Università degli Studi di Milano - Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

Invivo testing di applicazioni Android

Relatore: Leonardo Mariani

Co-relatore: Matteo Orrù

Relazione della prova finale di:

Alfredo Russo

Matricola 816536

Anno Accademico 2018-2019

Indice

Elenco delle figure **2**

1 Il problema della frammentazione di Android **5**

1.1 Introduzione 5

1.2 Esempio del problema della frammentazione di Android 7

2 Un framework per invivo testing: problematiche affrontate **9**

2.1 Importanza del framework 9

2.2 Sfide da superare per la realizzazione di un framework di in-vivo testing 10

2.3 Esempio applicativo 12

3 Invivo framework: achitettura e workflow **16**

3.1 Componenti framework 16

3.2 Interazione tra i componenti 20

3.3 Scelte implementative 22

4 Caso studio: ownCloud **29**

4.1 Processo di sviluppo 29

4.2 Recupero informazioni configurazione 32

4.3 Implementazione client e lavoro sul server 38

4.4 Simulazione scenari 43

Bibliografia **47**

Elenco delle figure

1.1 Distribuzione delle versioni di Android[1]. 6

2.1 ChatApp feature model [2]. 13

3.1 Scoperta di una nuova configurazione. 26
3.2 Esecuzione fase di testing. 27
3.3 Aggiornamento feature model. 28

4.1 Script per la creazione dell'immagine dell'*in-vivo* server. 31
4.2 Impostazioni dell'applicazione ownCloud. 33
4.3 Metodo che permette di scrivere la configurazione su file. 35
4.4 Metodo per controllare se la configurazione corrente deve essere salvata su file. 36
4.5 Configurazione di default. 37
4.6 39
4.7 Rappresentazione di due *OrderedList*. 42
4.8 Configurazione di tipo *Unknown*. 44

Introduzione

La fase di test riguardante un software è sempre stato un lavoro difficoltoso che grava non poco sulle software houses.

Una particolare categoria di bug emerge al variare delle configurazioni dei software e dei sistemi. In particolare negli ultimi anni c'è stata una diffusione di sistemi diversi tra loro e che sono anch'essi altamente configurabili. Questo ha reso ulteriormente difficoltoso testare un software. Tra i diversi sistemi prima menzionati ne troviamo uno in particolare, il sistema Android. Vista la sua crescente popolarità (e di conseguenza delle applicazioni che vi girano) e la sua diffusione non uniforme, data la variatà di dispositivi su cui è presente, il problema sopra menzionato risulta ancora più grave.

La presente tesi affronta il problema di aiutare gli sviluppatori nella scoperta e classificazione delle configurazioni che possono essere la causa dei bug prima citati. Per fare questo è stato utilizzato un framework che è stato sviluppato in parte durante un progetto di ricerca e quindi non tutto è stato prodotto dal mio lavoro. In particolare, durante l'attività di debigging si è scoperto che il framework presentava degli errori che andavano risolti, inoltre c'era bisogno di implementare le sue funzionalità all'interno dell'applicazione di ownCloud che ha ricoperto il ruolo di caso studio.

Il Capitolo 1 di questa tesi tratta più approfonditamente il problema della frammentazione di Android. Nel Capitolo 2 vengono introdotte le problematiche che sono state affrontate per la creazione di tale framework. Nel Capitolo 3, invece, è proposta una soluzione su come questo potesse essere implementato con le relative scelte che sono state effettuate. Nel Capitolo 4 vengono implementate le funzionalità del framework *in-vivo* in uno scenario che vede protagonista un'applicazione di nome

ownCloud.

Capitolo 1

Il problema della frammentazione di Android

1.1 Introduzione

Con lo sviluppo dei dispositivi mobili molte applicazioni già presenti in formato web e desktop sono state sviluppate anche per quest'ultimi. Molte altre invece sono state sviluppate in esclusiva. Lo sviluppo avvenuto riguarda tra tutti la potenza dei processori che sono sempre più simili a quelli che troviamo nei dispotivi fissi e le memorie di archiviazione che sono sempre più capienti. Questo ha portato sempre più a una maggiore complessità delle applicazioni. Tale complessità si è manifestata soprattuto sul numero di settaggi e preferenze che l'applicazione offre all'utente finale. Molte di queste opzioni sono diventate quasi uno standard, dovuto anche ai produttori di piattaforme mobili che hanno sempre più facilitato tale compito. Se prendiamo in esempio Android, uno dei più diffusi sistemi operativi per dispositivi mobili [3], attraverso l'AndroidX Preference Library [1] ha reso estremamente semplice agli sviluppatori creare e gestire le preferenze della loro applicazione, a meno che non ci sia bisogno di preferenze personalizzate.

Tuttavia non è tutto così semplice, infatti le versioni del sistema oprativo sono molteplici e molti dispositivi possiedono ancora versioni obsolete che obbligano gli sviluppatori a non poter usare tutte le nuove funzionalità del sistema se si vuole

[1] `https://developer.android.com/guide/topics/ui/settings`

puntare ad una maggiore diffusione del proprio software. Infatti come riportato in [3], nel 2015, erano presenti 24.000 diversi dispositivi prodotti e 1.000 diversi produttori. Le differenti versioni del sistema opretivo e dei dispositivi prodotti vanno a creare il cosiddetto *Android fragmentation problem* [4]. Questo problema riguarda la diffusione non uniforme delle nuove versioni del sistema operativo dato che molti modelli non vengono aggiornati rimanendo ad una versione obsoleta del sistema come si può notare nella Figura 1.1. Pie è l'ultima versione del sistema operativo ed è meno diffusa di altre sue antecedenti come Oreo, Nougat o Marshmallow. Tale problema, legato ai singoli produttori, va a generare anche dei problemi di sicurezza, esponendo i dispositivi più vecchi a diverse vulnerabilità.

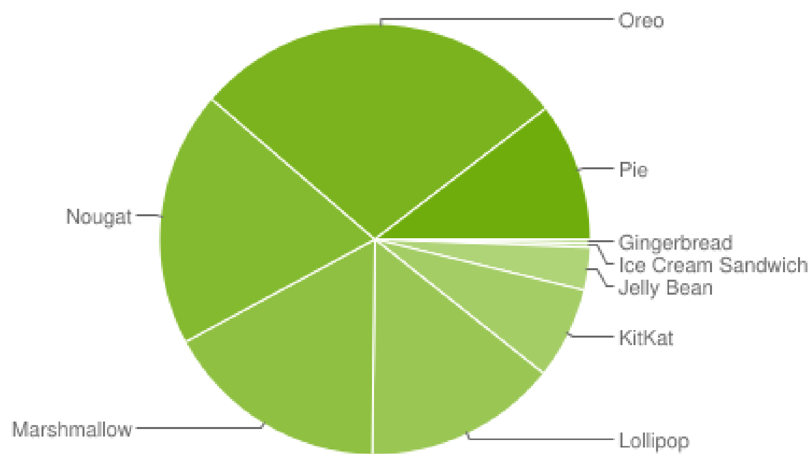


Figura 1.1: Distribuzione delle versioni di Android[1].

L'insieme formato, da una parte dalle diverse versioni del sistema operativo e dai diversi dispositivi e dall'altra l'elevato numero di opzioni possibili (preferenze e settaggi), forma le configurazioni di un'applicazione il quale spazio, tenendo conto dei numeri in questione, è davvero enorme. In fase di test per gli sviluppatori è quindi quasi impossibile riuscire a testare con efficacia tutte le possibili configurazioni prima che l'applicazione venga rilasciata, per non parlare dei bug che molte volte si verificano solo sul campo e quindi solo quando l'applicazione è nella mani dell'utente finale.

1.2 Esempio del problema della frammentazione di Android

Il problema presentato nel paragrafo precedente riguardante la frammentazione del sistema operativo Android è nato con il sistema stesso. Questo è nato con l'idea di proporsi come sistema operativo per dispositivi mobili ed è utilizzato da tanti dispositivi diversi sia sotto l'aspetto hardware che riguardante il loro brand.

Il sistema Android a seconda del brand del dispositivo che stiamo utilizzando può variare in alcune delle sue componenti. Infatti molti produttori prima di rilasciare l'ultima versione si occupano di personalizzare tale sistema. Si pensi per esempio a Samsung che ha sviluppato da poco una nuova interfaccia utente chiamata one UI. La personalizzazione di Android da parte dei brand rende la diffusione delle nuove versioni sempre più lenta.

Molti brand inoltre preferiscono puntare sulla quantità dei loro dispositivi presenti sul mercato e non tutti hanno caratteristiche hardware al passo con i tempi. Tali dispositivi non sono sempre in grado di supportare al meglio la nuova versione del sistema operativo, che quindi, vengono lasciati senza aggiornamenti da parte delle case produttrici.

Per i problemi citati in precedenza c'è stata sempre una disomogeneità delle versione di Android che andavano a complicare il lavoro svolto dagli sviluppatori. Un esempio lampante del problema della frammentazione di Android risale quasi agli inizi del suo rilascio. Infatti, quando si stava effettuando il passaggio dalla versione 2.1 alla 2.2, il famoso gioco angry birds ebbe bisogno di rilasciare due versioni diverse. Google stessa ebbe problemi di compatibilità per la sua applicazione Google Search, questa era presente sotto forma di applicazione per la versione 2.2 del sistema ma era integrata nella versione 2.1. Tuttavia su dispositivi il cui brand aveva personalizzato il sistema ed aveva eliminato tale integrazione non era possibile averla sotto forma di applicazione. Infatti se si provava a cercarla sul market con tali dispositivi questa non compariva, dato che la stessa Google provvedeva a verificare se un dispositivo era compatibile con l'applicazione in questione [2].

Quelli prima presentati sono solo pochi esempi delle difficoltà che sono scaturite dal problema della frammentazione di Android. Il framework *in-vivo* cerca di offrire

[2] `https://searchengineland.com/android-fragmentation-google-search-android-56582`

un supporto agli sviluppatori per agevolare il lavoro di scoperta e test di particolati bug che avvengono a runtime, data la numerosità e diversità delle versioni del sistema Android e dei dispositivi su cui questo è installato e le relative applicazioni che vi girano.

Capitolo 2

Un framework per invivo testing: problematiche affrontate

2.1 Importanza del framework

Il framework in-vivo si occupa di eseguire i test non appena viene rilevata una nuova configurazione. Come già detto però, lo spazio delle configurazione è veramente enorme, quindi c'è bisongo di una cernita delle configurazione rilevanti e quelle non rilevanti.

Quando viene scoperta una nuova configurazione c'è bisogno di classificarla in modo tale che si possa scoprire se tale configurazione è già stata vista in precedenza e magari testata o se invece è la prima volta che questa si presenta.

Nel caso tale configurazione non è mai stata trovata in precedenza devono essere eseguiti dei test e poi aggiungerla alle configurazione testate. I test vengono eseguiti sul dispostivo dell'utente tuttavia ci sono alcune tematiche da tenere in considerazione. Quando si eseguono i test sul campo bisogna fare molta attenzione a non intaccare l'esperienza utente. In particolare bisogna fare attenzione a non intaccare l'integrità delle informazioni presenti sul dispositivo, di non usare molte risorse andando a gravare sulle prestazioni generali del sistema. Inoltre è di particolare importanza, quando vengono scoperte nuove configurazioni, di estrarre solo le informazioni

necessarie per evitare problemi di privacy.

Infine l'utente non deve accorgersi del lavoro del framework *in-vivo* che deve lavorare in background e non essere invasivo.

2.2 Sfide da superare per la realizzazione di un framework di in-vivo testing

L'implementazione del framework *in-vivo* ci pone alcuni problemi da tenere in considerazione e a cui bisogna trovare una soluzione. Tra questi troviamo:

- **Spazio dei test:** nel definire tale spazio il framework deve riuscire a cogliere le configurazioni che sono rilevanti da quelle che non lo sono, tenendo anche conto dei valori assegnati alle feature [2]. Con la mancata gestione di una delle feature rilevanti si potrebbero verificare dei bug che non verrebbero così mai scoperti e testati. Non bisogna gravare molto sulle prestazioni del dispostivo in uso data la grandezza dello spazio dei test. Inoltre è importante un continuo aggiornamento delle feature, per esempio bisogna tener conto degli aggiornamenti del sistema operativo oppure dell'applicazione che si sta testando e l'attivazione del framework *in-vivo* deve essere non invasiva per l'utente finale.

- **Rilevanza configurazione:** la classificazione di una configurazione cambia col tempo, una configurazione che è gia stata testata o una configurazione classificata come *untested* che poi diviene *tested* dopo essere stata testata è poco rilevante agli occhi del framework *in-vivo*. Invece una classificata come *unknown* che a un certo punto viene scoperta è rilevante [2]. Il framework *in-vivo* è pensato per lavorare in un ambiente distribuito, questo è formato infatti da componenti che per svolgere il loro lavoro hanno bisogno di comunicare. Per esempio, quando c'è bisogno di testare una configurazione le componenti sia lato client che server hanno bisogno di comunicare tra loro. Il client ha bisogno di chiedere al server i test per testare la corrente configurazione. Le informazione relative alla classificazione delle configurazioni devono essere condivise dai device che eseguono la stessa applicazione, in modo tale che quando su un device viene scoperta una nuova configurazione, se questa è già stata scoperta

su di un altro non ci sia bisogno di testarla nuovamente. Supponiamo che due utenti stiano utilizzando la stessa applicazione ma con configurazioni differenti e che queste configurazioni siano già state testate. A questo punto uno dei due cambia la propria configurazione in modo tale che queste diventino uguali. Sul dispositivo (client) dell'utente che ha cambiato la propria configurazione questa non è stata ancora testata, così il client provvede a chiedere al server se questa è già stata testata in un qualsiasi altro dispositivo. Se questo è il caso non c'è quindi bisogno di testarla dato che è gia stato fatto in precedenza.

- **Esecuzione dei test:** quando una configurazione deve essere testata il framework deve richiedere i test relativi ed eseguirli. Su questo punto ci sono diverse problematiche riguardo quali test eseguire. Infatti deve essere scelto dal framework se sia meglio eseguire dei test in-vivo, ovvero dei test che possono essere eseguiti sul dispositivo e che sono supportati dal meccanismo di isolamento di quest'ultimo di cui parleremo nel punto successivo oppure dei test ex-vivo che possono essere isolati ed eseguiti soltanto dalla casa produttrice [2]. Tali test comportano un'altro problema, ossia, che è molto complicato replicare dalla casa la configurazione dell'utente come il suo hardware o le informazioni sensibili.

- **Meccanismo di isolamento:** il meccanismo di isolamento introdotto nel punto precedente è molto importante perchè ci da delle garanzie in fase di test di non andare a intaccare l'integrita del sistema e di prevenire effetti collateri, tra i quali possiamo trovare l'accesso a informazioni sensibili dell'utente o addirittura la loro cancellazione. Un meccanismo di sandboxing risulta quindi necessario per evitare inconvenienti, tuttavia può rendere l'efficacia del framework *in-vivo* totlamente inutile dato che viene limitato l'accesso alle informazione utente [2].

- **Prevenzione degli errori:** gli sviluppatori servendosi del framework *in-vivo* possono attivare una sorta di prevenzione degli errori mentre cercano una soluzione definitiva dell'errore che si è presentato. Per esempio tramite il framework si può evitare di far selezionare delle configurazioni che portano a quell'errore.

- **Costi prestazionali:** il lavoro che svolge il framework *in-vivo* deve essere impercettibile agli occhi dell'utente finale e deve essere poco dispendioso sotto tutti i punti di visti che riguardano sia l'aspetto prestazionale che l'uso di internet o della batteria.

2.3 Esempio applicativo

Per comprendere meglio il funzionamento del framework *in-vivo* procederemo con un esempio che riguarderà un'ipotetica applicazione di messaggistica che chiameremo *ChatApp* [2]. Tale applicazione ci permette di scambiare messaggi o file multimediali con altri untenti che ne sono in possesso. Tra le varie funzionalità è presente quella di poter scattare un fotografia e impostarla come foto del proprio profilo. Per fare questo *ChatApp* delega attraverso un *intent* l'operazione di scattare una fotografia a qualsiasi applicazione in grado di farlo. Nel delegare tale operazione si può incorrere in degli errori che possono essere dovuti sia all'applicazione chiamata per scattare una foto, sia alla configurazione dell'applicazione. Bisogna quindi esplorare tali scenari per testare correttamente l'applicazione.

Per esplorare gli scenari citati precedentemente facciamo uso del *feature model* che ci permette di tenere traccia delle configurazioni di un'applicazione il quale spazio può diventare davvero enorme. Inoltre è possibile che presenti dei vincoli che indichino che alcune preferenze possono presentarsi solo in certe condizioni.

Il *feature model* di *ChatApp* e rappresentato nella Figura 2.1. Come si può notare questo è raffigurato come un albero in cui i nodi rappresentano le feature e le foglie i valori che queste possono assumere. In quello di *ChatApp* le feature vengono divise in due categorie, una riguarda la configurazione hardware del dispositivo con l'applicazione a cui è possibile delegare il compito di scattare una foto, mentre l'altra riguarda i settaggi dell'applicazione. Questi in particolare ci fanno capire che l'applicazione *ChatApp* ci offre la possibilità di decidere se vogliamo caricare i file multimediali solo tramite rete mobile o wi-fi o se vogliamo farlo tramite entrambi. Inoltre è possibile decidere se vogliamo edffettuare il backup dei dati oppure no. Si può anche notare che la maggior parte delle feature sono obbligatorie, mentre ce ne sono altre che sono opzionali. In questo caso quelle opzionali sono quelle che ci indicano come possibile scelta dell'applicazione per scattare una foto la GoogleCam

o la MyCam. Rimanendo sempre in ambito della GoogleCamera, nel *feature model* sono presenti anche dei vincoli, infatti le varie versioni della GoogleCamera possono essere usate solo se c'è a disposizione una certa versione del sistema operativo. Per esempio se abbiamo intenzione di usare la versione 5-x della GoogleCamera sul nostro dispositivo c'è bisogno che la versione di Android sia la O. Ancora se stessimo usando un dispositivo Sony, l'hardware della fotocamera (CameraHw) può essere di due tipi che sono il modello IMX300 e IMX400.

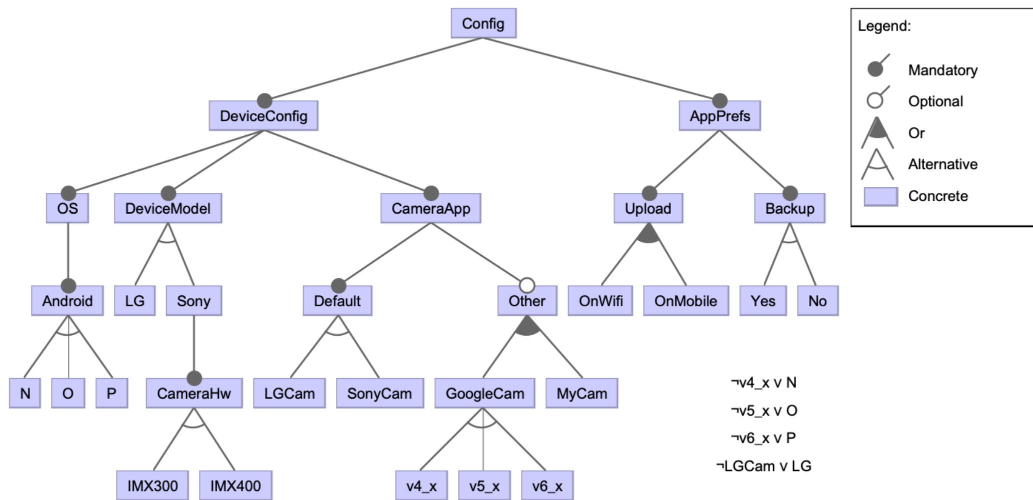


Figura 2.1: ChatApp feature model [2].

Tuttavia il *feature model* mostrato in precedenza è lontano da essere completo. Sono stati presi in considerazione solo alcune delle versione del sistema operativo, solo 3, in confronto alle altre che ancora oggi sono in circolazione. Se diamo uno sguardo ai modelli e a quanti produttori ci sono ci accorgiamo che quelli presi in considerazione sono un numero davvero irrisorio. Ancora se vogliamo citare le applicazioni che ci permettono di scattare foto o il numero di preferenze che di solito troviamo in un'applicazione, messe in confronto alle due citate, è davvero enorme. Rimane quindi, molto complicato riuscire a testare in modo esaustivo una spazio delle configurazioni così ampio. Magari si potrebbe pensare all'uso del *pairwise testing* generando quindi dei test che ci offrano una copertura massima [2]. Tramite questa tecnica però vengono lasciate *untested* molte configurazioni che potrebbero quindi causare un errore riducendo così l'efficienza di *in-vivo*.

Supponiamo che un insieme di configurazioni che sono state testate sono:

{N, LG, MyCam, OnWifi, No}

{O, LG, LGCam, OnWifi, No}

{P, LG, LGCam, OnMobile, No}

Abbiamo quindi ipotizzato che l'applicazione sia stata testata solo su un telefono LG, con tutte le versioni di Android, sia con la MyCam che con l'LGCam, con le impostazioni per il caricamento dei file multimediali sia sotto rete mobile che wi-fi e che il backup delle chat sia disabilitato. Lo spazio delle configurazioni prese in considerazione è davvero piccolo, questo però tende a crescere in maniera esponenziale e tenera traccia di tutte le possibili configurazioni può occupare molto memoria. Col tempo però molte delle configurazioni che sono state testate possono diventare obsolete. Si pensi a una versione del sistema operativo troppo vecchia e ormai non più in circolazione su nessun dispositivo. Per risparmiare memoria quindi, potrebbe essere necessario aggiornare il *feature model* non solo quando si scopre una nuova configurazione, ma anche quando una di queste diventa obsoleta, eliminandola.

Un'esempio rispettivamente di configurazioni *tested*, *untested*, *unknown* sono:

tested {O, LG, LGCam, OnWifi, No}

untested {N, Sony, SonyCam, IMX300, OnWifi, No}

unknown {N, Xiaomi, XiaomiCamera, OnWifi, No}

A seconda di come viene classificata la configurazione c'è un comportamento diverso da parte del framework *in-vivo*. Nel caso di una configurazione *tested* il framework non fa niente, se questa è *untested* si provvede ad eseguire i test per tale configurazione, infine se questa è *unknown* c'è bisogno dell'intervento di un persona che provveda ad aggiornare il *feature model*.

Consideriamo adesso uno scenario in cui *ChatApp* ha bisogno di scattare una foto per aggiornare quella esistente, per farlo manda una richiesta alla XiaomiCamera che però può essere aperta correttamente solo in maniera esplicita dall'utente. ChatApp che si aspetta una risposta non la riceve, lasciando così la richiesta con un riferimento settato con il valore *null*. Dopodichè quando in seguito si cerca di usare la foto viene catturata un'eccezione di *null pointer exception* che fa si che *ChatApp* smetta di funzionare. In questo caso in-vivo deve:

1. riconoscere la configurazione come *unknown*;

2. eseguire gli *in-vivo* test, se non sono disponibili localmente possono essere richiesti al server, per verificare che la fotocamera lavori nel modo corretto con l'applicazione XiomiCamera;

3. riconoscre l'errore di XiaomiCamera;

4. Far usare un'altra applicazione per scattare la foto o disabilitare tale funzione da *ChatApp* per evitare di provocare nuovamente un errore in fase di esecuzione dell'applicazione;

Dopo che è stata scoperta tale configurazione e classificata come *unknown* c'è bisogno di comunicarlo al server, così che possa esserci l'intervento di una persona che può così aggiornare il *feature model* e in questo caso venga aggiunta una nuova foglia sotto il nodo Other con il nome di XiaomiCamera.

Capitolo 3

Invivo framework: achitettura e workflow

3.1 Componenti framework

Il framework *in-vivo* fa parte di un progetto di ricerca. Questo implica che non tutto quello che viene presentato fa parte del mio lavoro. Durante l'attività di debugging sono emersi dei problemi riguardanti l'*in-vivo* server, c'è stato bisogno di implementare le funzionalità del framework *in-vivo* in ownCloud, creare il *feature model* per l'applicazione prima citata e simulare particolari scenari per testare il corretto funzionamento di tale framework (vedere Capitolo 4).

Il framework *in-vivo* presenta un'architettura di tipo client-server. Questi due componenti hanno, come già accennato in precedenza, bisogno di comunicare tra di loro.

Il client viene eseguito sul dispositivo lato utente e si occupa di gestire il processo di test in-vivo locale per l'applicazione da testare alla quale ci riferiremo da qui in avanti con il nome di "Application under test o AUT". Il client dell' in-vivo framework è parte integrante dell'AUT e presenta diverse interfacce e componenti.

L'utilità delle varie interfacce e la loro interazione con le altre componenti del client verrà poi spiegata in seguito. Tra le interfacce possiamo trovare:

- `ConfigurationInterface:` interfaccia che deve essere implementata da chi ha sviluppato l'AUT e che permette di ricevere le informazioni relative alla

configurazione dell'applicazione.

- `ConfigurationUpdateInterface:` sempre implementata dagli sviluppatori, permette, quando c'è stato un cambiamento nella configurazione dell'applicazione, di rilevarlo e avvisa il framework di tali cambiamenti mediante un processo di alert implementato lato server.

- `SelfHealingInterface:` Implementata anch'essa dagli sviluppatori serve per avviare delle procedure che permattano di risolvere anche temporaneamente il problema scaturito da una certa configurazione [1].

Ci sono poi, quattro componenti che sono gestiti dal `ClientService` che viene eseguito in background. Queste sono:

- `Configuration Manager:` questa componente ci permette di controllare la configurazione dell'applicazione, quindi le componenti hardware e le impostazioni di quest'ultima. Questo compito tuttavia non viene svolto tutto in maniera autonoma dal Configuration Manager.

  Questo avviene perchè tale componente è in grado di procurarsi in maniera autonoma le informazioni relative all'hardware del dispositivo e alla versione del sistema operativo che è installato su di esso. Invece quando si parla di settaggi dell'applicazione la situazione inizia a complicarsi. Il `Configuration Manager` non può accedere a tali dati, infatti questo non possiede nemmeno la posizione di dove queste impostazioni vengono salvate.

  Si spiega così l'utilità della `ConfigurationInterface` che viene implementata dagli sviluppatori dell'applicazione e che ci permette quindi di accedere ai dati relativi ai settaggi dell'applicazione che altrimenti sarebbe impossibile ottenere in automatico come avviene invece per quelli prima citati.

  C'è inoltre un altro modo in cui il `Configuration Manager` può accedere alle informazioni che riguardano i settaggi dell'applicazione. Questo riguarda la `ConfigurationUpdateInterface` che ogni volta che c'è un cambiamento nella corrente configurazione lo comunica al `Configuration Manager`. Questa componente ha anche il compito di testare una configurazione che non è ancora

[1] Tale lavoro tuttavia esula dal lavoro di tesi ed è previsto in una seconda fase del progetto

stata testata o comunicare al server se è stata trovata una configuratione di tipo *unknown.* Ogni volta che una configurazione viene trovata questa va analizzata a partire dal *feature model* per decidere se è di tipo *tested, untested* oppure *unknown.*

Nel caso ci trovassimo di fronte una configurazione *untested* il `Configuration Manager` deve comunicare con il `Test Manager` in modo che questo provveda a testare tale configurazione.

Quando invece viene trovata una configurazione *unknown* c'è bisogno che tale componente comunichi con il server in modo che il *feature model* venga in seguito aggiornato.

- `Test Manager:` ha il compito di eseguire i test in-vivo, comunicare i risultati all'in-vivo server e aggiornare la test suite locale. Infatti quando c'è bisogno di testare una configurazione di tipo *untested* il `Test Manager` deve controllare che questa non sia stata testata già su di un altro dispositivo quindi su un altro client. Se è questo il caso allora il `Test Manager` non ha bisogno di proseguire con la fase di test, procedendo così con l'aggiornamento delle informazioni relative alla configurazione in esame. Nel caso tale configurazione non sia stata testata ancora da nessun altro client c'è bisogno di eseguire dei test. Prima di fare questo però è molto importante che il `Test Manager` attivi il meccanismo di isolamento del dispositivo. Come vedremo in seguito, in modo più approfondito, tale meccanismo permette di separare il profilo utente dal *work profile* che è il profilo su cui viene testata l'AUT. Agendo in questo modo i dati sensibili dell'utente vengono salvaguardati evitando errori di questo genere. Dopodichè vengono eseguiti dei test per poi comunicare al server i risultati ottenuti ed in seguito, grazie a queste informazioni, verranno aggiornate le configurazioni testate.

- `Storage Manager:` si occupa di memorizzare le informazione che hanno bisogno di essere conservate col tempo. Per esempio queste possono essere il *feature model*, le configurazione che sono già state testate oppure la test suite di in-vivo. Questo ha bisogno di comunicare sia con il `Configuration Manager` che con il `Test Manager`, quando c'è un cambiamento nelle informazioni che

sono state memorizzate. Questo può capitare quando c'è bisogno di testare una configurazione e quindi vanno aggiornate le informazioni relative alle configurazioni testate. Ancora se viene trovata una configurazione di tipo *untested*, dopo aver eseguito le operazioni citate in precedenza c'è bisogno di aggiornare il *feature model* ed è per questo che c'è bisogno di tale comunicazione.

- `SelfHealing Manager:` Quando viene riscontrato un errore, questa componente si occupa di trovare una soluzione. Le contromisure che vengono usate possono anche essere state implementate dagli sviluppatori dell'applicazione tramite la relativa interfaccia. La soluzione inoltre può anche essere temporanea mentre si aspetta quella definitiva. Una soluzione temporanea potrebbe essere quella di bloccare la scelte dei settaggi che portano alla formazione della configurazione che ha portato all'errore [2].

Il framework *in-vivo* è composto anche da una componente lato server che chiameremo *in-vivo* server. Tale parte a sua volta è formata da tre componenti che vengono gestite dal `ServerService` che ha il compito di processare la richiesta che gli è stata inviata dal client e agire di conseguenza. Le componenti prima citate sono:

- `Configuration Manager:` quando viene testata una configurazione e questa diviene *tested*, il client deve notificare al server che questa è stata testata. Dopodichè il server deve aggiornare le configurazioni testate in modo che la nuova configurazione risulti testata anche *globalmente*. Una configurazione testata globalmente si riferisce a una configurazione che produca come risultato *tested* quando viene comparata con le configurazioni testate presenti nello `Storage Manager` lato server. Se invece viene scoperta una configurazione di tipo *unknown*, il client lo riporta al server che produce una richiesta di cambiamento del *feature model* da parte degli sviluppatori che in seguito possono provvedere a modificarlo inserendo la nuova configurazione trovata.

- `Test Manager:` tale componente riceve dal client i risultati dei test *in-vivo* per poi far partire i test *ex-vivo* se questi sono necessari. Può capitare infatti che il client non possa eseguire alcuni test sul campo nonostante il meccanismo

[2] Così come la SelfHealing Interface questa componente non è stata ancora svilupatta ed è presente in una successiva fase del progetto

di isolamento perchè questi possono produrre degli effetti collaterali. La configurazione osservata sul campo con cui si è trovati impossibilitati ad eseguire i test viene ricreata dagli sviluppatori per poterla testare in un ambiente sicuro.

- `Storage Manager:` questa componente serve per memorizzare, come nel caso del client, le informazioni che hanno bisogno di essere mantenute nel tempo. Tuttavia tali informazioni, sia *feature model* che *test suite*, possono essere modificate dagli sviluppatori.

3.2 Interazione tra i componenti

In questo capitolo verranno illustrate le interazioni che avvengono tra le diverse componenti del framework *in-vivo*, introdotte in precedenza, in diversi scenari che possono presentarsi. Per evitare fraintendimenti, se non verrà specificato, la componente a cui si sta facendo riferimento sarà quella relativa alla parte Client del framework *in-vivo*.

Scenario 1: scoperta nuova configurazione. Il primo scenario che prenderemo in esame è quando viene scoperta una nuova configurazione come illustrato nella Figura 3.1. In tale scenario il `Configuration Manager` svolge un ruolo fondamentale. Può capitare che durante l'uso dell'AUT l'utente cambi i settaggi andando così a generare una nuova configurazione. Tale configurazione verrà così prelevata dal `Configuration Manager` che si occuperà in seguito di inviarla al `ClientService`. È possibile poi che sia il `ClientService` a chiedere la configurazione corrente al `Cofiguration Manager` che a sua volta preleverà dall'applicazione ciò che gli è stato richiesto.

In entrambi i casi il `ClientService` dovrà provvedere a classificare la configurazione corrente. Per fare questo ha bisogno di comunicare con lo `Storage Manager` che provvederà a rispondergli se tale configurazione è già stata testata o meno oppure se questa è di tipo *unknown*.

Nel caso in cui lo `Storage Manager` comunicasse che tale configurazione non è ancora stata testata, il `ClientService` dovrà provvedere a confrontarsi con il `ServerService` per controllare che quest'ultima non sia di tipo *untested* anche globalmente. Il `ServerService` per effettuare tale controllo comunicherà con lo

`Storage Manager` lato Server. Se la risposta è che la configurazione corrente è di tipo *untested* anche globalmente allora si dovrà provvedere ad eseguire lo scenario che riguarda l'esecuzione dei test che vedremo in seguito. Nel caso la risposta fosse che la configurazione è gia stata testata globalmente allora il `ClientService` provvederà ad informare lo `Storage Manager` che provvederà quindi a classificare l'attuale configurazione come *tested*.

Infine se lo `Storage Manager` comunicasse al `ClientService` che la configurazione corrente è di tipo *unknown*, questo implicherebbe un intervento da parte degli sviluppatori. Per fare questo il `ClientService` comunicherà con il `ServerService` che notificherà che è stata trovata una configurazione *unknown*. Chi provvedrerà ad apportare i cambiamenti nel *feature model* dovrà poi riferirlo al `ServerService` che a sua volta li comunicherà allo *Storage Manager* lato server che provvederà ad aggiornare il *feature model*. Anche quest'ultimo scenario verrà visto in modo approfondito in seguito.

Scenario 2: testare una configurazione. Quando c'è bisogno di testare una configurazione il `ClientService` comunica con il `Test Manager` chiedendogli di testare la configurazione corrente come illustrato nella Figura 3.2. Per fare questo il `Test Manager` richiede la test suite allo `Storage Manager`.

Per eseguire ogni test presente all'interno della test suite c'è bisogno di attivare, per ognuno di essi, il meccanismo di isolamento. Se l'attivazione di quest'ultimo va a buon fine il test in questione viene eseguito e i risultati sono ritornati al `Test Manager`.

Nel caso in cui per qualche test non fosse possibile azionare il meccanismo di isolamento il `Test Manager` riporta l'errore al `ClientService` che a sua volta si mette in comunicazione con il `ServerService`. Dopodichè questo chiede al `Test Manager` lato server di eseguire i test che non è stato possibile effettuare lato client. Per l'esecuzione dei test lato server le operazioni eseguite e i componenti che interagiscono sono gli stessi. I risultati finali che sono stati ottenuti vengono poi ritornati al `ClientService` che può così continuare ad aggiornare le configurazioni testate.

Scenario 3: aggiornamento feature model. L'ultimo scenario da prendere in considerazione è quello in cui c'è bisogno di aggiornare il *feature model* (vedere Figura 3.3). Questo potrebbe accadere quando viene trovata una nuova configurazione di

tipo *unknown*. Lo scenario parte con il `ClientService` che richiede il *feature model* al `ServerService`. A sua volta c'è la comunicazione di quest'ultimo con lo `Storage Manager` lato server. Il nuovo *feature model* viene cosi ritornato al *ClientService* che lo comunica allo `Storage Manager` lato client che lo confronta con quello al suo interno. Nel caso il *feature model* ritornato fosse aggiornato rispetto a quello presente nello `Storage Manager` questo viene aggiornato.

3.3 Scelte implementative

Per poter recuperare la configurazione corrente dell'AUT c'è bisogno che questa venga modificata e implementata tale funzionalità. La configurazione deve essere recuparata ogni volta che l'utente apporta dei cambiamenti nelle impostazioni. Tuttavia la nuova configurazione deve essere confrontata con quella precedente per essere sicuri che le due differiscano. Nel caso queste non differiscano tra loro allora non c'è bisogno di processare la nuova configurazione.

Con la comunicazione tra lo `Storage Manager` e il `Configuration Manager` si deve riuscire a classificare la configurazione che è appena stata trovata. Per fare questo lo `Storage Manager` deve immagazinare dati come il *feature model* e le configurazioni che sono già state testate.

È stato quindi pensato come un oggetto che è formato dalle informazioni prima citate e tramite la configurazione corrente che viene prelevata è possibile ragionare su che tipo questa sia.

Per riuscire a costruire un'oggetto del genere è stata pensata una classe con dei *getters/setters* che è in grado di immagazzinare le informazioni che gli servono nel loro formato nativo. Invece per la nuova configurazione che viene prelevata c'è bisogno di poterla gestire.

Per fare questo è stato adottato un linguaggio di script chiamato *FAMILIAR*. Tale linguaggio acronimo di *FeAture Model scrIpt Language for manIpulation and Automatic Reasoning* viene decritto come un linguaggio per importare, esportare, comporre, decomporre, modificare e calcolare differenze del *feature model* [5]. È quindi perfetto per le operazioni che abbiamo bisogno di svolgere per gestire diversi *feature model* e configurazioni. Tale linguaggio ci permette anche di importare un

feature model a partire da un altro che è stato generato in un altro linguaggio come per esempio *SPLOT*. Tale linguaggio permette anch'esso di gestire *feature model*. Questa funzionalità di *FAMILIAR* è stata usata proprio all'interno dello `Storage Manager` per poter importare e gestire *feature model* che sono stati creati mediante il linguaggio *SPLOT*. Per eseguire le operazioni che sono state citate prima *FAMILIAR* mette a disposizione degli operatori.

Per poter infine stabilire il tipo di una configurazione c'è bisogno che lo `Storage Manager` abbia un *feature model* globale e uno che è formato solo dalle configurazioni che sono state già testate. In questo caso quando c'è bisogno di testare una configurazione si può andare a cercare per prima nel *feature model* formato dalle configurazioni che sono già state testate, se questa è presente allora non si deve procedere a svolgere nessuna operazione. Nel caso tale configurazione non sia presente allora si controlla che nel *feature model* globale questa sia presente.

Quando c'è bisogno di testare una configurazione la componente che si va ad utilizzare è il `Test Manager`. Molto importante per questo e per eseguire la test suite è il meccanismo di sandboxing per evitare errori che potrebbero portare all'alterazione dei dati dell'utente.

Tale meccanismo viene eseguito mediante l'uso del *work profile* che è stato inserito all'interno del sistema operativo Android a partire dalla versione 5.0. Il profilo prima citato viene controllato da un admin ed è separato da quello dell'utente. I dati dei due profili non vengono condivisi a meno non si abiliti tale funzionalità. Questo meccanismo ci permette di decidere anche quali applicazioni devono far parte del *work profile* e quali no, è possibile poi in seguito decidere di aggiungere o rimuovere un'applicaziona da tale profilo. Inoltre è possibile stabilire quali permessi deve avere un'applicazione. In questo caso è il `Test Manager` a fungere da admin per il *work profile*. In questo modo può creare o distruggere il *work profile*, aggiungere o rimuovere applicazioni da tale profilo e monitorare le applicazione al suo interno controllando i loro permessi.

Quando c'è bisogno di eseguire dei test il `Test Manager` aggiunge al *work profile* l'applicazione che bisogna testare. A questo punto tale componente invia un intent all'AUT per avviare il processo di testing. Tale intent viene ricevuto da un servizio sviluppato appositamente in tale applicazione. Vengono così eseguiti i test ed i risultati vengono immagazzinati e poi comunicati tramite un intent al `Test Manager`

che provvederà in seguito ad esaminarli. Dopodichè il `Test Manager` provvede ad eliminare l'AUT dal *work profile* per liberare spazio. È molto importante da tenere in considerazione che il *work profile* non è possibile crearlo ogni volta che bisogna eseguire la fase di test, dato che per farlo c'è bisogno di una richiesta da fare all'utente e per non essere troppo invasivi la sua creazione viene fatta una singola volta.

Il compito dell'*in-vivo* server è quello di processare e classificare le configurazioni che gli vengono inviate ed inoltre ha bisogno di gestire delle informazioni permanenti. Questo è stato pensato come una web application che espone una API REST.

Una web application è formata da una o più Java Servlet e/o JSP che sono eseguite all'intero di un servelt container che in questo caso è Tomcat, che si occuperà di gestire l'intero ciclo di vita di quest'utlime. Una Java Servlet non è altro che una classe java che si trova all'interno del servlet container. Mentre le Servlet sono scritte in linguaggio Java le JSP vengono scritte in HTML dove però sono presenti dei particolari tag che permettono di inserire al suo interno del codice Java. Quando il client invia una richiesta, in questo caso all'*in-vivo* server, è il web container, cioè Tomcat, a gestire e indirizzare la richiesta alla risorsa corretta, che sia la Java Servlet o JSP. A questo punto la Java Servlet o JSP si occupa di processare la richiesta e creare la risposta. Dopodichè una volta che Tomcat è entrato in possesso di tale risposta la invia al client. Tomcat inoltre si occupa di gestire le Java Servlet inizializzandole, invocando i suoi metodi a seconda della richiesta ricevuta e distruggendole. Inoltre quando c'è una nuova richiesta viene inizializzato un nuovo thread anzichè inizializzare una nuova Servlet risparmiando così tempo e memoria. Si occupa inoltre di compilare e convertire le JSP in Servlet in modo che possano essere gestite come quest'ultima.

L'*in-vivo* server espone una API REST che comunica con il client mediante il protocollo HTTP. Il paradigma REST permette di sfruttare il protocollo di livello applicativo HTTP per la trasmissione di dati. Questo è uno stile architerttuarale per sistemi distribuiti basato su dei princicpi fondamentali [6]. Una comunicazione HTTP è di tipo client-server, viene effettuata una richiesta HTTP REQUEST da parte del client ed il server provvede ad inviare una risposta HTTP RESPONSE. Dopo che il client ha effettuato la sua richiesta al server questo si disconnette, una volta che il server ha pronta la risposta questo cerca di ristabilire la connessione con il client per consegnarla. L'HTTP fornisce poi, diversi tipi di metodi che possono essere

utilizzati quando il client invia una richiesta, nel nostro caso sarà molto importante il metodo POST con cui il client chiederà al server di elaborare la configurazione che gli è stata inviata e produrre una risposta su che tipo questa sia (TESTED,UNTESTED, UNKNOWN).

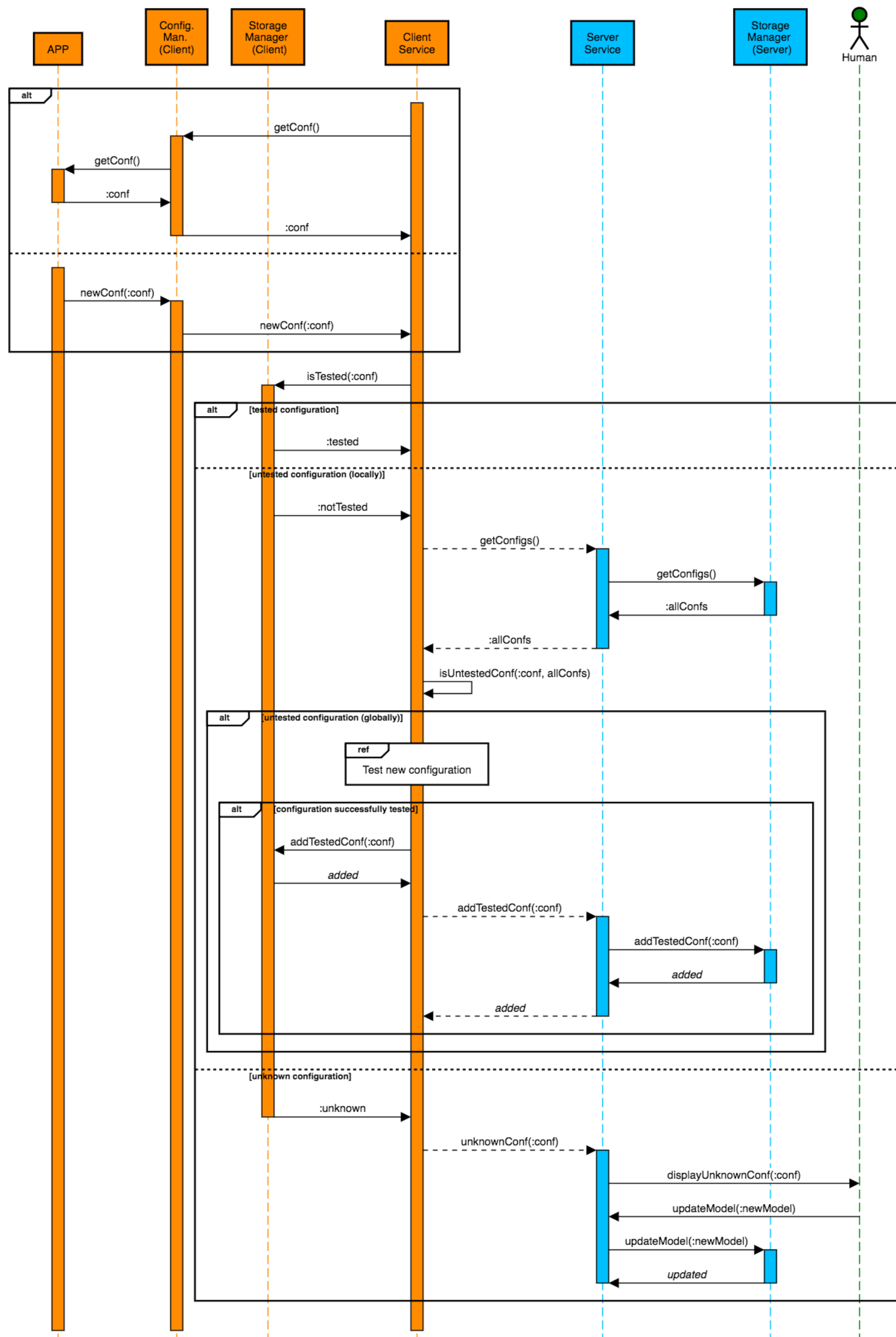


Figura 3.1: Scoperta di una nuova configurazione.

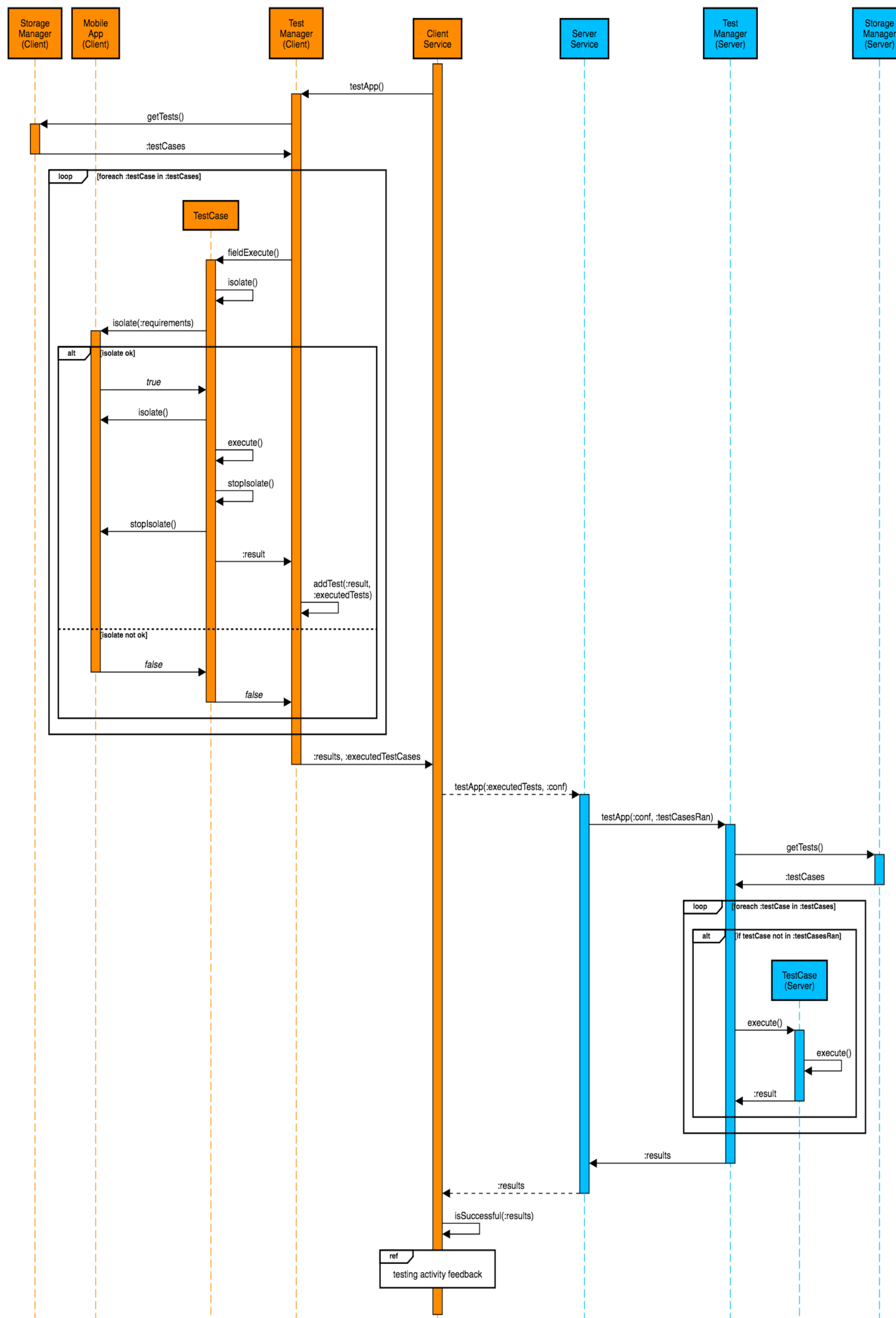


Figura 3.2: Esecuzione fase di testing.

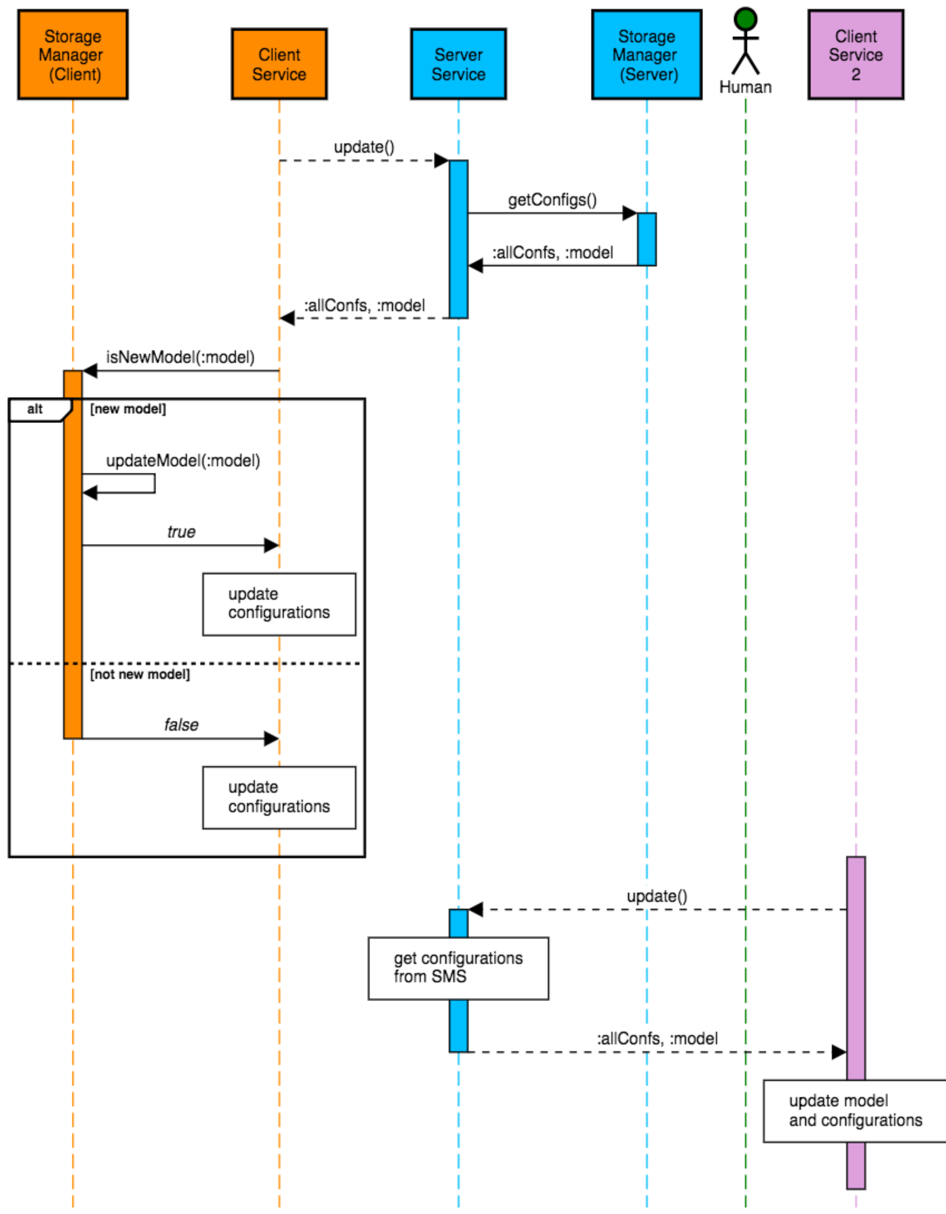


Figura 3.3: Aggiornamento feature model.

Capitolo 4

Caso studio: ownCloud

4.1 Processo di sviluppo

Per testare il funzionamento del framework *in-vivo* si è scelto, come caso studio l'applicazione di ownCloud [1]. Questo è un sofware che offre la possibilità di essere installato su un server web e possiede delle funzionalità volte alla gestione di file e cartelle permettendoci di creare un proprio cloud privato.

Tale software offre quindi un servizio di file hosting che non ha limitiazioni sullo spazio di archiviazione in quanto questo dipende dallo spazio di archiviazione che è presente sull'host su cui viene installato. OwnCloud nasce come un servizio web, quindi una volta installato è possibile configurarlo mediante l'utilizzo di un browser web. Nella configurazione iniziale ci verrà chiesto di scegliere le credenziali d'acesso e quale software usare per la gestione del database interno. Una volta completata la configurazione iniziale sempre tramite browser web sarà possibile iniziare a gestire il proprio cloud personale. La configurazione iniziale di ownCloud non è stata eseguita da me, ma è stata eseguita in precedenza da chi si è occupato dello sviluppo del framework *in-vivo*.

Tale software inoltre è open-source e consente di accedere alle risorse cloud anche utilizzando un'applicazione per dispositivi mobili, la quale però è a pagamento. Tale servizio mette a disposizione una funzionalità che permette di integrare lo spazio di archiviazione che abbiamo già in possesso da servizi come Google Drive, Dropbox o

[1] `https://owncloud.org/`

OneDrive. OwnCloud è provvisto inoltre di funzioni che ci permettono di gestire in modo esaustivo i file al suo interno, con la presenza anche di un editor che ci permette di modificare vari documenti anche simultaneamente con altre persone.

Uno degli strumenti che sono stati utilizzati per gestire in maniera efficiente il progetto *in-vivo* è stato Git. Questo appartiene alla categoria dei software per fare versionamento della base di codice, in particolare ai software con un sistema decentralizzato. I primi software lavoravano in locale, tuttavia non c'era possibilità di poter collaborare con altre persone. Furono così inventi dei software per fare versionamento del codice centralizzati. Questi utilizzavano un singolo server per salvare le diversi versioni del progetto e tenere traccia delle modifiche che venivano apportate. Tuttavia nel caso di un malfunzionamente lato server sarebbe stato impossibile accedere al repository. Per ovviare a tale problema nacquero dei software per fare versionamento del codice decentralizzati. Il progetto e tutte le sue versioni venivano salvate non solo sul server, ma anche su tutti i client. Quindi nel caso di un problema lato server potevano essere utilizzati i repository su quest'ultimi.

Git permette quindi la condivisione di codice tra gli sviluppatori che stanno lavorando a uno stesso progetto. Lo strumento tiene traccia di tutte le modifiche che sono state apportate ad ogni singolo file che sia tracciato. Questa è una funzionalità molto utile perchè qualora le modifiche effettuate portassero ad un problema che non si è in grado di risolvere è possibile tornare alla versione precedente del file oppure alla versione precedente dell'intero progetto. Tra le altre funzionalità troviamo anche la possibilità di confrontare i cambiamenti che sono stati effettuati ad ogni singolo file e lavorando in gruppo è possibile conoscere chi ha apportato determinate modifiche e, se è stato specificato il motivo, è possibile anche capire il perchè questo sia stato fatto.

Durante il processo di implementazione e sviluppo l'*in-vivo* server è stata la componente che che ha presentato le maggiori criticità. Questo per via del fatto che la realizzazione di tale componente era ancora in uno stato primordiale. Tali problematiche verranno trattatte in maggiore dettaglio nel seguito. Inoltre per poter implementare in maniera efficiente l'*in-vivo* server sono stati usati diversi strumenti, tra cui Docker e Apache Maven.

Docker è uno strumento per la gestione e lo sviluppo dei container. Questi ultimi sono dei contenitori in cui è possibile eseguire un'applicazione e le sue dipendenze

fornendo una virtualizzazione a livello di sistema operativo richiedendo così molte meno risorse di una normale virtualizzazione. Questo tipo di virtualizzazione permette di creare un ambiente isolato, nel caso venisse riscontrato un errore nel container questo riguarderebbe solo quest'ultimo e non andrebbe ad intaccare l'intera macchina.

Docker è stato utilizzato per pacchettizzare l'applicazione relativa all'*in-vivo* server. Una volta che si ha a disposizione l'applicazione, nel nostro caso quella dell'*in-vivo* server, prima di riuscire ad inserirla nel container c'è bisogno di creare la sua relativa immagine. Le immagini sono una parte fondamentale di Docker, dato che lo sviluppo delle applicazione all'interno dei container è basata proprio su quest'ultime. Inoltre queste sono organizzate in livelli, infatti quando l'applicazione viene modificata viene creato un nuovo livello all'interno del container, facendo si che nel caso le modifiche abbiano prodotto un errore sia facile poter tornare alla versione precedente dell'applicazione [2].

Per creare l'immagine relativa all'*in-vivo* server c'è stato bisogno di creare il Dockerfile. In tale file è stato specificato come creare il filesystem relativo al container. Una volta creato tale file si può procedere alla creazione dell'immagine tramite il comando *docker build* dopodichè è possibile lanciare il container con il comando *docker run*. Entrambi i comandi hanno poi diverse direttive che è possibile applicare, come per esempio specificare il nome del container che si crea oppure lanciare il container in background. L'esecuzione è stata automatizzata mediante uno script eseguibile da shell dei comandi e si presenta come segue:

```
mvn clean package
docker build -t invivo-server .
docker run -d -p 8080:8080 invivo-server
```

Figura 4.1: Script per la creazione dell'immagine dell'*in-vivo* server.

La prima riga riguarda maven e la vedremo in dettaglio nel punto successivo. La seconda come prima citato contiene il comando per creare l'immagine a partire dall'applicazione dell'*in-vivo* server. Infine l'ultimo comando serve per lanciare il container con l'immagine prima creata. Ci sono però alcune direttive che sono state utilizzate, tra queste troviamo il flag -d che permette di lanciare il container in

[2] https://www.criticalcase.com/it/blog/7-cose-che-devi-sapere-su-docker.html

background e il flag -p che permette di chiedere a docker di inoltrare il traffico in entrata dalla porta 8000 dell'host alla porta 8080 del container. Una volta che il container è stato creato avremo a dispozione tutte le funzionalità che può offrire l'*in-vivo* server che sono state ampiamente descritte nei capitoli precedenti.

Apache Maven è un tool per l'automazione dei processi di compilazione dei progetti, di esecuzione dei test e della creazione della documentazione. Tale strumento, quindi, viene utilizzato per una gestione efficiente dei progetti che possono essere scritti anche in diversi linguaggi.

Maven usa un file XML denominato pom per tenere traccia di tutte le dipendenze relative al progetto e quindi di tutte le librerie di cui il progetto ha bisogno e le dipendenze tra essi. Questo permette anche di riuscire ad organizzare in modo migliore il progetto dato che ci da la possibilità di separare le diverse cartelle, quelle relative ai file e quelle invece relative alle librerie.

Maven inoltre effettua in automatico il download di tutte le librerie che sono specificate nel file prima citato, così da averne la disponibilità in locale permettendo quindi di scaricare i vari file JAR, avendo così la possibilità di usare sempre le stesse librerie anche spostando il progetto in ambienti diversi.

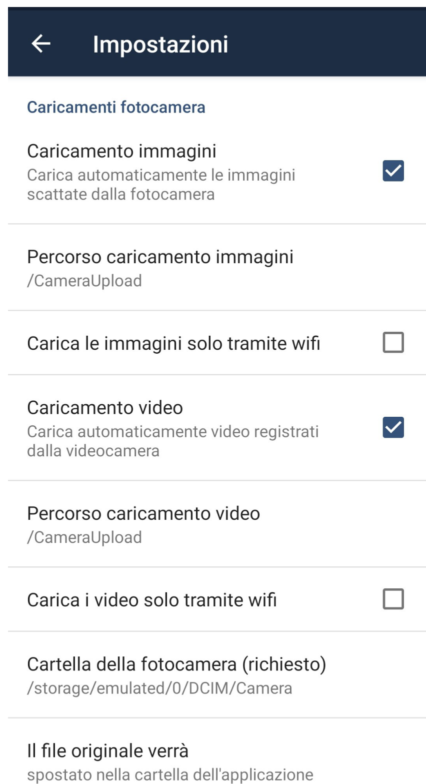
Riguardo la Figura 4.1 la prima riga ci permette di chiamare maven, il comando clean viene usato per eliminare tutte le precedenti risorse in modo che il progetto possa essere ogni volta lanciato come fosse la prima. Package permette di trasformare i codici con estenzione .java in file con estenzione .jar che vengono poi inseriti nella directory target del progetto.

Infine sia la pacchettizazione dell'applicazione dell'*in-vivo* server che l'implementazione di Apache Maven erano già state effettuate quando ho iniziate a lavorare a tale progetto.

4.2 Recupero informazioni configurazione

In questo paragrafo verranno introdotte le problematiche che si sono presentate per prelevare l'ultima configurazione dell'applicazione ownCloud e le soluzioni che sono state trovate.

Una configurazione di ownCloud sarà formata dalla versione del sistema operativo attualmente installato sul device, dal nome di quest'ultimo e dalle preferenze che



(a) Impostazioni preferenze file.

(b) Impostazioni sicurezza.

Figura 4.2: Impostazioni dell'applicazione ownCloud.

sono state selezionate dall'utente nelle impostazioni. Le impostazioni di ownCloud di nostro interesse si dividono in due categorie: quelle che riguardano le preferenze sulla gestione dei file e quelle riguardanti la sicurezza, come è possibile notare nella Figura 4.2.

Dobbiamo distinguere le impostazioni che sono rilevanti da quelle che non lo sono, per decidere quali devono essere prelevate. Riguardo le impostazioni sui file, possiamo notare che quelle che sono di nostro interesse e che faranno poi parte del *feature model* di ownCloud, sono quelle riguardanti il caricamento automatico delle immagini e dei video e le impostazioni su che tipo di connessione usare per farlo. Tra quelle che invece, al fine del nostro lavoro, non ci sono molto utili ci sono quelle riguardante la posizione di dove verranno salvati questi file localmente. Questo perchè, a differenza delle altre opzioni, non ci viene data una scelta limitata. Per

essere più chiari, il percorso su dove salvare la cartella della fotocamera può avere un numero enorme di differenti alternative. Inoltre può differire per ogni singolo utente rendendo molto complicato inserire le scelte che un utente può effettuare sulla preferenza di quell'opzione all'interno del *feature model*.

Riguardo, invece, alle impostazioni della protezione, queste sono tutte rilevanti. Possiamo inoltre notare che alcune voci possono essere selezionate solo se altre sono state precedentemente selezionate. Tra queste c'è lo sblocco con l'impronta digitale che può essere attivato solo se viene prima seleziona lo sblocco con codice o sequenza. Questo è un chiaro esempio di vincolo legato al *feature model* di un applicazione. Altri sono presenti anche nelle impostazioni riguardanti i file, ad esempio, la scelta di caricare immagini solo attraverso il wi-fi può essere selezionata solo se quella del caricamento immagini è già stata spuntata e lo stesso vale anche per le voci relative ai video. Tutti questi vincoli verrano poi riportati quando verrà creato il *feature model* di ownCloud.

Per riuscire a prelevare queste impostazioni con le relative preferenze è stata modificata la classe Preferences che si trovava già all'interno del progetto di ownCloud. L'idea è stata quella di prelevare le impostazioni con le relative preferenze e salvarle su un file in locale (vedere Figura 4.3).

Per farlo, il file dove viene salvata la configurazione viene creato per la prima volta quando l'utente entra e successivamente esce dalla sezione relativa alle impostazioni indifferentemente se siano state apporte modifiche o meno. Inoltre per verificare che sia la prima volta che viene prelevata la configurazione viene effettuato un controllo se esiste o meno un file dove è salvata una configurazione (vedere Figura 4.4). Nel caso non venga apportata nessuna modifica alle impostazioni, queste vengono scritte su file con i valori di default, come si può notare nella Figura 4.5, dato che l'utente non ha ancora selezionato le sue preferenze. In seguito se si entrerà nella sezione relativa alle impostazioni, prima di salvare la configurazione su file verrà eseguito un controllo tra le impostazioni correnti e quelle che erano state precedentemente salvate per assicurarsi che ci siano delle differenze (vedere Figura 4.4). Nel caso queste non ci fossero non viene creato nessun nuovo file.

Per riuscire a prelevare le scelte dell'utente relative alle impostazioni sono state usate le **SharedPreferences**[3]. Queste ultime, messe a disposizione dal sistema

[3] `https://developer.android.com/reference/android/content/SharedPreferences.html`

```
public class Preferences{
...

    public void PreferencesWriter(String type){

        //Create Map for save settings preferences
        Map<String,?> keys = getAppPreferences();

        //Create file for write map
        File file = null;

        switch (type) {

            case "FirstFile":
                file = new File(getFirstFileName());
                break;

            case "NotFirstFile":
                file = new File(getFileName());
                break;
        }

        Map<String, String> featureMap =
                convertToFeatureModelMap(keys);

        //Write Map to a file
        for (Map.Entry<String, ?> entry :
                featureMap.entrySet()) {
            appendStringToFile(file, entry.getKey() + "="
                    + entry.getValue().toString().trim()
                    + ";");

...

}
```

Figura 4.3: Metodo che permette di scrivere la configurazione su file.

```
public class Preferences {
...

    protected void onStop() {
    ...
        //Check if a configuration file already exist
        if(!fileExist(getFirstFileName())){
                PreferencesWriter("FirstFile");
        }
        //Check if current configuration is equals
        //to last file configuration
        else if(!isEqualsMap()) {
                PreferencesWriter("NotFirstFile");
        }
    }
...

}
```

Figura 4.4: Metodo per controllare se la configurazione corrente deve essere salvata su file.

Android, sono il metodo più comune utilizzato per salvare le impostazioni utente. Queste vengono usate per avere accesso alle informazioni non sensibili, questo perchè dovrebbero essere accessibili solo dall'applicazione ma ci potrebbero essere dei problemi di sicurezza che possono renderle accessibili anche al di fuori della stessa. Un problema, relativo a quanto detto, è stato riscontrato quando sono state prelevate le informazioni dalle impostazioni di ownCloud, se avessimo selezionato lo sblocco con il pin o con la sequenza, i dati immessi sarebbero stati salvati nelle SharedPreferences andando a creare un problema di sicurezza.

Le **SharedPreferences** tramite il metodo **getAll** permettono di accedere alle preferenze tramite una mappa java. Questa è una struttura dati formata da coppie chiave-valore in cui è possibile salvare qualsiasi tipo di dato. È proprio così che si riesce a confrontare le impostazioni correnti e quelle precedentemente salvate su file, queste infatti vengono prelevate dal file per poi essere convertite in una mappa che in seguito verrà confrontata con quelle delle impostazioni correnti.

Tuttavia alcune delle informazioni che venivano prelevate dalle SharedPreferences

```
preferences_pattern_lock=preferences_pattern_lock_false;
DeviceModel=Galaxy S10e;
preferences_video_uploads=preferences_video_uploads_false;
preferences_pincode_lock=preferences_pincode_lock_false;
preferences_picture_uploads=preferences_picture_uploads_false;
Android=9;
```

Figura 4.5: Configurazione di default.

erano superflue, come nel caso si decidesse di usare il pin per effettuare l'accesso questo veniva riportato nella mappa prima citata. Inoltre il *feature model* di ownCloud è stato scritto prima di effettuare il lavoro di prelevare la configurazione, così le chiavi che venivano prelevate avessero un valore diverso rispetto quelle presenti nel *feature model*. Quindi ho scritto un metodo a cui veniva inviata la mappa delle **SharedPreferences** con tutte le informazioni, dopodichè la mappa è stata esplorata e sono state prelevate solo le impostazioni rilevanti con le relative preferenze dell'utente. Inoltre tale metodo si occupava anche di modificare la chiave che veniva prelevata dalle impostazioni in quella che poi sarebbe dovuta essere confrontata con quella presente nel *feature model* di ownCloud. Per maggiore chiarezza ci tengo a precisare che la conversione del valore delle chiavi per far si che fossero uguali a quelle presenti nel *feature model* è stata una scelta dettata dalla prematura stesura di quest'ultimo. Un'altra soluzione che si sarebbe potuta adottare era quella di cambiare i valori delle feature presenti nel *feature model*, tuttavia non avrebbe portato nessun vantaggio particolare dato che avrei dovuto scrivere in ogni caso tale metodo per prelevare solamente le informazioni che mi sarebbero servite.

Un'altro problema che si è presentato è stato quello di prelevare la versione del sistema operativo Android del device attualmente in uso, dato che le SharedPreferences non sono in grado di accedere a tale informazione, ed aggiungerla alla configurazione attuale. Risolvere questo problema è stato semplice, dato che tramite il metodo **RELEASE** della classe **Build.VERSION** [4], messa a disposizione del sistema Android, è possibile accedere a tale informazione.

Un problema anagolo è stato prelevare il nome del device attualmente in uso.

[4]https://developer.android.com/reference/android/os/Build.VERSION

Risolvere tale problema è stato però un po' più complicato, dato che non c'era nessuna funzione che potesse accedere al nome commerciale dei device, ma solo a quelli di fabbrica. Più specificamente, nel caso di un Samsung Galaxy s7 saremmo potuti accedere solo al suo nome commerciale che è SM-G930F. Dato che il feature model è stato scritto usufruendo dei nomi commerciali dei device per non complicare la sua stesura, per risolvere tale problema mi sono avvalso di una liberia chiamata AndroidDevicesName [5] che permette di ottenere il nome commerciale del device a partire dal nome di fabbrica.

Risolti tutti i problemi prima citati si è riusciti ad arrivare all'obiettivo di prelevare una configurazione completa di ownCloud che potesse essere utilizzata successivamente dal client per poter, tramite server, stabilire di che tipo fosse.

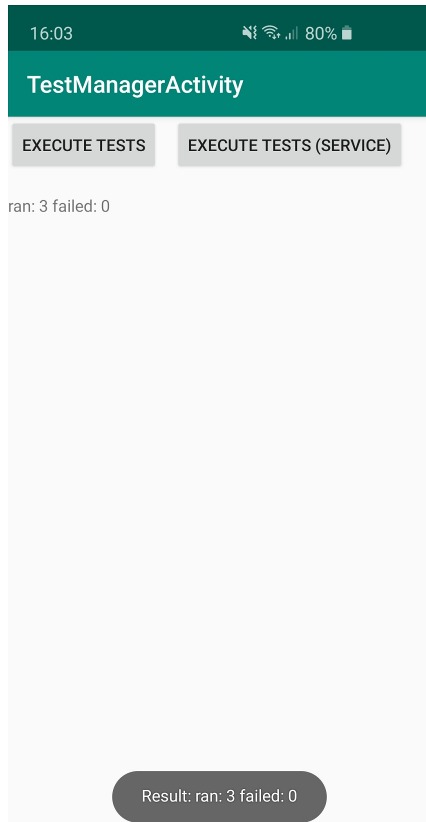
4.3 Implementazione client e lavoro sul server

Una volta aver sviluppato un meccanismo che ci permettesse di prelevare l'attuale configurazione dell'applicazione di ownCloud, c'era bisogno di usarla per poterla classificare. Questo lavoro viene svolto dall'applicazione del ClientService relativa ad ownCloud.

Tale applicativo si divide in due parti, il `Configuration Manager` e il `Test Manager`. La prima di queste per prelevare l'ultima configurazione si avvale, attraverso un intent, di una classe apposita che ha il compito di leggere l'ultimo file scritto riguardante le preferenze che sono state scelte dall'utente sotto forma di una stringa Java. Una volta in possesso della corrente configurazione sotto forma di stringa questa viene mostrata a schermo. Dopodichè il `Configuration Manager` procede con il suo invio all'*in-vivo* server in attesa di una risposta. A seconda del risultato ottenuto ci saranno comportamenti diversi:

- **Configurazione tested:** non verrà eseguita nessuna azione.
- **Configurazione untested:** dovranno essere eseguiti dei test, quindi c'è bisogno dell'intervento del `Test Manager` che verrà invocato all'instante attraverso un intent. Questo attraverso la pressione di un bottone permette di chiamare, sempre attraverso un intent, una classe presente all'interno di ownCloud che ci

[5] `https://github.com/jaredrummler/AndroidDeviceNames`



(a) Test Manager.

(b) Classe di ownCloud per l'esecuzione dei test.

Figura 4.6

permette di eseguire dei test (vedere Figura 4.6). Ovviamente tale classe è stata aggiunta successivamente e non faceva parte dell'applicazione prima che venisse modificata. Inoltre i test eseguiti non andavano a testare ownCloud, erano dei test di prova che sono stati sfruttati per verificare il corretto funzionamento del `Test Manager`.

Questo è stato fatto per dimostrare il funzionamento del framework *in-vivo*. Infatti, dato che tale framework dovrebbe essere usato dai programmatori in fase di sviluppo di un applicazione, questi potrebbero creare dei test appositi. Nel caso poi venisse trovato un bug, uno o più test potrebbero essere aggiunti in qualsiasi momento per cercare di avere più informazioni sull'errore avvenuto. In seguito vengono visualizzati a schermo i risultati dei test ed in questo caso

quanti test sono stati eseguiti con successo e quali invece hanno riportato degli errori.

- **Configurazione unknown:** in questo caso tutto il lavoro verrà lasciato all'*in-vivo* server, che dovrà provvedere a notificare che è stata trovata questo tipo di configurazione e aspettare l'intervento umano per poter aggiornare il *feature model* di ownCloud.

Nel verificare il corretto funzionamento dell' *in-vivo* server sono stati scoperti due errori. Prima di passare alla presentazione di tali problemi è doveroso capire come questa componente riesce a classificare le configurazioni che gli vengono inviate.

Il client invia, tramite una richiesta POST, la stringa che rappresenta la configurazione corrente. Tale stringa ha una precisa forma, la quale se non viene rispettata può provocare un errore quando questa viene processata. La forma in questione è:

fname1=fvalue1,fvalue2;fname2=fvalue1; ...; fname-n=fvalue-n; (4.1)

Fname indica il nome della feature, mentre *fvalue* il nome dei valori che tale feature può assumere. Come si può notare una feature può avere più di un valore ad essa associato: questi devono essere separati da una virgola, mentre le feature devono essere separate da un punto e virgola. Tale formattazione è molto importante per l'*in-vivo* server, dato che come prima operazione si occupa di trasformare la stringa passata in una mappa che chiameremo *ConfigurationMap* e che è del tipo seguente: Map<String, List<String>>. Questo ha però bisogno di due file fondamentali, che sono il *feature model* e il file in cui vengono salvate le configurazione che sono già state scoperte e testate.

Grazie all'uso di FAMILIAR e all'interazione del `Configuration Manager` e dello `Storage Manager`, il *feature model* viene letto e viene creata una lista, la *featureList*, dove sono presenti le chiavi delle feature che ci serviranno in seguito per effettuare diversi controlli. L'altro file serve per inizializzare una variabile, *testedTree*, che rappresenta un albero, il quale servirà anch'esso per effettuare dei controlli che ci permetteranno di arrivare a classificare la configurazione corrente. Dopo essere arrivati a questo punto l'*in-vivo* server deve controllare il tipo di configurazione

passata, per farlo si avvale della *featureList*, prima citata, contenente le chiavi delle feature e della *ConfigurationMap* che rappresenta la configurazione passata.

Tramite il metodo **getOrderedFeatures** controlla che nella *featureList* siano presenti le feature che ci sono all'interno della *ConfigurationMap*: se è questo il caso i valori relativi a tali feature vengono aggiunti ad un'altra lista che chiameremo *OrderedList*, altrimenti viene aggiunta un stringa vuota. Nel primo caso ci ritroveremo una lista che è formata dai valori delle feature che erano presenti nella configurazione e che facevano parte del feature model.

Dopodichè vengono effettuati dei controlli sull'*OrderedList* per poter verificare di che tipo è la configurazione. Il primo controllo che viene effettuato è sulla giusta formattazione delle stringhe, nel caso queste non siano corrette viene restituito come risultato *unknown*.

Successivamente viene effettuato un controllo sul *testedTree* prima citato: nel caso fosse vuoto significherebbe che non è stata ancora trovata nessuna configurazione e quindi qualsiasi fosse la configurazione passata in precedenza, questa sarebbe stata certamente *untested*.

Se nessuno dei precedenti controlli ha prodotto un risultato, allora viene cercata nel *testedTree* la lista relativa alla configurazione corrente, ovvero l'*OrderedList*. Nel caso tale controllo produca un risultato positivo la configurazione in esame viene classificata come *tested*, altrimenti questa è di tipo *untested*. Infine se la configurazione è di tipo *untested* allora questa viene aggiunta all'albero delle configurazioni testate nonchè al relativo file.

L'*in-vivo* server presentava però due errori che ne compromettevano il funzionamento. Infatti non era in grado di riconoscere tutte le configurazione di tipo *unknown* e inoltre restituiva come risultato *untested* tutte le volte che gli veniva passata una configurazione, indifferentemente se questa fosse già stata trovata e testata o meno. Quindi i problemi che presentava tale componente erano due:

- **Configurazione classificata come *untested* quando *tested*:** Questo problema era legato a quando il *feature model* veniva caricato dal file e salvato in una lista che prima abbiamo chiamato *featureList*. Il problema consisteva nel fatto che ogni volta che veniva prelevato il *feature model* dal file, le feature venissero prelevate sempre in ordine diverso.

Questo non sembrava un problema, ma ricordando il processo per classificare una configurazione lo diventa. Infatti la *featureList* era fondamentale per la creazione della successiva lista, l'*OrderedList*, su cui venivano effettuati i controlli. Dato che il *feature model* veniva caricato ogni volta con le feature in posizione diversa, quando la *featureList* veniva controllata e confrontata con la *ConfigurationMap* per la creazione dell'*OrderedList*, questa aveva le feature sempre in posizioni diverse.

La posizione delle feature è rilevante perchè quando l'*OrderedList* veniva cercata all'interno del *testedTree*, data una stessa configurazione, se la posizione delle feature fosse stata diversa allora queste sarebbero risultate due configurazioni diverse e quindi la configurazione corrente sarebbe stata classificata come *untested*, anche se in realtà non era così (vedere Figura 4.7).

```
---preferences_picture_upload=preferences_picture_upload_false---
----preferences_picture_upload=preferences_picture_upload_false--
```

Figura 4.7: Rappresentazione di due *OrderedList*.

I trattini presenti nella Figura 4.7 rappresentano la stringa vuota che viene inserita nell'*OrderedList* quando non sono presenti nella *ConfigurationMap* tutte le feature che sono presenti nella *featureList*.

Per risolvere il problema c'era bisogno che una volta prelevato il feature model e creata la *featureList* questa venisse ordinata ogni volta allo stesso modo. Per fare questo è stato usato il metodo **sort** relativo alle collezioni di java che permetteva di ordinare gli elementi presenti nella *featureList* in modo che questi risultassero sempre nello stesso ordine, risolvendo così il problema.

- **Configurazione *unknown* non sempre riconosciuta:** Prima di vedere le cause di questo problema occorre ricordare la definizione di una configurazione di tipo *unknown*. Una configurazione è tale solo se è formata da feature che non sono presenti all'interno del *feature model* oppure i valori associati ad almeno una feature non sono presenti all'interno dello stesso. Il problema è legato a una mancanza di controlli riguardanti le configurazioni che dovessero essere classificate come *unknown*.

L'unico controllo che era presente era sulla formattazione degli elementi all'interno dell'*OrderedList*. Questo comportava che una configurazione venisse classificata come *unknown* solo se questa era formata da almeno una feature che avesse almeno uno dei valori non presente all'interno del feature model.

Se infatti avessimo inserito una configurazione che avesse avuto almeno una feature che non era presente all'interno del feature model, l'*in-vivo* server non sarebbe stato in grado di riconoscerla come *unknown*. Per risolvere tale problema è stato aggiunto un ulteriore controllo che verificasse che le feature presenti all'interno della *ConfigurationMap* fossero tutte presenti all'interno della *featureList*, se così non fosse stato allora la configurazione corrente sarebbe stata classificata come *unknown*.

4.4 Simulazione scenari

Una volta aver risolto tutti i problemi relativi al framework *in-vivo* sono stati simulati alcuni scenari per osservare il suo corretto funzionamento.

- **Generazione e classificazione della prima configurazione:** per farlo c'era bisogno di generare la prima configurazione che sarebbe stata poi inviata, da parte del client, all'*in-vivo* server per essere testata.

  È stata così aperta l'applicazione di ownCloud che è stata modificata per poter recuperare la configurazione corrente, dopodichè si è entrati all'interno della sezione relativa alle impostazioni. A seconda se tali impostazioni venissero modificate o meno la configurazione sarebbe stata generata e scritta su file. Una volta generata la prima configurazione di ownCloud è entrato in gioco il client. Questo si è occupato di recuperare l'ultima configurazione dal file che era stato precedentemente creato e di inviarla all'*in-vivo* server. Questo ha processato tale configurazione per poterla classificare, essendo questa la prima configurazione che è stata prelevata e inviata al server la risposta restituita è stata *untested*. Tale configurazione inoltre è stata aggiunta alle configurazioni testate, infatti una volta che il client ha ricevuto la risposta dall'*in-vivo* server, questo ha provveduto immediatamente a richiamare la parte relativa all'esecuzione dei test (vedere Figura 3.1).

```
preferences_pattern_lock=preferences_pattern_lock_false;
DeviceModel=Xiaomi Mi9;
preferences_video_uploads=preferences_video_uploads_false;
preferences_pincode_lock=preferences_pincode_lock_false;
preferences_picture_uploads=preferences_picture_uploads_false;
Android=9;
```

Figura 4.8: Configurazione di tipo *Unknown.*

Mediante la pressione di un pulsante (vedere Figura 4.6) è stato così possibile eseguire i test sulla configurazione in esame e venire a conoscenza dei risultati dei test che sono stati effettuati.

- **Generazione e classificazione di una configurazione diversa dalla precedente:** è stato quindi simulato lo scenario in cui l'utente cambia le preferenze dell'applicazione di ownCloud, andando a generare una nuova configurazione che necessariamente deve differire da quella precedente. A questo punto il client ha provveduto ad inviarla all'*in-vivo* server che, come in precedenza, ha restituito come risposta *untested.* Anche questa volta sono stati eseguiti i test ed è stato possibile consultarne i risultati.

- **Generazione e classificazione di una configurazione uguale alla prima:** in seguito si è simulato l'utente che ancora una volta cambia le preferenze producendo però una configurazione identica alla prima. Il client ancora, ha prelevato e inviato tale configurazione all'*in-vivo* server, che tramite la consultazione delle configurazioni già testate ha potuto constatare che la configurazione corrente è stata già scovata in precedenza, restituendo così come risultato *tested.*

- **Generazione e classificazione di una configurazione Unknown:** infine si è simulato lo scenario in cui un utente stia usando un dispositivo il cui modello non è presente all'interno del feature model (vedere Figura 4.8).

  Il client quindi ha prelevato tale configurazione e l'ha inviata al server. Questo nei vari controlli che ha effettuato si è accorto che in tale configurazione è

presente una feature che inceve non lo è all'interno del *feature model*, restituendo così come risposta che la configurazione è di tipo *unknown*.

Si ricorda inoltre che il framework *in-vivo* è uno strumento che ha il compito di aiutare gli sviluppatori nella risoluzione di errori che possono verificarsi durante l'uso di un'applicazione. Gli scenari simulati e descritti in precedenza vogliono mostrare come il framework *in-vivo* è in grado di classificare una configurazione che ricordiamo sarà molto utile agli sviluppatori se ci fosse un errore dell'applicazione, nel caso tale configurazione non sia stata già scovata e quindi di tipo *untested* oppure non presa nemmeno in considerazione e quindi di tipo *unknown*, che potrebbe essere la causa dell'errore in questione.

Conclusione

Nella presente tesi è stato presentato il framework *in-vivo*, il quale, nelle nostre intenzioni, ha le potenzialità di rilevare i bug che avvengono al cambiare delle configurazioni di un'applicazione.

Si ricorca che lo sviluppo di tale framework non è stato svolto da me. Quello di cui mi sono occupato è stato implementare le sue funzionalità nell'applicazione di ownCloud, di creare il *feature model* di quest'ultima per poi testare il corretto funzionamento del framework *in-vivo*. Inoltre mi sono occupato di risolvere alcuni problemi dell'*in-vivo* server che sono emersi durante l'attività di debugging.

Mediante il salvataggio delle configurazioni il framework rende possibile riuscire a capire se un errore che è avvenuto a runtime possa derivare dal cambiamento di specifiche impostazioni da parte dell'utente. Per questo il salvataggio delle configurazioni avviene ogni qualvolta l'utente apporta dei cambiamenti alle impostazioni.

L'applicazione deve integrare l'*in-vivo* framework, le sue funzionalità devono essere implementate dagli sviluppatori di un'applicazione come è stato illustrato per ownCloud, un software open-source che può essere installato su un server web e permette di creare un proprio cloud privato.

Oltre alla modifica di ogni singola applicazione per poter implementare le funzionalità di recupero delle configurazioni deve essere modificata e personalizzata anche la parte relativa al ClientService. Questo per ogni applicazione deve essere capace di recuperare le configurazioni che sono state salvate e comunicare con il server per poterle classificare. Per fare questo il server utilizza il *feature model* che permette di tenere traccia di tutte le possibili configurazioni di un'applicazione.

Bibliografia

[1] Google, 2019. URL `https://developer.android.com/about/dashboards`. Online; accessed Ottobre 31, 2019.

[2] Mariano Ceccato, Luca Gazzola, Fitsum Meshesha Kifetew, Leonardo Mariani, Matteo Orrù, and Paolo Tonella. Toward in-vivo testing of mobile applications. In *1st International Workshop on Goverving Adaptive and Unplanned System of Systems (GAUSS), co-located with the 30th International Symposium on Software Reliability Engineering (ISSRE), Berlin*, 2019.

[3] Open Signal. Android fragmentation visualized (august 2015). *Techn. Ber*, 2015.

[4] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 226–237. IEEE, 2016.

[5] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming (SCP)*, 78(6):657–681, 2013.

[6] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces.* " O'Reilly Media, Inc.", 2011.