started in an activity and then also binding to it, or you can directly use the activity we created, RuntimeActivity, which automatically performs these operations within the various phases of its lifecycle.

We will describe the logics [methods, parameters and elements] used within it as we talk about its peculiarities, so as not to introduce topics that will only be examined later.

The elements that will be explained now are those useful for the correct functioning of the service, such as: a configuration file to allow customers to select the needed features; a check on the locked device status, in order to start the collection when it has the least impact on performance; a REST call with relative PayloadRequest, in order to send data to the multiple Servers.

5.1.1 Configuration

This is one of the most important elements of the library as it is through it that developers decide what data to collect and manage and how to do it.

In the Appendix Code [Configuration] below is reported the class and its parameters in order to give an idea on how the settings were saved and managed.

In particular we see how 2 main components are present:

- **HashMap <String, String> configurations** : it is a key value map where you can enable the various features, set the time interval, which defines how often to start collecting information, and the url, which tells the address of the server to send data to.

- **HashMap <String, List<String>> filters** : this instead is a map that contains the filters needed to extract intents and tags inside logs. Each of the 2 cases has a string list linked to 2 different keys, "LogFilters" and "IntentFilters".
To use it just access the class available in the tools repo\(^1\), which is included when all the package is downloaded. The developer must therefore modify the settings within Configuration and recompile the entire app so as to enable the services requested.

The initial idea was to create it as an interface in order to make the values static, unchangeable and recoverable with each run. Subsequently, however, the need arose to make the settings dynamic as at some point there may be the need, during the applications flow, to make changes. We have therefore converted this interface into a class leading us to a huge problem: although changing the values at runtime, these are no longer recoverable at the next run, as there would be a different initialization, we had so the need to save these data somewhere in order to recover them at any time. For this reason we have used SharedPreferences, in this way we will keep this information stored in a file locally, so that we can have access both in the case of changes and in the case of new apps runs.

Basically the flow works in this way: when the service starts it accesses the preferences file corresponding to the ”confFile” key, in case data are present, which means that they have been modified, then they are taken and set in a Configuration class which is then passed to the secondary thread, otherwise one with the default parameters is instantiated.

To make the dynamic settings work instead we made sure that the developer could set in the service the new configurations, then the preferences file is overwritten and passed to the other thread.

In this way we are able to handle all activations and disactivations of the functionalities.

### 5.1.2 Lock Device Status

The next step is to be able to assess which is the best and best performing time to be able to make the data collection and processing. As shown in the previous configurations, there is an interval that is enhanced and used to activate a timer, but it is not sufficient.

We considered the case of being able to check the CPU usage and find the best moment. However, having a service that runs continuously in background means that the best time to start the process is when the device is put on standby, so as not to influence the performance of any apps in foreground.

Below is the double check performed within the ServiceThread:

```java
public boolean isPhoneLocked() {
    boolean isLocked = false;
    // First we check the locked state
    KeyguardManager keyguardManager = (KeyguardManager)
        context.getSystemService(Context.KEYGUARD_SERVICE);
    isLocked = keyguardManager.inKeyguardMode();
    return isLocked;
}
```

\(^1\)https://github.com/KostandinCaushi/RuntimeCollector
boolean inKeyguardRestrictedInputMode =
    keyguardManager.inKeyguardRestrictedInputMode();

if (inKeyguardRestrictedInputMode) {
    isLocked = true;
} else {
    // If password is not set in the settings, the
    // inKeyguardRestrictedInputMode() returns false,
    // so we need to check if screen on for this case
    PowerManager powerManager = (PowerManager)
    context.getSystemService(Context.POWER_SERVICE);
    isLocked = !powerManager.isInteractive();
}
return isLocked;

Listing 5.2: RuntimeService - isPhoneLocked()

In fact, we have multiple ways to lock the device:

- **Screen-Lock**: with PIN or fingerprint or other, so it is sufficient to check
  that inputs are blocked until the device is opened.

- **Stand-by**: closing without protection measures, so we must verify that
  the screen is on or not.

We have to make the second check cause in case of no chosen security protec-
tion it will return always false.

This is actually the first check that is made on the service run loop, if it’s
satisfied than it will continue with the collection of data.

### 5.1.3 REST Call

When all the data has been collected, processed and ready to be sent, at
that point a custom PayloadRequest [whose fields are shown in the Appendix
Code - Payload Request] is created and then is sent to the server via REST
call.

In particular, at the end of each time interval and if the device is locked,
information extraction begins and they are gradually saved in this payload,
in the appropriate fields. Once the process is over, an HttpURLConnection is
opened towards the server (with the url specified in the Configuration “url”
field), a POST request is created, the content inserted inside and the whole
is finally sent. Nowadays this is the most used and practical way to exchange
information via internet.

For now, there are no security measures, such as specific protocols or en-
crypted data, since the thesis does not focus on this aspect, but it is possible
to integrate it as a future work.
5.2 New UI Components

Now let’s move the focus to the new UI components developed starting from those already present in the Android system and used to make data extraction automatic and efficient.

We will start with the analysis of the new RuntimeActivity which extends the existing AppCompatActivity, but which also includes the initialization and binding to the new RuntimeService. We will then move on to the RuntimeFragment which in turn extends the Fragment class and includes within it the functionality to capture the screen after the view is inflated. Finally we will talk about the 2 new listeners to be associated with the clickable graphic components in order to save the flow of the called touch methods.

It is therefore important, once the library has been imported, to use the new elements made available, which are ready and configured for use, except if there are some specific needs on customer side that will bring to the creation and usage of some custom parts.

Code parts essential for explanation and comprehension will also be shown, for the complete implementation instead consult the git repo.

5.2.1 RuntimeActivity

This is a new activity created starting from the already existing AppCompatActivity (it is an extension of that) which therefore contains inside it all the various phases of create, start, pause, etc..., but it has in addition, inside it, some checks and initializations that allow us to manage our service automatically and independently.

The first step is in fact to verify, during the onCreate() phase, that a similar service is not already running in background, in the negative case proceed by calling the startService() and at a later time the binding is performed.

The solution taken for the check is the following one:

```java
private boolean isRuntimeServiceRunning(Class<?> serviceClass) {
    ActivityManager manager = (ActivityManager)
        getSystemService(Context.ACTIVITY_SERVICE);
    for (ActivityManager.RunningServiceInfo service :
        manager.getRunningServices(Integer.MAX_VALUE)) {
        if (serviceClass.getName().equals(service.service.getClassName()))
            return true;
    }
    return false;
}
```

Listing 5.3: RuntimeActivity - isServiceRunning()
At first we are going to extract the ActivityManager which contains all the information and interactions with the various activities, services and processes. From this we get all the services currently active on the application, going then to check if the class of any of these matches the desired one (RuntimeService). All this is done in the first phases of the Activity, in order to be able, in the worst case, to start the service at that point in the cycle and do the binding on the next one, avoiding conflicts and delays.

So as said, the doBindService() is then inserted in the onResume() phase, allowing also activities that return in foreground to be able to re-bind to the service (obviously we also have a doUnbindService() in the onPause()). Inside this method we have added a crucial step, we are going to pass the Context reference, that the activity has, to the secondary thread. This allows us to have this parameter available at all times, thus managing to collect all the necessary data. As we saw on the previous chapters, this parameter is very important cause a lot of functionalities and data that we are using are accessible through it.

Moving on, once the service receives this reference it stores it and starts a process by calling "getView()”. Basically it will make some checks and based on that it will get access to the view components and collect their real position, size, id, class, etc...

```java
private String viewInfo() {
    String output = "";
    final ViewGroup parent = (ViewGroup) ((ViewGroup) ((Activity) context)
        .findViewById (android.R.id.content)).getChildAt (0);
    showViewInfo (parent, output);
    return output;
}
```

Listing 5.4: RuntimeActivity - getViewInfo()

Through this piece of code, having the context, we are able to derive the Activity to which it corresponds and from this its root element. In particular, we can get the ViewGroup parent of all and from then on we can get the components one by one and for each of them we are able to extract:

- the id, if it has been set;
- the corresponding class;
- the relative position with respect to the father node;
- the dimensions (height, width).

We can so explore and analyse the whole tree of the graphic structure.
All this leads to a significant performance problem, that is the execution of repeated scans, with the same output, every time the same activity is resumed. So if we leave the code by that, everytime the same activity is shown the service will analyse it and will return more or less the same output. In order to avoid that we resorted to the use of SharedPreference, where we can actually save in a local list all the corresponding tag (= name of the Activity class) of all the activities already seen. In this way the service before starting the `getView()` process will check if the corresponding tag is already present or not, avoid so unnecessary executions. This also means that at RuntimeService startup a `loadViewMap()` is made to load the map that contains all the activities or fragments already visited. Finally when the collection loop of the service is completed, if new views have been explored and sent to the server than a map needs to be saved, so another to preferences files is made and the new content is written.

![Figure 5.1: RuntimeActivity view capture flow](image)

The above shown flow will sum up the concepts we were talking about. Another point of our goal was actually to be able to record in some way the user experience by storing step by step the visited screens, so given that every-
time the `getView()` is called, regardless of the next check that is performed, as shown in the diagram above, we save the tag passed in a List<String> called `viewFlow`, thus going to outline a flow of the various opened views.

5.2.2 RuntimeFragment

It is very similar to the RuntimeActivity previously shown. As previously mentioned, an activity can have more fragments inside it and a FragmentManager is needed to switch between one another. Specifically, inside the activity a portion of the frame is reserved to set subsequently the layout of the fragment.

An important state in the lifecycle of this component is the `onCreateView()`, where through the LayoutInflater the desired xml is taken and shown to the user. It is important so to wait till the end of this step in order to be sure that the layout has been inflated in the activity frame. After that, on the `onResume()` state we inserted a call to the `getView()` method exposed by the service. All checks and subsequent unleashed actions are the same as those shown in the diagram of the previous subsection.

Note that when you are going to create your own custom fragment you have to extend the RuntimeFragment that we are offering and you have to override some of the required methods.

5.2.3 Listeners

We have built 2 listeners to respond to the request to keep track of the interactions the user has with the application. Specifically we managed 2 events: onClick and onLongClick because they are the most used and widespread, but the same technique could also be applied to the others.

The idea was to be able to use the already existing interfaces offered by the Android OS, inserting inside it a direct call to the RuntimeService, so it can keep track of the executed actions, all done automatically.

Going into detail we are going to analyse the developed `RuntimeOnClickListener`, but the same considerations and implementations also apply to the `RuntimeOnLongClickListener`. We must first start from the basic element which is precisely an interface to which we must override the `onClick(View v)` method by inserting the desired operations into it.

The idea was therefore to create a class that implements it and to insert inside this last method a direct call to the service, as `calledTouchMethod(String tag, HashMap<String, String> methodData)`, in this way everything will become automatic and hidden to the developer.

To be able to use it, therefore, it is essential that this specific type of listener is set to the graphic component concerned and when you go to override the method be careful to leave the call to `super.onClick()`, do not delete it, as shown in the code below.
Once the service has received the call, it will take a timestamp to associate with the data so as to be able to provide a temporal order for the various execution and subsequently it will add the tag and the information passed to the various lists in order to enrich the user’s flow.

5.3 Basic Information

Among all the useful information to be collected there are some basic ones useful for cataloging and filtering the device. Those are information that can uniquely identify the hardware being analysed. Specifically we are talking about Device Info and Screen Info. Both are managed at the beginning of the secondary thread of the service, as soon as it starts running these data are collected and saved in local variables, since given that they are static and invariable there is no need to look for them at each iteration.

On one side we have Device Info that are easily available, as described in chapter 3, just need the Build class and some other components.

More interesting is the extraction of the display specifications, not so much for density and width as for height measurement. In fact, having the Context available and accessing the DisplayMetrics present in the resources, we are able to obtain a height that is not the total height, in the case of non-fullscreen applications, but only that of the application frame that does not take the NavigationBar into account. For this we have carried out research to find the best solution in the calculation of this parameter and we report the final result below.
private int getNavigationBarHeight() {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.JELLY_BEAN_MR1) {
        DisplayMetrics metrics = new DisplayMetrics();
        ((Activity) context).getWindowManager().getDefaultDisplay()
        .getMetrics(metrics);
        int usableHeight = metrics.heightPixels;
        ((Activity) context).getWindowManager().getDefaultDisplay()
        .getRealMetrics(metrics);
        int realHeight = metrics.heightPixels;
        if (realHeight > usableHeight)
            return realHeight - usableHeight;
        else
            return 0;
    }
    return 0;
}

Listing 5.6: Screen Info - getNavigationBarHeight()

First of all it check if the version is greater than JellyBean or not, if so it needs to check if the real height of the activity is greater than the usable one, if it’s true it means that there is something in between that is filling that gap, the NavigationBar, so we then return its height.

5.4 Logs

As previously reported in chapter 3, logs are a very important part to be extracted as they are useful for a variety of purposes. However we have seen as it is not possible to access those of other apps due to the new restrictions introduced, but it is still possible to do it on the same one by using the "logcat" command.

Before going into more detail on the implementation, we need to see how and where the choreography is launched, specifically we decided to have the entire process managed by the secondary thread, which collects this data at each interval. This solution allows us to make the best use of the resources offered by the device, since first of all it does not affect the main thread and therefore the various foreground processes. The second thing is that the whole process is entirely executed when the device is in standby, providing in this way a good result in performance and management.

Now let’s move on to a more detailed analysis of the implementation. It all begins when the getLogs() method, present inside the ServiceThread, is invoked. It first goes into the current runtime using the component already exposed by the Android OS called Runtime, which allows applications to interface with the environment they are running in, then executes the
"logcat -d"2 command, which allows the dump of the log to the screen and exits.

We need so an InputStreamReader in order to be able to read this output and convert the content into a String that is going to be inserted into the payload. During the stream scan, the filtering functionality is also integrated, it means that a check is made to see if the analysed line contains one of the tags specified in the Configuration file, specifically we refer to the list of strings called logFilters. If a match is found, a new entry is added to a separate map, showing <Tag, LogLine>.

Once this collection has been made, for performance and management reasons, all the logs present up to that point in memory through a second "logcat -c" command are deleted. This also to avoid sending the same content several times and to deal with huge contents that take up a lot of memory and require more time to be sent.

5.5 Intents

On the basis of the analysis carried out we have seen how it is impossible to intercept explicit intents unless you are one of the 2 endpoints from which the information is sent or received or you are the system below that manages packet addressing. Given that in our case it is the service that handles everything, the only thing we can do is be able to collect all the messages sent in broadcast, extracting as much information as possible.

To fulfil this task we decided to use a BroadcastReceiver. However, we have not included it in the RuntimeActivity, that would then sent the data to the service, but we did the initialization directly in the secondary thread so that, even in this case, we would have a good management of the available resources, thus not going to affect the performance level.

The receiver is started, however, only if the functionality was chosen in the configuration file, in which case an override of the onReceive() method is performed and data collection is started. In particular, the logic that has been inserted inside the method consists in taking the received message, extracting the key and comparing its value with the filters inserted in the configuration list with the name of intentFilters. If a match is found, we are going to extract for each intent: Action, the general action to be performed; Component, the concrete component associated with the intent; Type, MIME type; DataString, data the intent is operating on; SerializableExtra, extended data from the intent; BundleExtra, same as before; Timestamp, this one is created by us to keep track of the temporal order in which the data arrive. Finally all this is converted into a string and inserted into a map that will then be added to the payload.

We are thus able to know at least a part of the communications that take place between various parts of the same app or between various apps or between the

---

2logcat command: https://developer.android.com/studio/command-line/logcat
app and the system.

5.6 SharedPreferences

Preferences are very important because based on the values they assume and the various combinations that can be obtained changes the behaviour and appearance of the application. They are also widely used to save data that you must then have to retrieve in different instants or different runs. Based on the analysis made we can see that, in this case, our purpose is to be able to access the "shared_prefs" folder, which is contained within the application package, so first we need to find the root folder and then look for the one we want.

Here too, for the umpteenth time, the context comes to our aid, since from it we can extract the ApplicationInfo, a class that contains all the information needed about a specific application, in this case the one in which the service is running. Among these there is one in particular called dataDir, which is actually the full path to the directory assigned to the package for its persistent data. We started so to explore its contents and we saw that right there was the folder we were looking for. Then from there we were able to extrapolate the names of all the files, paying attention to delete the last part that includes the extension and use the rest as a key to call the SharedPreferences component, which in turn returns all of its content. Infact in order to obtain the preferences usually you have to call getSharedPreferences(nameFile, ...), so we are actually using a trick in order to accomplish our goal.

Having also the service incorporated in the application we can access these files in MODE_PRIVATE, this means that we have all the permissions required to access all the preferences set by the app. Moreover, as for the features described above, this is also fully executed within the second thread and when the device is in standby, thus improving performance and management.

5.7 Others

We have left for last two features, not because they are less important, but because we have not found a smart solution to manage the requested information and also because they are closely related to each other, meaning that the resolution of one would also imply that of the other. The first of these is the CalledMethods, which is basically similar to the request to collect data on TouchEvents, whose solution was to create custom Listeners that can be used to automate the interception of the necessary data. In this case, however, we are talking about the custom methods that any Activity, Fragment or in general any class could have. We thought of possible solutions to automate this process too, but each of them required at least a call from the developer to an exposed method. At that point we decided that since we still needed to make a call, we might as well call directly the RuntimeService
and pass the necessary data to it. This is therefore our "manual" solution that for each method call allows you to register a specific tag and a data map to which we then add a timestamp to provide an order.

The tag is also used to enrich the lists in which all the interactions between user and app are saved, in order to reconstruct its experience flow.

The second functionality is that of being able to save the value that a certain object or parameter assumes at a given moment. Here too the solution would be to be able to intercept the data when a set() is performed with the new values and automatically send them to the service. Unfortunately, we didn’t found a smart way to do that so even here we have an exposed method that must be called ”manually” passing a tag and a value.

The correlation defined at the beginning consists in the fact that if the solution was found for the first, it could also be used in the second case.
Chapter 6

Validation

Finally we will begin to use the new developed components and to integrate them in some applications in order to test their actual functioning. In particular, given an app very similar to the ones we use every day, we’ll see how to perform the integration by making the small refactoring required. We will next activate some features and start collecting some data so that we can then analyse the actual output provided by the RuntimeService and also discuss about that.

Below we will begin to outline the context in which these tests will be performed, for example the devices that will be used and the available applications. We will then proceed by analysing the individual features one by one and showing the corresponding test cases, results, discussion and eventual problem detection.

6.1 Setup

Let’s start by talking about the context in which we will operate, in this specific case the tools we will use, be they physical or virtual. This step is very important as it allows us to interpret the results starting from the not inconsiderable initial assumptions.

We will now first see the devices that we will use with their information and technical specifications and then we will move on to the applications that we will use as testers.

6.1.1 Devices

Two medium-low end devices will be used to cover the average consumer segment. The one first is a Huawei p10 Lite that nowadays, with the present technologies and the new devices released in the market, can be considered low-end.

The second one is a Huawei p9 Lite, it’s almost similar to the previous one with just less CPU performance. The last one is a Samsung s10+, the first of its generation and therefore maybe a little less widespread but of a much higher level, let’s say a middle ground.

The other difference and perhaps the most important is that we have the first one and the second physically available, while for the last we will use a Virtual Machine to make tests, so the performances could be reduced.

In the next subsections we will provide all the useful details to identify the
devices so that the technical specifications are well highlighted and the outputs
easily interpretable depending on the cases in which we find ourselves.

6.1.1.1 Huawei p10 Lite

It is a smartphone released in February 2017 along with the big brothers p10 and p10 plus. In that year, however, it was the best-buy for its range.
It has a Kirin 658 that carries a quad-core 2.1 GHz Cortex-A53 and a quad-core 1.7GHz Cortex-A53 and as a GPU has a Mali-T830 MP2.
As for the memory, it has a 3 Gb RAM and a 32 Gb storage.
Uses as OS EMUI 8.0.0 which is a Huawei extension of Android 8 and therefore API level 26, released shortly after August 2017. Not very old considering that the last Android version 10 was released on 3 September 2019. It falls within the range of old but still supported versions, it means those for which the new applications still provide compatibility.
Finally, as a screen it has a 5.2-inch touchscreen with a resolution of 1920x1080 pixels and a density of 424 ppi.
We have to consider also that it is a smartphone that is used as a personal device in everyday life, so it is full of useful applications and services, so we must take into account that part of the resources offered are already used and we don’t have all the CPU and memory available.

6.1.1.2 Huawei p9 Lite

It is a lot similar to the previous one, except for the fact that it was released on market on February 2016, a bit earlier.
There are two main differences:

- **CPU**: it has a Kirin 650 with a quad-core 2.0 GHz Cortex-A53 and a quad-core 1.7GHz Cortex-A53.

- **Android Version**: it has EMUI 7.0.0 that means Android 7 and so API level 25, released on October 2016.

As concern screen resolution, GPU and so on their identical to the previous described model. Also this is a device daily used so has some running applications and services.

6.1.1.3 Samsung s10+ (VM)

It is not actually a real Samsung s10+ but we are emulating it, this means that the Virtual Machine in which the OS will run it’s going to work similar to that device but not equal. In fact when we created the emulator we had just to set the screen resolution and dpi, the ram we wanted to allocate for the machine and the Android version to associate but an important part is left out, CPU, that is the most important part that makes the difference.
Given these reported conditions we are not using this VM for the performance
tests, cause it will be useless, give that it doesn’t correspond to any real physical device. Instead we will use it a lot for the rest, cause it’s more practical.
Passing to the technical specification we set, it has a display of 6.4 inches with a resolution of 1440x3040 px with a dpi of 522.
I has also a RAM of 8 Gb and an Android 9 version.

6.1.2 Applications

We will describe in more detail the two applications that will be used to test the various functionalities, showing also the corresponding outputs.
The first is RuntimeCollector, an almost ”empty” application that contains only one, two activities and the RuntimeService. It was in fact developed together with IDEA in order to have a support on which test it.
The second one is EasyLib, a much more substantial and complex application, which needs more resources, a bit like all the apps we use every day. This will allow us to carry out more realistic tests.

6.1.2.1 RuntimeCollector

it is a very low content application, it includes in fact only two activities and two fragments, useful for testing all the functions inherent to the UI and the flow experience. In addition of course we find within it all the files and classes developed specifically to fulfil the tasks we have set ourselves, starting from the new service, to the listeners, configuration files, etc...
At the beginning the functionalities were activated by means of buttons and a text at the top showed the outputs, this allowed us to test the various solutions that we were able to gradually build, in fact we can see how in the MainActivity there are many buttons.
Subsequently all the processes were automated and therefore the new components were integrated and the calls were made directly from code. The text and the buttons mentioned above are therefore no longer usable, but they have been left to have a graphic that is richer in elements and therefore a richer content for our tests.
This application is the one found in the thesis project repository\(^1\) and is therefore available to everyone. If you want, you can also use it as a starting point for a new development, since in the end it is an empty project, if not for the components of IDEA.
We will mostly use it to do performance tests, since given the low workload it requires, it is the one that best advances to outline the added weight that the RuntimeService brings with it.

\(^1\)RuntimeCollector: https://github.com/KostandinCaushi/RuntimeCollector
6.1.2.2 EasyLib

They are two applications developed by me and a colleague of mine, Raffaele Bongo, for the course in Android Applications by professor Luciano Baresi.

These are intended to allow easier management of the many tasks that librarians and libraries’ users face every day. We specifically will only use EasyLib\(^2\), without EasyLib - Librarian, because it is the one with the most content and most attention to detail.

The application offers many features, in fact there are 13 goals to reach out, of which the most important: it allows you to search for the available libraries in the app and select the favourites; of these you can see all the contents such as contact information, timetables, recent news, all the available books any events with special guests; allows you to search for books by applying many different types of filters, such as a title, author, genre and library name; for each book it shows a small plot, a rating and the list of the libraries in which it is available and eventually if it is available or reserved; it allows to scan a QRcode applied to each book in order to directly open its own dedicated page; allows you to reserve a book and receive notifications when it has been delivered; and so much more.

As you can see, in this case a lot of information is handled and the processing is much more complex, so it can be considered a realistic application.

It has a lot of activities and fragments that we will use in particular to test the UI, the user flow, any bug detection and in addition it can provide us some real data on the requested performances.

6.2 Tests

Now we will finally begin to show the results of our work, in particular for each type of data that we have analysed, evaluated and for which we have developed a solution that is able to extract it, we will build some normal test cases that simulate daily behaviour.

We will mainly use EasyLib and therefore in a first phase we will also briefly describe how to integrate IDEA into the application in order to be able to exploit that specific functionality. We will then show the output created by the RuntimeService and finally we will make the necessary considerations.

6.2.1 Basic Information

6.2.1.1 Test Cases

The purpose of this test is to derive the basic parameters that are collected only once at the beginning of the service run, because these does not change over time and it would be a waste of time and resources to do it again.

Given the following goal we decided to use the less substantial app, because it

\(^2\)EasyLib: https://github.com/filus95/DIMA-Project
is easier and faster to run, and to execute it in all three devices described in
the previous section, as they are of different brands and/or models and they
all have Android versions one different from the others, so it is interesting to
be able to compare them in order to understand the different management of
the basic information, that is which ones are saved in the main files and how
they are saved.
To do this we simply included IDEA in the app project, we defined the service
in the AndroidManifest and made the MainActivity extend RuntimeActivity.
Finally we set in the map of the "configurations" the value to true of the
GET UI key, thus doing so that the extractor can get, in addition to the
DeviceInfo, also the ScreenInfo.

6.2.1.2 Results

<table>
<thead>
<tr>
<th>Device Info</th>
<th>HUAWEI p10 Lite</th>
<th>HUAWEI p9 Lite</th>
<th>Samsung s10+ [VM]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brand</td>
<td>HUAWEI</td>
<td>HUAWEI</td>
<td>google</td>
</tr>
<tr>
<td>Model</td>
<td>WAS-LX1A</td>
<td>HUAWEI</td>
<td>Model : Android SDK built for x86</td>
</tr>
<tr>
<td>Build n°</td>
<td>WAS-LX1A</td>
<td>VNS-L31</td>
<td>Build n° : sdk.gphone_x86-userdebug</td>
</tr>
<tr>
<td></td>
<td>8.0.0.376(C02)</td>
<td>VNS-L31C02B346</td>
<td>Build n° : 9 PSR1.180720.012</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4923214 dev-keys</td>
</tr>
<tr>
<td>CPU model</td>
<td>null</td>
<td>null</td>
<td>CPU model : Virtual CPU</td>
</tr>
<tr>
<td>Processors n°</td>
<td>8</td>
<td>8</td>
<td>Processors n° : 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Screen Info</th>
<th>HUAWEI p10 Lite</th>
<th>HUAWEI p9 Lite</th>
<th>Samsung s10+ [VM]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Density</td>
<td>3.0 - xxhdpi</td>
<td>3.0 - xxhdpi</td>
<td>3.0 - xxhdpi</td>
</tr>
<tr>
<td>Density Dpi</td>
<td>480</td>
<td>480</td>
<td>480</td>
</tr>
<tr>
<td>Width</td>
<td>1080px - 360.0dp</td>
<td>1080px - 360.0dp</td>
<td>1440px - 480.0dp</td>
</tr>
<tr>
<td>Height</td>
<td>1920px - 640.0dp</td>
<td>1920px - 640.0dp</td>
<td>3040px - 1013.3333dp</td>
</tr>
</tbody>
</table>

6.2.1.3 Discussion

As you can see from the results we can get all the data highlighted by the
analysis of chapter 3, except the CPU model as it is one of those data that is
managed differently depending on the type of device and the Android version.
In fact we note that it is null in the two Huawei and "Virtual CPU" in the
Samsung [VM], but in this specific case we have also to consider that we are
on a virtual machine.
We have also verified that for each device the data shown correspond to the
real one and except for the last one, everything perfectly coincides.
With regard to the Screen Info we want to emphasize that density does not
follow the real screen size but corresponds to the logical density of the display.
It is actually a scaling factor for the Density Independent Pixel unit, where 1 DIP
is equal to 1 pixel on an 160 dpi screen. We need it to convert the reported
width and height dimensions from px to dp, cause this last one is usually used as a reference to create the min width layouts for different screens.

6.2.2 View

6.2.2.1 Test Cases

We now want to test one of the most important features of IDEA, the collection of views at real time, so that we can then, on the server side, understand if the look of the application is as expected or if there are discrepancies. In this case we used EasyLib in order to have more complex and content-rich activities available. Talking about the integration part, we have extended our RunTimeActivity in the MainActivity and the RunTimeFragment in the ProfileFragment. In this way the main activity will start the RunTimeService automatically at startup, give that the service is not running, and will bind to id, thus also passing the context, useful for collection operations. Once it has finished its main checks and initializations, it calls the FragmentTransaction and sets the previously cited fragment in the activity frame, this will active the procedure explained previously in chapter 3. To sum up, it will begin by inflating the layout of the fragment on the first steps and next, to be more specific on the onResume() state of its lifecycle, it will automatically call the service to gather the runtime ui information.

Remember that everything is done on background and most important it’s completely handled by the second thread so the main one will not have slowdowns due to a drop in performance.

Moreover we have set to true the GET VIEW configuration key, in order to enable the functionality, otherwise after calling the getView() method to the service the whole process will stop and no extraction will be executed.

Now we will discuss the results that we will report later showing the structure of the view actually displayed in the device Huawei p10 Lite with a graphic image alongside.

6.2.2.2 Discussion

As expected, we obtained a long list of all the nodes in the layout and for each one we highlight the properties we selected: id, class, x and y relative positions (it means the delta from the top-left corner of the parent), width and height.

The entire content has not been reported as in this precise screen there are many RecyclerViews with as many elements inside, so the structure is quite branched and consistent. Specifically, the information reaches the texts present under the frame containing the user’s info, that is, up to the words ”Your User ID” and ”59”.

As you can well see the root node starts from the main node of the external Activity, the first ConstraintLayout that you see, this because despite the getView() has been called from within the Fragment class, to perform data
collection it is used the Context that is passed by the activity itself. However we cannot call it directly from the latter as we have to make sure that the fragment has finished its initializations and settings and therefore what better place to do it if not in the fragment itself.

Regarding the processing of these data, or the execution of particular checks and verifications, it has been left in the hands of the server-side managers so that they can use them in the way they think is right. In the next section, we will in fact suggest some solutions, reporting also a case that we ourselves have tested.

6.2.2.3 Problem Detection

There are many solutions that can be adopted, in particular we highlight one for a large-scale analysis and a second one for a more detailed study:

- **Constraints**: consists in applying some restrictions on the properties of certain components or ids, depending on how you want to manage the various elements. For example we tried to set to certain buttons, of which we wanted to check the width since they hooked to some ConstraintLayouts, the id equal to ”testButton”. We then ran the app on some devices, including tablets, and collected data from the activities in which they were present. Finally we checked that their length was not more than 300dp, as stylistically incorrect in that case, and we found that for some dpi we had this problem. This is just an example, but checks can be made on any type of data that can be extracted from those we collect.

- **Blueprint**: use a graphical tool like the one from Android Studio that converts an xml file into an image, allowing a graphic visualization of the code. You can do the same thing here cause we are providing all the needed data to create a graphical representation of some boxes, with empty content, that will represent the various nodes.

![Figure 6.1: EasyLib - ProfileFragment Blueprint](image-url)
6.2.2.4 Results

Figure 6.2: EasyLib Profile View and Layout Structure
6.2.3 SharedPreferences

6.2.3.1 Test Cases

User preferences or in general the local data saved from the application are very important because they define the "constants" that have a considerable impact on the processes executions. It should be considered the fact that among those information there could be some sensitive ones, peculiar to the user, it is therefore necessary to offer concealment countermeasures that allow the respect of privacy and the non disclosure of these to third parties if not directly requested. For the moment, no security measures have been provided as it was not within the scope of our project. In order to test this feature we have decided to use the viewMap which is saved in the preferences when the various activities are visited in order to collect information only once. Suppose we do not know how to call the file on which to read these data and see if the implementation made in any case can return the desired information. First we made sure that the renowned getView() was called by multiple components, then we put to true the GET_PREFERENCES on Configuration and finally the device in standby to start so all the service processes. Below are the obtained results and a subsequent discussion on those.

6.2.3.2 Results

```json
{
    "preferences": [
    {
        "key": "viewFile",
        "value": [
        {
            "key": "viewStringMap",
            "value": "{"com.easylib.dima.easylib.Activities.Fragments .MainActivity..."
        }
    },
    {
        "key": "Login",
        "value": [
        {
            "key": "User ID",
            "value": "8jf5a263"
        }
    ]
    ]
}]
```
6.2.3.3 Discussion

As you can see from the results obtained we have a list of preferences which contains all the files saved by the app on the device with the corresponding key, value maps. In particular, we see how there is a "Login" file inserted by me and my colleague to save a token that allows automatic authentication in the test phase. So we are actually recovering a file that has nothing to do with IDEA. Returning to the subject of sensitive data, we see how a clear parameter has been exposed here that, if misused, can damage the user’s privacy.

As for the saved screens, we see that the "viewFile" file is present, which contains a map with an entry. Specifically it is a Map <String, String> so, in order to insert the list of views in the value, we have converted it into a string, a parsing is therefore required for reading and writing.

6.2.4 User Experience Flow

6.2.4.1 Test Cases

In this case we want to focus on being able to get the user experience with the application in the best possible way, so to be able to have not only the data on the visited screens but, where possible, also the methods called and the interactions made with EasyLib, in this case.

For the integration we used the same settings of the previous functionalities plus we performed a refactoring of some clickListeners so that the default ones are not used but those offered by IDEA that allow the autonomous call to the service.

The application flow to test will be the following one: at the beginning will start from the MainActivity with the HomeFragment set; from there we will touch the profile icon in the bottom navigation bar, which will trigger the 

\[\text{onClick()}\]

method of the listener and then the call to the service; after that the ProfileFragment will appear. We have always chosen the same view as we have already provided an image of how it is done, so it is easier to visually follow the steps. From the profile we will press the "ALL FAVOURITE LIBRARIES" button, another interaction with the listeners, thus arriving at the actual screen of the list of favourite libraries.

6.2.4.2 Discussion

We have taken a snapshot of the parameters in the test results while we were debugging in order to give a general idea of both this functionality and the previous ones and show how they fit together.

As you can see there are many maps that deal with saving the various information that gradually arrives, specifically:

- \text{methodsAndViewFlow}: saves the various opened screens and called methods (touch or not) in order of arrival, is therefore the most important one of all since it gives us an overall idea of what the user is doing.
– **methodsMap**: keeps all the called methods (touch or not) in order.

– **touchMethodsMap**: all touch methods are ordered, i.e., all the actual interactions between the user and the device, so the application.

– **viewFlow**: it deals in an orderly way with the sequence of screens shown on the display.

Once the device is switched off or put on standby, the RuntimeService collects data from all the maps to compose and send the request to the server, after which it cleans all of them so as not to take up too much memory and to save different experiences in distinct flows.

### 6.2.4.3 Results

#### 6.2.5 Intents

#### 6.2.5.1 Test Cases

To test the intents, that is, the communications between the various internal and external parts of the app, EasyLib is very useful because the link between the ui thread and the back-end logic has been implemented by sending broadcast messages in.

We remind you that based on the analysis carried out in chapter 3 we have
seen how the only Intents we are able to collect are those sent to everyone, otherwise we will have to be either the system that manages them all or one of the two endpoints that sends or receives them.

To test then we use EasyLib and given the by now familiarity with the flow HomeFragment and ProfileFragment we have continued to keep the same. First of all We went to the home class in order to take the IntentFilters that it uses and set them in the BroadcastReceiver of the RuntimeService, thus registering the receiver for those types of messages. Finally we obviously activated the functionality in the Configuration file.

In the following discussion we will also describe the calls made by the MainActivity to switch from the Home to the Profile, so as to see even if those match.

6.2.5.2 Results

`get_events_per_user=Action : get_events_per_user
Component :
Type : null
DataString : null
SerializableExtra : [AnswerClasses.Event@efe13a8]
BundleExtra :
Timestamp : Fri Nov 22 03:50:34 GMT+01:00 2019,

get_user_preferences=Action : get_user_preferences
Component :
Type : null
DataString : null
SerializableExtra : [AnswerClasses.LibraryDescriptor@4cdd4d2,
AnswerClasses.LibraryDescriptor@ee7c9a3]
BundleExtra :
Timestamp : Fri Nov 22 03:50:34 GMT+01:00 2019,

user_login=Action : user_login
Component :
Type : null
DataString : null
SerializableExtra : AnswerClasses.User@953a789
BundleExtra :
Timestamp : Fri Nov 22 03:50:34 GMT+01:00 2019,

get_read_books=Action : get_read_books
Component :
Type : null
DataString : null
SerializableExtra : [AnswerClasses.Book@e520619,
AnswerClasses.Book@71e66de,
6.2.5.3 Discussion

As you can see from the results, we are mapping the following parameters: Action, which is the key from which we filter the messages; Component; Type; Datastring; SerializableExtra; BundleExtra; Timestamp. Some of these parameters are null or empty in this case because for the purposes of our application it was not useful to pass them. We used the AnswerClasses you see mapped to extras. In fact if we analyze the intents in details we can see that we have a "user_login" through which it receives a User object that contains all the user data in order to fill the fields in the profile. Subsequently the other three instead are used to add the contents to the RecyclerView below, if you go look at the image shown several sections ago you can have proof of it, and in fact they pass Event, LibraryDescriptor and Book. Among other things, this is also an excellent solution to use in any context where there is the need to propagate data, cause it’s easy to implement and use. However, it has a counter, which means that if they were sensitive data they would not be very safe.

6.2.6 Performance

6.2.6.1 Test Cases

Last but not least we need to make some performance tests in order to see if this whole process can perform well or not. We will not deal with the Heap and RAM as those are more "static" quantities and quite similar, also between different devices, but we will focus on the CPU, in order to understand which is the required workload and if there are some reachable limits. To do this we will use both the Hauwei devices and applications, mixing them to see any differences and peculiarities. We have also activated all the features of the service to guarantee the maximum load.
6.2.6.2 Results

![Figure 6.4: CPU Performance - EasyLib, RuntimeCollector](image)

6.2.6.3 Discussion

As we can see the performances between different devices are very similar, but there is a clear difference between the two applications. On the left we have the executions of the two Huawei with the EasyLib. We see that despite an initial peak that is normal, since at that moment we move from the point where the app is running in foreground to when the phone is on standby, but then it settles at around 20% / 30%, a sum that is not insignificant if we were running in the main thread and in the foreground, but this is not the case.

On the right side we have the RuntimeCollector executions that after less than 10 minutes tend to a totally irrelevant workload. This disparity is completely understandable given the fact that the latter app has no consistent logs, does not have many interactions with the user to be saved and has no intents. The time interval between scans of the RuntimeService has been set up to 10 minutes.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

Despite the great changes and the great limitations introduced by Android in the subsequent releases and despite the lack of official documentation that has to report these, we were able to carry out an in-depth analysis on the information that the operating system allows us to extract and in addition we were able to build IDEA, a tool that allows you to collect this data mostly automatically and efficiently and that is simple to integrate into your application development.

In particular in chapter 3 we have reported all the results obtained from the various analyses and tests, so for each type of data we are able to say whether it is actually possible to derive it, how to do it and above all if there are limitations in doing so. This is the crucial part on which the thesis is based, which wants to formalize through a research documentation and by testing the few but above all little news on these drastic changes in the Android world.

Subsequently, based also on considerations made regarding the already existing tools that due to the new releases have become unusable because the functionalities to which they accessed are no longer available, we have created IDEA showing a first architectural solution at the end of the chapter 4.

From there on the implementation and integration of the new components that our project aims to offer starts, that is target solutions that allow the right extraction of the requested data in ways that are as automated as possible. The whole chapter 5 deals with the creation and description of the individual classes, services and configuration files, thus going into more detail at a technical level. Specifically we talk about the RuntimeService, the main component of our library that manages all the logic and communication between the various parts and then some UI components or elements that interface with it (Activity, Fragment, Listener), that allow you to automate the processes for collecting and sending data.

Finally we integrated IDEA in some applications, also explaining the whole process, and with the use of more than one device we performed tests to show what the real output was and to discuss any problems or particular cases.
7.2 Future Work

- **Test:**
  - Provide more test cases with different device, with various brands and models, we unfortunately have physically available just hose two. In this way we will be able to see eventual differences in performance and evaluate the efficiency in last generation devices.
  - perform tests on more commercial and famous applications, since they are based on technologies and tools that are better built and more efficient.

- **UI:**
  - provide view structure data in an xml/json format, so to be easily used, edited and elaborated.
  - provide some feature for the elaboration and filtering of those data.

- **Intent:**
  - given the intercepted intents, provide a smart way get information about the extras value.

- **Time Lapse:**
  - integrate one of the already present solutions to IDEA so to have a unique collection that provides all the needed functionalities.

- **Performance:**
  - give the possibilities to collect those information in a separate interval of time, for example making an async call, in order to have more/less extracted values. For example we can have normal loop set to 1 day interval, while for CPU usage we may want every 6/7 hours.
  - provide cpu usage in a specific time of interval, it may be needed to test specific functionalities performance.

- **Listener:**
  - extend also the other event listeners in order to cover all cases and user interactions with the device.

- **Configuration:**
  - provide a graphic tool that will allow the user to select the wanted features and which will next automatically create the corresponding Configuration file.
• **Security:**

  – for now, there is no secure connection established with the server, so it may be useful to introduce some security layers or protocols.
  
  – also there is no encryption applied to the sent data, and given that inside can be also sensible data it is needed to mask them in some way.

• **Server-side Tool:**

  – provide a tool that is able to elaborate the sent data, it can graphical or not.
  
  – it can integrate the Configuration graphical tool and based on the chosen selection create a custom library to integrate, that can be downloaded by the user.
  
  – given the performance values, it can elaborate them and create some diagrams and make them available to the user. They can also be configurable be selecting the time intervals or values.
  
  – given the view information it can create a black-print of how the screen looks like or can just make some predefined checks in some values.
public class Configuration {

    // Time in Seconds
    public static final String INTERVAL = "interval";
    // SharedPreferences
    public static final String GET_PREFERENCES = "getPreferences";
    // CPU Usage
    public static final String GET_CPU_USAGE = "getCpuUsage";
    // RAM Usage
    public static final String GET_RAM_USAGE = "getRamUsage";
    // HEAP Usage
    public static final String GET_HEAP_USAGE = "getHeapUsage";
    // UI Info
    public static final String GET_UI_INFO = "getUiInfo";
    public static final String GET_VIEW = "getView";
    // Methods
    public static final String GET_METHODS = "getMethods";
    public static final String GET_TOUCH_METHODS = "getTouchMethods";
    // Obj values
    public static final String GET_OBJ_VALUES = "getObjValues";
    // Logs
    public static final String GET_LOGS = "getLogs";
    public static final String LOG_FILTERS = "LogFilters";
    // Intents
    public static final String GET_INTENTS = "getIntents";
    public static final String INTENT_FILTERS = "IntentFilters";
    // REST call data
    public static final String URL = "url";

    public HashMap<String, String> configurations = new HashMap<String, String>() {
        put (INTERVAL,"30");
        put (GET_PREFERENCES, "true");
        put (GET_CPU_USAGE, "false");
        put (GET_RAM_USAGE, "true");
        put (GET_HEAP_USAGE, "false");
        put (GET_UI_INFO, "true");
        put (GET_VIEW, "true");
        put (GET_METHODS, "false");
    };
}
```java
public HashMap<String, List<String>> filters = new HashMap<String, List<String>>() {
    // Logs
    List<String> logFilters = new ArrayList<>();
    //    logFilters.add("something");
    put(LOG_FILTERS, logFilters);

    // Intents
    List<String> intentFilters = new ArrayList<>();
    //    intentFilters.add("something");
    put(INTENT_FILTERS, intentFilters);

    ...
};
```
Payload Request

```java
public class PayloadRequest {

    // Device Info
    private String deviceInfo;
    private String uiInfo;

    // Preferences
    private Map<String, ?> preferences;

    // Performance
    private String cpuUsage;
    private String ramUsage;
    private String heapUsage;

    // LOGS
    private String log;
    private HashMap<String, String> logTagsMap;

    // Maps For Data Collected
    private HashMap<String, String> objectValuesMap;
    private HashMap<String, String> intentsMap;

    // UI
    private HashMap<String, String> viewMap;
    private List<String> viewFlow;
    private List<String> methodsAndViewFlow;

    // Methods
    private HashMap<String, HashMap<String, String>> methodsMap;
    // Contains also touchMethods
    private HashMap<String, HashMap<String, String>> touchMethodsMap;

    ...
}
```
# List of Figures

2.1 Android Architecture ........................................... 6
2.2 Android Activity Lifecycle ..................................... 8
2.3 Android Service Lifecycle ...................................... 10
2.4 Context Structure .................................................. 11
2.5 Intents Types and Actions ...................................... 12

4.1 Tinycore and CPU.Z .............................................. 24
4.2 Catlog ............................................................... 26
4.3 Functional Solution ................................................ 29

5.1 RuntimeActivity view capture flow .............................. 37

6.1 EasyLib - ProfileFragment Blueprint ........................... 51
6.2 EasyLib Profile View and Layout Structure .................. 52
6.3 Debug Snapshot of EasyLib flow ................................. 55
6.4 CPU Performance - EasyLib, RuntimeCollector ............. 58
## Listings

| 5.1 | RuntimeService - Android Manifest definition | ... | 32 |
| 5.2 | RuntimeService - isPhoneLocked()         | ... | 33 |
| 5.3 | RuntimeActivity - isServiceRunning()     | ... | 35 |
| 5.4 | RuntimeActivity - getViewInfo()           | ... | 36 |
| 5.5 | RuntimeOnClickListener - onClick()        | ... | 39 |
| 5.6 | Screen Info - getNavigationBarHeight()    | ... | 40 |
Bibliography


