COMPUTER SCIENCE AND ENGINEERING

POLITECNICO
MILANO 1863

Runtime Monitor

Un servizio di sistema per il monitoraggio delle applicazioni Android

MATTEO COLOMBO - 883114

Anno accademico 2019/2020
Relatore: Luciano Baresi

Sommario

Al giorno d'oggi nel mondo esistono più di 2 miliardi di dispositivi marchiati Android e, solamente sul Google Play Store, più di 2.8 milioni di applicazioni. Se dal punto di vista degli utenti finali la possibilità di scegliere fra centinaia di dispositivi diversi fra di loro che si distinguono sia per caratteristiche hardware che per caratteristiche software offre grandi vantaggi e permette a ciascuno di scegliere i dispositivi che meglio si adattano al proprio utilizzo, per gli sviluppatori non sempre è una cosa positiva.

Gli sviluppatori si trovano infatti a dover sviluppare e successivamente a verificare il funzionamento delle applicazioni per dispositivi con caratteristiche non solo hardware, ma anche software molto diverse fra di loro. La vastità di dispositivi diversi presenti sul mercato rende praticamente impossibile per gli sviluppatori testare le proprie applicazioni su ogni combinazione hardware-software possibile e questi devono quindi limitarsi a un numero ristretto.

Lo scopo di questa tesi è quello di studiare una possibile soluzione che permetta agli sviluppatori di verificare il funzionamento per poter correggere gli errori e valutare i cambiamenti delle proprie applicazioni anche dopo che queste sono state pubblicate sui vari marketplace.

Durante lo svolgimento della tesi è stato svolto uno studio incentrato sul funzionamento del sistema operativo, in particolare su come questo gestisce l'esecuzione delle applicazioni e dei servizi di sistema, con lo scopo di capire come fosse possibile modificarlo per poter implementare un sistema di monitoraggio in grado di raccogliere le informazioni sul funzionamento e sull'interfaccia grafica delle applicazioni per poi trasmetterle a un server remoto in modo che gli sviluppatori possano verificarne lo stato e correggere eventuali problemi.

Il risultato di questo studio è una modifica al codice sorgente del sistema operativo stesso per permettere l'implementazione di un servizio di sistema con il compito di monitorare le applicazioni. La modifica costituisce la base di un possibile sistema di monitoraggio più completo che i vari produttori di dispositivi potrebbero migliorare e sviluppare per metterlo a disposizione degli sviluppatori di applicazioni, così da fornire un sistema completo di monitoraggio in grado di valutare il funzionamento anche dopo la pubblicazione.

Il sistema, dopo l'implementazione delle modifiche, è stato compilato ed è stata generata un'immagine di sistema eseguibile sull'emulatore fornito da Google nell'SDK per sviluppatori. Questa immagine è stata utilizzata per verificare il funzionamento e l'efficienza della soluzione proposta.

Summary

Nowadays in the world there are more than 2 billion active devices running Android and, only on the Google Play Store, more than 2.8 million applications. If from a final user point of view the possibility of choosing among hundreds of devices differing not only by hardware characteristics but also by software features is a great advantage as it allows anyone to choose the devices that fit the best their needs, from a developer point of view it isn't always a positive thing.

In fact, developers find themself in a situation where they need to develop and test applications that will run on devices with very different hardware and software features that may perform in a really different way. These spread differences make almost impossible for the app developers to test their work on every possible hardware and software combinations and they are limited to a restricted number of devices.

During the execution of this thesis, it was carried out a study to understand the functioning of the Android operative system. The main focus of the study was about understanding how the OS handles the execution of applications and system services. The goal was to find a possible modification to implement a monitoring system able to collect information about UI and performances and then transmit them to a remote server to allow developers to monitor the distribution and to fix possible errors.

The results of this study is a modification of the Android sources that allowed the creation a custom system service able to monitor applications and to collect information. These modifications can be considered as the first basic step of a future monitoring system that OEMs could implement in their devices so that app developers could monitor and evaluate their work even after the publication of their apps.

After implementing the modifications, the source code was recompiled and it was generated a system image able to be run on the standard Android emulator provided by Google in the developers SDK. This system image was also used to perform tests to evaluate the performance and the correctness of the identified solution.

Indice

1 Introduzione **1**
1.1 Obiettivo 1
1.2 Risultati 2
1.3 Struttura del documento 2

2 Android **4**
2.1 Cos'è Android 4
2.2 Come funziona lo sviluppo 5
2.3 L'Architettura 6
2.3.1 Il kernel Linux 6
2.3.2 L'Hardware Abstraction Layer 6
2.3.3 L'Android System Services 6
2.3.4 Il Binder IPC 7
2.3.5 L'Application Framework 7
2.4 L'esecuzione delle applicazioni 9
2.5 La fase di avvio del sistema 11
2.6 La comunicazione fra processi 12
2.7 I componenti "front-end" del sistema 13
2.7.1 Le Activity 14
2.7.2 Le View e i Widget 15
2.7.3 I Fragment 15
2.7.4 I servizi 16
2.7.5 Gli Intent e i Broadcast Receiver 16

3 Situazione corrente e possibili soluzioni **18**
3.1 Il problema 18
3.1.1 Frammentazione software 19
3.1.2 Frammentazione hardware 20
3.1.3 Altri problemi 21
3.2 Soluzioni attualmente disponibili 21
3.2.1 Another Monitor 22

3.2.2 Developer Assistant 23
3.2.3 CPU-Z 24
3.2.4 CatLog 25
3.2.5 Google Play Developer Console 25
3.3 Valutazione delle possibili soluzioni 25
3.3.1 Soluzioni individuate 26
3.3.2 Libreria per applicazioni 26
3.3.3 Applicazione di monitoraggio 27
3.3.4 Modifica del codice compilato 28
3.3.5 Modifica del sistema operativo 29
3.3.6 Soluzione scelta 30

4 I dati collezionabili 31
4.1 Informazioni sul dispositivo 31
4.2 Informazioni riguardo le prestazioni 33
4.2.1 Prestazioni della CPU 33
4.2.2 Utilizzo della memoria 34
4.3 Informazioni riguardo l'interfaccia utente 35
4.3.1 Informazioni riguardo al Display 35
4.3.2 Informazioni riguardo alle View 36
4.4 File di Log 37

5 Implementazione 38
5.1 Creazione di un servizio di sistema 38
5.1.1 Creazione dei file del servizio 39
5.1.2 Estensione della classe SystemService.java 40
5.1.3 Esecuzione del servizio in fase di boot 41
5.1.4 Esposizione del servizio 41
5.2 Il funzionamento del servizio 42
5.3 Fase di inizializzazione 42
5.3.1 Il gestore della rete 42
5.3.2 Creazione del Process Observer 43
5.3.3 Inizializzazione del database 44
5.3.4 Registrazione dell'observer per lo schermo 45
5.3.5 Raccoglimento delle informazioni di base 45
5.4 Fase di monitoraggio 46
5.4.1 Selezione dei processi osservabili 46
5.4.2 Il Thread di monitoraggio 47
5.4.3 Registrazione dei messaggi di Log 53
5.4.4 Modifiche al ProcessRecord 54
5.5 Sincronizzazione dei dati 55

6 Risultati e valutazione **57**
6.1 Metodo di valutazione 57
6.2 Test di verifica del funzionamento 59
6.3 Test di comparazione con il sistema originale 64

7 Conclusioni e possibili sviluppi **69**
7.1 Conclusioni 69
7.2 Possibili sviluppi 71

Capitolo 1

Introduzione

Al giorno d'oggi nel mondo esistono più di 2 miliardi di dispositivi che sfruttano il sistema operativo Android [1] e, solo sul Google Play Store, sono presenti più di 2.8 milioni di applicazioni [2].

Una delle maggiori sfide degli sviluppatori è quella di riuscire a creare applicazioni la cui interfaccia utente sia in grado di adattarsi a tutti i possibili dispositivi e allo stesso tempo riuscire a garantire prestazioni sufficienti in modo da permettere al maggior numero possibile di utenti di poterle utilizzare.

A causa dell'elevata frammentazione che distingue i dispositivi Android, sia a livello hardware che a livello software, questa sfida è tutt'altro che semplice.

1.1 Obiettivo

L'obiettivo di questa tesi è quello di studiare il funzionamento del sistema operativo Android con lo scopo di individuare una possibile modifica dello stesso che permetta ai produttori di dispositivi di implementare facilmente una soluzione di monitoraggio nei propri smartphone, in modo da fornire agli sviluppatori di applicazioni una piattaforma che permetta loro di verificare il comportamento e le prestazioni delle applicazioni anche dopo la loro pubblicazione.

Le modifiche effettuate dovranno fornire un servizio di raccoglimento delle informazioni delle varie applicazioni installate sui dispositivi, sia per quanto riguarda le prestazioni durante l'esecuzione, sia per quanto riguarda l'interfaccia grafica. Le informazioni dovranno essere salvate in locale sul dispositivo e poi caricate su un server esterno, la cui creazione non è oggetto di

questa tesi, che si occuperà di elaborare i dati e metterli a disposizione degli sviluppatori.

1.2 Risultati

Per raggiungere gli obiettivi descritti nella sezione precedente si è studiato il funzionamento del sistema operativo concentrandosi in particolare sulle applicazioni e sui servizi cosiddetti di sistema, che sono fondamentali per l'esecuzione di Android.
I risultati ottenuti sono quindi:

- Un'analisi del funzionamento del sistema operativo Android, concentrata in particolare su come il sistema gestisce l'esecuzione delle varie applicazioni e dei processi ad esse collegati.
- Una modifica del sistema operativo per permettere la creazione di un servizio di sistema in grado di raccogliere le informazioni riguardo al funzionamento delle applicazioni in esecuzione e la modifica di alcuni dei servizi e dei componenti già presenti nel sistema in modo da poter fornire le informazioni necessarie al servizio di monitoraggio.
- La creazione di un'immagine di sistema in grado di essere eseguita dall'emulatore Android presente nell'SDK standard fornito da Google.
- Test comparativi tra due emulatori dotati entrambi di Android 9, uno dei quali presenta la versione originale, mentre l'altro una versione appositamente modificata.

1.3 Struttura del documento

- Il capitolo 2 ha il compito di spiegare il funzionamento del sistema operativo e dei suoi componenti, con lo scopo di introdurre i concetti fondamentali per il proseguimento del documento;
- Il capitolo 3 ha lo scopo di fornire una descrizione generale del problema e di presentare lo stato attuale, mostrando alcuni esempi delle soluzioni attualmente disponibili per gli sviluppatori e analizzando le possibili soluzioni individuate;
- Il capitolo 4 ha lo scopo spiegare sulla raccolta di quali informazioni ci si è concentrati e dove queste informazioni sono reperibili all'interno del sistema operativo stesso;

- Il capitolo 5 ha il compito di descrivere come è stata svolta l'implementazione della soluzione individuata, spiegando quali sono state le difficoltà e come si è deciso di procedere per l'implementazione;
- Il capitolo 6 si occupa di mostrare i risultati ottenuti e di descrivere i test comparativi che sono stati effettuati per valutare l'efficienza della soluzione individuata e il suo impatto sulle risorse di sistema.
- Il capitolo 7 ha lo scopo di mostrare qualche considerazione e di trarre le conclusioni finali. Propone inoltre alcune possibili futuri cambiamenti.

Capitolo 2

Android

Lo scopo di questo capitolo è quello di fornire sia una descrizione generale di Android con lo scopo di introdurre alcune nozioni fondamentali per poter comprendere il funzionamento del sistema operativo stesso e le scelte effettuate nel proseguimento di questa tesi.

2.1 Cos'è Android

Android è un sistema operativo per dispositivi mobili basato su una versione modificata del kernel Linux, il cui codice è completamente Open Source [3]. Il suo sviluppo è gestito dalla Open Handset Alliance [4], un consorzio di aziende guidate da Google che ne costituisce il maggior contribuente.

La maggior parte del codice di Android è costituito dall'Android Open Source Project (AOSP) che, grazie al fatto che è rilasciato sotto licenza Apache, ne ha permesso lo sviluppo per numerosi dispositivi elettronici come TV e console, oltre che per i più comuni smartphone tablet.

Android costituisce la base di vari ecosistemi closed e open source, il più famoso dei quali è quello di cui Google è proprietaria e che è costituito da vari servizi e applicazioni che solitamente sono preinstallati su smartphone e tablet in commercio. Questa suite chiamata Google Mobile Services (GMS), che include tra le varie funzionalità il Google Play Store, solitamente viene data in licenza ai vari produttori di dispositivi, a patto che i loro smartphone e tablet rispettino determinate caratteristiche e superino una serie di test imposti da Google stessa.

2.2 Come funziona lo sviluppo

Lo sviluppo del sistema operativo, come spiegato al punto precedente, viene gestito da Google che decide privatamente tutti i futuri sviluppi e i cambiamenti nel funzionamento del sistema, mentre ai contributori viene permesso solamente di effettuare suggerimenti per correzioni di errori, miglioramenti di applicazioni e altri cambiamenti che non vanno ad alterare il funzionamento delle API di sistema.

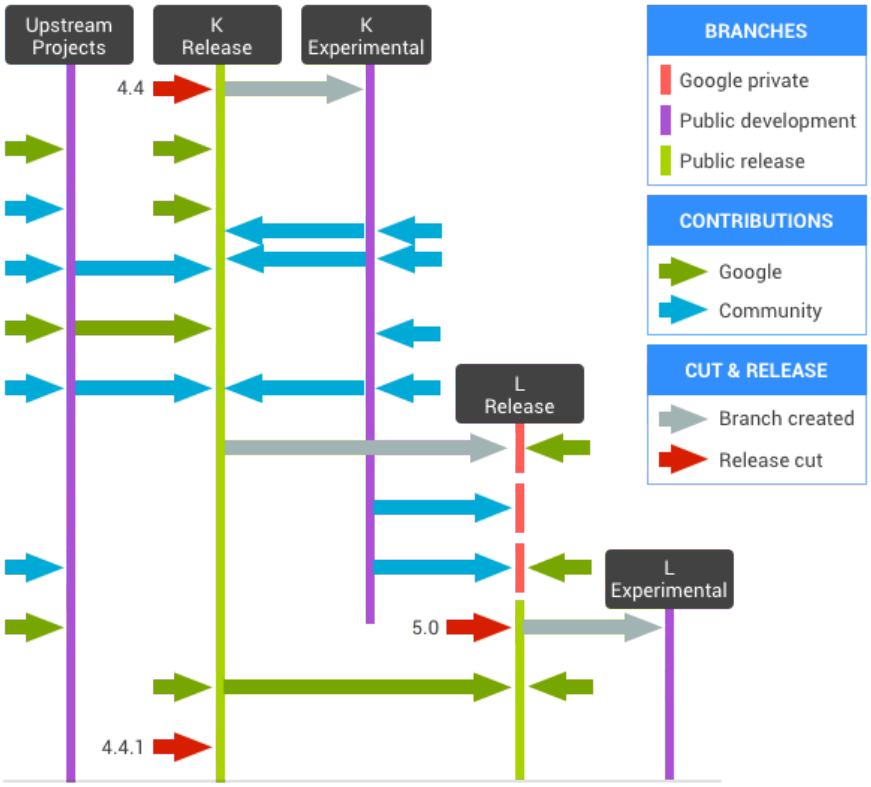

Figura 2.1: Metodo di sviluppo dell'AOSP

Il codice del sistema operativo è organizzato in più di 400 repository git chiamati *Upstream Projects*, ognuno dei quali è organizzato in vari branch che rappresentano le varie release del sistema operativo. In particolare, come si può vedere in figura 2.1, esistono due branch per ogni versione del sistema operativo:

- Release;
- Experimental.

Release rappresenta la versione corrente del sistema operativo, quella che gli OEM utilizzano per la creazione dei proprio dispositivi e per gli aggiornamenti di sistema, mentre Experimental rappresenta un ramo che può essere utilizzato sia da parte degli OEM che da parte della community per suggerire correzioni. In caso le correzioni vengano approvate, saranno incluse anche nel ramo Release al rilascio successivo. Oltre a questi due rami Google lavora anche su un terzo ramo, privato, nel quale sviluppa le future versioni di Android, che rimangono segrete fino a quando Google stessa non decide di rilasciarle [5].

2.3 L'Architettura

Come si può vedere dalla figura 2.2, Android è costituito da un'intera pila software alla cui base vi è il kernel Linux [6].

2.3.1 Il kernel Linux

Alla base dello stack troviamo una versione modificata del kernel Linux che viene utilizzato sia dal resto del sistema, sia dalle applicazioni in caso di necessità e che offre una base stabile e testata in grado di fornire buone prestazioni e un livello di sicurezza molto funzionale.

2.3.2 L'Hardware Abstraction Layer

Sopra al kernel Linux è posizionato l'Hardware Abstraction Layer (HAL), che è un'interfaccia che viene fornita agli sviluppatori di hardware per implementare i driver per i loro componenti. L'utilizzo di questa interfaccia permette di standardizzare il modo in cui i componenti del sistema accedono all'hardware del dispositivo e al tempo stesso permette di semplificare la creazione dei dispositivi e i futuri aggiornamenti del sistema in quanto introduce una separazione fra il sistema stesso e il software che si interfaccia con l'hardware.

2.3.3 L'Android System Services

Sopra all'HAL troviamo i servizi di sistema di Android, che rappresentano il cuore del sistema operativo e insieme al Binder IPC e all'Application Framework costituiscono l'Android Framework. I servizi di sistema sono divisi in due categorie: quelli che fanno parte del Media Server e quelli che fanno

parte del System Server. I primi sono quelli che si occupano della riproduzione e dell'acquisizione dei contenuti multimediali, mentre i secondi sono quelli che si occupano della gestione e del funzionamento del sistema.

2.3.4 Il Binder IPC

Il Binder IPC [7] è un meccanismo che permette la comunicazione fra i vari processi del sistema ed è fondamentale per permettere alle applicazioni che sfruttano i componenti dell'Application Framework di dialogare con i servizi di sistema o fra di loro.

2.3.5 L'Application Framework

L'ultimo livello del framework è costituito dall'Application Framework che è scritto in Java ed è costituito da una serie di API pubbliche che vengono messe a disposizione alle applicazioni create dagli sviluppatori indipendenti che possono quindi sfruttarle per la creazione delle applicazioni e per il dialogo con i servizi di sistema.

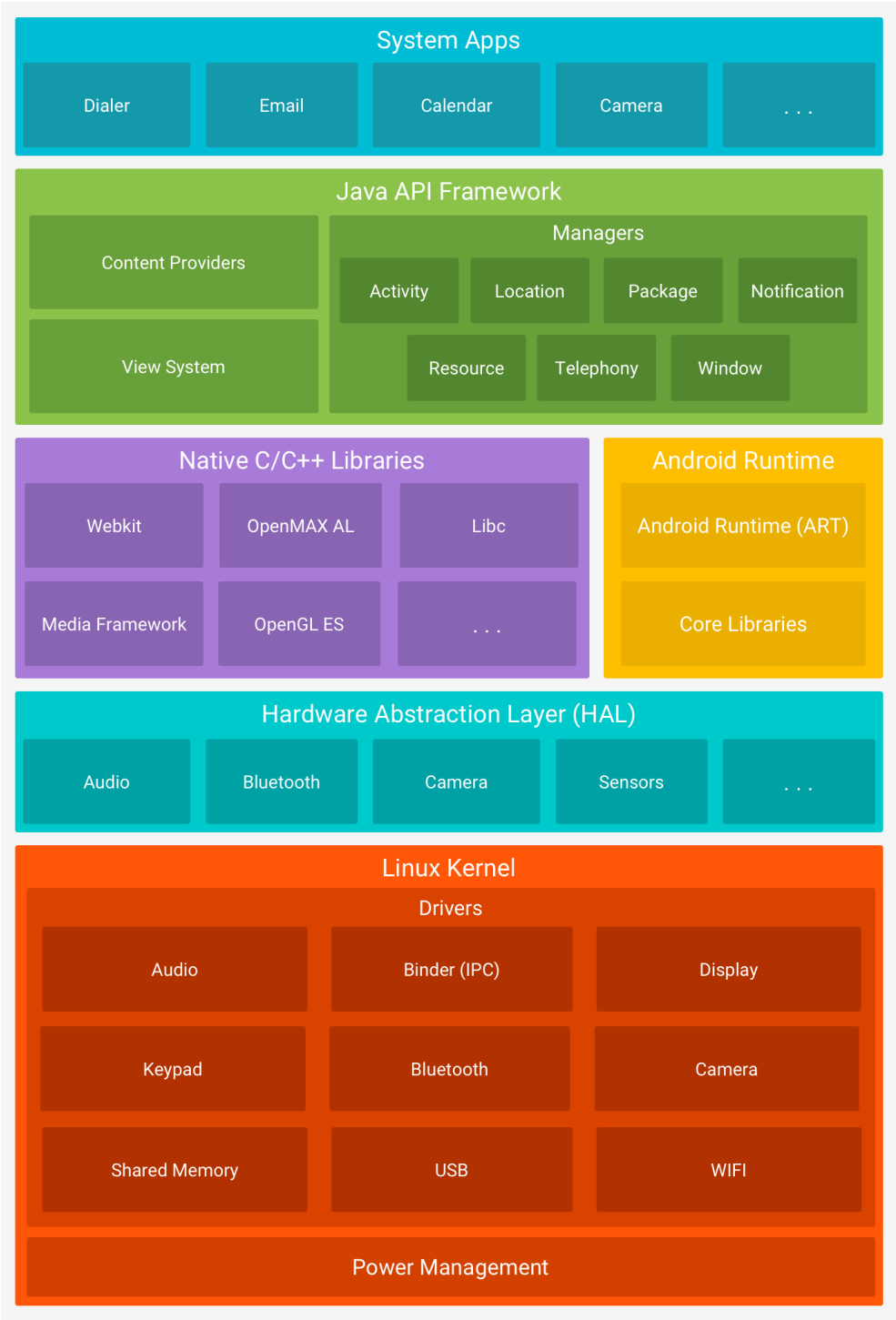

Figura 2.2: L'Architettura di un sistema Android

2.4 L'esecuzione delle applicazioni

Quando un'applicazione viene avviata, il sistema le assegna un processo dedicato e genera un Runtime dedicato che si occuperà di eseguirla. Sia per l'esecuzione delle applicazioni Android e per gran parte del framework stesso, viene utilizzato l'Android RunTime [8] (ART), un componente in grado di eseguire bytecode nel formato Dex: un bytecode pensato per Android che viene generato direttamente dal codice compilato Java.

Precedentemente, al posto di ART, veniva usata Dalvik [9]: una Java Virtual Machine anch'essa in grado di eseguire codice Dex. A partire da Android 5.0 Dalvik è stata prima affiancata e poi sostituita da ART, che offre numerosi vantaggi sia a livello di prestazioni che di ottimizzazione del codice una volta che l'applicazione è stata installata.

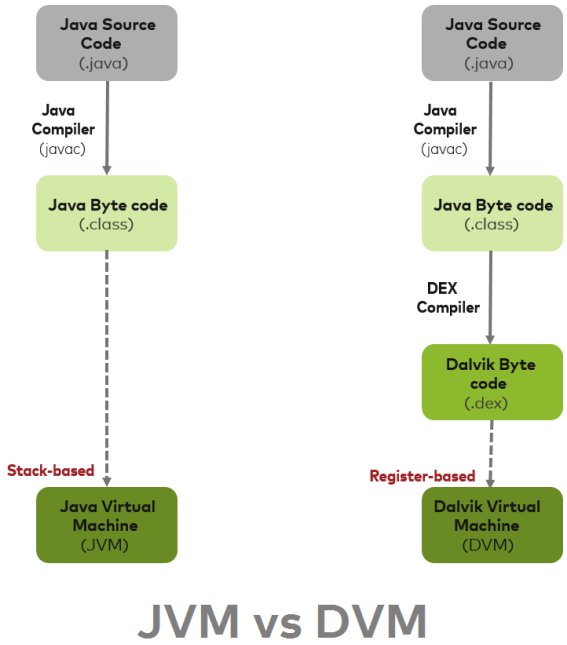

Figura 2.3: Confronto fra Java Virtual Machine e Dalvik Virtual Machine

Quello che avviene [10], a differenza della Java Virtual Machine comune (figura 2.3), è che il codice compilato nel formato Class non viene convertito direttamente in machine code dal Runtime, ma viene prima convertito in Dalvik ByteCode (Dex) al momento dell'installazione e poi, a seconda che il Runtime sia ART o Dalvik, viene convertito e ottimizzato in modi differenti, come mostrato in figura 2.4.

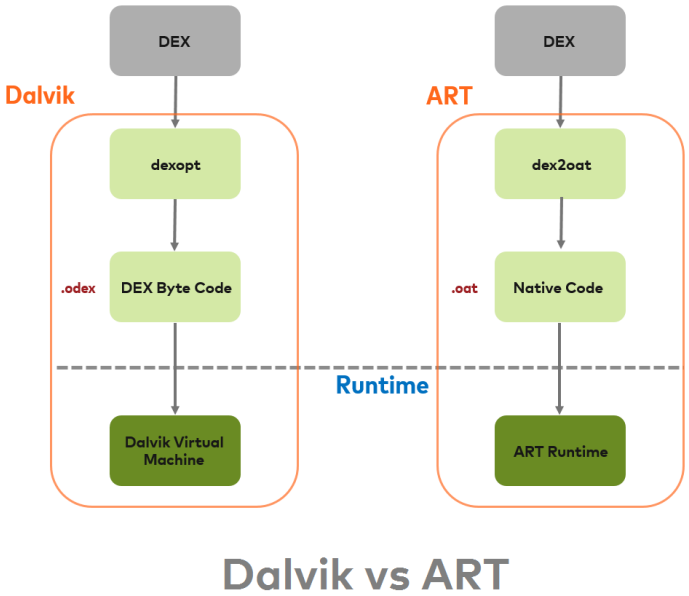

Figura 2.4: Differenze fra Dalvik e ART

L'utilizzo di una virtual machine per l'esecuzione delle applicazioni offre vari vantaggi, fra i quali:

- L'isolamento dell'applicazione dal resto del sistema, che si rivela utile in caso di applicazioni malware;
- Maggiore compatibilità delle applicazioni, che possono essere installate su dispositivi con architetture hardware differenti senza la necessità di modifiche da parte degli sviluppatori.

2.5 La fase di avvio del sistema

La fase di avvio del sistema, come mostrato in figura 2.5, è divisa in vari passaggi in cui vengono inizializzati i vari componenti e termina quando i servizi di sistema sono stati caricati e gli utenti possono avviare le applicazioni installate [11][12][13].

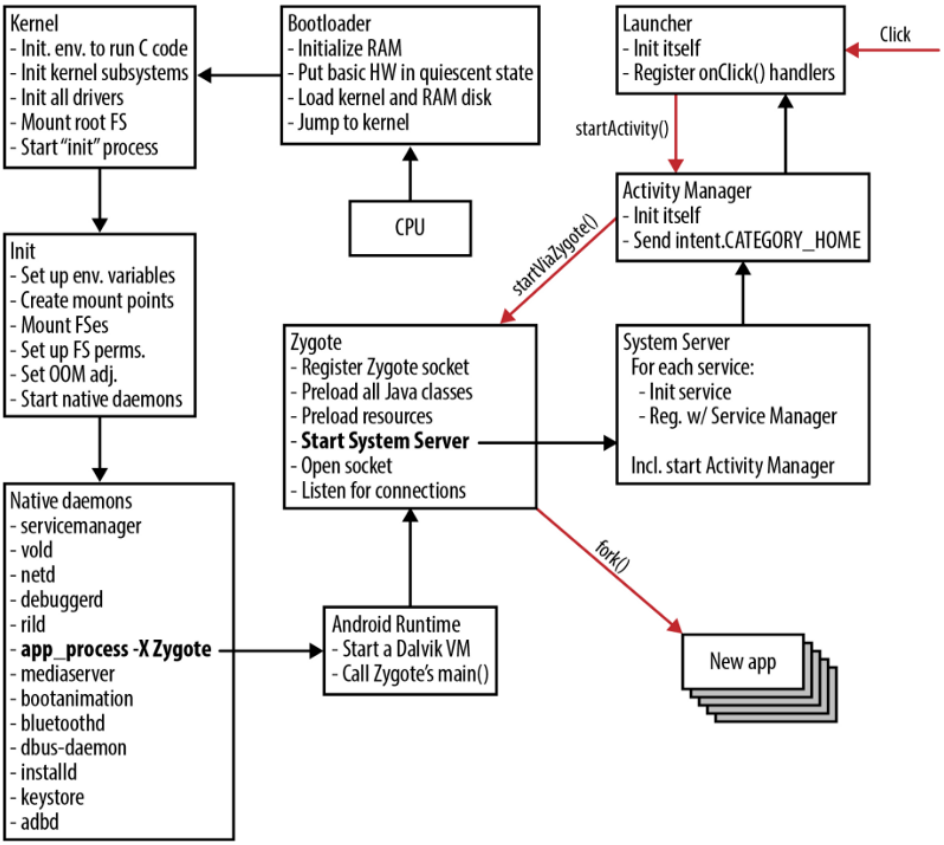

Figura 2.5: Sequenza di boot di un dispositivo Android

La prima cosa che viene avviata è il bootloader, che procede ad avviare il kernel Linux; quest'ultimo compie varie operazioni di inizializzazione e, fra queste, esegue il processo Init.

Il processo Init ha vari compiti, alcuni dei quali sono avviare il demone del Service Manager e chiamare il comando app_process, che sia occupa di creare la prima istanza della virtual machine ART e poi di chiamare il comando main del processo Zygote.

Zygote costituisce il primo processo di Android che viene avviato ed è il padre di tutti i processi che verranno creati successivamente. Esso ha il

compito di precaricare tutte le risorse di sistema e le classi che verranno successivamente usate dal framework Android, in modo da poter sfruttare la condivisione di queste con i processi figli e velocizzare il caricamento delle applicazioni. Zygote infatti eseguirà un fork e genererà un figlio ogni qual volta che il framework avrà la necessità di creare un nuovo processo per il sistema stesso o per una qualsiasi applicazione.

Dopo aver cariato le risorse, Zygote esegue il suo primo fork ed avvia il System Server, un componente del sistema che si occupa di avviare e mantenere attivi tutti i servizi di sistema, chiamati Manager. In caso anche uno solo di questi si interrompa, il System Server procederà al riavvio di tutti i servizi.

Fra i servizi avviati dal System Server, uno dei più importanti è l'Activity Manager, che ha il compito di gestire il ciclo di vita delle Activity, il loro stack e la creazione dei thread e dei processi associati.

Una volta terminata l'inizializzazione, l'Activity Manager ha anche il compito di comunicare a Zygote l'intenzione di avviare il Launcher delle applicazioni, che può essere considerata la "Homepage", per poi mettersi in ascolto in attesa che questa invii il comando di avvio per altre applicazioni.

2.6 La comunicazione fra processi

La comunicazione fra i vari processi è fondamentale in quanto consente alle applicazioni di contattare i servizi di sistema e ai vari servizi di comunicare fra di loro, permettendo il funzionamento del framework stesso.

Per implementare la comunicazione IPC sono necessari quattro componenti:

- Il servizio.
- Delle API pubbliche;
- Un oggetto chiamato Proxy;
- Un oggetto chiamato Stub.

Il servizio è costituito da uno o più file e la sua istanza è un singleton, unico per tutto il sistema. Viene avviato dal System Server e deve implementare lo Stub, che è un'interfaccia definita in un file con formato AIDL [14] che specifica quali metodi il servizio deve fornire.

Le API pubbliche sono costituite anch'esse da uno o più file e sono solitamente un singleton nella virtual machine in cui è in esecuzione l'applicazione. Questo implica che possano esistere contemporaneamente più istanze di una

determinata API pubblica che dialogano con il singleton del servizio. Per dialogare con il servizio, le API pubbliche devono ottenere il riferimento ad esso chiedendolo al Service Manager, un servizio che si occupa di fornire i riferimenti ai servizi di sistema. Una volta ottenuto il riferimento al servizio, le API pubbliche dialogano con il servizio tramite una sua rappresentazione, che è il Proxy [15], come mostrato in figura 2.6.

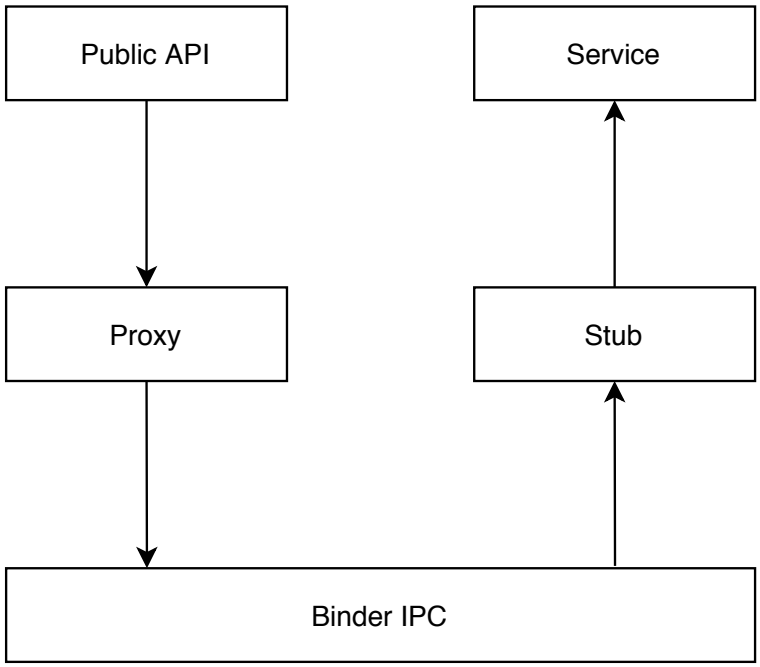

Figura 2.6: Rappresentazione di una comunicazione IPC

Perché questo sistema possa funzionare in sicurezza, il framework sfrutta anche il modulo di sicurezza Security-Enhanced Linux (SELinux) del kernel Linux, che permette di specificare in che modo i processi possono cercare e dialogare con i servizi di sistema. Per esempio è possibile abilitare delle API solamente alle applicazioni privilegiate [16].

2.7 I componenti "front-end" del sistema

In quest'ultima parte del capitolo saranno spiegati i componenti di Android che vengono utilizzati per la maggior parte dagli sviluppatori o che sono effettivamente visibili agli utenti, cioè quelli relativi al funzionamento dell'interfaccia grafica o delle applicazioni in generale.

2.7.1 Le Activity

Le Activity fungono da punto d'ingresso delle applicazioni e permettono l'interazione degli utenti. Forniscono una finestra, in cui l'applicazione disegna la propria interfaccia grafica, e gestiscono la risposta all'interazione degli utenti con lo schermo del dispositivo. Solitamente le applicazioni sono composte da più di una schermata e quindi, da più di un'Activity. Queste sono fra di loro generalmente indipendenti, anche se può esistere una gerarchia e anche se possono comunicare; per esempio un'Activity con lo scopo di chiedere dei dati all'utente potrebbe comunicare le risposte all'Activity padre che la ha invocata [17].

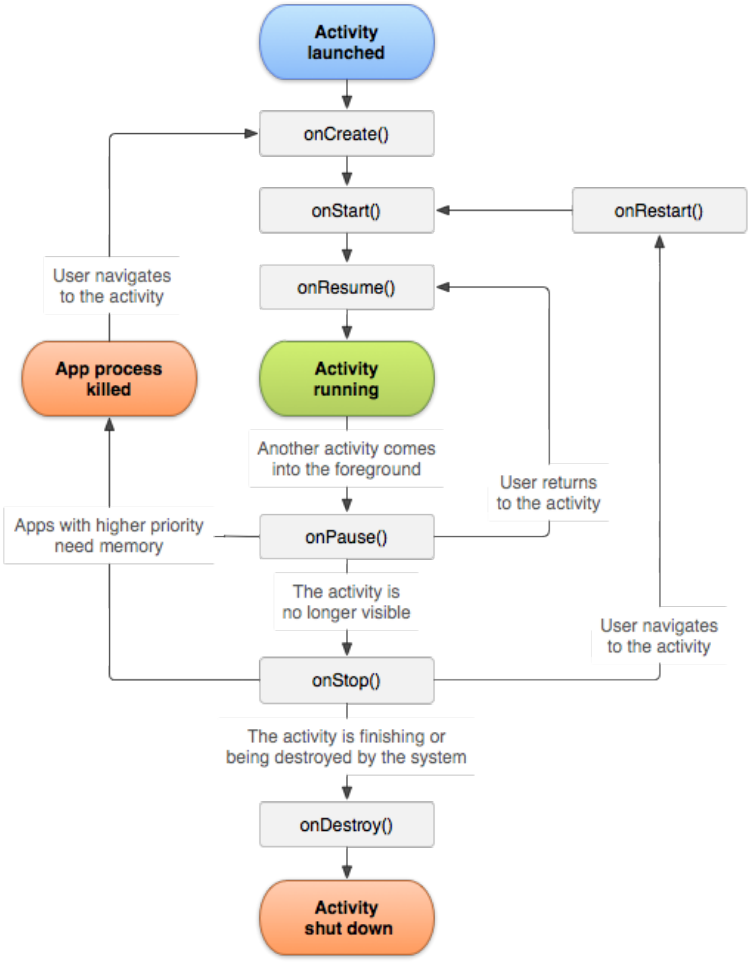

Figura 2.7: Ciclo di vita di un'Activity Android

Le Activity hanno un ciclo di vita (mostrato in figura 2.7) che viene gestito dall'Activity Manager e che dipende sia dall'interazione dell'utente che dallo stato del dispositivo [18].

Come mostrato dal grafico relativo al ciclo di vita, le Activity possono trovarsi in vari stati che dipendono dalla visibilità della loro interfaccia grafica:

- Foreground: quando l'Activity è visibile in primo piano ed il focus dell'utente è su di essa;
- In pausa: quando l'Activity è visibile, ma non è più in primo piano e il focus dell'utente è su qualcos'altro;
- In background: quando l'Activity non è più visibile.

2.7.2 Le View e i Widget

Le View sono i componenti che rappresentano un singolo blocco dell'interfaccia utente delle applicazioni Android. Occupano un'area rettangolare sullo schermo e gestiscono sia gli eventi che la propria renderizzazione. La classe View è la classe padre da cui derivano altri due tipi di componenti: i ViewGroup e i Widget [19].

I Widget sono dei componenti interattivi dell'interfaccia utente che permettono all'utente di effettuare operazioni di input/output. Sono componenti come i bottoni, i campi di testo e le etichette. I ViewGroup sono invece dei componenti che fungono da contenitori di altre View e sono utilizzati per costruire il layout dei componenti nelle Activity.

Tutte le View sullo schermo sono organizzate in un singolo albero alla cui radice vi è un elemento chiamato ViewRoot, mentre le foglie sono costituite dai Widget e i nodi intermedi dai ViewGroup.

2.7.3 I Fragment

I Fragment sono dei componenti modulari delle Activity che hanno un ciclo di vita proprio, ricevono gli input separati e possono essere rimossi o aggiunti a piacimento quando l'Activity è in esecuzione [20].

Quando un Fragment viene aggiunto in un'Activity, mediante un ViewGroup, genera un proprio layout con View, ViewGroup e Widget indipendenti da quello dell'Activity padre; anche se possono esistere Fragment senza interfaccia grafica.

La gestione dei Fragment viene fatta da un componente del sistema chiamato FragmentManager, che oltre ad aggiungerli e rimuoverli, si occupa anche di visualizzare gli effetti di transizione e in caso venga richiesto dallo sviluppatore, mantiene una pila LIFO chiamata Back Stack che permette la navigazione fra di essi come se fossero schermate diverse dell'applicazione.

2.7.4 I servizi

I servizi sono componenti pensati per eseguire operazioni in background, senza fornire un'interfaccia utente. Una volta che un servizio di un'applicazione viene avviato, può rimanere attivo a lungo anche dopo che l'utente ha cambiato applicazione. Inoltre un servizio può fornire comunicazione IPC, anche se in modo leggermente diverso da quella dei servizi di sistema, e permette ad altri componenti di collegarsi ad esso [21].

Esistono tre tipi di servizi:

- Foreground: un tipo di servizio che viene avviato e che esegue operazioni che sono visibili dall'utente, come può essere un riproduttore musicale. Per questo motivo, Android richiede che un servizio in Foreground mostri una notifica per avvisare l'utente della sua esecuzione.

- Background: un tipo di servizio che viene avviato dall'applicazione ma che non viene direttamente notato dall'utente, per esempio un servizio di ottimizzazione dei file.

- Bound: un tipo di servizio che rimane in esecuzione fino a quando un componente è collegato ad esso. Quando un componente di un'applicazione si collega ad esso, il servizio offre una comunicazione di tipo client-server che permette al componente di inviare richieste, ricevere risposte ed effettuare comunicazione IPC.

2.7.5 Gli Intent e i Broadcast Receiver

Gli Intent [22] sono delle strutture dati che contengono informazioni riguardo ad azioni che devono essere compiute. Possono essere utilizzati per avviare Activity, inviare messaggi che saranno ricevuti dai BroadcastReceiver [23] e per comunicare con i servizi in background. In pratica sono delle strutture dati usate per scambiare messaggi fra componenti diversi della stessa applicazione, o fra applicazioni differenti.

Esistono due tipi di intent:

- Impliciti
- Espliciti

Mentre i primi sono generici e specificano solamente l'azione che dovrà essere compiuta, i secondi specificano anche il destinatario che sarà quindi l'unico a cui saranno recapitati.

I destinatari per ricevere gli Intent sfruttano i BroadcastReceiver, che sono dei componenti che permettono di registrarsi per la ricezione di eventi di sistema, come possono essere gli Intent, a prescindere dallo stato dell'applicazione.

Capitolo 3

Situazione corrente e possibili soluzioni

La prima parte di questo capitolo si concentra sul descrivere quali sono i principali problemi che rendono più difficoltoso il testing delle applicazioni e quali sono le soluzioni attualmente disponibili sul mercato che possono essere utilizzate dagli sviluppatori. Nella seconda parte vengono invece esposte alcune possibili soluzioni che sono state individuate e vengono successivamente analizzate.

Nonostante sul mercato esistano molti dispositivi che non utilizzano i servizi offerti da Google, per esempio i nuovi dispositivi marchiati Huawei o tutti i dispositivi attivi in Cina o nei paesi dove Google non è presente, l'analisi in questo e nei capitoli seguenti si concentrerà solamente sui dati ufficiali forniti dal Google Play Store.

3.1 Il problema

Come spiegato nel capitolo introduttivo, quello che mi ha spinto a svolgere questa tesi è la difficoltà che spesso gli sviluppatori incontrano nel testare le applicazioni su tutti i possibili tipi di dispositivi e nel verificarne il funzionamento una volta che queste sono state pubblicate e scaricate dagli utenti.

Ho cercato di individuare quali sono le possibili cause di queste difficoltà, in modo da poter studiare una soluzione, e ne ho identificate alcune. Le tre principali cause individuate sono le seguenti:

- Frammentazione software;
- Frammentazione hardware;

- Modifiche apportate dagli OEM.

3.1.1 Frammentazione software

La frammentazione software consiste nella poca omogeneità della distribuzione del sistema operativo nei dispositivi che risultano attualmente in uso da parte degli utenti. Questo problema è dovuto al fatto che gli aggiornamenti dei dispositivi vengono gestiti direttamente dai produttori e non in modo automatico da Google. Ogni volta che una nuova versione di Android viene rilasciata, perché questa venga resa disponibile per un dispositivo è necessario che i produttori la implementino e superino i test di compatibilità, per poi rilasciarla agli utenti.

Spesso però i produttori, soprattutto per quanto riguarda i dispositivi di fascia bassa, trascurano gli aggiornamenti o li forniscono in ritardo rispetto a quello che è il ciclo di sviluppo e di rilascio di Google. Oltre a questi ritardi, è necessario considerare anche il fatto che il supporto agli aggiornamenti è garantito, per la maggior parte dei telefoni, solamente per 2 anni [24].

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.3%
4.1.x	Jelly Bean	16	1.2%
4.2.x		17	1.5%
4.3		18	0.5%
4.4	KitKat	19	6.9%
5.0	Lollipop	21	3.0%
5.1		22	11.5%
6.0	Marshmallow	23	16.9%
7.0	Nougat	24	11.4%
7.1		25	7.8%
8.0	Oreo	26	12.9%
8.1		27	15.4%
9	Pie	28	10.4%

Figura 3.1: Distribuzione delle versioni Android

Tenendo conto di quanto appena detto e nonostante gli sforzi di Google per semplificare gli aggiornamenti, è sufficiente osservare l'ultimo aggiornamento della tabella delle distribuzioni di Android (figura 3.1) fornita da Google per rendersi conto di quanto il sistema sia frammentato [25].

Questa frammentazione è un problema per gli sviluppatori in quanto introduce la necessità di testare le applicazioni su tutte le versioni del sistema operativo supportate dall'applicazione, dato che nonostante la presenza di librerie di supporto per fornire retrocompatibilità, a volte a causa di API nuove o deprecate e restrizioni imposte da Google il funzionamento del sistema potrebbe cambiare da una versione all'altra.

3.1.2 Frammentazione hardware

La frammentazione hardware consiste nella poca omogeneità della distribuzione dei componenti hardware nei vari dispositivi presenti sul mercato, per un esempio si veda la figura 3.2. Questa frammentazione deriva dal fatto che i vari produttori sono liberi di scegliere i componenti senza alcuna restrizione e possono introdurne di nuovi, per esempio telefoni con display pieghevoli o con aspect ratio differenti da quelli esistenti.

I principali problemi che derivano dalla frammentazione hardware sono due:

- Problemi di interfaccia utente;
- Problemi di prestazioni.

I problemi di prestazioni dipendono dal fatto che sul mercato esistono dispositivi con componenti hardware come CPU, RAM e GPU molto diversi fra di loro e le applicazioni che effettuano operazioni pesanti o che richiedono l'utilizzo della rete potrebbero comportarsi in modo diverso sui vari dispositivi.

I problemi di interfaccia utente derivano dalle differenti dimensioni e densità di pixel dei display, la cui combinazione può cambiare il modo in cui l'applicazione viene renderizzata. Android permette lo sviluppo di differenti interfacce grafiche, in modo da selezionare automaticamente quella che meglio si adatta alle caratteristiche del dispositivo, ma resta comunque difficile per gli sviluppatori verificare il corretto funzionamento delle applicazioni su tutte le possibili combinazioni.

	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small	0.4%				0.1%	0.1%	0.6%
Normal		0.9%	0.3%	24.0%	37.7%	23.6%	86.5%
Large		2.4%	1.9%	0.6%	1.6%	1.7%	8.2%
Xlarge		3.1%		1.3%	0.6%		5.0%
Total	0.4%	6.4%	2.2%	25.9%	40.0%	25.4%	

Figura 3.2: Distribuzione hardware dei display dei dispositivi Android

3.1.3 Altri problemi

Altri problemi possono essere introdotti dalle differenti implementazioni che gli OEM scelgono di effettuare sui dispositivi o dalle modifiche con cui questi li personalizzano, che possono cambiarne sia le funzionalità che l'interfaccia grafica.

Google permette infatti ai produttori di dispositivi di personalizzare la versione di Android che monteranno, a patto che vengano superati i test di compatibilità. Al di fuori dei test di compatibilità i produttori possono però effettuare modifiche a proprio piacimento, sia alterando il funzionamento del sistema, sia aggiungendo o rimuovendo nuove funzionalità.

Queste modifiche sono impossibili da testare per gli sviluppatori, dato che sono strettamente legate al dispositivo e quindi impossibili da prevedere. Un esempio di queste differenze è il Media Player, il componente che si occupa di riprodurre audio e video, integrato nel sistema operativo che i produttori possono implementare in modi differenti.

Questo problema non fa altro che rendere più difficile il test delle applicazioni e mostra come vi sia la necessità di un sistema di monitoraggio che permetta di verificarne il funzionamento sui dispositivi in possesso degli utenti finali.

3.2 Soluzioni attualmente disponibili

Ho effettuato una ricerca per scoprire quali soluzioni sono disponibili sul mercato e ne ho trovate alcune che sono in grado di raccogliere le informazioni, anche se nessuna soddisfa completamente le nostre necessità.

3.2.1 Another Monitor

AnotherMonitor [26] è un'applicazione che può essere installata sui dispositivi e permette agli utenti di raccogliere le informazioni sull'utilizzo della CPU e della memoria in tempo reale. Queste informazioni possono essere raccolte sia in generale che relativamente a un singolo processo/applicazione e possono essere visualizzate tramite dei grafici o esportate tramite dei file CSV.

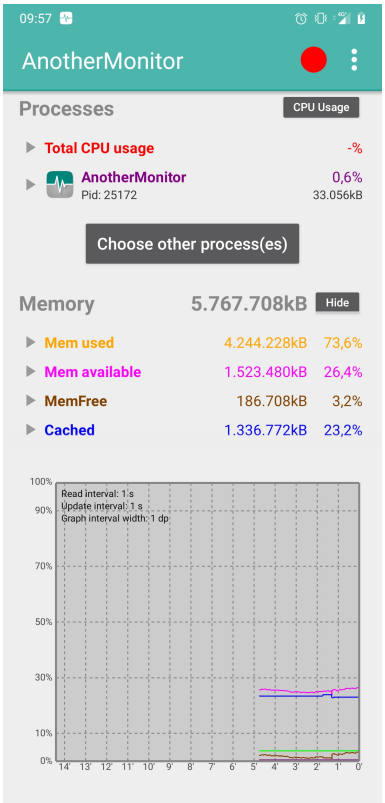

Figura 3.3: Un esempio dei dati raccoglibili da AnotherMonitor

A seguito dei vari aggiornamenti di Android, Google ha però introdotto varie limitazioni alle informazioni a cui le applicazioni possono accedere; infatti è prima stata limitata la possibilità di conoscere quali altri processi sono attivi e poi è stata limitata la possibilità di risalire alle informazioni sull'utilizzo della CPU, anche dell'applicazione stessa [27]. A causa dei motivi appena descritti, l'applicazione risulta inutile per le ultime versioni del sistema operativo.

3.2.2 Developer Assistant

Developer Assistant [28] è un'applicazione in grado analizzare l'interfaccia grafica dell'applicazione attualmente in esecuzione. Permette di verificare la struttura dell'albero delle View, verificare il layout, gli stili e le traduzioni direttamente dal dispositivo.

Le informazioni che l'applicazione riesce ad ottenere sono complete e dettagliate, ma le funzionalità sono comunque limitate in quanto per potervi accedere è necessario impostare l'applicazione come assistente dell'utente ed è necessario che sia l'utente stesso ad avviarla manualmente.

Una volta che l'applicazione è stata richiamata, riesce a leggere il contenuto dell'Activity in primo piano e ad analizzarla grazie alle Google Assist API [29].

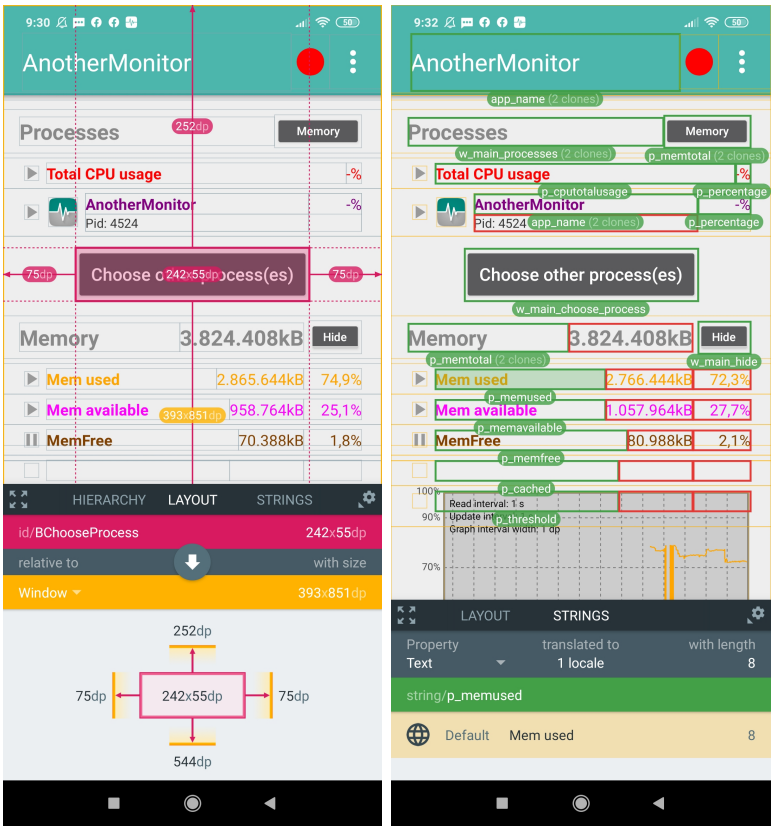

Figura 3.4: Analisi di AnotherMonitor tramite Developer Assistant

3.2.3 CPU-Z

CPU-Z [30] è la versione per Android di un popolare programma per computer Windows che permette di visualizzare varie informazioni sul sistema:

- Informazioni sulla CPU: nome, architettura e velocità di ogni core;
- Informazioni sul sistema: modello e marca del dispositivo, risoluzione dello schermo, RAM e memoria;
- Informazioni sulla batteria: livello, stato, temperatura e capacità;
- Informazioni sui sensori.

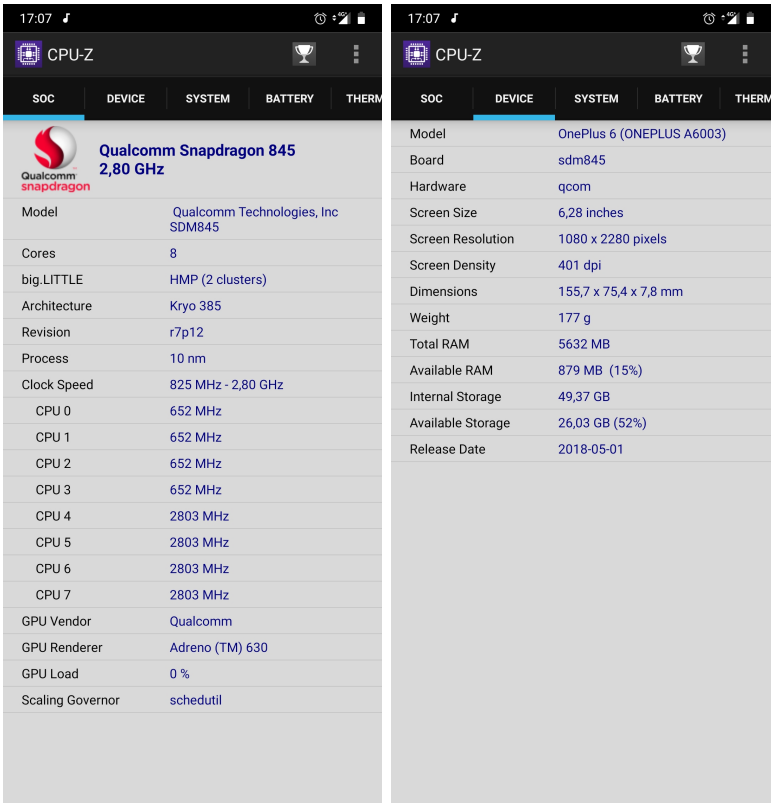

Figura 3.5: Un esempio dei dati raccoglibili da CPU-Z

Anche in questo caso è l'utente che avviando l'applicazione può visualizzare le informazioni riguardo al proprio dispositivo.

3.2.4 CatLog

CatLog [31] è un'applicazione molto famosa che veniva utilizzata per poter leggere le informazioni di Log delle applicazioni Android. L'Applicazione permette di filtrare i risultati in modo da selezionare solamente i messaggi relativi ai processi e al livello di priorità d'interesse.

A partire da Android 4.3 Google ha però bloccato l'accesso ai log delle altre applicazioni per motivi di sicurezza, rendendo di fatto l'applicazione inutilizzabile [32], anche se gli utenti possono comunque accedere ai log collegando il dispositivo a un computer e utilizzando il comando shell Logcat.

3.2.5 Google Play Developer Console

La Google Play Developer Console [33] è lo strumento che Google mette a disposizione agli sviluppatori per gestire la pubblicazione delle applicazioni sul Google Play Store.

La Developer Console mette a disposizione degli sviluppatori statistiche sulle installazioni e i commenti degli utenti; inoltre, sfruttando i Google Play Services installati sui dispositivi certificati, mette a disposizione degli sviluppatori dati importanti come la distribuzione dell'installazione delle applicazioni sui diversi dispositivi, raggruppati per modello o per versione di Android. Sempre sfruttando i servizi Google installati sui dispositivi fornisce anche una panoramica generale del funzionamento delle applicazioni, riportando gli errori e gli arresti anomali rilevati durante il funzionamento dei dispositivi degli utenti.

Purtroppo, la console per gli sviluppatori non fornisce informazioni più dettagliate come l'utilizzo della CPU, della RAM o sul funzionamento dell'interfaccia grafica.

3.3 Valutazione delle possibili soluzioni

Per poter risolvere i problemi descritti nelle sezioni precedenti ho effettuato una ricerca in modo da individuare alcune possibili soluzioni che permettessero di implementare un sistema di monitoraggio in grado di raccogliere i dati delle applicazioni e di trasmetterli a un server remoto. Lo studio esposto in questa sezione, come la successiva implementazione, è stato svolto utilizzando Android 9 che, nonostante Android 10 fosse già disponibile, era la versione di Android maggiormente diffusa sul mercato all'inizio di questa tesi.

Inoltre, non si esclude la presenza di altri possibili metodi che non sono stati individuati o presi in considerazione dallo studio svolto per questa ricerca.

3.3.1 Soluzioni individuate

Nella fase di analisi ho individuato quattro possibili soluzioni al problema, ognuna delle quali ha sia dei vantaggi che degli svantaggi.

- Libreria per applicazioni;
- Applicazione di monitoraggio;
- Modifica del codice compilato;
- Modifica del sistema operativo.

3.3.2 Libreria per applicazioni

L'idea di questa soluzione è quella di costruire una libreria di monitoraggio che gli sviluppatori possano integrare nelle proprie applicazioni.

Grazie al fatto che la libreria sarebbe parte stessa dell'applicazione, questa soluzione potrebbe fornire molte possibilità di monitoraggio ed è probabilmente la soluzione con il maggior potenziale e con la maggior possibilità di personalizzazione da parte degli sviluppatori.

La libreria potrebbe fornire sia una parte di monitoraggio automatica, sia una manuale.

La parte di monitoraggio automatica potrebbe occuparsi di raccogliere in maniera autonoma informazioni come caratteristiche del dispositivo, log, utilizzo delle risorse di sistema e dettagli riguardo alla renderizzazione dell'interfaccia utente; mentre la parte di monitoraggio manuale potrebbe consistere in una serie di metodi messi a disposizione dalla libreria agli sviluppatori che questi potrebbero decidere di utilizzare nei punti di maggior interesse dell'applicazione per raccogliere i dati che necessitano.

Per la creazione di una libreria del genere sarebbe necessaria la creazione di un sistema che si occupi non solo di raccogliere i dati, ma anche di immagazzinarli e successivamente sincronizzarli con un server remoto. Questo probabilmente comporterebbe la necessità di estendere le funzionalità di alcuni componenti delle applicazioni come le Activity, i Fragment o anche le stesse View per facilitare l'accesso ai dati e automatizzarlo il più possibile. Gli sviluppatori dovrebbero poi utilizzare i componenti dell'applicazione per la creazione delle loro applicazioni e non quelli forniti dall'SDK di Android o da librerie di terzi. L'applicazione potrebbe per esempio avviare, in automatico, un thread dedicato che si occupi del monitoraggio delle risorse di sistema e il cui ciclo di vita sia associato a quello dell'applicazione

stessa; mentre le Activity e i Fragment potrebbero analizzare e salvare automaticamente la struttura dell'albero delle View in modo da automatizzare la scansione dell'interfaccia utente e renderla la più precisa possibile.
I vantaggi di questa soluzione sono:

- Nessuna modifica del sistema operativo richiesta;
- Può funzionare su qualsiasi tipo di dispositivo;
- Ottima integrazione con l'applicazione;
- Possibilità di personalizzazione da parte degli sviluppatori.

Gli svantaggi invece sono i seguenti:

- Richiede lavoro extra da parte degli sviluppatori che dovrebbero integrare manualmente la libreria;
- Scarsa compatibilità con librerie di terzi che potrebbe limitarne il funzionamento e le velocità di aggiornamento;
- Può monitorare solo l'applicazione in cui è stata implementata.

3.3.3 Applicazione di monitoraggio

L'idea alla base di questa soluzione è quella di creare un'applicazione in grado di monitorare il funzionamento delle altre applicazioni installate sul dispositivo. Sfortunatamente questa soluzione non è fattibile e dopo una breve analisi è stata scartata, dato che le limitazioni che per motivi di sicurezza e di privacy Android impone ne renderebbero impossibile il funzionamento.

Durante lo studio per verificare la fattibilità di questa soluzione ho scoperto che esiste un particolare tipo di applicazioni Android che sono considerate privilegiate e che godono di permessi aggiuntivi rispetto alle altre applicazioni installate dagli utenti. Queste applicazioni possono ottenere i privilegi in due modi:

- Avendo la stessa firma del sistema operativo stesso;
- Venendo installate in una cartella dedicata alle applicazioni privilegiate.

Nel secondo caso ho però scoperto che a partire da Android 8 perché all'applicazione vengano assegnati i privilegi richiesti è comunque necessario effettuare una modifica al sistema per aggiungerla a una whitelist [34], altrimenti i permessi non verrebbero assegnati e il sistema non si avvierebbe.

Prendendo in considerazione la necessità di dover comunque modificare il sistema, anche se in modo molto limitato, e tenendo conto del fatto che anche le applicazioni privilegiate non disporrebbero di tutte le API necessarie per monitorare le altre applicazioni, per esempio non potrebbero accedere alle informazioni sull'utilizzo dell'HEAP, ho deciso di scartare questa soluzione. Se questa soluzione fosse stata fattibile avrebbe comunque offerto vari vantaggi:

- Applicazione indipendente dalle altre;
- Nessun carico di lavoro extra per gli sviluppatori;
- Possibilità di monitorare più applicazioni installate nel sistema.
- Possibilità di personalizzazione.

3.3.4 Modifica del codice compilato

L'idea alla base di questa soluzione è quella di modificare l'applicazione una volta che gli sviluppatori hanno terminato il proprio lavoro e hanno generato il file installabile sui dispositivi in modo da integrare un sistema di monitoraggio. La principale differenza dalla prima soluzione proposta a questa è che mentre nel primo caso sarebbe compito dello sviluppatore quello di integrare la libreria di monitoraggio, in questo caso lo sviluppatore non dovrebbe compiere alcuna azione extra in fase di sviluppo, ma il sistema di monitoraggio sarebbe aggiunto andando a modificare il bytecode in formato Class o Dex.

Una soluzione del genere sarebbe probabilmente la più complessa da implementare in quanto richiederebbe sia lo studio del codice compilato Dex [35] o Class [36], sia la scrittura di un interprete in grado di interpretare il codice, capire il suo funzionamento e innestare dell'altro codice in grado di effettuare il monitoraggio.

La fattibilità di questa soluzione si scontra anche con la firma che garantisce l'autenticità delle applicazioni installate sui dispositivi, un servizio fornito dai servizi Google [37]. Tutte le app installate sul dispositivo tramite il Google Play Store sono infatti firmate in modo da garantire l'autenticità e la non alterazione del codice da parte di terzi, i servizi Google hanno il compito di verificare le applicazioni prima di procedere all'installazione e in caso di discrepanze l'installazione viene bloccata.

Per poter ovviare a questo problema una possibile soluzione sarebbe quella di andare a modificare il bytecode direttamente sul dispositivo, cosa che comporterebbe la necessità di una modifica al sistema operativo stesso. Questa modifica potrebbe essere implementata in modi differenti:

- Sistema di monitoraggio centralizzato;
- Sistema di monitoraggio indipendente per ogni applicazione.

La seconda opzione, a causa della mancanza di un sistema centrale in grado di gestire il monitoraggio, verrebbe svolta per qualsiasi applicazione, indipendentemente dalla volontà dello sviluppatore di monitorare l'applicazione e oltre a non offrire possibilità di personalizzazione richiederebbe anche la modifica del funzionamento dell'applicazione dovendo aggiungere i permessi per la connessione alla rete Internet ed eventualmente un sistema di salvataggio delle informazioni raccolte.

La prima opzione permetterebbe invece l'implementazione della soluzione in modo più efficiente, fornendo un sistema centralizzato di gestione del monitoraggio che potrebbe venire attivato o disattivato a seconda del volere degli sviluppatori e degli utenti. Permetterebbe anche una migliore possibilità di personalizzazione del servizio di monitoraggio da parte degli sviluppatori e non richiederebbe eccessive modifiche al codice dell'applicazione se non quelle per rilevare i dati e comunicarli al sistema.

3.3.5 Modifica del sistema operativo

L'idea alla base di questa soluzione è quella di modificare il sistema operativo stesso in modo da integrare un sistema di monitoraggio che possa funzionare con tutte le applicazioni installate.

Il monitoraggio potrebbe essere fatto tramite un servizio di sistema in grado di dialogare con gli altri e in grado di monitorare le applicazioni in esecuzione. Il fatto di implementare il sistema di monitoraggio tramite un servizio di sistema permetterebbe di sfruttare la API del framework Android e allo stesso tempo di godere di tutti i privilegi di cui i processi di sistema godono.

L'idea è simile alla seconda opzione esposta nella proposta precedente, in quanto anche in quel caso un servizio di monitoraggio centralizzato sarebbe stato creato tramite un servizio di sistema. La principale differenza però deriva dal fatto che in questo caso le applicazioni non verrebbero modificate e tutte le alterazioni al codice di Android verrebbero fatte a livello del framework, sostituendo le eventuali modifiche al bytecode con modifiche ai componenti stessi di Android come Widget, Activity e Fragment, oltre che agli altri servizi di sistema già implementati.
I possibili vantaggi della soluzione:

- Soluzione integrata nel sistema operativo;

- Trasparente agli sviluppatori;
- Trasparente agli utenti;
- Possibilità di personalizzazione;
- Possibilità di monitorare più applicazioni.

Mentre alcuni degli svantaggi sarebbero:

- Necessità di ricompilare il sistema operativo;
- Aggiornamenti del servizio collegati agli aggiornamenti del sistema;
- Scarsa compatibilità con librerie esterne.

3.3.6 Soluzione scelta

Dopo aver analizzato le varie soluzioni identificate, la scelta è ricaduta sull'ultima opzione.

Mentre la seconda opzione, quella di un'applicazione di monitoraggio è stata scartata per l'infattibilità, la terza opzione è stata scartata per l'eccessiva complessità. Fra le due opzioni rimanenti il fattore decisivo di scelta è stata la possibilità di costruire un sistema di monitoraggio completo che l'ultima opzione offre.

Infatti, la quarta proposta permette di sviluppare un sistema completo in grado gestire il monitoraggio di tutte le applicazioni in maniera trasparente sia per gli utenti che per gli sviluppatori; un vantaggio che, secondo me, vale la pena di sfruttare anche a costo di sacrificare un po' delle possibilità di personalizzazione che la prima soluzione permetterebbe agli sviluppatori. Inoltre, costruendo un servizio di monitoraggio centralizzato e gestito dagli OEM invece che dagli sviluppatori stessi è anche possibile tutelare maggiormente gli utenti fornendo un servizio più sicuro e maggiormente testato.

Capitolo 4

I dati collezionabili

Questo capitolo descrive le tipologie di dati che il servizio potrebbe raccogliere e che potrebbero rivelarsi interessanti per gli sviluppatori di applicazioni. Sono descritti i dati, quale è la loro utilità e dove sono localizzati nel sistema. Non tutti i dati presentati in questo capitolo sono stati inclusi nell'implementazione, ma lo studio effettuato resta valido.

4.1 Informazioni sul dispositivo

Questo gruppo di dati rappresenta le informazioni base che servono a caratterizzare il dispositivo. È costituito da un insieme di informazioni che dovrebbero essere inviate al server remoto ogni qual volta si voglia effettuare il caricamento dei dati raccolti:

- Marca del dispositivo;
- Nome del dispositivo;
- Modello del dispositivo;
- Versione delle API di Android;
- Versione della build installata;
- Modello della CPU;
- Numero di core della CPU;
- Quantità totale di RAM.

Queste informazioni sono statiche e non possono cambiare se non effettuando un aggiornamento software o modificando l'hardware di un dispositivo.

Le prime cinque informazioni (marca, modello, dispositivo, API e numero di build) sono tutte informazioni facilmente ottenibili e a cui qualsiasi applicazione installata sul dispositivo può accedere. Le API di Android mettono infatti a disposizione una classe chiamata *Build* [38] che contiene informazioni riguardo al sistema stesso e utilizzandola è possibile ottenere le informazioni che servono.

Le informazioni sul modello della CPU e il numero di core di questa sono invece più difficili da ottenere attraverso le API di Android in quanto differenti dispositivi potrebbero implementare queste informazioni in modo diverso o non implementarle affatto. Ho quindi deciso di raccogliere le informazioni riguardo alla CPU tramite i file di sistema di Linux.

Nonostante Android abbia introdotto nuove restrizioni che impediscono l'accesso alle applicazioni ad alcune aree di memoria, essendo la soluzione proposta un servizio di sistema che gode dei diritti di amministrazione, non sono presenti vincoli ed è possibile accedere a qualsiasi file. Nei sistemi Linux è presente una cartella chiamata */proc* [39] che contiene informazioni riguardo ai processi e statistiche sulla loro esecuzione. Fra questi file ne è presente uno chiamato */proc/cpuinfo* [39] che raccoglie le informazioni riguardo alla CPU e che ho deciso di utilizzare per ottenere il nome e il numero di core da cui questa è composta.

Informazione	**Dove trovarla**
Marca	Classe Build.java
Nome	
Modello	
Versione API	
Versione Build	
Modello CPU	/proc/cpuinfo
Numero di Core	
Dimensione RAM	Activity Manager Service

Tabella 4.1: Dove è possibile trovare le informazioni riguardo al dispositivo

4.2 Informazioni riguardo le prestazioni

Per valutare le prestazioni delle applicazioni ho individuato due categorie di parametri: quelli collegati all'utilizzo della CPU e quelli collegati all'utilizzo della memoria.

4.2.1 Prestazioni della CPU

Le informazioni riguardo alle prestazioni della CPU sono necessarie per poter capire il comportamento dell'applicazione sui vari dispositivi ed effettuare una valutazione delle prestazioni. Per capire come trovare le informazioni riguardo all'utilizzo della CPU di ogni processo, ho deciso di cercare le informazioni relative al processo Linux associato all'applicazione, concentrandomi su come Linux registra le informazioni riguardo l'utilizzo della CPU per ogni processo. Ho scoperto che nella cartella di sistema */proc* nominata precedentemente è presente una cartella per ogni processo in esecuzione nel sistema che raccoglie le informazioni riguardo ad esso. In particolare, nella cartella è presente un file */proc/[PID]/stat* che è un file di testo contenente le varie informazioni sull'utilizzo della CPU aggiornate dal kernel Linux.

Fra i dati presenti in questo file ne ho individuati alcuni:

- Utilizzo corrente della CPU;
- Utilizzo medio della CPU;
- VSZ (Virtual Memory Size);
- RSS (Resident Set Size).

Per poter raccogliere questi dati ho deciso di utilizzare due utility Linux: *Top* e *PS*. Entrambi i tool sono in grado di fornire informazioni riguardo ai processi in esecuzione nel sistema ed entrambi sfruttano i file presenti nella cartella di sistema /proc per elaborarle.

Top è in grado di fornire informazioni dinamiche in tempo reale ed è possibile utilizzarlo per ottenere l'utilizzo corrente della CPU [40] , mentre PS fornisce una fotografia del sistema [41] ed è possibile usarlo per estrarre le informazioni in un determinato istante di tempo. Si può quindi utilizzare PS per ottenere le informazioni riguardo all'utilizzo medio della CPU, che viene calcolato come rapporto fra tempo per cui il processo ha utilizzato la CPU e tempo totale di esecuzione di quest'ultimo, e riguardo all'utilizzo della memoria.

4.2.2 Utilizzo della memoria

Anche l'utilizzo della memoria è uno degli aspetti fondamentali per valutare le prestazioni delle applicazioni sui vari dispositivi ed è importante capire come l'applicazione utilizzi la memoria allocata in modo statico o in modo dinamico.

Ho cercato di capire come ottenere le informazioni relative all'utilizzo della memoria RAM del dispositivo e ho scoperto che l'Activity Manager Service mette a disposizione un metodo che restituisce le informazioni relative alla memoria. In alternativa, anche in questo caso, si sarebbero potute ottenere le informazioni mediante un file posizionato nella directory di sistema */proc*: */proc/meminfo* [42] che contiene varie informazioni riguardo alla memoria RAM e che è utilizzato anche dall'Activity Manager Service. Android calcola però alcuni parametri relativi alla memoria in modo differente, per esempio la soglia di memoria disponibile sotto il quale il sistema si considera nello stato Low Memory viene calcolata in modo dinamico e dipende dal dispositivo, motivo per cui ho deciso di non leggere i file di sistema ma di utilizzare le informazioni fornite dal servizio.

Le informazioni riguardo alla RAM che sono state individuate sono le seguenti:

- **totalMem**: La quantità totale di memoria RAM accessibile dal kernel Linux;
- **availMem**: La quantità di memoria RAM disponibile per il sistema;
- **lowMemory**: Una variabile booleana che indica se il sistema si considera in una condizione di disponibilità limitata di memoria RAM;
- **threshold**: Il valore di soglia sotto il quale il sistema si considera con disponibilità limitata di memoria RAM.

Oltre a queste informazioni mi sono concentrato anche sulle informazioni riguardo all'utilizzo dell'HEAP da parte delle applicazioni. Le informazioni dell'HEAP sono facilmente accessibili tramite la classe *Runtime* [43], che viene implementata dalla virtual machine (ART o Dalvik) in cui l'applicazione è in esecuzione, ma che non fornisce API pubbliche in modo che gli altri processi possano ottenerle. Per questo motivo, per accedere a questi dati è stato necessario implementare una modifica al sistema per effettuare chiamate inter-processuali.

Le informazioni riguardo all'allocazione della memoria dinamica che ho individuato sono le seguenti:

- Dimensione massima dell'HEAP;
- Quantità di memoria attualmente utilizzata;
- Quantità di memoria attualmente disponibile.

Informazione	**Dove trovarla**
Utilizzo della RAM	Activity Manager Service
Utilizzo della CPU	Process Record - /proc/[PID]/stat
Informazioni sull'HEAP	Activity Thread

Tabella 4.2: Dove è possibile trovare le informazioni riguardo le performance

4.3 Informazioni riguardo l'interfaccia utente

Le informazioni riguardo l'interfaccia utente sono fra le più importanti per gli sviluppatori in quanto permettono di capire come l'applicazione viene visualizzata sui dispositivi degli utenti e in caso li aiuta a correggere i problemi.

4.3.1 Informazioni riguardo al Display

Raccogliere le informazioni riguardo al display del dispositivo è fondamentale per permettere agli sviluppatori di correggere i possibili problemi sostituendo lo smartphone o il tablet degli utenti con un emulatore che abbia le stesse caratteristiche.

Le caratteristiche del display possono essere sia hardware che software, infatti se da un lato la dimensione fisica del display e il numero di pixel sono parametri determinati dalla progettazione, ci sono altri parametri come la densità che sono variabili e possono essere configurati via software dal produttore o dagli utenti.

Le informazioni che ho ritenuto essenziali sono le seguenti:

- Larghezza [pixel];
- Altezza [pixel];
- Densità Dpi;
- Larghezza [dp];

- Altezza [dp];
- Ingrandimento;
- Fattore di scala.

4.3.2 Informazioni riguardo alle View

Le informazioni riguardo alle View si riferiscono a come ogni singolo componente dell'interfaccia utente viene renderizzato sullo schermo. Raccogliere questo tipo di informazioni permette agli sviluppatori di riprodurre quello che gli utenti vedono sui propri dispositivi e di confrontare il risultato con quello che era il risultato atteso e individuare possibili problemi da correggere.

Le informazioni riguardo alle View sono disponibili all'interno dell'Activity e possono essere recuperate in modo ricorsivo partendo dal cosiddetto *Root View*, la View che funge da radice dell'albero. Partendo da questa View è possibile ricostruire l'albero verificando verificando se gli elementi hanno o meno dei figli.

Le informazioni che ho identificato sono le seguenti:

- Nome della classe;
- Tag;
- Id;
- Larghezza;
- Altezza;
- Distanza dal margine sinistro del contenitore;
- Distanza dal margine superiore del contenitore;
- Eventuali figli;
- Eventuale contenuto testuale.

Informazione	**Dove trovarla**
Informazioni sul Display	Display Manager Service
Struttura delle Activity	Activity Thread

Tabella 4.3: Dove è possibile trovare le informazioni riguardo l'UI

4.4 File di Log

I log costituiscono uno strumento molto utile per permettere agli sviluppatori di individuare e correggere gli errori nelle applicazioni o di monitorare il flusso delle operazioni eseguite. Vengono utilizzati sia dal sistema in modo automatico, sia dagli sviluppatori in modo manuale.

Android gestisce i Log dividendoli in più canali, ognuno con uno scopo diverso:

- Main;
- Radio;
- Events;
- System;
- Crash.

A ogni messaggio inviato in ciascun canale viene inoltre assegnato un livello di importanza, in ordine crescente:

- Verbose;
- Debug;
- Information;
- Warning;
- Error.

I log vengono scritti tramite i metodi forniti dalla classe *Log* [44] su un buffer circolare che può essere letto tramite il comando Linux *logcat*. Durante la lettura è possibile filtrare i log tramite vari parametri, fra i quali il numero del processo e la priorità.

Ho ritenuto che per gli sviluppatori fossero importanti i log con livello di priorità maggiore o uguale a Warning e appartenenti ai canali Main e Crash.

Informazione	**Dove trovarla**
Log	logcat (comando shell)
Log	Log.java

Tabella 4.4: Dove è possibile trovare i messaggi di Log

Capitolo 5

Implementazione

Lo scopo di questo capitolo è quello di spiegare come è stata svolta l'implementazione del servizio di monitoraggio. Durante l'implementazione ho riscontrato varie difficoltà, che derivano per la maggior parte dalla scarsa documentazione presente.

Dopo aver scartato la possibilità di implementare il servizio tramite un'applicazione privilegiata mi sono concentrato sulla creazione di un servizio di sistema, che offre più libertà e può integrarsi il più possibile con gli altri componenti del framework.

Il primo approccio che ho scelto è stato quello di effettuare meno modifiche possibili agli altri componenti, ma di utilizzare tutte le API già disponibili per raccogliere le informazioni; mi sono però scontrato con l'assenza di alcune API fondamentali per la raccolta di dati come le informazioni riguardo alle View o all'HEAP dei processi.

Ho quindi deciso di modificare anche altri componenti del sistema, cercando di effettuare modifiche il meno invasive possibile, ma tenendo comunque in considerazione l'efficienza in modo da non alterare troppo le prestazioni del sistema operativo.

5.1 Creazione di un servizio di sistema

Per la creazione di un servizio di sistema non esiste una documentazione ufficiale, quindi si è resa necessaria la ricerca di guide non ufficiali. I risultati della ricerca hanno fornito vari tutorial, molti dei quali non aggiornati o incompleti. Integrando questi articoli con l'analisi dei sorgenti del framework, ho scoperto dove i servizi di sistema sono localizzati nei sorgenti e come crearne uno.

I passaggi fondamentali che ho identificato per la creazione di un servizio di sistema sono quattro:

1. Creazione dei file del servizio;
2. Estensione della classe SystemService.java;
3. Esecuzione del servizio in fase di boot;
4. Esposizione del servizio agli altri processi.

5.1.1 Creazione dei file del servizio

Ho scoperto che i servizi di sistema sono localizzati nella cartella */frameworks/base/services/*. La cartella contiene varie sottocartelle in cui sono organizzati tutti i servizi, il System Server e le varie implementazioni in codice nativo.

In caso si voglia creare un servizio, a seconda delle esigenze, si può creare o un singolo file o una cartella dedicata nella directory */frameworks/base/services/core/java/com/android/server*. In caso di servizi semplici come l'Alarm Manager [45] una singola classe potrebbe rivelarsi sufficiente, mentre in caso di servizi più complessi può rendersi necessaria la creazione di una sottocartella in grado di contenere tutti i file necessari.

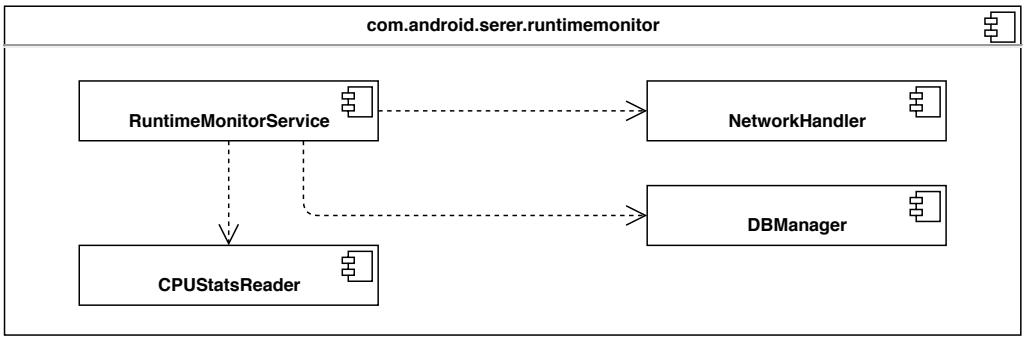

Figura 5.1: Principali componenti del servizio

Per questa implementazione ho optato per la seconda opzione e ho creato una cartella chiamata *runtimemonitor* in cui ho creato il file RuntimeMonitorService.java e tutti gli altri file contenenti le classi necessarie per il funzionamento. In figura 5.1 sono mostrati i principali componenti del servizio.

5.1.2 Estensione della classe SystemService.java

Ogni servizio in esecuzione nel sistema è costituito da un singleton avviato in fase di boot e istanziato dal SystemServer. Per far si che i servizi ricevano le notifiche durante tutte le fasi in cui il sistema operativo è in esecuzione, dall'avvio allo spegnimento, è necessario che implementino una classe astratta chiamata *SystemService* e localizzata in */frameworks/base/services/core/java/com/android/server*.

Una volta implementata questa classe sarà possibile avviare il servizio tramite il SystemServer e ricevere le notifiche; fra le quali la *PHASE_BOOT_COMPLETED*, che segnala ai vari servizi di sistema il completamento della fase di avvio e li invita a iniziare il normale funzionamento.

Per poter implementare la classe SystemService non è necessario che l'implementazione venga effettuata direttamente dalla classe principale che costituisce il servizio, ma può essere anche svolta da una classe secondaria, un cui singleton rappresenterà poi il servizio di sistema.

In questo caso ho deciso di creare una classe Lifecycle, interna alla classe principale RuntimeMonitorService, il cui compito è quello di implementare SystemService. Lifecycle gestisce le varie notifiche ricevute durante le fasi di esecuzione e avvio del sistema e una volta ricevuta la notifica di completamento della fase di boot si occupa di avviare il servizio vero e proprio chiamando il metodo start della classe RuntimeMonitorService.

La figura 5.2 mostra la classe RuntimeMonitorService e le sue classi interne.

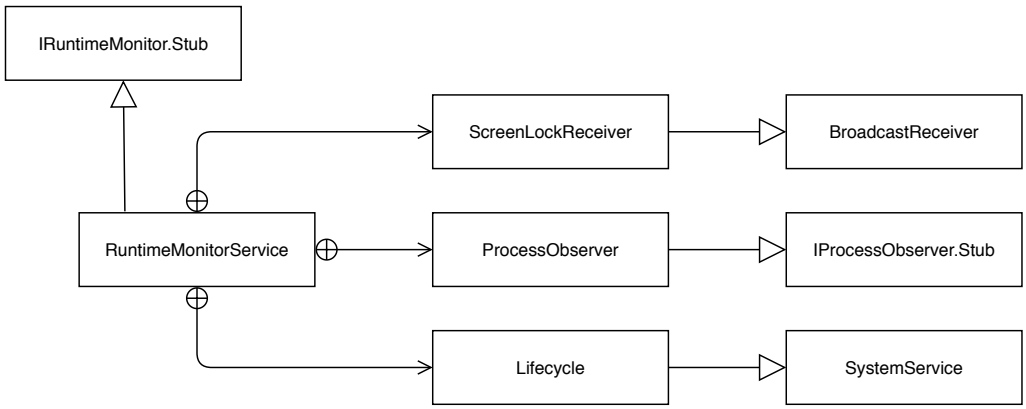

Figura 5.2: Classe RuntimeMonitorService e le sue classi interne

5.1.3 Esecuzione del servizio in fase di boot

Una volta creato il servizio, per fare in modo che venga avviato durante la fase di avvio e riavviato in caso di crash è necessario registrarlo nel SystemServer.

Il SystemServer è infatti il primo servizio avviato da Zygote ed è quello che si occupa dell'avvio di tutti i servizi e i processi fondamentali per il framework Android. In caso anche uno solo di questi si interrompa, il SystemServer procede a terminarli tutti e a riavviarli.

L'avvio dei servizi avviene in più fasi, in ordine di importanza. I servizi sono raggruppati nelle varie fasi in base alla loro essenzialità per il funzionamento del sistema:

1. Avvio dei servizi critici (Bootstrap Services);
2. Avvio dei servizi di base (Core services);
3. Avvio degli altri servizi.

Per registrare il servizio è quindi necessario modificare la classe */frameworks/base/services/java/com/android/server/SystemServer.java* e istanziare il servizio all'interno del metodo preferito.

Nel caso in questione ho deciso di istanziare il servizio nel gruppo "altri servizi" in quanto non lo ritengo essenziale per il funzionamento del sistema, ma solamente un servizio accessorio, inoltre dato che per il funzionamento vengono sfruttati altri componenti del sistema, è necessario che venga avviato solamente quando tutti i servizi da cui dipende siano già in funzione.

5.1.4 Esposizione del servizio

Una volta che il servizio è stato creato ed avviato, perché gli altri processi possano comunicare con esso tramite IPC è necessario che siano in possesso dell'autorizzazione e che il servizio sia stato esposto. Questa comunicazione IPC viene gestita direttamente a livello kernel tramite SELinux, un sistema integrato nel kernel Linux che permette di implementare un Mandatory Access Control (MAC) a cui devono sottostare anche i processi in esecuzione con i privilegi di root/superuser.

Una volta creato un servizio di sistema, perché gli altri processi possano identificarlo e comunicare con esso è quindi necessario definire una policy che specifichi quali tipi di processo possono trovarlo quando effettuano una ricerca e quali azioni questi possono compiere. Per definire una policy è necessario modificare i file presenti nella cartella */system/sepolicy*, dove è possibile definire sia API pubbliche che private [46].

5.2 Il funzionamento del servizio

Una volta che il servizio viene avviato, esegue una fase di inizializzazione interna che serve a caricare tutti i componenti necessari e una volta terminata, quando riceve la notifica dal sistema del completamento della fase di boot, inizia a il monitoraggio delle altre applicazioni.

Il monitoraggio è composto da una fase attiva e da una fase più o meno passiva.

Nella fase attiva è il servizio che periodicamente si occupa di raccogliere le informazioni e se necessario contatta gli altri processi o gli altri servizi di sistema tramite comunicazione IPC per richiedere le informazioni necessarie.

L'idea che ho seguito è quella di creare un servizio che monitori il cambiamento dei processi in foreground e in base al package name dell'applicazione associata ai processi, decide se questo merita di essere monitorato o meno e, se la risposta è affermativa, lo aggiunge a una lista contenente il PID di tutti i processi osservabili.

Ogni volta che la lista contenente i processi osservabili ha almeno un elemento, viene creato un thread dedicato al monitoraggio che entra in esecuzione ogni 30 secondi e si occupa di raccogliere le informazioni necessarie e di contattare gli altri processi.

5.3 Fase di inizializzazione

Nella fase di inizializzazione il servizio si occupa di istanziare gli oggetti che rappresentano tutti i componenti fondamentali per il monitoraggio delle applicazioni e per il dialogo con il server remoto.
I passi svolti sono i seguenti:

1. Viene istanziato il gestore della rete;
2. Viene creato il Process Observer;
3. Viene inizializzato il database;
4. Viene registrato un observer per lo schermo;
5. Vengono raccolte le informazioni di base.

5.3.1 Il gestore della rete

Il gestore della rete è la classe, istanziata come singleton, che si occupa di effettuare tutte le comunicazioni di rete necessarie per la sincronizzazione

dei dati necessari al funzionamento del servizio e per il caricamento delle rilevazioni effettuate.
In particolare si occupa di:

- Aggiornare la lista dei processi osservabili;
- Effettuare il caricamento dei log rilevati dal sistema;
- Effettuare il caricamento delle informazioni riguardo alla visualizzazione delle view delle applicazioni;
- Effettuare il caricamento dei dati di monitoraggio delle prestazioni.

Per poter effettuare il caricamento e il download viene utilizzato l'Alarm Manager che periodicamente notifica il servizio di dover effettuare una sincronizzazione. In caso di errore o mancata connessione al momento della notifica da parte dell'Alarm Manager, la sincronizzazione viene sospesa e viene effettuata una richiesta al Connectivity Manager del sistema che, quando la connessione tornerà disponibile, invierà una nuova notifica per permettere la ripresa della sincronizzazione.

5.3.2 Creazione del Process Observer

Per il monitoraggio dei cambiamenti nei processi in foreground ho deciso di utilizzare ProcessObserver, un'interfaccia presente nel package dell'ActivityManagerService che mette a disposizione due metodi:

- onForegroundActivitiesChanged;
- onProcessDied.

Il primo viene utilizzato per notificare del cambiamento delle Activity in foreground e viene chiamato per ogni Activity coinvolta nel cambiamento in foreground, con i seguenti parametri:

- PID (Process ID);
- UID (User ID);
- foregroundActivities (Indica se l'Activity associata al PID è in foreground dopo i cambiamenti).

Il secondo metodo invece viene chiamato quando il sistema decide di terminare un processo in quanto non più associato a un'applicazione e quindi inutile.

Nelle versioni successive di Android rispetto a quella utilizzata per questa tesi è presente un terzo metodo che permette di essere notificati anche quando un servizio e non solo un'Activity passa in foreground.

Per poter utilizzare l'interfaccia ProcessObserver ho deciso di creare una classe interna alla classe RuntimeMonitorService che implementi l'interfaccia e che gestisca tutti i cambiamenti.

Ogni volta che il processo in foreground cambia, il servizio utilizza il PID per recuperare il package name dell'applicazione e in base a quest'ultimo decide se deve monitorarla. In caso positivo, il processo viene aggiunto alla lista di quelli attualmente sotto osservazione e se il thread di monitoraggio non è già attivo, viene creato.

Quando un processo viene terminato viene rimosso dalla lista dei processi sotto osservazione.

5.3.3 Inizializzazione del database

Per salvare temporaneamente le informazioni prima di caricarle sul server remoto ho deciso di utilizzare un database locale. Android mette a disposizione un database SQLite 3 integrato nel sistema e accessibile sia dalle applicazioni che dai servizi.

Il database è un servizio di sistema che viene avviato dal SystemServer. Per essere sicuri di avere a disposizione il DBMS quando si istanzia il servizio è quindi necessario assicurarsi tramite SystemServer di avviare prima il servizio SQLite 3 e successivamente il servizio che deve farne uso.

Una volta avuta la garanzia del funzionamento del database per utilizzarlo è sufficiente seguire le istruzioni dedicate alle applicazioni [47], le informazioni verranno salvate su file, nella stessa cartella in cui è localizzato il servizio di sistema.

Per il funzionamento e la gestione del database ho quindi creato tre classi suddivise in altrettanti file:

- DBHelper;
- DBManager;
- DBStrings.

DBHelper costituisce la classe che si occupa di istanziare il database e creare le tabelle, fornisce inoltre l'istanza di un database scrivibile o leggibile al DBManager.

Il DBManager è la classe che contiene tutti i metodi usati per interrogare il database ed è quella che viene utilizzata dalle altre classi del servizio per dialogare con il database.

DBStrings è una classe di supporto che contiene tutte le stringhe utilizzate per la creazione delle varie tabelle del database.

5.3.4 Registrazione dell'observer per lo schermo

Per evitare che il servizio continui il monitoraggio attivo anche quando lo schermo del dispositivo è spento e quindi quando l'utente non sta utilizzando il dispositivo ho deciso di interrompere l'esecuzione del thread di monitoraggio ogni volta che lo schermo viene spento e di riprenderla ogni volta che viene riattivato.

Per ricevere una notifica ogni volta che lo stato dello schermo cambia è sufficiente creare un BroadcastReceiver e registrarlo per i seguenti Intent:

- android.intent.action.SCREEN_OFF;
- android.intent.action.SCREEN_ON.

Anche in questo caso, per implementare il BroadcastReceiver ho creato una classe interna alla classe principale del servizio; ogni volta che lo stato cambia il receiver modifica il valore di una variabile, che viene utilizzata dal thread di monitoraggio per valutare se deve continuare l'esecuzione o se deve terminare. Nell'ultimo caso il thread completerà le operazioni e poi uscirà dal ciclo. Una volta che lo schermo verrà nuovamente attivato sarà compito del BroadcastReceiver cambiare lo stato della variabile e riattivare il thread.

5.3.5 Raccoglimento delle informazioni di base

Per il raccoglimento delle informazioni statiche ho deciso di creare una classe nel file StaticInfo.java, la classe contiene un metodo *readInfo* che istanzia un nuovo thread che ha lo scopo di raccogliere autonomamente alcune informazioni statiche che non cambiano mai durante l'esecuzione del sistema e che è necessario inviare ogni volta sul server remoto per distinguere il dispositivo dagli altri:

- Informazioni riguardo al modello del dispositivo;
- Informazioni riguardo alla versione di Android;
- Informazioni riguardo alla CPU;

- Informazioni riguardo alla RAM.

Per poter notificare la lettura dei dati statici e abilitare quindi la sincronizzazione sul server, ho creato anche un'interfaccia chiamata *InfoReadyCallback* che gestisce dati di tipo generico e specifica un metodo chiamato *signalInfoReady*. Il metodo *readInfo* della classe StaticInfo riceve come parametro un oggetto di tipo InfoReadyCallback e quando la lettura dei dati è terminata, per segnalare che questi sono pronti ed è possibile iniziare la sincronizzazione con il server remoto, chiama il metodo signalInfoReady.

5.4 Fase di monitoraggio

Una volta terminata la fase di inizializzazione il servizio entra in quella che è la fase di monitoraggio, che continua fino allo spegnimento del sistema o fino al riavvio dei servizi da parte del System Server, in caso uno di questi si blocchi.

Una parte di monitoraggio attiva viene svolta da un thread dedicato che si occupa di rilevare periodicamente i dati. Periodicamente quest'ultimo entra in esecuzione e verifica se esistono processi osservabili attivi e in caso positivo procede con il raccoglimento delle informazioni.

5.4.1 Selezione dei processi osservabili

Ogni qual volta il processo in foreground cambia è necessario stabilire se è necessario osservarlo o meno. Questa operazione viene svolta nel Process Observer che dopo aver creato un thread dedicato, come prima operazione, controlla se il PID associato al processo è già conosciuto e se è da osservare; In caso positivo procede ad avviare il thread di monitoraggio, se questo non è già attivo.

Per poter svolgere questo compito vengono sfruttate due strutture dati. La prima è una mappa che associa a ogni PID il rispettivo package name, che viene utilizzata per verificare se il processo è già conosciuto; mentre la seconda è una lista che contiene il PID di tutti i processi che al momento sono monitorabili.

In caso il PID del processo non sia presente nella prima mappa e sia quindi sconosciuto, la prima cosa da fare è risalire al package name dell'applicazione, in modo da poter verificare se compare nella lista delle applicazioni osservabili. Per recuperare questa informazione ho individuato varie possibilità, ma ho deciso che il modo più conveniente consiste nell'interrogare l'ActivityManagerService che fornisce un metodo *getRunningAppProcesses* che restituisce

una lista di RunningAppProcessInfo, oggetti che rappresentano i processi delle applicazioni correntemente in esecuzione. Confrontando il PID associato ai processi presenti in questa lista con quello fornito dal metodo del ProcessObserver è possibile risalire all'oggetto associato al processo di interesse e da questo è possibile l'attributo contenente il process name.

Una volta ottenuto il nome del processo viene interrogato il gestore delle connessioni di rete, nel quale ho creato un metodo chiamato *isObservable* che riceve come parametro un package name ed effettuando una ricerca nella lista di tutti i nomi delle applicazioni restituisce un booleano che indica se il processo è da monitorare o meno. Sia in caso negativo che in caso positivo il PID e il package name del processo vengono aggiunti alla lista dei processi noti, mentre solo in caso positivo il PID del processo viene aggiunto alla lista dei processi da osservare.

Anche in questo caso il ProcessObserver procede ad avviare il thread di monitoraggio se questo non è già attivo.

5.4.2 Il Thread di monitoraggio

Una volta avviato, il thread di monitoraggio si attiva periodicamente ogni 30 secondi fino a quando ci sono processi da osservare o fino a quando lo schermo del dispositivo non viene spento. La prima cosa che esegue in ogni ciclo è verificare che lo schermo del dispositivo sia acceso, grazie a una variabile booleana chiamata *screenOn* che viene gestita dall'observer che monitora il display. In caso lo schermo sia spento il thread esce dal ciclo, imposta la variabile *monitorThread*, che funge da riferimento al thread corrente, come nulla e termina.

Il passaggio successivo è quello di verificare se ci sono applicazioni osservabili e, con la stessa procedura appena descritta, in caso negativo termina automaticamente.

A questo punto il thread recupera la lista dei PID di tutti i processi osservabili e, per ciascuno dei processi, richiede all'ActivityManagerService il corrispondente ProcessRecord. Il ProcessRecord è una classe privata del package a cui appartiene l'ActivityManagerService, che contiene molte informazioni riguardo ai processi associati alle applicazioni.

L'utilizzo della classe ProcessRecord è utile in quanto permette di ottenere i seguenti dati:

- L'ApplicationThread associato all'applicazione;
- Se l'applicazione ha Activity in foreground;
- Se l'applicazione ha Servizi in foreground;

- Le informazioni riguardo al Display in cui l'applicazione è in esecuzione;
- Il nome dell'Activity attualmente in foreground;
- Il token associato all'Activity attualmente in foreground.

Dato che la classe ProcessRecord è privata nel package ActivityManagerService e dato che la lista dei ProcessRecord viene mantenuta dall'ActivityManagerService, per poter accedere ai dati ho valutato varie possibilità, fra le quali quella di creare un altro servizio collegato all'ActivityManager che potesse essere interrogato per ottenere l'istanza ProcessRecord relativa all'applicazione di interesse, un'altra opzione che ho preso in considerazione è la modifica dell'ActivityManager con lo scopo di aggiungere un metodo getter per ognuna delle singole informazioni del ProcessRecord utile per il funzionamento del servizio.

Ho però deciso di scartare queste opzioni a causa del modo in cui l'ActivityManagerService gestisce la lista dei ProcessRecord in quanto creando un metodo getter per ognuna delle informazioni, i tempi di ricerca si sarebbero allungati notevolmente. I ProcessRecord vengono infatti organizzati in una struttura dati chiamata SparseMap che salva le informazioni in array e usa un tipo di ricerca binaria. Per questo motivo ho deciso di ridurre al minimo le ricerche e di effettuare le seguenti modifiche:

- Modifica della visibilità della classe ProcessRecord: la visibilità della classe è stata modificata da protected a public, in modo che il servizio di monitoraggio una volta ricevuta un'istanza di un ProcessRecord possa usarlo per estrarre le informazioni necessarie;
- Creazione dell'interfaccia ActivityCollector: un'interfaccia estesa dall'ActivityManagerService che richiede l'implementazione di un metodo getter che restituisce il ProcessRecord associato al package name dell'applicazione, che viene passato come parametro del metodo;
- Modifica della classe ActivityManagerService: modifica della classe per l'implementazione del metodo getter per effettuare la ricerca del ProcessRecord necessario.

Una volta effettuate le modifiche appena elencate, il thread di monitoraggio può interrogare l'Activity Manager chiedendo, per ogni PID presente nella lista dei processi osservabili, di ottenere il ProcessRecord. Una volta ottenuto il ProcessRecord il thread controlla se questo ha un servizio o un'Activity in foreground e, in caso positivo, procede con il raccoglimento delle informazioni.

I dati che vengono raccolti dal thread di monitoraggio sono i seguenti:

- Informazioni sull'HEAP;
- Informazioni sulla RAM;
- Informazioni sulla CPU;
- Informazioni sull'interfaccia utente.

Mentre le informazioni riguardo all'utilizzo di RAM e CPU sono accessibili direttamente dal servizio, per ottenere i dati riguardo all'HEAP e all'interfaccia utente è necessario effettuare una richiesta IPC dal processo del servizio a quello in cui è in esecuzione l'applicazione. Le informazioni sull'HEAP sono infatti fornite dalla classe Java chiamata *Runtime*, la cui implementazione viene eseguita direttamente dalla Java Virtual Machine che esegue l'applicazione e che è diversa da quella che esegue il servizio di monitoraggio. Per quanto riguarda l'interfaccia utente, ho deciso di implementare questa richiesta in modo da interrogare direttamente l'oggetto che rappresenta l'Activity in primo piano nell'applicazione.

A causa dei motivi appena descritti, si è rivelato necessario effettuare una modifica alla classe ActivityThread e in particolare alla sua sottoclasse ApplicationThread, che è la classe che rappresenta il thread principale di un'applicazione. Tramite questo thread è possibile effettuare chiamate al metodo Runtime e quindi ricevere informazioni dalla VM, oppure interrogare direttamente l'Activity.

Per poter effettuare queste chiamate, ho deciso di utilizzare dei metodi non bloccanti (asincroni) con cui il servizio, tramite il thread di monitoraggio, richiede all'ActivityThread le informazioni e poi continua. Quando l'ActivityThread avrà elaborato le informazioni si occuperà di contattare il servizio e comunicare i dati.

Le seguenti modifiche, effettuate nella cartella */frameworks/base/core/-java/android/app/*, sono necessarie:

- Modifica del file IApplicationThread.aidl per specificare l'aggiunta di due metodi, uno per richiedere le informazioni dell'HEAP e uno per le informazioni sulle view;
- Modifica del file ActivityThread.java per l'implementazione dei due metodi specificati al punto precedente;
- Creazione del file IRuntimeMonitor.aidl per specificare i due metodi utilizzati per comunicare al servizio le informazioni richieste;

- Creazione del file RuntimeMonitor.java che rappresenta l'interfaccia pubblica del servizio di monitoraggio, che l'ActivityThread utilizzerà per comunicare con esso.

A queste si aggiungono poi le modifiche necessarie al servizio per poter elaborare i dati ricevuti in risposta dall'ActivityThread:

- Aggiunta dei metodi per la ricezione dei dati in RuntimeMonitorService.java;
- Modifica dei file SELinux per l'esposizione del servizio alle applicazioni;
- Modifica del file */frameworks/base/Android.bp*.

Grazie alle modifiche appena effettuate il thread di monitoraggio effettua una chiamata asincrona all'ActivityThread, per ogni processo con un servizio o un'Activity in foreground, per richiede le informazioni sull'HEAP e poi si mette automaticamente in pausa per altri 30 secondi.

I vari ActivityThread, una volta ricevuta la richiesta, recuperano le informazioni sull'HEAP tramite la classe Runtime e poi richiedono il servizio RuntimeMonitor tramite le API pubbliche, le quali si occuperanno di identificare l'istanza singleton del servizio di monitoraggio e chiameranno il metodo per riportare le informazioni richieste.

Una volta ricevute le informazioni sull'HEAP da parte di un processo, il servizio può continuare con la raccolta dei dati. Procederà quindi con l'istanziazione di un oggetto della classe DynamicInfo, che ho creato per poter contenere le informazioni dinamiche, in cui inserirà i dati sull'HEAP. Successivamente procederà a raccogliere le informazioni sull'utilizzo della memoria RAM e della CPU; le prime effettuando una chiamata all'ActivityManagerService, che fornisce un metodo pubblicato dedicato, mentre per le seconde è necessario effettuare una lettura del file di sistema localizzato in */proc*. Per facilitare la lettura di questi file, ho optato per l'utilizzo di due comandi da terminale di Linux che funzionano anche sui dispositivi Android: Top e PS. Utilizzando il comando PS si riesce a ottenere le informazioni sull'utilizzo medio di CPU, VSZ e RSS; mentre con il comando top si ottiene l'utilizzo istantaneo della CPU.

Una volta raccolte queste informazioni il thread di monitoraggio le affida al DB Manager, che procederà con il salvataggio, mentre il primo procede con il passo successivo: la raccolta delle informazioni sull'interfaccia utente. In figura 5.3 si può vedere un sequence diagram che rappresenta un esempio di esecuzione del thread di monitoraggio.

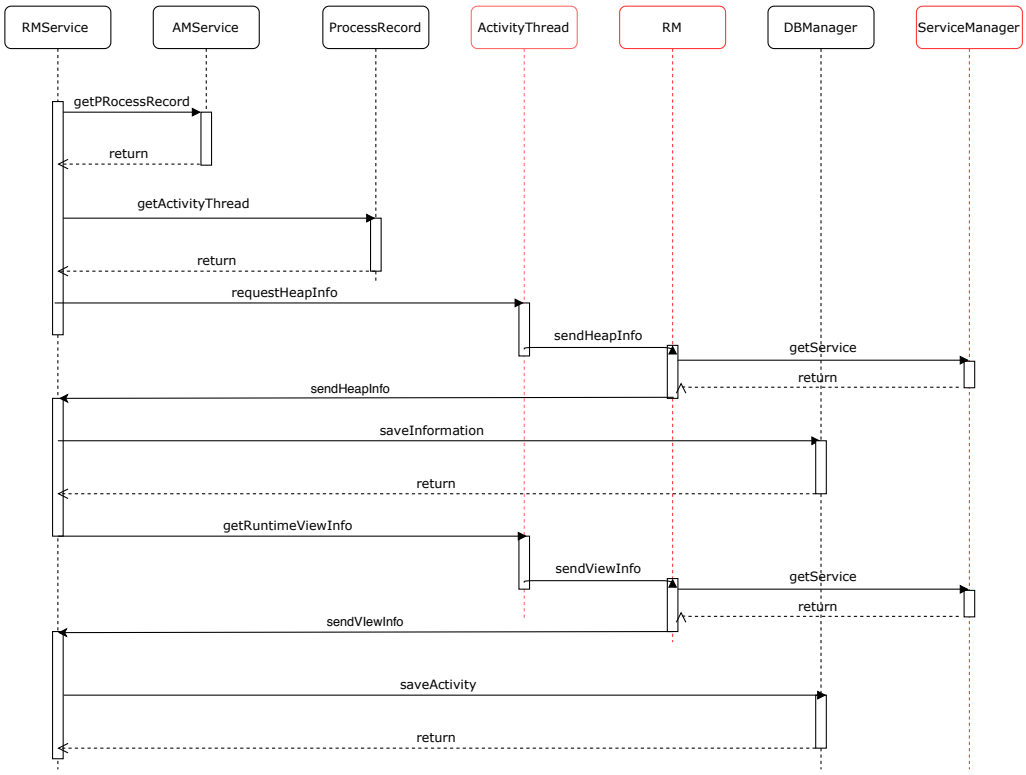

Figura 5.3: Sequence Diagram del Thread di monitoraggio

Dato che le informazioni sull'interfaccia utente potrebbero rivelarsi più pesanti da raccogliere, ho deciso di effettuare un solo monitoraggio per Activity per ogni periodo di sincronizzazione dei dati del servizio. Questo implica che in caso l'interfaccia grafica di un'Activity sia già stata scansionata, fino alla successiva sincronizzazione dei dati locali con il server remoto, questa non verrà più analizzata. Inoltre, in caso nell'interfaccia utente siano presenti componenti dinamici o fragment, il servizio non si accorgerà dei cambiamenti e quindi la rilevazione della UI non sarà sempre precisa o non mostrerà tutte le possibili schermate. Con ulteriori modifiche al sistema di monitoraggio delle View sarebbe possibile raccogliere anche quelle informazioni, ma ho deciso di lasciare questa possibilità aperta per possibili sviluppi futuri e di non includerla in questa tesi.

Dopo aver raccolto il resto delle informazioni dinamiche, sempre nel metodo che gestisce la ricezione dell'utilizzo dell'HEAP, viene effettuato un controllo per verificare se l'Activity corrente è già stata analizzata. In caso negativo viene effettuata una nuova chiamata all'ActivityThread per richiedere la scansione e il nome dell'Activity viene salvato nel database fra quelle già scansionate, in modo da evitare controlli futuri.

Il metodo nell'ActivityThread che gestisce questa richiesta riceve come parametro un ActivityToken, che utilizza per risalire all'Activity e da questa ottiene la RootView. Una volta ottenuta quest'ultima è possibile ricostruire la struttura ad albero delle View presenti nell'Activity.

Per la ricostruzione della struttura e il raccoglimento dei dai riguardo alle varie View ho deciso di creare una classe chiamata RuntimeView che viene utilizzata per creare in modo ricorsivo un albero contenente tutte le informazioni ritenute rilevanti. Ogni View in android possiede una variabile che indica se questa ha o non ha dei figli, quindi se è un container o se è una foglia, come può essere una TextView o un EditText.

La informazioni raccolte dalla classe RuntimeView sono le seguenti:

- Nome della classe;
- Id del Widget;
- Larghezza;
- Altezza;
- Distanza dal margine sinistro del contenitore;
- Distanza dal margine superiore del contenitore;
- Se il Widget è un container e in caso positivo l'elenco dei figli sempre sotto forma di RuntimeView;
- In caso il Widget sia una TextView, il contenuto.

Dopo aver effettuato la conversione e creato l'albero di RuntimeView, l'ActivityThread può inviare i dati al servizio di monitoraggio. Per poter inviare un tipo di dati non primitivo fra due processi IPC è però necessario effettuare alcune modifiche in modo che il sistema sia in grado di gestire la trasmissione. È necessario creare un file *RuntimeView.aidl* che specifica che RuntimeView può essere trasferito ed è necessario che la classe implementi l'interfaccia Parcelable, che è necessaria per definire come la struttura dati può essere prima serializzata e poi deserializzata, una volta ricevuta dal servizio.

Quando il servizio riceve le informazioni sulla struttura dei Widget di una Activity recupera le informazioni riguardo al display tramite il Process Record e poi le salva, insieme a quelle sulle View, in un file JSON localizzato nella cartella */data/system/runtime_monitor* e nominato con il formato *nome_package_nome_classe*.

5.4.3 Registrazione dei messaggi di Log

Per la raccolta dei file di Log ho valutato due possibili implementazioni: la prima è quella di una lettura di tipo polling in cui periodicamente, mediante il comando di sistema logcat si possono leggere i log ed è possibile filtrare sia per livello di priorità, sia per data che per PID dell'applicazione; mentre il secondo tipo prevede un funzionamento di tipo client server in cui il servizio attende che gli vengano inoltrati i messaggi di Log.

Come primo approccio ho deciso di seguire il primo metodo. Periodicamente, con un thread diverso da quello di monitoraggio, il servizio monitorava i log; ogni volta che il thread entrava in esecuzione eseguiva il comando Shell logcat per ogni processo presente nella lista dei processi osservabili, filtrando per PID, per livello di priorità e per ultima di lettura dei log del determinato processo. Una volta letti tutti i log, questi venivano salvati nel database e il thread veniva messo in pausa fino alla lettura successiva. Dopo un'analisi di questo metodo ho individuato vari problemi che mi hanno invogliato a cambiare approccio:

1. Ogni volta che un Log va a capo, viene riportato come due (o più) Log separati nel logger di sistema. Questo implica la necessità di dover ricostruire i log dei vari processi in modo da fornire risultati leggibili e raggruppati;

2. In caso un processo termini, vi era il rischio che il thread non leggesse i log di quest'ultimo in quanto già rimosso dalla lista dei processi osservabili. Questo problema sarebbe stato risolvibile tramite la creazione di una lista di processi osservabili dedicata ai Log, in modo da poter rimuovere il processo solo dopo la lettura dei Log;

3. Il metodo è macchinoso e potenzialmente più esoso di risorse di sistema.

Per risolvere i problemi appena descritti ho ideato a una nuova soluzione in cui sia il logger di sistema a comunicare direttamente con il servizio di monitoraggio. Ho quindi aggiunto un metodo dedicato alla ricezione dei log nel servizio e modificato i seguenti altri file:

- */frameworks/base/core/java/android/app/RuntimeMonitor.java*;
- */frameworks/base/core/java/android/app/IRuntimeMonitor.aidl*;
- */frameworks/base/core/java/android/util/Log.java.*

Le prime due modifiche hanno lo scopo di aggiungere un metodo all'interfaccia pubblica del servizio, in modo che la comunicazione IPC possa funzionare; mentre la modifica al file Log.java modifica i vari metodi utilizzati per effettuare i log in modo da aggiungere anche una chiamata alle API pubbliche del servizio ogni volta che il livello di log sia superiore o uguale a Warning.

Dato che il sistema stesso utilizza il logger, anche prima che il servizio di monitoraggio entri in funzione, per evitare che tutto il sistema si blocchi durante la fase di avvio, viene effettuato un controllo per verificare se lo UID associato al processo è relativo a un'applicazione o se è un processo di sistema. Nel primo caso si procede con la segnalazione del log al servizio, mentre nel secondo caso viene ignorato e il log viene scritto solamente nel buffer circolare. In figura 5.4 è presentato un esempio di segnalazione di un log da parte di un'applicazione.

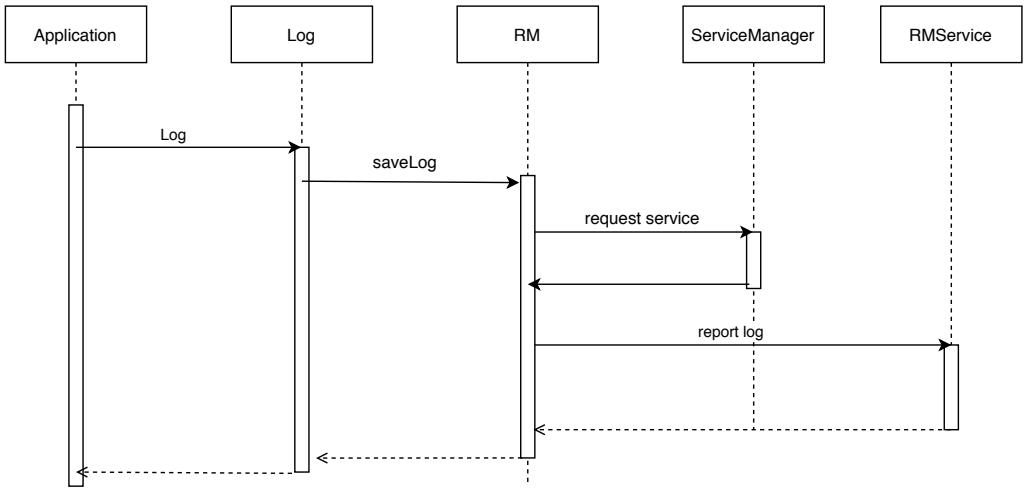

Figura 5.4: Sequence diagram che rappresenta la segnalazione di un log

5.4.4 Modifiche al ProcessRecord

Il ProcessRecord contiene molte informazioni riguardo al processo a cui è associato, purtroppo però molte di queste informazioni hanno visibilità privata o limitata al package dell'ActivityManagerService. Per poter accedere a questi dati, oltre a modificare la visibilità della classe da protected a public, ho aggiunto vari metodi getter per poter accedere alle informazioni:

- hasForegrounServices: un metodo che indica se il processo ha dei servizi in foreground;

- hasForegrounActivities: un metodo che indica se il processo ha delle Activity in foreground;
- getApplicationThread: un metodo che restituisce l'ApplicationThread associato al processo;
- getVisibleActivityName: un metodo che restituisce il package name dell'Activity visibile;
- getVisibleActivityToken: un metodo che restituisce il token univoco che identifica l'Activity attualmente visibile, necessario per l'ApplicationThread;
- getDisplayInfo: un metodo che restituisce le informazioni relative al display nel quale l'applicazione è in esecuzione.

5.5 Sincronizzazione dei dati

Come detto in precedenza, tutta la gestione dei dati che richiede la comunicazione con il server remoto viene effettuata dal gestore della rete. Per il gestore della rete ho deciso di creare una classe chiamata NetworkHandler che è completamente indipendente dal resto del servizio e che si occupa di gestire tutte le connessioni di rete. La sua composizione è mostrata nella figura 5.5.

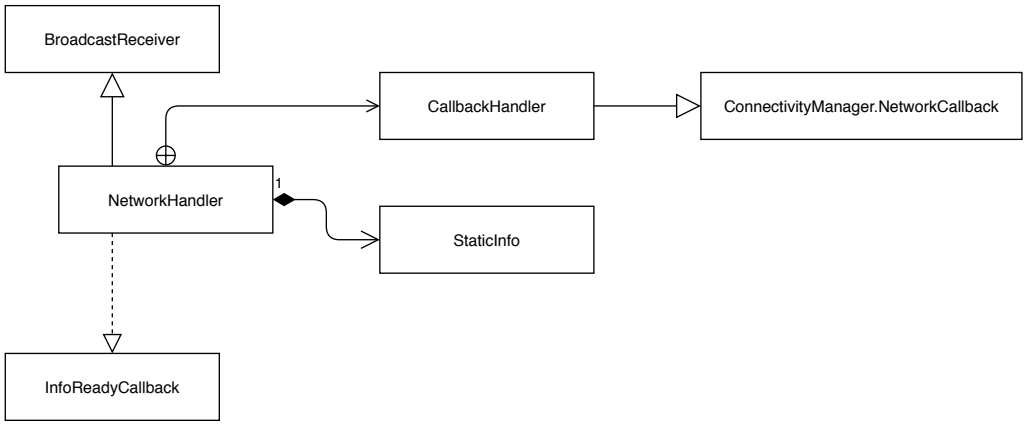

Figura 5.5: Class Diagram relativo al Network Handler

Quando il NetworkHandler viene avviato, la prima cosa che fa è creare un'istanza della classe StaticInfo che, come precedentemente descritto nel capitolo, si occupa del raccoglimento e del mantenimento in memoria delle

informazioni statiche del dispositivo, in modo da poterle recuperare in modo asincrono solamente all'avvio del servizio ed evitare futuri accessi ai file di sistema.

Il NetworkHandler recupera poi i riferimenti ai due servizi di sistema AlarmManager e ConnectivityManager che sfrutta rispettivamente per programmare la sincronizzazione dei dati in modo periodico e per verificare la disponibilità di una connessione di rete. Dopo aver ottenuto i riferimenti ai due servizi di sistema, viene programmata la prima sincronizzazione dei dati locali con il server remoto, mentre il NetworkHandler si registra al ConnectivityManager passando una callback che viene chiamata ogni volta che si verifica una variazione dello stato della connessione.

Quando viene attivata la Callback che notifica della disponibilità della connessione Internet, a seconda del valore della variabile booleana downloadOnConnect, il NetworkHandler decide se è necessario riprendere o meno la sincronizzazione dei dati.

La sincronizzazione dei dati può essere divisa in due categorie distinte:

- Download della lista delle applicazioni osservabili;
- Caricamento dei dati raccolti dal sistema di monitoraggio sul server remoto.

Le due fasi della sincronizzazione vengono gestite in modo indipendente e in parallelo ma, mentre il caricamento dei dati avviene periodicamente se e solo se è presente la connessione internet e in caso di fallimento non viene ripreso, il download della lista dei processi osservabili oltre ad avvenire periodicamente, in caso di errore dovuto a un problema di rete, riprende non appena il NetworkHandler riceve la notifica di disponibilità.

Una volta che il download della lista dei processi osservabili è terminata, l'elenco di questi viene salvato in una lista di stringhe, mentre viene fornito un metodo che permette al servizio di monitoraggio di verificare se il package name dell'applicazione compare fra quelli osservabili.

La fase di caricamento dei dati raccolti, come quella di download, viene effettuata su un thread dedicato ed è divisa in tre fasi:

1. Caricamento dei dati dinamici;
2. Caricamento delle scansioni delle Activity;
3. Caricamento dei log delle applicazioni.

In ogni fase, i dati da caricare vengono trasformati in formato JSON e associati alle informazioni statiche per essere poi caricati sul server remoto. In caso di successo dell'operazione il NetworkHandler si occupa poi di eliminare dal database i dati non più necessari, così da liberare spazio.

Capitolo 6

Risultati e valutazione

Lo scopo di questo capitolo è quello di esporre i metodi utilizzati e i risultati ottenuti dalle valutazioni effettuate per valutare la qualità dell'implementazione e il suo funzionamento.

6.1 Metodo di valutazione

Lo scopo di questa sezione è quello di spiegare quali sono state le metodologie utilizzate per valutare il funzionamento e l'efficienza della soluzione proposta e della sua implementazione.

Sono stati effettuati due tipi di test:

- Test per valutare il funzionamento della soluzione;
- Test per valutare l'efficienza della soluzione.

Per il primo tipo di test è stato utilizzato un singolo dispositivo equipaggiato con una versione di Android in grado di eseguire il sistema di monitoraggio e si è verificato che quest'ultimo registrasse i dati attesi.

Per il secondo tipo di test sono invece stati utilizzati due dispositivi: un primo dispositivo con la versione originale di Android 9 compilata dai sorgenti AOSP, mentre il secondo con la versione modificata mediante l'implementazione del servizio di sistema.

Mentre l'obiettivo del primo tipo di test era quello di verificare la correttezza della soluzione, lo scopo del secondo tipo era quello di confrontare il sistema originale con quello modificato valutando le differenze nell'utilizzo delle risorse di sistema.

Per l'esecuzione di entrambi i test sono state generate due immagini di sistema in grado di essere eseguite sull'emulatore Android fornito dall'SDK

per sviluppatori da Google. Entrambe le immagini sono state compilate per essere eseguite su dispositivi con architettura x86_64 e sono state eseguite su un computer con le seguenti caratteristiche:

- Sistema operativo: Windows 10;
- Processore: Ryzen 5 3600;
- Memoria RAM: 16GB.

Entrambe le immagini di sistema sono state eseguite su un emulatore clone del Google Pixel 2 XL con le seguenti caratteristiche hardware:

- 1.5GB di memoria RAM;
- CPU multi-core con 4 core;
- Display da 5.99";
- Display con risoluzione 1440x2880;
- Display con densità 560 dpi.

Dato che per il funzionamento del sistema di monitoraggio la presenza di un server esterno che permetta la sincronizzazione dei dati è fondamentale, anche se la creazione di questo server non fa parte di questa tesi, ho deciso di creare un server di tipo REST nel linguaggio Python per poter rispondere alle necessità del servizio.

Il compito del server è quello di fornire un file statico, contenente la lista delle applicazioni osservabili, e di rispondere a tre chiamate HTTP di tipo POST:

- Chiamata per l'invio dei dati dinamici;
- Chiamata per l'invio dei log delle applicazioni;
- Chiamata per l'invio delle analisi delle Activity.

Ogni volta che il server riceve una chiamata di tipo POST si occupa anche di salvare il contenuto di quest'ultima su un file in formato JSON in modo che possa essere visualizzato e confrontato con i dati attesi.

6.2 Test di verifica del funzionamento

Per i test di verifica del funzionamento dell'applicazione e della correttezza dei dati raccolti ho deciso di creare un'applicazione composta da due Activity (figura 6.1). Ogni riferimento all'applicazione presente in questa sezione sarà relativo a quella appena specificata.

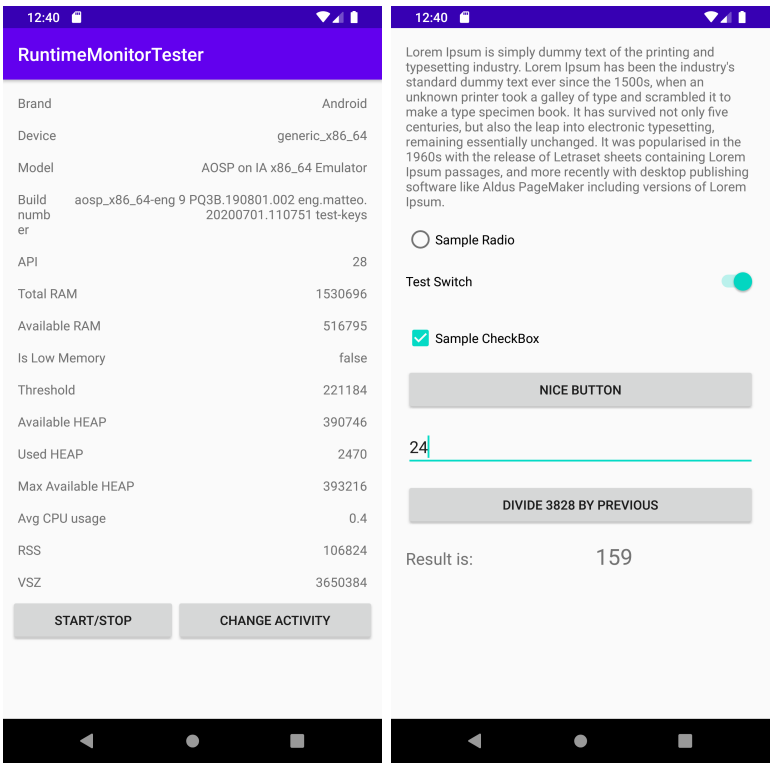

Figura 6.1: Applicazione utilizzata per testare il servizio

La prima Activity Mostra un elenco di dati che corrispondono ai dati dinamici e ai dati statici a cui l'applicazione può accedere e sono presenti due bottoni; il primo per avviare o mettere in pausa l'aggiornamento delle informazioni dinamiche, mentre il secondo per aprire la seconda Activity.

La seconda Activity mostra alcuni dei Widget più usati nella costruzione delle interfacce utente di Android e, fra questi, sono presenti due bottoni. Il primo bottone genera dei log casuali, mentre il secondo permette di generare un'operazione illegale che causa l'interruzione dell'applicazione.

Sfruttando i dati raccolti e confrontandoli con le informazioni conosciute e con i dati attesi è stato possibile verificare la correttezza delle informazioni raccolte dall'applicazione.

Listing 6.1: Esempio di raccolta di dati dinamici


```
{
    "brand": "Android",
    "device": "generic_x86_64",
    "model": "AOSP on IA x86_64 Emulator",
    "build": "aosp_x86_64-eng 9 PQ3B.190801...",
    "API": "28",
    "cpuModel": "AMD Ryzen 5 3600 6-Core Pr...",
    "coreNr": "4",
    "RAM": "1530696",
    "data": [
        {
            "package": "it.polimi.rmtester",
            "timestamp": "1594190982602",
            "availRam": "961772",
            "isLowMem": "false",
            "threshold": "221184",
            "availHeap": "386062",
            "usedHeap": "7154",
            "maxAvailHeap": "393216",
            "cpuUsage": "8.0",
            "avgCpuUsage": "5.1",
            "RSS": "109912",
            "VSZ": "3611800"
        }
    ]
}
```


Il listato 6.1 mostra un esempio di raccoglimento dei dati dinamici durante l'esecuzione dell'applicazione in oggetto. Mentre per i primi dati, cioè quelli statici, è possibile verificare la correttezza, per quelli dinamici è impossibile farlo con sufficiente precisione. Infatti, dato che il servizio effettua rilevazioni periodiche, è impossibile conoscere i valori esatti nel momento in cui il servizio effettua il raccoglimento e ci si deve limitare a fare una valutazione approssimativa. Si può però notare che i dati rilevati dal servizio sono spesso uguali o simili a quelli raccolti dall'applicazione ed è quindi plausibile ipotizzare che i dati raccolti dal servizio siano corretti.

Il listato 6.2 mostra un esempio di log generato dall'applicazione in seguito a una divisione per 0. In questo caso è stato sfruttato il bottone presente nella seconda Activity dell'applicazione per effettuare una divisione per 0, senza cattura degli errori, che ha generato un errore e l'interruzione dell'applicazione. Da questo e dagli altri test effettuati è risultato come l'applicazione riesca

a catturare i log di qualsiasi applicazione osservabile con livello di priorità maggiore o uguale a Warning senza alcun problema.

Listing 6.2: Esempio di raccolta di log d'errore

```
{
    ...
    "data": [
        {
            "package": "it.polimi.rmtester",
            "log": "AndroidRuntime ERROR
            FATAL EXCEPTION: main
            Process: it.polimi.rmtester, PID: 3156
            java.lang.IllegalStateException: Could not
                execute method for android:onClick
            android.view.View$DeclaredOnClickListener.
                onClick(View.java: 5634)
            android.view.View.performClick(View.java:
                6597)
            android.view.View.performClickInternal(View
                .java: 6574)
            android.view.View.access$3100(View.java:
                778)
            android.view.View$PerformClick.run(View.
                java: 25885)
            android.os.Handler.handleCallback(Handler.
                java: 873)
            android.os.Handler.dispatchMessage(Handler.
                java: 99)
            android.os.Looper.loop(Looper.java: 193)
            android.app.ActivityThread.main(
                ActivityThread.java: 6612)
            java.lang.reflect.Method.invoke(Native
                Method)
            com.android.internal.os.
                RuntimeInit$MethodAndArgsCaller.run(
                RuntimeInit.java: 493)
            com.android.internal.os.ZygoteInit.main(
                ZygoteInit.java: 858)

            Caused by: java.lang.reflect.
                InvocationTargetException
            java.lang.reflect.Method.invoke(Native
                Method)
```

```
                android.view.View$DeclaredOnClickListener.
                    onClick(View.java: 5629)
                ... 11 more

                Caused by: java.lang.ArithmeticException:
                    divide by zero
                it.polimi.rmtester.Component.
                    computeDivision(Component.kt: 22)
                ... 13 more"
            }
        ]
    }
```

Il listato 6.3 mostra invece un estratto di un'analisi della seconda Activity dell'applicazione. Come si può notare, per ogni View il servizio è in grado di recuperare il nome della classe e le informazioni riguardo alla posizione e alle dimensioni del layout. Per i Widget di tipo testuale viene anche mostrato il contenuto. Alla fine della rilevazione il servizio allega anche le informazioni relative al display. La correttezza delle informazioni è stata verificata confrontandole con i dati noti, come le caratteristiche hardware dell'emulatore o i file XML utilizzati per definire il layout in fase di creazione dell'applicazione, e sfruttando anche le funzionalità dell'applicazione Developer Assistant che è stata presentata nel capitolo 3.

Listing 6.3: Esempio di analisi di una View

```
{
    ...

    {
    "class": "android.widget.LinearLayout",
    "id": -1,
    "width": 1440,
    "height": 2121,
    "marginLeft": 0,
    "marginTop": 0,
    "children": [
        {
            "class": "android.widget.TextView",
            "id": 2131296462,
            "width": 1328,
            "height": 636,
            "marginLeft": 56,
            "marginTop": 56,
```

```
            "text":"Lorem Ipsum is simply dummy text of
                the printing and typesetting industry.
               Lorem Ipsum has been the industry\'s
               standard dummy text ever since the 1500s
               , when an unknown printer took a galley
               of type and scrambled it to make a type
               specimen book. It has survived not only
               five centuries, but also the leap into
               electronic typesetting, remaining
               essentially unchanged. It was
               popularised in the 1960s with the
               release of Letraset sheets containing
               Lorem Ipsum passages, and more recently
               with desktop publishing software like
               Aldus PageMaker including versions of
               Lorem Ipsum."
        },
        {
            "class": "android.widget.RadioButton",
            "id": 2131296409,
            "width": 1328,
            "height": 112,
            "marginLeft": 56,
            "marginTop": 748,
            "text": "Sample Radio"
        },
        {
            "class": "android.widget.Switch",
            "id": 2131296448,
            "width": 1328,
            "height": 95,
            "marginLeft": 56,
            "marginTop": 916,
            "text": "Test Switch"
        },
        ...
        ]
    },
    ...
    "pxWidth": 1440,
    "pxHeigth": 2880,
    "dpi": 560,
    "dpWidth": 411,
```

```
        "dpHeight": 822,
        "activity": "it.polimi.rmtester.Component"
    }
```

In conclusione, si è potuto verificare come la rilevazione dei dati da parte del servizio sia abbastanza precisa e corretta. Lo svolgimento di questo test è stato utile non solo per evidenziare criticità e mancanze nelle funzionalità del servizio, ma anche nella correzione degli errori che sono stati rilevati e prontamente risolti in fase di test. Per esempio è stato corretto un bug che preveniva la raccolta dei log sul canale di errore e che causava quindi la perdita di informazioni molto importanti per gli sviluppatori.

Per quanto riguarda le criticità e le mancanze evidenziate, come precedentemente detto, i dati raccolti non sono sempre precisi ma possono essere considerati plausibili e abbastanza indicativi del consumo delle risorse di sistema da parte delle applicazioni. Le maggiori mancanze del servizio si notano però nelle funzionalità dedicate all'analisi del layout delle applicazioni.

Il servizio non supporta infatti le Activity con layout dinamico, che in caso di Fragment invisibili potrebbero causare errori nella scansione o, nel peggiore dei casi, scansioni completamente non effettuate in caso di Fragment aggiunti dopo che l'Activity è già stata scannerizzata, in quanto il servizio non riesce a percepire i cambiamenti nelle Activity. Inoltre il servizio non è stato pensato per supportare le più recenti implementazioni delle ultime versioni dei dispositivi Android come la possibilità di avere più applicazioni attive sullo stesso display, oppure come la possibilità di avere dispositivi con più display o con applicazioni fluttuanti.

6.3 Test di comparazione con il sistema originale

Lo scopo di questi test è quello di valutare l'impatto sull'utilizzo delle risorse di sistema dell'implementazione da me effettuata.

Come metro di giudizio per valutare l'efficienza della soluzione, per questi test sono stati usati:

- Consumo medio della memoria RAM da parte del dispositivo;
- Consumo medio della CPU da parte del dispositivo.

Per tutti i test effettuati sono stati rilevati i consumi di CPU e RAM ad intervalli regolari a 1 secondo di distanza gli uni dagli altri e una volta terminati i test è stata calcolata la media dei valori ottenuti.

Per il monitoraggio delle risorse di sistema è stato utilizzato il comando shell Top tramite il comando 6.4 che ha salvato i risultati su un file di testo.

Listing 6.4: Esempio dell'utilizzo del comando top

```
top -m 1 -n 750 -d 1 > out.txt
```

Per effettuare i test di monitoraggio sono state utilizzate le applicazioni, visibili in figura 6.2 installate di default in un'immagine di sistema compilata dai codici sorgenti di Android, in aggiunta all'applicazione creata per verificare il funzionamento del servizio.

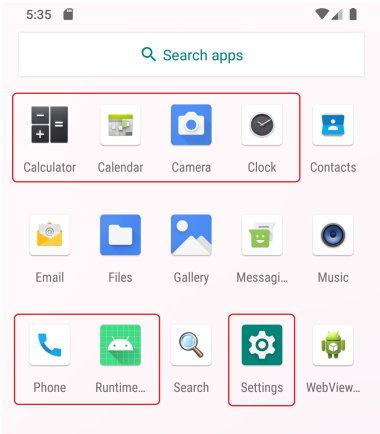

Figura 6.2: Applicazioni installate sul dispositivo e applicazioni osservabili

Fra queste applicazioni ne sono state selezionate alcune, evidenziate in figura 6.2, che sono state impostate come applicazioni osservabili e utilizzate per effettuare i confronti fra i due sistemi, quello originale e quello modificato.

Per eseguire i test è stato ridotto il tempo fra una sincronizzazione e l'altra e sono stati utilizzati Monkey e MonkeyRunner, due tool forniti da Google, per effettuare dei test riproducibili in maniera sistematica in modo da poter confrontare l'utilizzo delle risorse di sistema. Grazie all'utilizzo dei due tool è stato possibile riprodurre lo stesso esatto comportamento dell'utente sui due dispositivi e quindi a effettuare un monitoraggio il più preciso possibile.

Monkey è un programma che viene eseguito direttamente sul dispositivo tramite shell ed è pensato per testare le singole applicazioni [48]. Il suo funzionamento è molto semplice e richiede di fornire un seme che viene usato per la generazione pseudo casuale di eventi che consistono in azioni che l'utente può compiere sul dispositivo. L'utilizzo del seme permette la generazione delle stesse azioni in modo riproducibile anche su dispositivi diversi e quindi permette di riprodurre il test e di confrontare le prestazioni.

Un esempio dell'utilizzo di Monkey è dato da 6.5.

Listing 6.5: Esempio di comando per l'esecuzione di un test con Monkey

```
adb shell monkey -s 1 --throttle 3000 --ignore-
    crashes --ignore-security-exceptions --ignore-
    timeouts 1000
```

Per eseguire i test con monkey sono stati generati vari seed con valore casuale, il dispositivo è stato riavviato e ed è stata eseguita la seguente procedura per ognuno dei semi generati:

1. Scelta del seed;
2. Avvio del comando top per monitorare l'utilizzo delle risorse;
3. Avvio del comando monkey per simulare gli eventi;
4. Attesa della terminazione dei due comandi;
5. Chiusura di tutte le applicazioni avviate;

Al termine dell'esecuzione della procedura per tutti i semi inizialmente generati sono stati estratti dal dispositivo tutti i file generati da top e la procedura è stata ripetuta nello stesso identico modo sul secondo dispositivo.

Al termine dei test su entrambi i dispositivi i file generati sono stati analizzati e le informazioni riguardo all'utilizzo della CPU e della RAM sono state confrontate.

Le figure 6.3 e 6.4 mostrano le differenze fra l'utilizzo di CPU e RAM fra i due sistemi in 5 campioni.

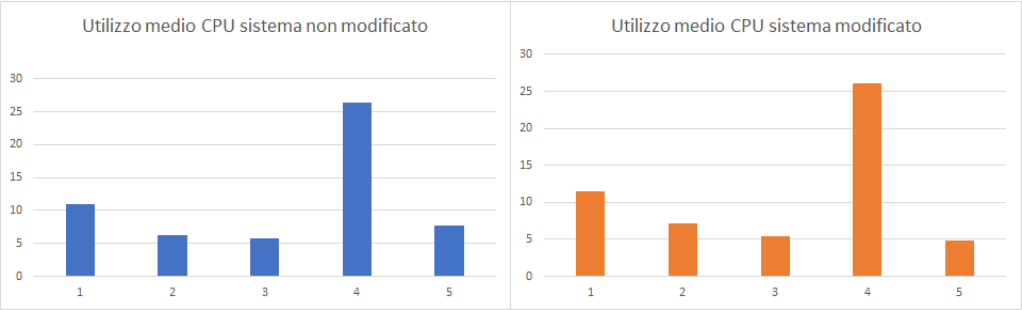

Figura 6.3: Confronto sull'utilizzo della CPU

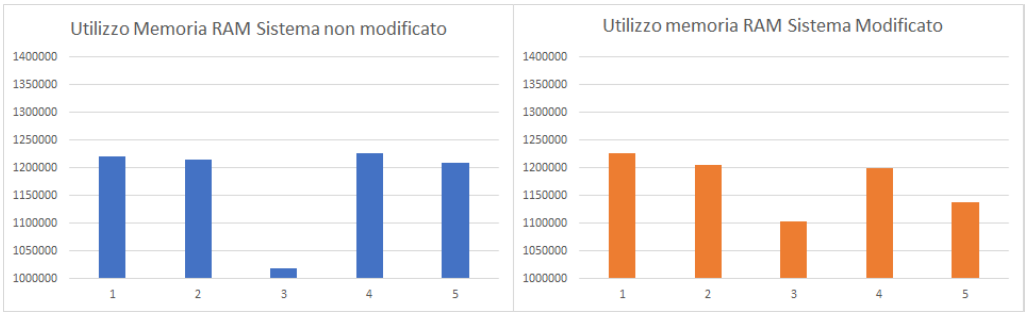

Figura 6.4: Confronto sull'utilizzo della RAM

Come si può notare dalle figure, dai test effettuati con Monkey è risultato come l'utilizzo delle risorse di sistema da parte dei due dispositivi sia più o meno simile e a seconda delle applicazioni e dei servizi in esecuzione in background il sistema, impossibili da controllare, risulta maggiore o inferiore nell'uno o nell'altro. Dai dati ottenuti da questi test non è quindi possibile quantificare l'impatto che l'implementazione ha sulle risorse di sistema, ma bisogna limitarsi a dire che l'utilizzo del sistema di monitoraggio non influisce se non in maniera marginale sull'utilizzo della CPU e della RAM.

MonkeyRunner [49] è un programma il cui funzionamento è simile a quello di Monkey, con la differenza che viene eseguito non sul dispositivo, ma sul computer a cui questo è collegato. Mentre Monkey è pensato per effettuare test relativi alle singole applicazioni, MonkeyRunner è pensato per eseguire dei test a livello di sistema e permette ai programmatori di decidere quali eventi o azioni generare.

MonkeyRunner è stato utilizzato come conferma alle rilevazioni effettuate con Monkey. Per effettuare le valutazioni ho scritto un programma in grado di simulare le azioni degli utenti che utilizzasse quasi esclusivamente applicazioni segnate come osservabili, in modo da evidenziare il più possibile le differenze di utilizzo delle risorse fra le due versioni del sistema operativo.

Le applicazioni utilizzate da MonkeyRunner sono le seguenti:

- RMTester (applicazione creata per verificare il funzionamento);
- SMS;
- Orologio;
- Impostazioni;
- Fotocamera;
- Telefono.

Seguendo una procedura simile a quella utilizzata per Monkey, i dispositivi sono stati riavviati e successivamente il programma è stato eseguito tramite shell utilizzando MonkeyRunner, mentre lo stesso comando Top utilizzato nella fase precedente si occupava di effettuare il monitoraggio dell'utilizzo delle risorse di sistema.

Anche in questo caso non sono state evidenziate differenze nette nell'utilizzo delle risorse di sistema. Più precisamente, è risultato un consumo leggermente superiore sia nell'utilizzo della CPU che in quello della RAM nella versione modificata di Android rispetto a quella compilata dai sorgenti puri dell'AOSP, ma il margine ridotto non permette di concludere che la differenza derivi completamente dalle alterazioni non da altri processi in esecuzione sui dispositivi.

In conclusione, tenendo conto di come il sistema di monitoraggio funziona e dei test effettuati, penso di poter affermare che l'impatto del sistema di monitoraggio da me implementato sulle risorse di sistema sia minimo. Sicuramente è presente un aumento dell'utilizzo della CPU e della RAM ma, mentre il secondo è pressoché irrisorio, il primo è poco influente sul resto del sistema in quanto in genere le operazioni richieste non hanno una durata prolungata nel tempo.

Capitolo 7

Conclusioni e possibili sviluppi

Lo scopo di questo capitolo è quello di presentare alcune considerazioni e di trarre le conclusioni del lavoro svolto. Nella seconda sezione verranno inoltre presentate alcune possibili future modifiche che, se implementate, renderebbero il sistema di monitoraggio più completo e funzionale.

7.1 Conclusioni

Per riassumere, posso dire che il lavoro svolto durante questa tesi può essere diviso in due fasi: una di studio e una di implementazione.

La fase di studio è quella che ha richiesto più tempo ed è stata la più impegnativa. In un primo momento è stata concentrata sul capire che cosa è Android e come funziona, mentre in un secondo momento si è concentrata sui componenti che lavorano "dietro alle quinte", cercando di capire quali sono, cosa fanno e come interagiscono fra di loro.

Una volta scoperto come funziona il framework, lo studio si è concentrato sull'analisi delle possibili soluzioni al problema originale, cioè la creazione di un sistema di monitoraggio, e successivamente, una volta identificata la soluzione, sul capire come fosse possibile crearla e integrarla con il resto del sistema e quali file fosse necessario modificare per raggiungere l'obiettivo.

Quest'ultimo compito si è rivelato tutt'altro che facile, soprattutto a causa della scarsa e spesso assente documentazione presente sia nei sorgenti stessi del sistema operativo, sia sul sito di riferimento per il progetto AOSP. Per risolvere queste mancanze spesso è stato necessario affidarsi a guide di terzi trovate online o interpretare il codice sorgente di altre classi e usarle come esempio per capire il funzionamento dei componenti. Mentre nel primo caso gli articoli trovati online si sono rivelati spesso datati o incompleti ed errati, nel secondo caso le maggiori difficoltà sono derivate dalla complessità

del codice; infatti non è raro trovare classi con lunghezza anche superiore a 20000 righe, come l'ActivityManagerService, che di certo non semplificano la comprensione.

I risultati di questa fase di ricerca sono comunque stati buoni e mi hanno permesso di scoprire non solo come funzionano il framework di Android e i suoi componenti, ma come è strutturato l'intero sistema operativo basato sul kernel Linux.

Ho scoperto cosa sono e come funzionano le applicazioni privilegiate e i servizi di sistema. Ho scoperto come viene gestita l'esecuzione dei vari processi associati alle applicazioni e che ruolo i servizi di sistema ricoprono nella loro gestione e nel loro funzionamento. Ho anche scoperto come avviene la comunicazione fra più processi e sono riuscito a creare un progetto per l'implementazione di un sistema di monitoraggio all'interno del framework esistente. Data la complessità del codice non escludo comunque la possibilità che possano esistere soluzioni migliori o più semplici e che richiedano meno alterazioni al codice sorgente.

La seconda fase della tesi, quella di implementazione, si è rivelata più facile di quella di studio anche se pure in questo caso ci sono state varie difficoltà che hanno rallentato lo sviluppo. I problemi principali che hanno rallentato lo sviluppo derivano però non dal codice ma dall'ambiente di sviluppo. La mancanza di un IDE con supporto a funzioni come intellisense e controllo degli errori ha reso lo sviluppo più difficoltoso. A queste difficoltà si sono aggiunte quelle dovute alla compilazione. I tempi di compilazione infatti, a causa delle dimensioni del progetto, non sono affatto ridotti e spesso i messaggi d'errore non sono chiari. Spesso si è rivelato necessario compilare l'intero sistema operativo per scoprire l'esito della compilazione e correggere eventuali problemi.

Nel complesso la fase di sviluppo è stata comunque molto più veloce della fase di studio. Nonostante ciò, anche in questo caso la mancanza di documentazione si è fatta sentire. Per esempio, per l'esposizione del servizio tramite SELinux è stato necessario integrare le istruzioni fornite da Google con altre istruzioni tradotte dal cinese in modo da riuscire a correggere gli errori generati in fase di compilazione.

Il risultato è stato comunque quello della creazione di un sistema e di una sua immagine in grado di essere eseguita sull'emulatore per Android. Il sistema, nonostante la semplicità, credo funzioni discretamente e offra una buona base per lo sviluppo di una possibile implementazione da parte di un produttore.

7.2 Possibili sviluppi

Sia nella fase di studio che in quella di implementazione e testing ho individuato varie possibili migliorie e alcuni problemi a cui la soluzione non pone attualmente rimedio ma che potrebbero essere implementati in versioni future.

Dal punto di vista del funzionamento del servizio si potrebbe effettuare un'ottimizzazione delle funzioni di raccoglimento delle risorse in modo che il thread di monitoraggio si attivi, per esempio, anche in seguito a un evento e non solamente in modo periodico. Il raccoglimento dei dati potrebbe essere fatto ogni volta che viene aperta un'Activity o che un Fragment nell'applicazione viene sostituito con un altro e i dati raccolti riguardo all'utilizzo delle risorse di sistema potrebbero essere raccolti non solo in modo istantaneo ma anche per un determinato periodo di tempo. Si potrebbe quindi migliorare il metodo di raccoglimento delle informazioni per ottimizzarli e fornire informazioni più significative e fedeli a quello che avviene effettivamente sul dispositivo.

Per quanto riguarda i dati che vengono raccolti, si potrebbero introdurre nuove categorie di dati che potrebbero essere anch'esse utili agli sviluppatori, come statistiche riguardo all'utilizzo della rete da parte delle applicazioni, consumo energetico, tempo totale di utilizzo da parte degli utenti e tempo per cui l'applicazione è stata utilizzata in background. Si potrebbe poi prestare maggiore attenzione alle librerie esterne e ai vari componenti non integrati nel sistema operativo per capire come poterli analizzare e come includere i risultati nell'analisi effettuata.

Si potrebbe inoltre implementare un sistema di personalizzazione che permetta agli sviluppatori di selezionare quali dei dati preferiscano che vengano raccolti per le loro applicazioni, in modo da concentrarsi solo sui dati che interessano realmente. Infine, per permettere l'implementazione in un sistema reale sarebbe necessario aggiungere anche un meccanismo che permetta agli utenti di abilitare e disabilitare manualmente le funzioni di monitoraggio, in modo da rispettare maggiormente la loro privacy e le loro preferenze.

In conclusione, come detto in precedenza, credo che la soluzione funzioni in modo efficiente ma non escludo che possano esistere implementazioni migliori che sono sfuggite a questo studio. Il lavoro potrebbe costituire la base di lavoro su cui uno o più OEM potrebbero costruire un sistema di monitoraggio dedicato agli sviluppatori da implementare nei loro dispositivi. Implementando le possibili modifiche appena descritte si potrebbe ottenere un sistema completo e molto funzionale.

Elenco delle figure

2.1 Metodo di sviluppo dell'AOSP 5
2.2 L'Architettura di un sistema Android 8
2.3 Confronto fra Java Virtual Machine e Dalvik Virtual Machine 9
2.4 Differenze fra Dalvik e ART 10
2.5 Sequenza di boot di un dispositivo Android 11
2.6 Rappresentazione di una comunicazione IPC 13
2.7 Ciclo di vita di un'Activity Android 14

3.1 Distribuzione delle versioni Android 19
3.2 Distribuzione hardware dei display dei dispositivi Android . . 21
3.3 Un esempio dei dati raccoglibili da AnotherMonitor 22
3.4 Analisi di AnotherMonitor tramite Developer Assistant 23
3.5 Un esempio dei dati raccoglibili da CPU-Z 24

5.1 Principali componenti del servizio 39
5.2 Classe RuntimeMonitorService e le sue classi interne 40
5.3 Sequence Diagram del Thread di monitoraggio 51
5.4 Sequence diagram che rappresenta la segnalazione di un log . . 54
5.5 Class Diagram relativo al Network Handler 55

6.1 Applicazione utilizzata per testare il servizio 59
6.2 Applicazioni installate sul dispositivo e applicazioni osservabili 65
6.3 Confronto sull'utilizzo della CPU 66
6.4 Confronto sull'utilizzo della RAM 67

Elenco delle tabelle

4.1 Dove è possibile trovare le informazioni riguardo al dispositivo 32
4.2 Dove è possibile trovare le informazioni riguardo le performance 35
4.3 Dove è possibile trovare le informazioni riguardo l'UI 36
4.4 Dove è possibile trovare i messaggi di Log 37

Listings

6.1 Esempio di raccolta di dati dinamici 60
6.2 Esempio di raccolta di log d'errore 61
6.3 Esempio di analisi di una View 62
6.4 Esempio dell'utilizzo del comando top 65
6.5 Esempio di comando per l'esecuzione di un test con Monkey . 66

Bibliografia

[1] Android Police Taylor Kerns. *There are now more than 2.5 billion active Android devices.* URL: https://www.androidpolice.com/2019/05/07/there-are-now-more-than-2-5-billion-active-android-devices/.

[2] J. clement. *Google Play: number of available apps 2009-2020.* URL: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[3] *About the Android Open Source Project.* URL: https://source.android.com/.

[4] *Opne Handset Alliance.* URL: https://www.openhandsetalliance.com/.

[5] *Android Software Management.* URL: https://source.android.com/setup/start/codelines.

[6] *Android Architecture.* URL: https://source.android.com/devices/architecture.

[7] *Binder IPC.* URL: https://elinux.org/Android_Binder.

[8] *Android Runtime.* URL: https://source.android.com/devices/tech/dalvik.

[9] *Dalvik Virtual Machine.* URL: https://source.android.com/devices/tech/dalvik.

[10] Android Java Point. *Closer Look At Android Runtime: DVM vs ART.* URL: http://androidjavapoint.blogspot.com/2017/04/closer-look-at-android-runtime-dvm-vs.html.

[11] URL: https://dzone.com/articles/android-internals-the-android-os-bootup-process.

[12] URL: https://community.nxp.com/docs/DOC-102546.

[13] URL: https://www.geeksforgeeks.org/android-boot-process/.

[14] *Android Interface Definition Language (AIDL)*. URL: https://developer.android.com/guide/components/aidl.

[15] AndroidPub Budhdi Sharma. *Android Binder Framework*. URL: https://android.jlelse.eu/android-binder-framework-8a28fb38699a.

[16] *Security-Enhanced Linux in Android*. URL: https://source.android.com/security/selinux.

[17] *Introduction to Activities*. URL: https://developer.android.com/guide/components/activities/intro-activities.

[18] *Activity Lifecycle*. URL: https://developer.android.com/reference/android/app/Activity#activity-lifecycle.

[19] *View*. URL: https://developer.android.com/reference/android/view/View.

[20] *Frament*. URL: https://developer.android.com/guide/components/fragments.

[21] *Service*. URL: https://developer.android.com/guide/components/services.

[22] *Intent*. URL: https://developer.android.com/reference/android/content/Intent.

[23] *Broadcast Receiver*. URL: https://developer.android.com/guide/components/broadcasts.

[24] The Verge Jacob Kastrenakes Russell Brandom. *Google mandates two years of security updates for popular phones in new Android contract*. URL: https://www.theverge.com/2018/10/24/18019356/android-security-update-mandate-google-contract.

[25] Google. *Distribution Dashboard*. URL: https://developer.android.com/about/dashboards.

[26] *AnotherMonitor di Antonio Redondo*. URL: https://github.com/AntonioRedondo/AnotherMonitor.

[27] *How to find out which app hoggs CPU on your Android*. URL: https://mobileinternist.com/cpu-usage-android.

[28] *Developer Assistant*. URL: https://github.com/jwisniewski/android-developer-assistant.

[29] *Implementing Your Own Assistant*. URL: https://developer.android.com/training/articles/assistant.html#implementing_your_own_assistant.

[30] *CPU-Z*. URL: https://play.google.com/store/apps/details?id=com.cpuid.cpu_z.

[31] *Cat Log*. URL: https://play.google.com/store/apps/details?id=com.nolanlawson.logcat.

[32] *https://nolanlawson.com/2012/09/02/catlog-jives-with-jelly-bean-goes-open-source/*. URL: https://nolanlawson.com/2012/09/02/catlog-jives-with-jelly-bean-goes-open-source/.

[33] *Google Play Developer Console*. URL: https://play.google.com/apps/publish/?account=6204455412943667213.

[34] *Privileged Permission Whitelisting*. URL: https://source.android.com/devices/tech/config/perms-whitelist.

[35] *Dalvik Executable format*. URL: https://source.android.com/devices/tech/dalvik/dex-format#dalvik-enclosingclass.

[36] *Java Bytecode*. URL: https://en.wikipedia.org/wiki/Java_bytecode.

[37] *Sign your app*. URL: https://developer.android.com/studio/publish/app-signing.

[38] *Build*. URL: https://developer.android.com/reference/android/os/Build.

[39] */proc/*. URL: https://man7.org/linux/man-pages/man5/proc.5.html.

[40] *Top*. URL: https://man7.org/linux/man-pages/man1/top.1.html.

[41] *PS*. URL: https://man7.org/linux/man-pages/man1/ps.1.html.

[42] *Understanding /proc/meminfo file (Analyzing Memory utilization in Linux)*. URL: https://www.thegeekdiary.com/understanding-proc-meminfo-file-analyzing-memory-utilization-in-linux/.

[43] *Runtime*. URL: https://developer.android.com/reference/java/lang/Runtime.

[44] *Log*. URL: https://developer.android.com/reference/android/util/Log.

[45] *Alarm Manager*. URL: https://developer.android.com/reference/android/app/AlarmManager.

[46] *Android 9.0 (P version) Service linux permissions in SystemServer*. URL: http://www.jeepxie.net/article/678521.html.

[47] *Save data using SQLite*. URL: https://developer.android.com/training/data-storage/sqlite.

[48] *UI/Application Exerciser Monkey*. URL: https://developer.android.com/studio/test/monkey.

[49] *MonkeyRunner*. URL: https://developer.android.com/studio/test/monkeyrunner.