Alma Mater Studiorum · Università di Bologna
Sede di Cesena

SECONDA FACOLT DI INGEGNERIA CON SEDE A CESENA
Corso di Laurea in Ingegneria Informatica

TuCSoN on Android

Tesi di Laurea in Sistemi Distribuiti

Relatore:
Chiar.mo Prof.
Andrea Omicini

Correlatore:
Dott. Ing.
Stefano Mariani

Presentata da:
Andrea Ridolfi

Secondo appello - Terza sessione
Anno Accademico 2011/2012

Sommario

TuCSoN (*Tuple Centres Spread over the Network*) è un modello di coordinazione per processi distribuiti o agenti autonomi. Il modello di TuCSoN viene implementato come un *middleware* distribuito Java-based, distribuito Open Source sotto la licenza LGPL tramite Googlecode.

Il fatto che lo stesso sia Open Source e Java-based ha reso possibile il suo porting su Android, rendendo il noto sistema operativo di Google un possibile agente partecipante ad un sistema TuCSoN.

La tesi descrive il percorso che ha portato dallo studio dell'infrastruttura TuCSoN e del sistema Android alla realizzazione dell'applicazione Android, rendendo possibile a qualsiasi dispositivo Android di partecipare ad un sistema TuCSoN. Nel particolare l'obiettivo finale dell'applicazione Android, e di questa tesi, è rendere lo smartphone un nodo TuCSoN funzionante[1].

La tesi non si pone l'obiettivo di analizzare ed esplorare le funzionalità e le possibilità delle due tecnologie principali trattate (Android e TuCSoN) nel loro singolo, quanto quello di esplorare le criticità che un porting di questo tipo comporta, quali ad esempio le differenze intrinseche fra la JVM e la DalvikVM e come aggirarle, o le funzionalità di Android e come utilizzarle allo scopo di realizzare un applicazione che funga da server ad una infrastruttura distribuita, oppure le differenze a livello di gestione della GUI fra Android e plain-java, e di analizzare le soluzioni trovate per risolvere (o dove non era possibile risolvere evitare) tali problemi al fine del raggiungimento dell'obiettivo che ci si era prefissati.

[1] Analizzeremo in seguito il significato di Nodo Tucson e la sua funzione all'interno dell'infrastruttura

0.1 Introduzione

Eretogeneità, indipendenza e autonomia sono gli aspetti chiave di una qualsiasi entità all'interno di un sistema distribuito.

Quando tali autonome entità cooperano in un unico sistema uniforme, abbiamo una serie di aspetti fondamentali da tenere in considerazione, quali ad esempio la coordinazione e la comunicazione fra di esse, l'accesso a dei dati che possono essere condivisi, il passaggio di informazioni e la codifica delle stesse ecc ecc.

Allo scopo di gestire tutti questi aspetti nascono i *middleware*.

Nel contesto delle applicazioni distribuite il *middleware* è il *software* che provvede alla comunicazione ed alla gestione dei dati del sistema distribuito, coordinando i vari agenti allo scopo di semplificare la progettazione di sistemi anche molto complessi.

Il *middleware* si occupa di risolvere i problemi relativi a sintassi, semantica, connessioni, accesso e modifica a dati condivisi, e tutte quelle problematiche che nascono nel momento stesso in cui si distribuisce topologicamente un sistema, offrendo ai vari componenti un'interfaccia con la quale comunicare; nascondendo al sistema complessivo la tecnologia, la struttura, il comportamento o altri aspetti che caratterizzano fortemente le singole entità eterogenee, ma che risultano tuttavia inutili relativi al corretto funzionamento del sistema nel suo complesso.

Nasce con questo Intento, all'interno dei laboratori di APICe, il progetto di un *middleware* distribuito Java-based, TuCSoN (*Tuple Centres Spread over the Network*).

Data la sempre maggiore diffusione degli smartphone Android, l'ormai celebre sistema operativo per smartphone e tablet firmato Google, e considerato il fatto che Android è Java-based, è nata spontanea l'idea di rendere TuCSoN utilizzabile anche su uno di questi dispositivi, realizzando un *porting* della piattaforma.

Tuttavia questo tentativo di *porting* si scontra con problemi che, alla creazione dell'infrastruttura, non erano stati considerati: la scarsità di risorse hardware disponibili su uno smartphone rispetto ad un comune PC, le differenze intrinseche fra la Java Virtual Machine e la Dalvik Virtual Machine, la mobilità dello smartphone, un sistema operativo completamente differente rispetto a quelli su cui abitualmente funziona la JVM e sulla quale di conseguenza è stato progettato TuCSoN, e tutta una serie di caratteristiche per cui uno smartphone è ben diverso da un comune PC.

La tesi si pone l'obiettivo di indagare queste differenze e di garantire una piena funzionalità di un nodo TuCSoN su un dispostivo Android, modificando ed ottimizzando laddove necessario TuCSoN stesso, allo scopo di garantire tutti quegli aspetti per cui il *middleware* stesso è stato concepito.

Viene in questo documento trattato ed approfondito il percorso che ha portato dalla nascita dell'idea del *porting* alla sua realizzazione, cercando di

restare il più fedeli possibile a quello che effettivamente è stato l'evolversi del progetto.

A questo scopo la tesi viene divisa in più parti:

La prima parte è un riassunto della fase di studio ed approfondimento preliminare, parte che ha richiesto probabilmente la parte maggiore di tempo. Essa riguarda lo studio dell'infrastruttura TuCSoN ed in particolare del funzionamento del sistema operativo Android e della sua architettura interna.

La seconda parte analizza i problemi riguardanti al *porting* in se, studiando in che modo si può rendere un dispositivo Android un server vero e proprio, quali problemi si sono incontrati e come sono stati risolti. Vengono inoltre approfondite tematiche riguardo la realizzazione di altri componenti oltre al nodo stesso, quali il CLI[2] o il LogCat[3].

La terza parte infine descrive l'applicazione risultato di questa ricerca. Vengono descritte le riflessioni finali riguardo al lavoro svolto, le cose che possono essere migliorate, e degli spunti dai quali partire per eventuali lavori futuri.

[2]Command Line Interpreter, un interprete di comandi TuCSoN per scrivere all'interno dello spazio di memoria condiviso base del funzionamento dell'intera infrastruttura TuCSoN. Verrà meglio trattato nelle sezioni seguenti.

[3]Il meccanismo Android che si occupa di mostrare all'utente output di sistema per il debug

Indice

Parte I

Le tecnologie utilizzate

Capitolo 1

TuCSoN

Questo capitolo ha lo scopo di presentare, in maniera quanto più sintetica e chiara possibile, come si presenta e cosa è l'infrastruttura TuCSoN creata nei laboratori di APICe dell'università di Bologna.

Verrà trattato il modello di base dell'infrastruttura, le sue componenti, nonché qualche dettaglio tecnico.

Verrà inoltre approfondito il concetto di nodo TuCSoN e la sua funzione, essendo la parte centrale inerente alla tesi.

1.1 Aspetti base di TuCSoN

1.1.1 Modello base

TuCSoN[1], come già stato menzionato nell'introduzione, è un modello per la coordinazione di processi distribuiti, così come di agenti autonomi.

L'intera infrastruttura TuCSoN può essere sintetizzata in tre entità di base, le quali sono:

- Agenti TuCSoN, i quali sono le entità base da coordinare all'interno del sistema distribuito;
- Centri di Tuple ReSpecT, che sono il mezzo sul quale avviene la coordinazione;
- Nodi TuCSoN, che rappresentano l'astrazione topologica di base, sul quale sono ospitati i centri di tuple.

In sintesi un sistema TuCSoN è un insieme di agenti e centri di tuple i quali lavorano insieme, in maniera coordinata, in un insieme di nodi, possibilmente distribuiti.

L'interazione e la comunicazione all'interno del modello TuCSoN vede protagonisti le due entità di base prima descritte, gli agenti e i centri di tuple.

Gli agenti ricoprono il ruolo di entità attive, avendo la possibilità di sfruttare i centri di tuple per comunicare. Questa comunicazione avviene per gli agenti tramite il linguaggio di coordinazione TuCSoN, dato da un insieme di primitive le quali permettono lo scambio, la scrittura o la lettura di tuple all'interno di un qualsiasi centro di tuple.

I centri di tuple hanno invece il ruolo di entità passive, fornendo agli agenti lo spazio condiviso sul quale questi ultimi possono comunicare per mezzo delle tuple. Il centro di tuple è inoltre caratterizzato da un comportamento reattivo: sebbene infatti quest'ultimo sia passivo, non avendo lo scopo di comunicare con altri agenti, può tuttavia essere programmato dinamicamente per reagire al maneggiamento di tuple al suo interno, fornendo in questo modo un modello base di coordinazione per gli agenti, allo scopo di raffinare e personalizzare il funzionamento di un centro di tuple.

La topologia di un sistema TuCSoN appare quindi subito molto chiara: si tratta di un modello client-server, dove i client sono gli agenti tucson e i server sono i nodi TuCSoN, sui quali troviamo i centri di tuple.

1.1.2 Naming

Essendo sia gli agenti, sia i centri di tuple distribuiti in rete, appare subito fondamentale definire una ferrea sintassi per i nomi.

[1] Guardare sezione Bibliografia per riferimenti al manuale ed al sito ufficiale di TuCSoN.

Ogni centro di tuple viene definito tramite il nome del centro di tuple associato all'identificatore univoco del nodo sul quale è presente, rendendo in questo modo univoco in rete ogni singolo centro di tuple, anche se omonimo con un altro centro di tuple presente su un nodo differente, seguendo la sintassi generale:

```
tname @ netid :  portno
```

dove `tname` è il nome del centro di tuple, `netid` è l'indirizzo ipv4 del nodo e `portno` è la porta sulla quale il nodo TuCSoN è in ascolto.

In maniera del tutto simile ogni agente TuCSoN deve possedere un identificatore univoco all'interno della rete: questo viene realizzato dall'insieme del nome dell'agente, definito dall'utente[2], ed un identificatore univoco fornito automaticamente dal sistema TuCSoN nel momento in cui l'agente ne fa per la prima volta l'accesso.

La sintassi dell'identificatore appena descritto è la seguente:

```
aname :  uuid
```

Dove `aname` è il nome definito dall'utente e `uuid` è il codice identificativo dato dal sistema TuCSoN.

1.1.3 La comunicazione

Come già accennato la comunicazione di base in un sistema TuCSoN avviene mediante le primitive definite dal linguaggio di coordinazione TuCSoN, le quali consentono agli agenti di scrivere, leggere o rimuovere tuple all'interno di un centro di tuple, allo scopo di comunicare e sincronizzarsi.

Una qualsiasi operazione TuCSoN viene fatta da un agente verso un centro di tuple, che ha il compito di eseguirla.

L'operazione ha due fasi: l'*invocation*, e cioè la richiesta fatta dall'agente al centro di tuple, contentente tutte le informazioni necessarie all'esecuzione dell'operazione stessa, e la *completion*, la quale è semplicemente il risultato ottenuto dall'operazione all'interno del centro di tuple, nonché le informazioni relative all'esecuzione della stessa.

Per le primitive di base di TuCSoN si consiglia di fare riferimento alla documentazione ufficiale di TuCSoN[1].

[2] L'unica restrizione riguardo al nome fornito dall'utente è che non abbia spazi né caratteri speciali, e che non contenga lettere maiuscole.

1.2 Architettura di TuCSoN

Ricordiamo che l'obiettivo finale della tesi è il *porting* del nodo TuCSoN su Android: a questo scopo verrà approfondito in questa sezione il funzionamento del nodo, le componenti che ne fanno parte e gli aspetti che avranno poi importanza una volta messe di fronte alle esigenze del sistema operativo Android.

Verranno perciò approfondite in maniera minore tematiche quali ReSpecT e tuProlog, o riguardo al funzionamento dell'agente, non per la loro minore importanza in senso assoluto all'interno del modello, in quanto anch'essi aspetti chiave per il corretto funzionamento di TuCSoN, ma in quanto non al pari rilevanti riguardo all'obiettivo finale di questo percorso.

Si consiglia pertanto ancora una volta qualora interessati di far riferimento alla documentazione ufficiale per eventuali autonomi approfondimenti riguardo alle tematiche non presenti nel seguente documento.

1.2.1 Il Nodo TuCSoN

Il nodo è la parte principale di in un sistema TuCSoN che più si avvicina a quello che viene comunemente definito server all'interno di un sistema server-client.

Esso infatti viene fortemente caratterizzato dal device fisico che lo ospita, essendo definito all'interno del sistema tramite l'indirizzo ip del device stesso e dalla porta[3] sulla quale il nodo si mette in ascolto delle richieste degli agenti facenti parte del sistema TuCSoN. È inoltre proprietario dello spazio di memoria condiviso dagli agenti, ha infatti il compito di costruire i centri di tuple, che come già detto sono il mezzo tramite il quale viene garantita coordinazione agli agenti.

In sintesi il nodo TuCSoN ha il compito di mettersi in ascolto, sulla porta specificata, di eventuali richieste, passarle al centro di tuple bersaglio (creandolo qualora non esistente) e di fornire all'agente richiedente il risultato del'operazione eseguita.

In linea di principio deve essere infatti possibile eseguire una qualsiasi operazione primitiva TuCSoN da un qualsiasi agente TuCSoN su un qualsiasi centro di tuple fornito da un qualsiasi Nodo TuCSoN, ed il nodo ha il compito esclusivo di assicurarsi che ciò sia possibile.

A questo scopo ogni nodo ha un centro di tuple di default (chiamato per l'appunto default) sul quale vengono eseguite quelle operazioni che non hanno specificato un determinato centro di tuple[4].

[3] La porta di Default sulla quale opera TuCSoN è la numero 20504.

[4] Al contrario non viene definito alcun tipo di nodo di default. Se l'operazione viene eseguita su un nodo non attivo, non raggiungibile o di sintassi sbagliata viene generata un'eccezione.

1.2.2 TuCSoN e tuProlog

TuCSoN, come già specificato, è un *middleware* Java-based.

Tuttavia esso è anche Prolog-based: si appoggia a tuProlog (anch'esso Java-based) per vari scopi, quali ad esempio la definizione delle tuple e dei loro template, le operazioni di parsing di primitive e identificatori, o per la definizione del linguaggio ReSpecT.

TuProlog è un sistema Prolog Java-based studiato appositamente per applicazioni e infrastrutture distribuite, disegnato con l'idea di snellire e comprendere solo le specifiche essenziali di Prolog per essere poi configurato a runtime, tramite caricamento di librerie on-demand.

Nella versione standard di TuCSoN[5], all'interno del pacchetto .jar scaricabile, è già presente tuProlog, non essendo quindi necessario scaricarlo a parte, tuttavia nel nostro progetto è stato necessario scaricarlo e ricompilarlo individualmente, per motivi che tratteremo in seguito.

1.2.3 Role-Based Access Control (RBAC) e Agent Coordination Context (ACC) in TuCSoN

Il modello RBAC (standard NIST[6]) è un modello che integra organizzazione, coordinazione e sicurezza all'interno di un sistema; si basa sulla definizione di ruoli assegnati ai processi (gli agenti TuCSoN nel nostro caso) in base all'organizzazione alla quale appartengono all'interno del sistema, in modo da controllare come e a quali servizi del sistema possono accedere.

Questo modello viene utilizzato all'interno del sistema sia per garantire sicurezza, definendo ad esempio ruoli che non possono accedere a determinate organizzazioni, o che non possono compiere determinate operazioni, sia per garantire coordinazione fra i processi stessi, tramite un processo di suddivisione dei ruoli.

I centri di tuple in TuCSoN possono essere strutturati come organizzazioni, rendendo quindi possibile implementare il modello RBAC all'interno del modello TuCSoN; nasce così in TuCSoN il concetto di ACC. Quest'ultimo consiste in un'interfaccia assegnata ad un agente che gli consente di effettuare operazioni su determinati centri di tuple facenti parte di una specifica organizzazione, differente a seconda dell'ACC assegnatogli.

Questa interfaccia, fornita dall'infrastruttura, ha il duplice scopo di fornire un mezzo tramite il quale comunicare con il sistema, sia di limitare la comunicazione stessa alle organizzazioni specifiche. L'ACC viene inoltre utilizzato in TuCSoN anche per garantire coordinazione fra gli agenti e il centro di tuple stesso, che ricordiamo è il mezzo di comunicazione sul quale si basa l'intero sistema TuCSoN.

[5] Quella che si può scaricare liberamente dalla directory googlecode ufficiale di TuCSoN[2].

[6] http://csrc.nist.gov/groups/SNS/rbac/

1.2.4 Architettura del Nodo TuCSoN

Il nodo TuCSoN, o richiamando il nome della classe Java il `TucsonNodeService`, può essere diviso in varie componenti:

- L'`ACCProvider`;
- Il `WelcomeAgent`;
- Il `NodeManagementAgent`;
- L'`ACCProxyNodeSide`.

L'insieme di questi componenti ha lo scopo di garantire tutte le funzionalità di un Nodo TuCSoN, ricoprendo un ruolo fondamentale in quello che in seguito sarà la parte centrale del progetto.

Il funzionamento del nodo è relativamente semplice. Viene lanciato tramite la classe `TucsonNodeService` utilizzando uno dei suoi costruttori, i quali danno la possibilità di scegliere su quale porta il nodo debba essere lanciato, oppure di caricare al lancio un file di configurazione ReSpecT per personalizzare il comportamento del nodo[7] e chiamando conseguentemente il metodo `install`, il quale ha il compito di caricare i file di configurazione e far partire individualmente i vari componenti prima elencati.

Il `WelcomeAgent` ha il compito di mettersi in ascolto di nuove richieste effettuate al nodo TuCSoN sulla porta da lui indicatagli. Nel momento in cui riceve una richiesta entra in gioco l'`ACCProvider`, il quale ha il compito di creare l'`ACCProxyNodeSide` e di fornire l'ACC all'agente richiedente.

L'`ACCProxyNodeSide` si occuperà di tutte le comunicazioni fra l'agente e il cenro di tuple. Esiste un rapporto biunivoco all'interno di TuCSoN fra numero di agenti e `ACCProxyNodeSide`: per ogni agente connesso ad un nodo, il nodo crea il Proxy incaricato di servire le sue richieste. Lato agente, in maniera molto simile, viene creato un `ACCAgentAgentSide`, che ha uno scopo molto simile: la comunicazione fra agente e centro di tuple non avviene mai quindi direttamente, bensì viene mediata dai due Proxy.

Il `WelcomeAgent` ha quindi il solo compito di ricevere la prima connessione di un agente, in quanto gli agenti già connessi possiedono il riferimento al proprio `ACCProxyNodeSide`.

Infine il `NodeManagementAgent` rappresenta l'agente il quale ha il compito effettivo di effettuare le operazioni richieste dagli agenti sui centri di tuple del nodo.

[7] In caso non venga fornito un file personalizzato TuCSoN possiede un file di configurazione di default. Questa funzionalità non è ancora supportata nella release finale di Android, nella quale viene sempre caricato il file di default.

Capitolo 2

Android

Molteplici sono gli aspetti e le sfaccettature dell'ormai noto sistema operativo firmato Google, e difficile è sintetizzare e descrivere la sua complessità all'interno di una tesi di questo tipo, data sia la quantità di materiale riguardante tale argomento, sia la diversità di livelli a cui ci si può soffermare.

Verrà in questo capitolo descritta sinteticamente l'architettura di un sistema Android e le sue caratteristiche, soffermandosi principalmente sul lato di sviluppo delle app, nonché sui concetti basilari che verranno affrontati durante il *porting* di TuCSoN vero e proprio su questa piattaforma; verrano pertanto approfonditi concetti quali service, activity e fragment, verrà analizzata l'organizzazione della GUI in android ed il funzionamento della Dalvik VM, inoltre verranno presentati i principali tool di sviluppo supportati da Android ed il loro utilizzo.

2.0.5 La nascita di Android

Android nasce nel 2003 in California, da Andy Rubin, Rich Miner e Chris White, i quali si uniscono con lo scopo di creare dei dispositivi più intelligenti di quelli che erano i cellulari secondo lo standard dell'epoca.

Lavorando segretamente ad un progetto riguardo a software per cellulari, nel 2005 Google ingloba Android Inc., lasciando tuttavia a capo del progetto Rubin, Miner e White.

Due anni più tardi, nel 2007, Google richiede la registrazione di vari brevetti riguardanti tecnologie software molto vicine alla tecnologia mobile, stringe inoltre accordi commerciali con i più grandi produttori e con le più grandi aziende operanti nel settore della telefonia mobile (HTC, Samsung e T-Mobile) e della produzione hardware (Qualcomm e Texas Instruments), alimentando i già presenti rumors riguardo un eventuale ingresso nel settore mobile della compagni di Mountain View.

Rumors che trovano veridicità nel 2008, quando Google svela il suo progetto presentando il primo smartphone Android, l'HTC Dream.

Il fatto che Android sia distribuito in formato open-source, sotto licenza Apache, porta nei 5 anni a seguire ad una enorme diffusione della piattaforma; sebbene Android sia infatti nato per dispositivi mobili touchscreen, potendo essere liberamente distribuito e modificato ha esteso la sua diffusione ai più svariati dispositivi, quali ad esempio televisioni e console videoludiche.

È tuttavia nel mondo degli smartphone che Android detiene il record per il sistema operativo più diffuso, con più del 75% di market share, grazie anche al supporto fornito da Google alle migliaia di developers che vogliono cimentarsi nella creazione di app; applicazioni che estendono le funzionalità di un dispositivo Android, come quella che verrà realizzata alla fine di questa tesi.

2.0.6 Perché Android?

Perché si è quindi deciso di utilizzare Android per estendere le possibilità dell'infrastruttura TuCSoN? Gli aspetti da considerare che hanno portato questa scelta sono molteplici.

In primis l'enorme diffusione a livello mondiale di Android: all'autunno 2012 sono contati più di 500 milioni di dispositivi Android attivati nel mondo, con un ritmo di 1.3 milioni di attivazioni giornaliere di nuovi dispositivi, una diffusione che ha portato in breve Android a diventare un brand mondiale.

Va inoltre considerata la sua licenza Apache, comportando il fatto di essere completamente open-source, oltre ad una facile reperibilità delle API android e di ottimi tutorial e guide reperibili online che introducono agevolmente e rapidamente alla programmazione nel mondo Android[3].

Un'altro punto a favore di Android è la possibilità di poter scrivere le applicazioni utilizzando Eclipse, un IDE anch'esso open-source ufficialmente supportato da Google tramite plugin appositamente sviluppati, che rende semplice lo studio e lo sviluppo di una applicazione su questa piattaforma.

Uno degli aspetti più importanti inoltre risiede nel fatto che, nonostante l'intero sistema operativo sia basato su Linux, le applicazioni livello utente siano scritte primariamente in Java, rendendo possibile l'utilizzo della versione già esistente di TuCSoN senza, in certi casi, dover neppure modificare la release che viene comunemente utilizzata per PC[1].

Inoltre va considerata l'enorme mobilità data da uno smartphone, che apre scenari e possibilità uniche riguardo a quella che è la distribuzione di un sistema.

Ricollegandoci alla diffusione di Android e alla sua licenza Apache bisogna considerare la sua possibilità di girare su una quantità enorme ed eterogenea di dispositivi, siano essi tablet, smartphone, televisioni, console o addirittura automobili e apparecchiature domestiche, rendendo quindi virtualmente possibile interfacciare qualsiasi cosa ad un sistema TuCSoN.

Infine l'indubbia solidità dell'azienda Google, le quali rendono Android un sistema stabile, in continuo sviluppo e che difficilmente cadrà in disuso e declino a breve.

[1] Questo non è del tutto vero, come vedremo in seguito, Android necessita di qualche piccola modifica.

2.1 Architettura di Android

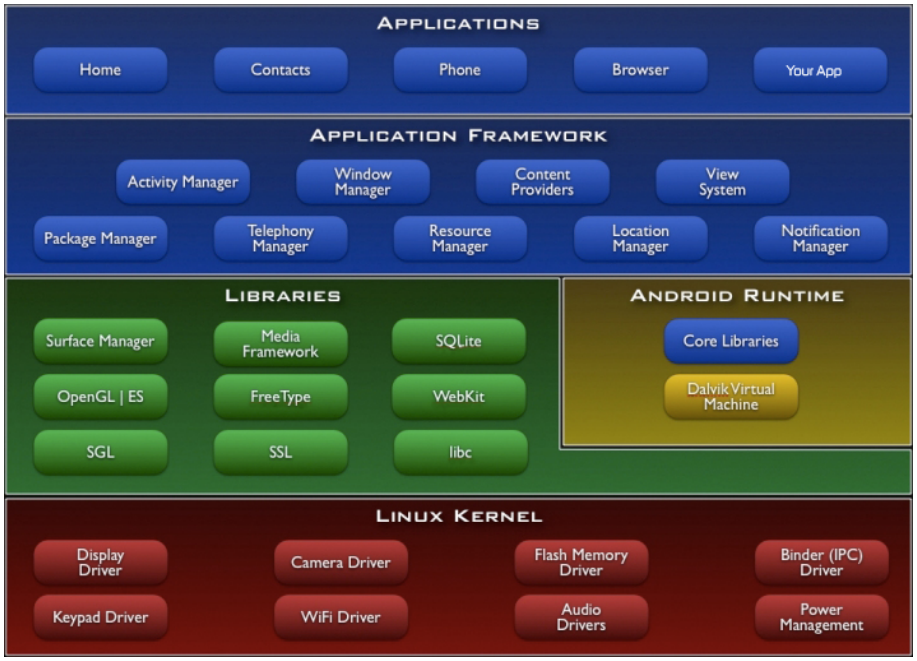

Figura 2.1: Architettura di Android

Android è un sistema operativo a livelli (*Layered Operating System*), in cui ogni livello è diviso e comunica solo con il sottostante tramite apposite interfacce. Ogni livello ha quindi il compito di servire il livello a lui superiore sfruttando quello a lui inferiore; genericamente in questo tipo di sistemi, ed è così che Android è strutturato, si pongono le applicazioni lato utente al livello più alto e le risorse di sistema al livello più basso.

Il principale vantaggio di un sistema operativo di questo tipo è la sua sicurezza, in quanto avendo ogni livello accesso univoco al sottostante viene limitata la quantità di codice avente il potere di accedere direttamente alle risorse di sistema, rendendo più sicuro e stabile il sistema stesso nel suo complesso.

Un sistema a livelli offre inoltre vari vantaggi a livello progettuale: rende ad esempio possibile la sostituzione di un intero livello senza compromettere il resto del sistema operativo, oppure semplifica il procedimento di progettazione, permettendo di partire dal livello inferiore fino a quello più alto garantendo un corretto funzionamento del sistema.

Un sistema di questo tipo ha tuttavia anche degli svantaggi, il più grande dei quali è una performance delle applicazioni peggiore di quello che si avrebbe in un sistema ad esempio monolitico, in quanto ogni applicazione

per accedere alle risorse hardware deve prima passare dai livelli sottostanti, non potendoci accedere direttamente.

In figura 2.1 possiamo vedere la divisione del sistema Android in quattro livelli principali: partendo dal livelo più basso il Kernel Linux, le librerie di sistema, il Framework delle applicazioni ed infine le applicazioni utente, al livello più alto.

Come si vede in figura Android si basa sul Linux Kernel versione 2.6, e a partire dalla versione di Android 4.0 *Ice Cream Sandwich* alla sua più recente versione 3.x, con middleware, librerie e API scritte in C, mentre le applicazioni operano su un framework che include librerie compatibili con il linguaggio Java basate su *Apache Harmony*[2], venendo quindi scritte in Java.

Android utilizza la Dalvik virtual machine al posto della classica JVM, dotata di Just-In-Time compilation (JIT) per eseguire Dalvix dex-code (gli eseguibili Dalvik), tradotti dal bytecode Java.

Vediamo ora in dettaglio i singoli livelli del sistema Android.

2.1.1 Linux Kernel

Il Kernel Linux di Android ha molte modifiche architetturali compiute da Google che lo differenziano molto dal tipico Kernel Linux operante su un qualsiasi sistema Unix; Android ad esempio non possiede nativamente un sistema X Window[3] né supporta le librerie standard GNU[4], rendendo difatti assai complicato il *porting* di applicazioni o librerie Linux esistenti in Android.

Alcune aggiunte effettuate da Google al kernel Linux, quali ad esempio la creazione del concetto di Wakelocks per la gestione dell'energia, sono state inizialmente rifiutate dal team di sviluppo del kernel, in quanti questi ultimi temevano che Google non fosse intenzionato a mantenere il codice. Tuttavia gli ultimi sviluppi hanno portato il team a ritornare sui propri passi, annunciando un inizio di collaborazione fra il team di Google e quello di Linux.

Linus Torvals stesso ha annunciato, nell'agosto 2011, la possibilità che in un futuro non troppo remoto i due kernel torneranno ad essere lo stesso.

Il kernel in Android ha comunque la funzione standard che possiede in un qualsiasi sistema Unix: contiene i driver per le varie risorse hardware e si occupa dell'accesso alle stesse, essendo al livello più basso; ha inoltre il compito di gestire tutte le funzionalità al cuore di un sistema operativo, quali

[2] Un'implementazione di Java open source, sviluppata da Apache Software Foundation

[3] Conosciuto comunemente come X11, è un sistema software ed un protocollo di rete indipendente dall'architettura hardware di un sistema, creato con lo scopo di fornire un set di comandi utilizzabili per creare programmi riutilizzabili su qualsiasi computer che implementi X

[4] Un sistema operativo Unix-Like composto unicamente da free-software

la gestione della memoria e dei processi, le comunicazioni di rete, policy di sicurezza riguardo all'accesso di dati in sezioni di memoria protette, etc.

Android eredita dalla quantità di sistemi hardware che possono supportare agevolmente un kernel Linux la sua semplicità nell'adottare un'ampia varietà di hardware, il che spiega (in parte) la sua enorme diffusione nella più variegata quantità di device.

2.1.2 Librerie di Android

Il livello successivo al kernel sono le librerie standard di Android. Queste librerie, scritte in linguaggio c o c++, permettono al device di gestire diversi tipi di dati e sono hardware-specific.

Alcune delle librerie più importanti presenti in un sistema Android sono le seguenti:

- Surface Manager: è utilizzato per la composizione di finestre con off-screen buffering. L'off-screen buffering significa che è possibile disegnare finestre direttamente sullo schermo, ma queste vanno nell'off-screen buffer, dove vengono combinate con altre finestre per formare la schermata finale che viene mostrata all'utente. Questo procedimento è alla base del concetto di popup e di transizione che spesso si ha nel normale utilizzo di un device Android.

- Media Framework: contiene vari codecs, permettendo la riproduzione, registrazione, lettura e scrittura di differente formati multimediali, quali fotografie, file audio o riprese video.

- SQLite: SQLite rappresenta il database utilizzato da Android per lo storage dei dati.

- WebKit: il motore browser utilizzato per la conversione di documenti HTML.

- OpenGL: Apic grafiche, usato per il rendering grafico di contenuti 2D o 3D.

2.1.3 Android Runtime

Sebbene non sia un livello vero e proprio, in questa parte risiede il cuore di un sistema Android: ne fanno infatti parte le librerie Java e la Dalvik Virtual Machine.

La Dalvik Virtual Machine utilizzata nei dispositivi Android ha il compito di eseguire le applicazioni, ed essendo sviluppata per dispositivi mobili è ottimizzata per dispositivi con scarsa potenza computazionale e scarsa memoria disponibile.

A differenza della JVM, la Dalvik non esegue file .class, bensì file .dex. I file .dex vengono generati dai file .class a runtime, ed hanno una maggiore efficienza in caso di scarse risorse disponibili.

La Dalvik VM inoltre permette la creazione di più istanze di macchine virtuali in grado di operare simultaneamente, portando sicurezza e isolamento qualora necessario, supportando il multi-threading e implementando politiche di gestione della memoria.

Le librerie Java, sebbene leggermente differenti dalle librerie Java SE e Java ME, possiedono quasi tutte le funzionalità definite nelle librerie Java SE.

2.1.4 Application Framework

Questo livello è quello con cui interagiscono tutte le applicazioni lato utente.

Ha il compito di gestire le funzioni base del telefono come gestione delle risorse, gestione delle chiamate etc.

Come sviluppatore, questi sono gli strumenti dati dal sistema operativo google coi quali costruire la propria applicazione.

Alcuni dei più importanti blocchi sono:

- Activity Manage: gestisce il ciclo di vita delle activity, cuore del funzionamento delle applicazioni Android.
- Content Providers: gestisce lo scambio di dati e informazioni fra le varie activity.
- Telephony Manage: gestisce le chiamate vocali. Viene utilizzato in una applicazione qualora si voglia accedere alla parte telefonica del device.
- Location Manager: gestisce la topologia del dispostivo, sfruttando il modulo GPS o la rete GSM/UMTS per definire la propria posizione.
- Resourse Manager: gestisce i vari tipi di risorse che si vogliono utilizzare all'interno di una applicazione.

2.1.5 Applications

Le applicazioni sono il livello più alto dell'architettura Android, ciò che del sistema operativo viene a contatto con l'utente e con il quale l'utente sfrutta il dispositivo.

La particolarità di Android è che qualsiasi app, qualora gliene sia dato il permesso, può accedere senza limiti a qualsiasi tipo di risorsa presente sul dispositivo, dando quindi allo sviluppatore la massima libertà nel creare applicazioni che amplino o migliorino le funzionalità del dispositivo stesso.

2.2 Le Applicazioni Android

L'ultimo livello del sistema operativo Android è parte centrale del lavoro svolto, in quanto l'obiettivo finale della tesi è la creazione di un'applicazione Android.

Per giungere a questo risultato è importante analizzare i vari aspetti di un applicativo Android; sicurezza, Privacy, multi-threading, comunicazione, sfruttamento delle risorse all'interno dell'ambiente Android sono concetti fondamentali per la realizzazione di un applicazione di qualità, ed in quanto tali occorre conoscere perfettamente il funzionamento o le complicazioni che questi aspetti comportano nel processo di sviluppo e progettazione di una qualsiasi applicazione android.

Analizzeremo e approfondiremo pertanto nelle prossime sezioni questi aspetti ed il loro funzionamento, verrà spiegato cosa è in Android una activity, come queste ultime comunicano fra di loro, come viene gestita l'interazione utente-dispositivo, oppure come vengono gestiti dal sistema aspetti fondamentali quali privacy e protezione dei dati.

2.2.1 Sicurezza e Privacy

La così detta *mobile security* è diventata al giorno d'oggi un aspetto sempre più importante nei sistemi distribuiti, specialmente quando ne fanno parte gli smartphone, i quali per loro stessa natura possiedono al loro interno una quantità elevatissima di informazioni personali e private.

Allo scopo di difendere tali informazioni le applicazioni Android sono mandate in esecuzione in una *sandbox*, un'area del sistema isolata la quale non ha accesso alle risorse del sistema, a meno che gliene sia dato esplicito consenso dall'utente.

Tale consenso viene dato durante l'installazione dell'applicazione, nel momento della quale appare una lista di risorse alla quale l'applicazione ha il permesso di accedere: ad esempio una applicazione per l'invio di messaggi potrà accedere alla parte telefonica del dispositivo, un gioco potrà accedere alla memoria per salvare i progressi di gioco, o un browser potrà accedere alla connessione dati.

La *sandbox* Android ha inoltre lo scopo di separare l'ambiente di esecuzione delle applicazioni: ogni applicazione infatti viene eseguita con un proprio livello di utenza specificato dal sistema Linux sottostante, limitando in questo modo eventuali problemi alla singola applicazione, che non può quindi disturbare o danneggiare l'esecuzione delle altre[5].

[5]Un esempio è il bug trovato poco dopo l'uscita ufficiale del primo cellulare Android, il T-Mobile G1, riguardante il web browser. Grazie a questo tipo di progettazione, il problema rimase confinato al browser, che venne sistemato in breve tramite una patch, senza danneggiare in questo modo la normale esecuzione delle altre applicazioni.

Il livello al quale le applicazioni vengono eseguite inoltre non è mai quello di root[6], il quale non viene mai dato all'utente, per garantire la sicurezza del dispositivo. Di base quindi una applicazione Android non può accedere a partizioni protette del sistema come `/system`, a meno di sfruttare particolari programmi esterni per poter acquisire tale livello di esecuzione effettuando quella che, per l'appunto, viene chiamata procedura di root; una volta effettuata tale procedura risulta in ogni caso possibile, tramite apposite applicazioni, controllare quali applicazioni vengono eseguite dal sistema Android a livello di root e quali invece a livello utente.

2.2.2 I componenti delle applicazioni

Abbiamo quindi spiegato come le applicazioni siano divise all'interno del sistema, ognuna di queste essendo eseguita in una sezione protetta e univoca del sistema.

Ma come sono composte le applicazioni al loro interno?

Le applicazioni sono costruite da vari componenti, i quali hanno le loro particolarità e possono comunicare fra di loro. I componenti più importanti sono 4: Activity, Service, Content provider e Broadcast receiver.

Activity

Le activity sono il cuore di una applicazione, ne definiscono l'interfaccia grafica ed hanno il compito di interfacciarsi con l'utente. Tipicamente ogni activity possiede il proprio layout, avendo quindi la propria personale interfaccia grafica.

Il sistema passa da un activity all'altra a seconda delle interazioni dell'utente, ed una sola activity viene eseguita per volta, mentre tutte le altre sono sospese o in *background*.

Al contrario di Java classico, in Android non esiste un `main()` dal quale parte l'esecuzione di un programma.

Una activity ha un ciclo di vita che viene eseguito a seconda dell'interazione con l'utente e del bisogno di risorse del sistema, che effettua quindi delle chiamate all'activity seguendo un ciclo ben definito, riprodotto in figura 2.2.

L'activity viene per la prima volta creata dal sistema in varie situazioni, tipicamente quando l'utente clicca l'icona nel launcher[7] o quando un'altra applicazione manda un'Intent[8] all'activity in questione.

In questo scenario il sistema chiama prima il metodo `onCreate()`, qualora l'activitiy non sia ancora stata creata, successivamente il metodo

[6]Livello di utenza amministrativo in un sistema Unix.

[7]Il desktop di Android

[8]Il mezzo con il quale comunicano i componenti e le applicazioni fra di loro, verrà approfondito nella sezione seguente

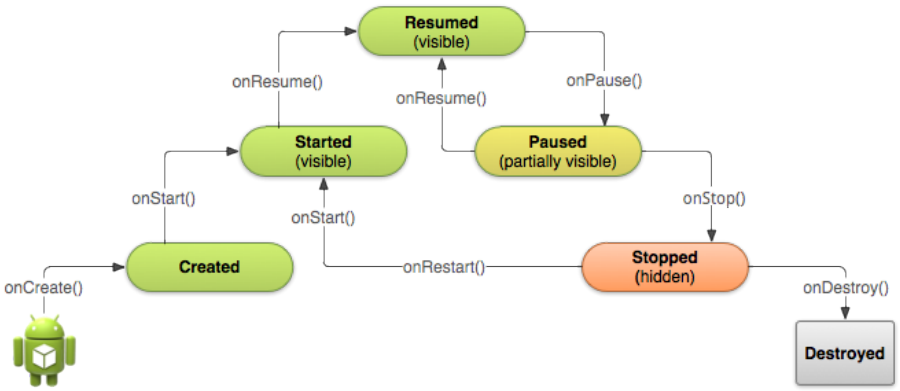

Figura 2.2: Lifecycle di una activity Android

`onStart()`, ed infine `onResume()`, portando in *foreground* l'applicazione, che ha quindi ora il focus del sistema ed interagisce con l'utente.

Quando si passa ad un'altra activity (ad esempio se stiamo per ricevere una chiamata) il sistema chiama prima di passare alla nuova activity il metodo `onPause()` seguito dal metodo `onStop()` dell'activitiy attualmente in *foreground*, mandandola in pausa in *background*. Un'activity in questo stato mantiene le informazioni immesse dall'utente, tuttavia non può eseguire alcun tipo di codice.

Qualora una activity in *background* venga richiamata dall'utente (ad esempio finita la chiamata) il sistema richiama il metodo `onResume()`, ripristinandola a come era prima del metodo `onStop()`.

Nel caso in cui il sistema Android abbia bisogno di liberare risorse, le activity in *background* possono essere chiuse completamente. In questo caso il sistema chiama il metodo `onDestroy`, liberando completamente la memoria relativa all'activity in questione.

Per creare una activity perciò è sufficiente sovrascrivere questi metodi, inserendo le operazioni che vogliamo compiere nelle varie situazioni di esecuzione nella quale si può trovare la nostra activity. Potremmo ad esempio voler disegnare l'interfaccia grafica alla chiamata dell'`onCreate()`, salvare dei dati quando l'activity viene chiusa tramite il metodo `onStop()`, o definire la nostra routine centrale all'interno del metodo `onResume()`.

Service

Il service è concettualmente simile all'activity, le sue principale differenze rispetto a quest'ultima sono la possibilità di eseguire codice anche mentre è in *background* e la sua mancanza di un'interfaccia, che al contrario caratterizza fortemente una activity.

I service vengono genericamente utilizzati per compiere operazioni che non necessitano di avere il focus dell'utente per essere eseguite: ad esempio per la riproduzione musicale o il download di un file, cose che un utente potrebbe voler fare mentre in *foreground* è presente qualcos'altro, come un gioco o un browser.

I service, oltre che per garantire il multi-threading, sono utilizzati anche per calcoli computazionali per una activity, aleggerendo il carico delle stesse, mantenendo così reattiva l'interfaccia utente.

Esistono due forme principali di service: Started e Bound.

Un service è di tipo Started quando viene lanciato da una applicazione tramite il metodo `startService()`, ed una volta lanciato viene eseguito fino al termine del suo lavoro.

È compito dello sviluppatore definire quando questo tipo di service deve interrompere la propria esecuzione, chiamando o il metodo `stopSelf()` all'interno del service stesso, o tramite il metodo `stopService()` da una applicazione qualsiasi (solitamente dalla stessa la quale ha fatto partire l'esecuzione in primo luogo).

In alternativa un service può essere lanciato da un componente come - Bound; in questo caso la vita del service è strettamente legata al componente che ne ha avviato l'esecuzione, in quanto se quest'ultimo viene interrotto il service viene interrotto di conseguenza.

Il vantaggio nell'utilizzare service Bound risiede nella semplicità con cui questi ultimi possono comunicare con i componenti che ne hanno avviata l'esecuzione; nel momento della creazione del service tramite il metodo `bindService()` viene infatti fornita una interfaccia utilizzabile per la comunicazione. Inoltre è possibile effettuare il bind di più componenti con lo stesso service: in questo caso il service verrà terminato nel momento in cui tutti i componenti a lui collegati vengono terminati.

Content Provider

Questi componenti hanno il compito di gestire le informazioni e i dati relativi alle applicazioni, utilizzando l'interfaccia di un database relazionale[9], nonché quello di fornire meccanismi per la sicurezza dei dati.

Ogni Content Provider è associato ad una *Authority* univoca, utilizzabile dagli altri componenti dell'applicazione per eseguire query SQL allo scopo di scrivere o leggere dati. Da notare il fatto che in Android tutti i dati vengono gestiti con questo metodo, per cui ad esempio anche la semplice condivisione di un file avviene appoggiandosi a questo componente.

I Content Provider costituiscono inoltre l'interfaccia standard del sistema tramite la quale i dati di un processo vengono resi disponibile al codice eseguito all'interno di un altro processo, come vedremo nella sezione successiva.

[9]Nel sistema Android viene utilizzato SQLite.

Broadcast Receiver

Infine abbiamo i Broadcast Receiver, l'ultimo tipo di componente di una applicazione Android.

Il Broadcast Receiver funge da casella mail, per così dire, dell'applicazione, ricevendo messaggi dalle altre applicazioni.

Solitamente all'interno del sistema Android i messaggi vengono inviati a destinazioni implicite; è necessario quindi che i Broadcast Receiver si iscrivano attivamente al tipo di messaggi che vogliono ricevere, che in Android vengono chiamati Intents.

Nella sezione seguente vedremo in maniera più approfondita la comunicazione fra diverse applicazioni in Android, approfondendo il funzionamento di Content Provider e Broadcast Receiver.

Fragment

Sebbene non siano dei veri e propri componenti di una applicazione, in quanto molto simili e strettamente dipendenti alle activity, i Fragment sono diventati parte fondamentale nella programmazione di una applicazione in Android.

I Fragment sono stati introdotti relativamente tardi nello sviluppo di Android[10], con l'avvento cioè della prima versione Android studiata per i Tablet, caratterizzati da uno schermo molto più ampio rispetto a quello che si trova comunemente in uno smartphone.

Con uno schermo così ampio ci si è sentiti limitati dal concetto di Activity, e si è sentito il bisogno di avere una interfaccia modulare, definibile a runtime a seconda delle dimensioni dello schermo sul quale è in esecuzione l'applicazione.

Nasce così il concetto di Fragment: un frammento di interfaccia modulare e riutilizzabile, avente il proprio layout ed il proprio comportamento. Si può pensare a questi frammenti come a delle sub-activity, utilizzabili in diverse Activity, i quali possono essere composti a piacere per realizzare il layout e le funzionalità desiderate.

Come possiamo vedere in figura 2.3, l'activity diventa quindi un semplice contenitore di diversi Fragment. Una combinazione di diversi Fragment da vita all'interfaccia finale lato utente, la quale verrà disegnata a seconda del tipo di dispositivo in uso direttamente a Run Time. Nell'esempio in figura abbiamo una lista, creata dal Fragment A, e delle informazioni relative ad ogni lista, rese a display dal Fragment B.

Mentre nello smartphone abbiamo un comportamento realizzabile tramite più Activity (cliccando un oggetto della lista appare una nuova Acti-

[10]Sono stati introdotti con Android 3.0 (API level 11). Esistono tuttavia librerie di supporto da implementare in un progetto, qualora si voglia creare una applicazione per Android 2.x con le funzionalità dei Fragment.

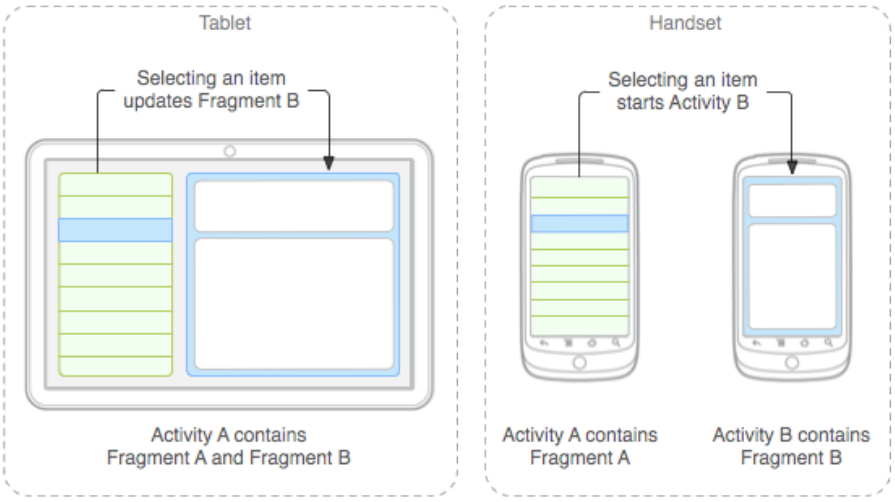

Figura 2.3: Un esempio della stessa Applicazione gestita tramite Fragment.

vity mostrante le informazioni inerenti), utilizzando i Fragment è possibile mostrare a schermo entrambi i Fragment contemporaneamente utilizzando una sola Activity, aggiornando il fragment B a seconda dell'elemento del Fragment A che viene selezionato.

Tuttavia è sbagliato separare Fragment e Activity concettualmente, in quanto questi ultimi caratterizzano fortemente l'esecuzione dei primi. Ad esempio un Fragment non può essere eseguito se non all'interno di una Activity, venendone fortemente influenzato; per esempio nel caso in cui sull'activity host venga chiamato dal sistema il metodo `onPause()`, quest'ultimo viene chiamato anche su tutti i Fragment da essa contenuti. La cosa però non vale nel verso opposto: mentre l'Activity è in esecuzione, è infatti possibile manipolare, sostituire, interrompere od eseguire i Fragment in maniera completamente indipendente; queste transizioni sono comunque gestite dall'Activity, la quale ne tiene traccia in un apposito *back stack*. In questo modo è ad esempio possibile ritornare ad un Fragment precedentemente sostituito tramite il tasto *back*, rendendo intuitiva la navigazione all'interno delle applicazioni.

Il ciclo di vita di un Fragment, in figura 2.4 è molto simile a quello di una Activity, a parte per l'aggiunta di 4 metodi: `onCreateView()` e `onDestroyView()`, i quali hanno il compito di disegnare l'interfaccia del Fragment; `onAttach()` e `onDetach()` vengono invece invocati nel momento in cui ne viene richiesta l'aggiunta/rimozione nel layout di una Activity.

È interessante notare che, al contrario dell'Activity, il cui layout è definito nella fase di sviluppo e non modificabile a Run Time, i Fragment caricano il layout che intendono utilizzare durante la chiamata del metodo

`onCreateView()`, rendendo possibile perciò per un Fragment caricare layout differenti, a Run Time, a seconda delle necessità.

2.2.3 Comunicazione fra i vari componenti

La comunicazione fra applicazioni diverse avviene in Android tramite gli *Intent*, dei messaggi contenenti l'indirizzo di un componente e dei dati.

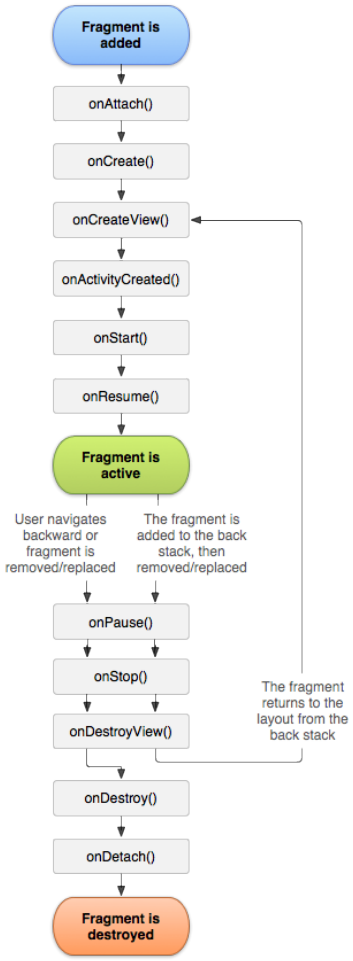

Figura 2.4:
Il ciclo di vita di un Fragment.

Android fornisce delle API che accettano Intent di vario tipo, utilizzati ad esempio per fare partire delle Activity (`startActivity(Intent)`), dei Service (`startService(Intent)`) o per mandare dei messaggi (`sendBroadcast(Intent)`).

L'esecuzione di questi metodi avverte il framework Android riguardo alla necessità di eseguire una determinata porzione di codice dell'applicazione indirizzata dall'Intent utilizzato. Questo processo di comunicazione viene chiamata in Android *action*.

Una delle caratteristiche più potenti di Android è l'utilizzo che viene fatto di questo meccanismo di comunicazione. Infatti non è obbligatorio indirizzare un Intent ad una applicazione ben specifica, si possono bensì utilizzare anche nomi impliciti[11]. In questo caso il sistema utilizza il programma predefinito ad eseguire l'Intent richiesto, e qualora più applicazioni installate siano in grado di eseguire l'action specificata il sistema fa scegliere all'utente quale utilizzare.

Ad esempio per aprire una pagina web si crea un Intent di tipo `ACTION_VIEW`, si aggiunge a questo Intent i dati utili all'*action*, quali l'indirizzo web della pagina che vogliamo sia aperta. All'esecuzione di questa *action* tramite (`startActivity(Intent)`) il sistema cercherà le varie applicazioni di browsing installate, lanciando il browser alla pagina indicata dall'Intent.

Questo processo di passaggio di dati e comunicazione necessita di un nuovo livello di sicurezza, che in Android

[11] Per nomi espliciti si intende quando all'interno dell'Intent viene dichiarato il nome specifico dell'applicazione da eseguire.

vengono gestiti tramite dei filtri nel file `AndroidManifest.xml` presente in ogni applicazione Android.

Il `manifest` ha il compito di dire al sistema le caratteristiche fondamentali dell'applicazione: quali Activity e Service sono presenti nell'applicazione, la versione di quest'ultima, i permessi richiesti per la sua esecuzione, ma soprattutto gli Intent filters.

Questi filtri hanno lo scopo di informare il sistema riguardo a quali Intent l'applicazione vuole rispondere e quali Intent vuole ricevere, quando ad esempio altre applicazioni usano il metodo `sendBroadcast(Intent)`[12]. In questo modo viene limitata la possibilità che altre applicazioni interagiscano con la nostra applicazione in maniera accidentale.

[12]Un Intent esplicito viene sempe consegnato all'applicazione bersaglio, indipendentemente dai filtri.

2.3 Memory Management in Android e Android Run Time

Quelli finora descritti sono aspetti di base riguardo al funzionamento di una applicazione Android e le sue componenti.

Dobbiamo ora tenere in forte considerazione uno degli aspetti più innovativi del sistema Android quando, come nel nostro caso, si ha l'intenzione di sviluppare una applicazione che avrà il compito di fungere da Server all'interno di un sistema distribuito: stiamo parlando della politica di gestione della memoria interna e del funzionamento della Dalvik Virtual Machine, molto differente rispetto alla tradizionale Java Virtual Machine.

La Dalvik Virtual Machine

Le difficoltà più grandi incontrate nello sviluppo del sistema operativo Android sono state la grande eterogeneità di device che devono essere supportati, possibilmente il più ampia possibile, e la carenza di risorse hardware che la maggioranza di questi hanno.

Android si pone infatti l'obiettivo di garantire un funzionamento fluido, reattivo ed efficace ad una gamma quanto più ampia possibile di device, fornendo allo stesso tempo agli sviluppatori un set di API solide e performanti su cui lavorare serenamente senza doversi preoccupare della compatibilità verso specifici set hardware.

Per realizzare tutto ciò Android ha preferito implementare una propria virtual machine, piuttosto che utilizzare una tradizionale Java virtual machine (VM) come ad esempio Java ME (Java Mobile Edition), la quale soddisfasse tutti i requisiti richiesti, chiamata Dalvik Virtual Machine (o Dalvik VM).

La prima differenza che si nota fra Java ME e la Dalvik risiede nei file eseguibili: quest'ultima infatti utilizza file eseguibili ottimizzati per il funzionamento in ambienti con scarsa memoria disponibile, creati trasformando i file .class generati dall'SDK di sviluppo in file .dex, chiamati Dalvik EXecutables.

Inoltre, come già accennato in precedenza, ogni applicazione possiede la propria, isolata istanza all'interno della Dalvik, che si occupa del suo funzionamento, relegando ad Android Run Time il compito di monitorare la memoria e i processi dell'intero sistema; quest'ultimo ha infatti il compito di controllare lo stato complessivo del sistema, avendo la possibilità di bloccare o terminare determinati processi per liberare risorse.

Questa peculiarità del poter agire sul ciclo di vita dei processi è un'altra delle caratteristiche peculiari di Android: Android infatti ha la possibilità di terminare in qualsiasi momento qualsiasi applicazione, indipendentemente dalla sua importanza o dal lavoro che sta eseguendo, qualora tale applicazione soddisfi certi requisiti (ad esempio se è in pausa o bloccata da molto

tempo). Questo aspetto risulta di vitale importanza quando, come nel nostro caso, si vuole costruire un servizio il quale ha necessità di essere sempre attivo.

Application Priority e stato dei processi

Implementare una funzionalità come quella sopra descritta, in cui il sistema ha in qualsiasi momento la possibilità di terminare o bloccare un qualsiasi processo, porta conseguentemente con se la necessità di creare una politica di priorità e classificazione dei processi stessi, con lo scopo di ordinare, in caso di bisogno di risorse, le applicazioni che dovranno essere terminate per fare spazio a quelle che stanno per essere mandate in esecuzione.

A questo scopo Android possiede una lista, continuamente aggiornata, contenente le applicazioni da terminare in caso di necessità. Tutte le applicazioni Android infatti, una volta lanciate, restano in memoria finché non terminate attivamente dal sistema.

Android effettua questa classificazione dotando i processi di tre livelli di priorità, dalla più alta alla più bassa. Il livello di priorità di un'applicazione è dato dal livello più alto tra tutti i suoi componenti.

Nel caso in cui due applicazioni abbiano entrambe lo stesso livello di priorità (cosa che accade abbastanza spesso, avendo soltanto tre livelli) l'applicazione che vi è da più tempo occupa una posizione più alta nella lista.

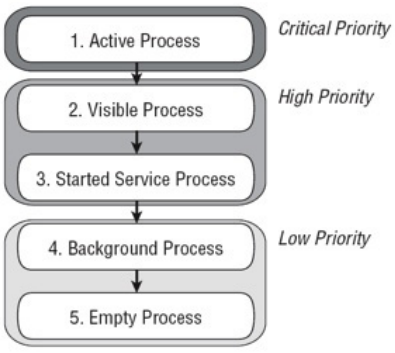

Figura 2.5: Priorità dei processi

La posizione in lista di una applicazione può essere affetta anche dalle comunicazioni fra processi: se ad esempio una applicazione dipende da un Content Provider di una seconda applicazione, quest'ultima avrà almeno lo stesso grado di priorità dell'applicazione che supporta.

Risulta importante strutturare una applicazione in modo che la sua priorità nella lista sia coerente con il compito che tale applicazione deve svolgere: altrimenti si può rischiare che il sistema decida di terminare la nostra applicazione anche se ne è nel mezzo di qualche operazione importante.

Per fare ciò occorre analizzare bene quali sono questi livelli di priorità e a che stato dei processi corrispondono, di cui una sintesi efficace è mostrata in figura 2.5.

Il livello massimo di priorità la possiedono i processi delle applicazioni i cui componenti stanno attivamente interagendo con l'utente. Questi sono

i componenti per i quali in genere viene richiesta la terminazione di altre applicazioni, in pochi hanno tale livello di priorità.

Il livello medio di priorità la possiedono invece i processi visibili o che sono già stati avviati e che possiedono service ancora in esecuzione (Started Service Processes). In genere questo tipo di processi viene difficilmente terminato a meno di necessità di grosse quantità di risorse e mancanza di processi al livello minimo di priorità.

Il livello più basso viene invece assegnato ai processi in *background* o vuoti, quando cioè non sono presenti service in esecuzione. Riguardo ai processi vuoti, questi processi appartengono ad applicazioni che hanno terminato la loro esecuzione. Vengono mantenuti in memoria per mantenere la cache corrispondente, per velocizzare un eventuale riutilizzo dell'applicazione.

In genere questi sono i processi che più facilmente e più frequentemente vengono terminati per liberare risorse a favore dei processi con priorità maggiore.

2.4 L'IDE di sviluppo: Eclipse

Per supportare lo sviluppo di applicazioni Android, Google ha creato un plugin per il noto IDE di sviluppo open-source Eclipse, disegnato con lo scopo di fornire un ambiente di sviluppo potente, flessibile e completo.

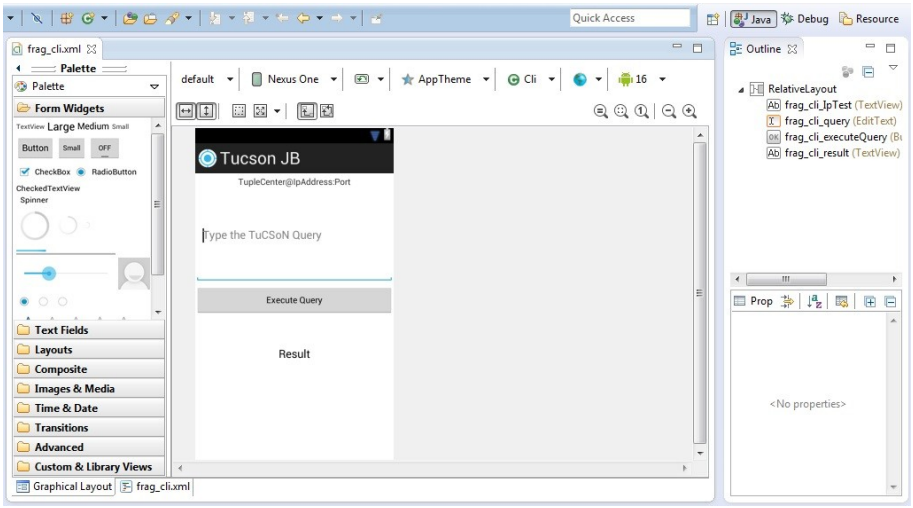

Figura 2.6: Un esempio dell'interfaccia Eclipse e della funzionalità WYSIWYG per creare l'interfaccia grafica di una applicazione. Rimane tuttavia possibile scrivere manualmente il file .xml, utilizzando questa nuova modalità solamente per vedere poi i risultati finali, come è consigliabile fare nello sviluppo di interfacce complesse, allo scopo di mantenere pulito ed efficace il codice scritto.

Tale plugin, chiamato Android Development Tools (ADT), estende le funzionalità di Eclipse implementandone di completamente nuove; permettendo di creare nuovi progetti Android, di creare la UI delle applicazioni sia tramite file xml sia con una pratico editor di tipo WYSIWYG[13], come mostrato in figura 2.6, di aggiungere package facenti parte dell'Android Framework API, aggiunge inoltre un terminale apposito per il debug delle applicazioni, ed infine rende possibile esportare il proprio progetto in formato .apk, già pronto all'installazione su qualsiasi device fisico.

All'interno dell'Android SDK è inoltre possibile fare il download di varie macchine virtuali, per simulare in un pratico emulatore un qualsiasi dispositivo esistente, permettendo di vedere il comportamento della nostra applicazione simulata in svariati device e tablet, prima dei test su device fisici. Possiamo vedere un esempio dell'emulatore in funzione in figura 2.7.

[13]What You See Is What You Get, connotazione tipica utilizzata per indicare programmi che tramite interfaccia grafica scrivono automaticamente il codice corrispondente.

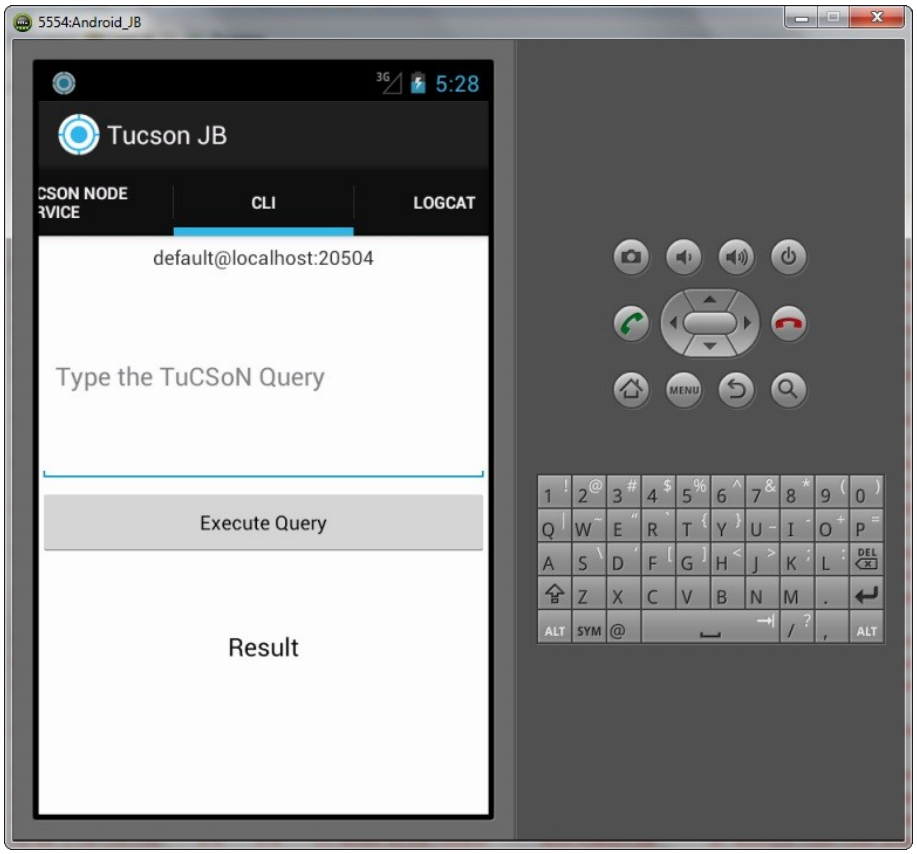

Figura 2.7: Uno screenshot dell'emulatore Android, in grado di riprodurre un qualsiasi dispositivo esistente. Nello screenshot possiamo vedere l'interfaccia CLI dell'applicazione su un nodo già avviato.

Lo scopo del poter emulare un qualsiasi tipo di dispositivo è quello di fornire allo sviluppatore la possibilità di decidere quali tipi di device verranno supportati dall'applicazione, e testare l'applicazione in detti device senza bisogno di comprarne uno reale sul quale effettuare i test. È possibile infatti dichiarare all'interno del file Manifest dell'applicazione quale versione di Android verrà supportata[14], scaricare la relativa macchina virtuale e fare su quest'ultima test, con la certezza che si otterrà lo stesso comportamento nel device reale.

Google inoltre mette a disposizione un intero website[15] nel quale sono contenuti tutorial per iniziare ad entrare nel mondo Android, nonché una spiegazione nel dettaglio di ogni aspetto del sistema Android, con codici

[14]Nello specifico quale livello di Api: ogni versione di Android ha un suo livello di API, dal più basso al più alto, incrementando di uno ad ogni Major Version uscita. Ad esempio Android 2.3 ha livello 9, mentre la 4.0 ha livello 15.

[15]developer.Android.com[4]

e applicazioni di esempio liberamente scaricabili, nonché la possibilità di navigare le API del sistema Android.

Questo facilita enormemente l'inserimento nel mondo Android, i paradigmi e i pattern di sviluppo, e la generale filosofia che sta dietro alle applicazioni ed al sistema stesso: velocità, reattività, semplicità e design sono infatti le parole chiave che lo sviluppatore deve tenere a mente quando inizia ad addentrarsi nello sviluppo delle applicazioni Android.

Parte II

Il Porting

Capitolo 3

Una prima versione

Abbiamo fino ad ora delineato ed approfondito le tecnologie che verranno utilizzate nel nostro progetto di porting. Nel particolare è stato presentato il funzionamento di un nodo TuCSoN e i concetti basilari che stanno dietro al funzionamento dell'infrastruttura. È inoltre stato dato ampio spazio al sistema operativo Android, alla sua architettura e nel particolare all'architettura delle sue applicazioni.

A questo punto del lavoro, finite le doverose (ed abbondanti) premesse teoriche, possiamo iniziare ad occuparci del *porting* vero e proprio, il quale consisterà nello sviluppo di una applicazione Android in grado di sfruttare e partecipare ad un sistema TuCSoN, sia attivamente, tramite componenti come il CLI, sia passivamente, diventando un nodo.

A questo scopo dovremo sfruttare le funzionalità di Android descritte nel capitolo precedente, a nostro vantaggio laddove siano migliori di quelle della JVM, o arginando i problemi qualora tali funzionalità limitino le capacità del nodo TuCSoN.

Ricordiamo infatti che TuCSoN è stato creato per funzionare su JVM standard, senza mai dare peso a concetti e problemi che si verificano in un sistema estremamente mobile come è invece uno smartphone Android.

La prima versione dell'applicazione è una versione piuttosto grezza, dalle funzionalità limitate e con parecchi bug da risolvere.

Per garantire il supporto al numero maggiore possibile di device si è deciso di sviluppare l'applicazione prendendo come target iniziale Android 2.3.

3.1 TuCSoN on Android

Per prima cosa, preparato l'IDE seguendo i facili tutorial sul sito ufficiale[1], è stata clonata l'ultima versione di TuCSoN disponibile online.

Qui si è subito presentato il primo problema: mentre TuCSoN, nella versione corrente, viene sviluppato utilizzando Java 7, Android, anche nella sua ultima versione, è ancora fermo ad una versione di Java corrispondente a Java 6.

Android infatti, utilizzando una propria implementazione di Java diversa da quella standard, da pieno supporto soltanto alle versioni di Java 5 e 6, non garantendo il funzionamento di tutte le nuove funzionalità introdotte da Oracle con Java 7, alcune delle quali vengono utilizzate nella versione corrente di TuCSoN.

Sebbene questo problema non abbia ripercussioni dirette sul codice, in quanto è stato sufficiente creare un .jar di TuCSoN compilato in Java 6 da includere nel nostro progetto Android[2], questo primo problema ha grosse ripercussioni a livello teorico sul nostro progetto.

Se infatti l'obiettivo finale era quello di far interagire un dispositivo Android con un qualsiasi sistema TuCSoN, questa prima fondamentale differenza pone un serio problema di compatibilità fra sistemi TuCSoN già esistenti, i quali utilizzano Java 7, con la nostra nuova applicazione, la quale necessariamente utilizzerà Java 6.

A questo scopo l'unica soluzione possibile è a livello progettuale: se uno sviluppatore prevede o vuole abilitare l'utilizzo dell'infrastruttura TuCSoN da lui disegnata ad eventuali dispositivi Android, dovrà necessariamente utilizzare TuCSoN compilato in Java 6.

3.2 TuProlog on Android

All'interno della distribuzione .jar di TuCSoN, scaricabile dal repository ufficiale[2], troviamo già incluso il motore tuProlog nella sua ultima versione disponibile.

Essendo tuttavia stati costretti a ricompilare TuCSoN, è stato necessario procurarci anche l'ultima versione di tuProlog da inserire nel nostro progetto Android.

L'unico problema riscontrato con la versione attuale di tuProlog riguarda la versione delle classi serializzabili, non definita nel codice ma automaticamente dal compilatore. Sebbene la cosa non causasse problema nella release ufficiale di TuCSoN, Android generava identificativi di versione differenti da quelli che venivano generati automaticamente dalla JVM.

[1] developer.Android.com

[2] Cosa che ha comunque portato a qualche correzione nel codice originale di TuCSoN, per non generare errori nella compilazione in Java 6, che risultasse comunque utilizzabile in Java 7.

Questo generava delle eccezioni in Android Run Time, in quanto i sistemi Android e PC davano automaticamente diverse `serialVersionUID` alla stessa classe, generando un'eccezione di tipo `InvalidClassException`.

È stato sufficiente fare presente il problema agli sviluppatori che si occupano del progetto tuProlog, i quali hanno definito seriali specifici alle classi in oggetto (eliminando i `@SuppressWarnings(serial)` presenti nelle classi utilizzate per eliminare i problemi presentati da questo difetto, dandoci quindi una versione di tuProlog perfettamente compatibile con Android.

3.3 Activity vs Service

A questo punto, ottenute le infrastrutture compatibili con l'ambiente Android, non rimane altro da fare che creare il progetto della nostra applicazione, inserendo le librerie di TuCSoN e tuProlog all'interno della directory dell'applicazione[3].

Si era inizialmente deciso di lanciare il Nodo TuCSoN all'interno dell'activity dell'applicazione, idea che è stata scartata subito per la sua inefficienza. Mentre infatti il codice e il progetto risultavano molto semplici, una Activity non può essere una scelta corretta per implementare un servizio di tipo server.

Come descritto nel capitolo relativo ad Android infatti le activity vengono bloccate tramite il metodo `onPause()` e `onStop()` molto frequentemente, rendendo il nodo non solo instabile e spesso non raggiungibile, ma venendo addirittura spento automaticamente dal sistema nel caso del semplice utilizzo dello smartphone.

Il Thread sul quale risiede il nodo risultava infatti essere interno all'activity: nel caso in cui quest'ultima venisse bloccata, per un qualsiasi motivo, il nodo veniva arrestato di conseguenza, rendendo non più raggiungibile il server. Mentre un comportamento del genere può essere tollerato in certi casi, come ad esempio nel caso in cui un agente TuCSoN manda informazioni inserite dall'utente ad un nodo, non può invece essere accettato per una funzionalità di tipo server, in cui l'applicazione deve essere sempre attiva e pronta a ricevere richieste da parte di agenti distribuiti.

Con un'implementazione di questo tipo è sufficiente una semplice chiamata telefonica, o una pressione del tasto back, per far bloccare il nodo, venendo chiamato il metodo `onStop()` all'activity sulla quale risiede quest'ultimo.

La scelta obbligata è stata pertanto quella di implementare il nodo TuCSoN come un service, dichiarandolo nel Manifest, sollevando tuttavia numerosi altri problemi e decisioni da intraprendere.

[3] In automatico, quando eclipse genera l'.apk, inserisce le librerie utilizzate, a patto che siano nella cartella /libs a radice del progetto.

3.4 Il nodo come Service Android

Sorge quindi spontanea la necessità di lanciare il nodo TuCSoN come service Android. Come precedentemente descritto, il service è un componente dell'applicazione in grado di essere eseguito anche in *background*.

L'ipotesi di creare un buond service è stata subito scartata, in quanto avremmo avuto gli stessi inconveniente che si presentano creando un nodo come activity, e cioè che quando l'utente, nel suo normale utilizzo dello smartphone, uscirà dalla nostra applicazione, il nodo verrà interrotto, in quanto l'activity a cui è bindato viene bloccata ed in seguito terminata dal sistema Android.

Viene pertanto implementato il nodo come started service, lanciato e bloccato dall'utente tramite apposita interfaccia. È importante fornire all'utente un modo per bloccare l'esecuzione del nodo, in quanto come è stato spiegato in precedenza uno started service rimane sempre in esecuzione fino a che o venga terminata la sua esecuzione, caso che non accade mai in un servizio di tipo server come è il nodo TuCSoN, oppure una activity chiami esplicitamente il metodo `stopService(Intent)`, dichiarando come Intent il service che si vuole terminare.

A questo punto ci si pone il problema della comunicazione fra service e applicazione: come facciamo a comunicare con l'utente da un service, se questo non è bindato alla nostra activity?[4] In Android infatti, per garantire sicurezza, soltanto le activity (o i fragment) proprietari possono modificare le proprie interfacce, ad esempio per scrivere o leggere dei testi.

A questo scopo Android introduce gli `handler`, dei messaggi contententi porzioni di codice sotto forma di `Runnable`. Gli Handler possono avere due utilizzi principali: programmare messaggi o porzioni di codice che andranno in esecuzione in un punto specifico nel futuro, oppure essere utilizzati per eseguire del codice in un Thread diverso da quello in cui sono stati inizializzati; possiamo perciò modificare l'interfaccia utente anche all'interno del

Un altro problema è il livello di priorità del service: deve essere sempre al livello massimo, non vogliamo infatti che il sistema Android termini l'esecuzione del nodo per liberare risorse.

A questo scopo Android mette a disposizione il metodo `startForeground(int, notification)`, il quale, chiamato all'interno del service, permette allo stesso di avere priorità massima rispetto agli altri processi. Certamente in caso di un funzionamento anomalo del sistema anche questi ultimi possono essere terminati per liberare risorse, tuttavia è un caso molto remoto, considerabile al pari di un guasto meccanico.

[4] Come è stato scritto in precedenza, un service di tipo bound restituisce un oggetto di tipo `binder`, usabile come interfaccia per comunicare fra il service le activity bindate a quest'ultimo; al contrario un service di tipo started non fornisce nessuna interfaccia di questo tipo.

Ponendo il nostro service in questa maniera tuttavia dobbiamo accertarci che le sue interazioni non siano troppo gravose per il device stesso, altrimenti avrenno un calo di reattività e di prestazioni che può andare contro all'utilizzo comune del dispositivo.

Quando utilizziamo il metodo `startForeground(int, notification)` dobbiamo fornire inoltre un oggetto di tipo `Notification`: questo oggetto non è altro che la notifica con la quale comunichiamo all'utente che il nostro servizio è attivo, come vediamo in figura 3.1.

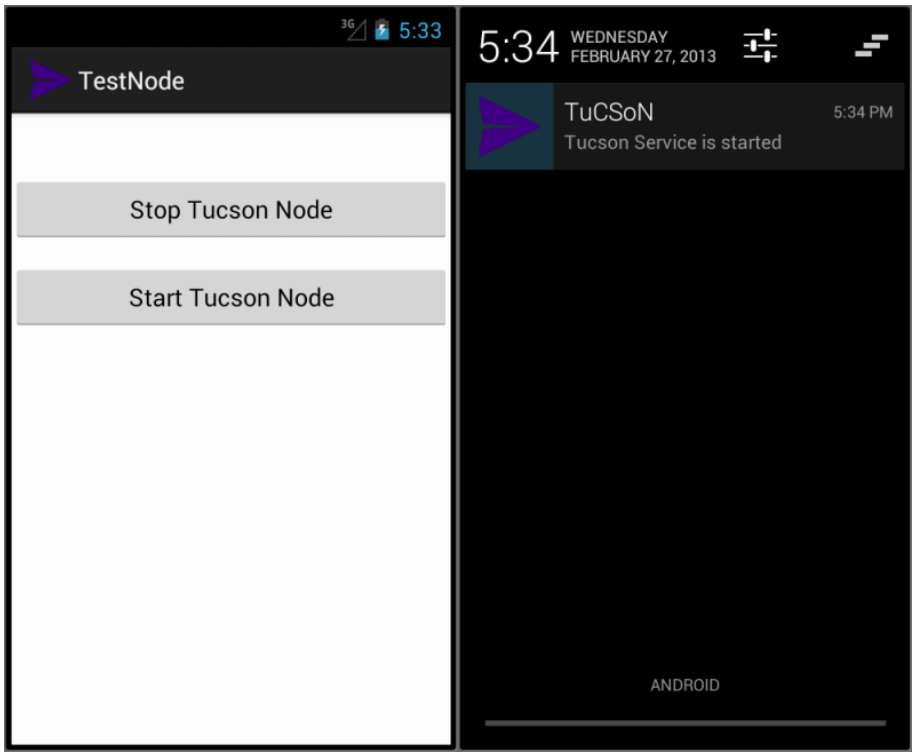

Figura 3.1: Uno screenshot della prima versione dell'app. Sulla sinistra vediamo una semplice activity con i due pulsanti per avviare o fermare il service del nodo TuCSoN, sulla destra troviamo invece la notifica per avvertire l'utente riguardo al servizio correntemente attivo

A questo punto una versione base del *porting* può considerarsi pronto: abbiamo un nodo TuCSoN capace di funzionare in *background* fino ad una interruzione generata dall'utente, in quanto difficilmente il sistema Android deciderà di terminarlo autonomamente per liberare risorse.

Tuttavia non abbiamo modo di monitorare il corretto funzionamento del nodo, né abbiamo modo di interagire con lo stesso in caso di necessità. Non possiamo neppure decidere su quale porta mettere in ascolto il nodo, né possiamo lanciare contemporaneamente due nodi su due porte differenti.

L'interfaccia inoltre è tutt'altro che user-friendly, né rispecchia la nuova generazione di applicazioni Android.

Tutte queste considerazioni ci portano a considerare un miglioramento generale dell'applicazione, oltre alla implementazione di varie altre utility che possono essere utilizzate al di à del semplice avvio di un nodo TuCSoN.

3.5 Supporto 2.3 in futuro

La versione appena descritta era inizialmente sviluppata per Android 2.3, come già detto. Tuttavia, per migliorare il look&feel generale dell'applicazione è stato scelto di utilizzare per le versioni successive i fragment, disponibili solo a partire da Android 3.0, e passare all'utilizzo delle API di livello 16 (Android 4.1).

Questo comporta tuttavia ad un taglio di molti dispositivi attualmente in circolazione, che vuoi per mancanza di supporto, vuoi per carenze a livello hardware sono impossibilitati all'utilizzo di Android 4.1.

Per questo motivo si potrà, in futuro, creare l'applicazione utilizzando le librerie `Android.support.v4` e `Android.support.v13`, librerie di supporto che hanno il compito di emulare le funzionalità introdotte in versioni Android successive, rendendo possibile ad esempio l'utilizzo dei fragment anche su Android 2.3.

Capitolo 4

Verso una Stable Release

Vista la necessità di implementare nuove funzionalità e di migliorare il look&feel generale dell'applicazione, cercando di mantenere al contempo facilmente utilizzabile ed intuitiva l'applicazione, è stato deciso di passare ad una implementazione tramite fragment, alzando il livello di API richieste al 16 ed utilizzando tutta una serie di nuove funzionalità che Android mette a disposizione a partire da questa versione.

Questo comporta un miglioramento sia a livello prestazionale, dato semplicemente da uno stadio di Android più avanzato e più performante, sia di design, altro aspetto fondamentale nella progettazione di applicazioni Android, sia a livello di usabilità, in quanto tutte le funzionalità sono molto più immediate e facilmente raggiungibili rispetto ad Android 2.3.

Ne abbiamo un esempio nella funzionalità dell'`Actionbar`, che ci permette in qualsiasi momento di tornare alla schermata home, o nella gestione dei menu, più semplice ed immediata, o nel poter effettuare lo *swipe* fra una schermata e l'altra, o la possibilità di lanciare dei popup per l'inserimento di valori o per mandare messaggi all'utente.

Vedremo nelle sezioni seguenti come sono stati sviluppati questi concetti, inoltre verranno presentate due nuove funzionalità introdotte all'app, allora scopo di fornire un metodo di comunicazione diretto con il nodo.

4.1 Il passaggio ai Fragment

Una applicazione basata solo su activity rende meccanica, poco intuitiva e lenta la navigazione attraverso i menù e le funzionalità fornite, in quanto viene essenzialmente strutturata da una activity home, solitamente la prima ad essere lanciata all'avvio dell'applicazione, contentente i collegamenti a tutte le altre activity tramite dei bottoni cliccabili, come succedeva nella nostra prima implementazione.

Con l'utilizzo dei fragment invece possiamo creare una sola activity, la quale contiene al proprio interno tutti i fragment di cui abbiamo bisogno. Ogni fragment avrà le proprie funzionalità ed il proprio ciclo di vita, mentre la stessa activity resterà sempre in foreground; per passare da una funzionalità all'altra basterà usare quello che viene chiamato *swipe* fra una schermata e quella successiva, permettendo all'utente di passare agevolmente e velocemente da una funzionalità all'altra, come mostrato in figura 4.1.

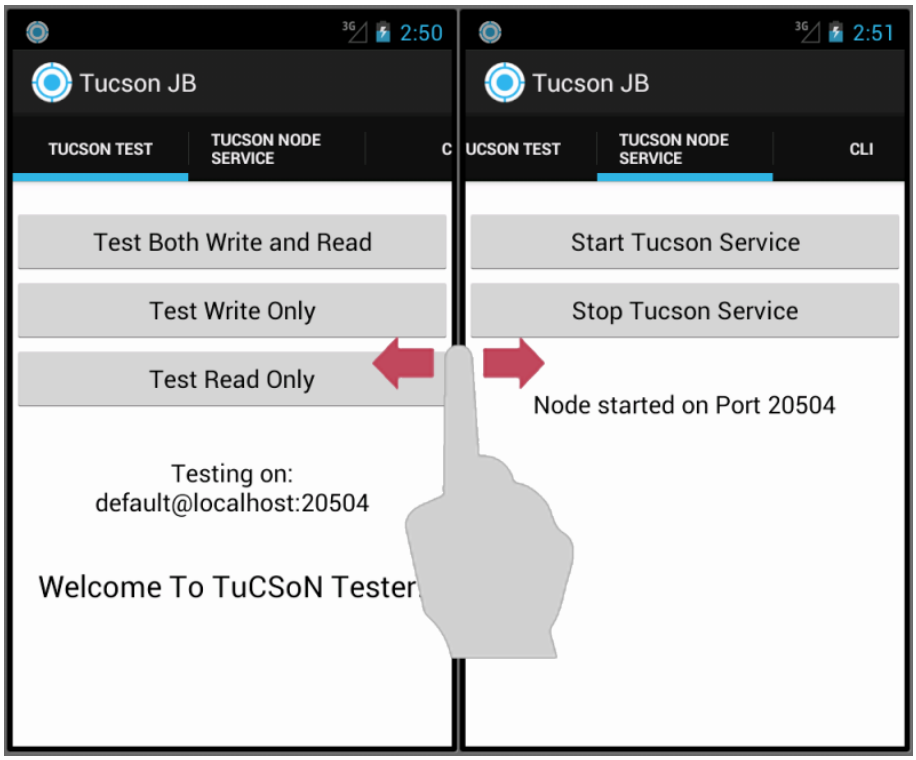

Figura 4.1: Screen che mostra intuitivamente il funzionamento dei fragment. Allo *swipe* l'activity home rimpiazza il fragment nella finestra principale con quello precedente o successivo.

In questo modo non solo miglioriamo notevolmente l'usabilità dell'applicazione, ma aumentiamo anche le possibilità di estensione del codice;

qualora si voglia inserire una funzionalità (come è stato fatto in seguito) basta infatti inserire un nuovo tab nell'activity home, e successivamente implementare un fragment come se fosse una semplice activity, contentente le funzionalità richieste.

È importante, quando si implementa questo tipo di interfaccia, ricordarsi che il codice eseguito nei Fragment va in esecuzione nello stesso Thread dell'activity: occorre quindi evitare grossi calcoli all'interno del fragment, che deve essere utilizzato solo per la costruzione e l'aggiornamento della GUI. Tutti i calcoli che possono rallentare il sistema devono essere eseguiti in Thread separati, onde evitare il blocco completo della GUI e conseguente crash dell'applicazione. Per comunicare con la GUI si può utilizzare lo stesso principio utilizzato per i Service, ovvero degli Handler passati al fragment contententi le istruzioni per l'aggiornamento.

4.2 Tucson Tester

La prima funzionalità implementata è stata implementata più per necessità di debug che per fornire vere e proprie funzionalità utili ad un sistema TuCSoN.

Essenzialmente si basa su un piccolo agente, in grado di scrivere o leggere una semplice tupla contentente un messaggio identificativo sul proprio nodo[1]. La sua utilità inizialmente è molteplice: controllare che il nodo sia attivo e raggiungibile, controllare che i comandi TuCSoN sullo smartphone siano effettivamente funzionanti, testare in locale il nodo TuCSoN.

Il motivo per cui si è deciso di mantenerlo nella release ufficiale, sebbene sia nato come semplice ed immediato strumento di debug è perché nella sua semplicità può comunque risultare utile per gli stessi motivi per cui è risultato utile nella fase di sviluppo. Con una mobilità estremamente elevata, quale è quella di uno smartphone, può essere utile un piccolo strumento in grado di controllare se il proprio nodo sia funzionante e raggiungibile.

Inoltre è stato implementato un timer, in modo da poter effettuare una specie di ping del nodo TuCSoN, utilizzabile per controllare in quanto tempo il nodo risponde alla lettura o scrittura di una tupla, in modo da poter verificare la funzionalità e reattività del un nodo.

Come possiamo vedere in figura 4.2, l'interfaccia del tester è piuttosto semplice. Possiamo infatti decidere se scrivere, leggere o fare entrambe le operazioni sul nodo bersaglio, specificato sotto per semplicità[2], ed un piccolo testo che riassume il risultato dell'operazione oltre al tempo impiegato a concluderla. In caso di errori o problemi il testo scriverà il `printStackTrace()` dell'eccezione incontrata.

[1] All'indirizzo `default@localhost:20504`.

[2] In quanto, nella release finale, sarà possibile effettuare il test su qualsiasi nodo TuCSoN presente in rete

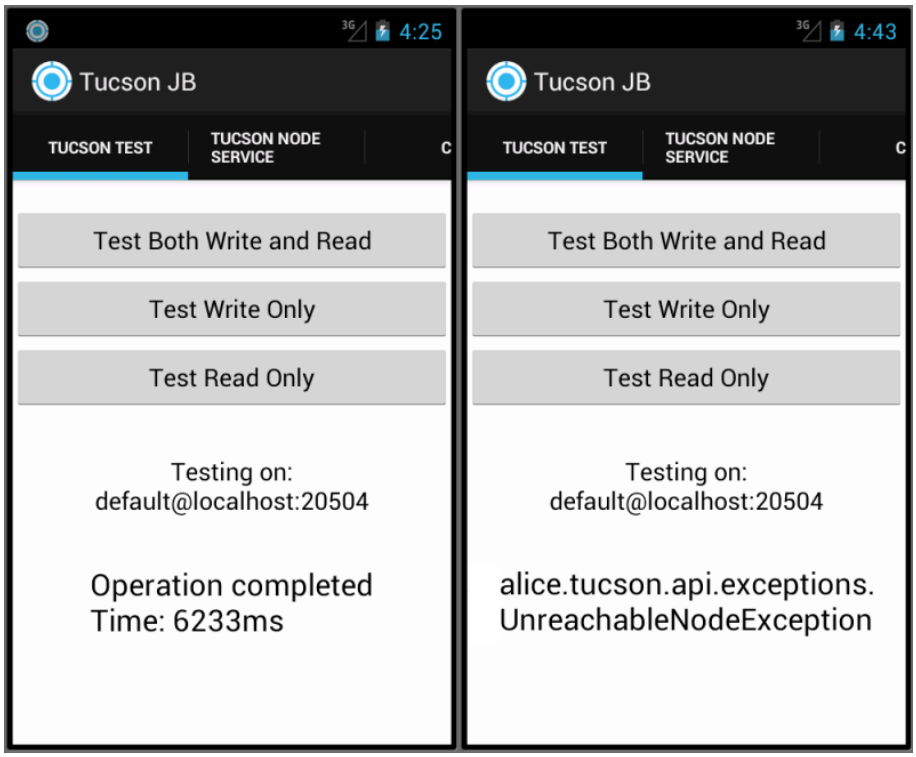

Figura 4.2: Nello screen il fragment relativo al tester TuCSoN. Come possiamo notare sulla sinistra possiamo leggere il valore di ping, molto elevato trattandosi di un test effettuato su macchina virtuale, mentre sulla destra ci viene fornita l'eccezione, che in questo caso ci avvisa che il nodo TuCSoN non è attualmente raggiungibile. Come possiamo notare in alto infatti non è neppure presente l'icona di notifica, il che indica che il nodo non è attualmente installato sul dispositivo (I test sono effettuati su localhost).

4.3 Il Logcat

Finora, con l'applicazione creata, abbiamo la possibilità di far partire un nodo e di testare se è effettivamente funzionante.

A questo punto ci si pone il problema di come fare a monitorare il corretto funzionamento del nodo dal dispositivo stesso; TuCSoN infatti è programmato per scrivere su terminale varie informazioni durante la sua esecuzione, per la maggior parte stampe di controllo e di debug. Può essere utile avere la possibilità di leggere tali stampe anche direttamente dal dispositivo. Per ora infatti l'unico modo che abbiamo di vedere tali stampe è collegando il device sul quale abbiamo installato la nostra applicazione ad un pc, avente i driver

ADB [3] installati e funzionanti e abilitando l'opzione di USB Debugging[4] nello smartphone.

Questo ci permette di monitorare varie cose del nostro smartphone direttamente da Eclipse tramite l'ADT, fra le quali ad esempio il logcat.

Il logcat altro non è che il sistema di log fornitoci da Android per ricevere da varie porzioni di sistema e dalle applicazioni varie informazioni di debug, rese poi disponibili tramite vari comandi `logcat`.

La differenza rispetto a Java è l'assenza di terminali indipendenti, nei quali ogni applicazione scrive il proprio output (ad esempio, su pc, lanciando il CLI e il nodo avremo due terminali, nel quale ognuno scriverà le stampe relative a se stesso tramite `System.out.println()`), abbiamo invece un unico terminale nel quale vengono raccolte le stampe relative a tutte le applicazioni ed al sistema Android stesso.

Il nostro obiettivo a questo punto è fornire un Fragment nel quale è possibile leggere il logcat direttamente dallo smartphone, senza la necessità di collegare il cellulare via cavo ad un pc, rendendo possibile il monitoraggio del corretto funzionamento del nodo direttamente dal device stesso sul quale è in esecuzione.

A questo scopo abbiamo la necessità di prelevare le informazioni di nostro interesse dal logcat di sistema, e mostrarle all'utente tramite un semplice campo di testo. Visto che le informazioni sono raccolte insieme in ordine cronologico, indipendentemente dalla fonte, Android fornisce dei comandi che ci permettono di filtrare tali informazioni, dandoci la possibilità di scegliere solo le stampe alle quali siamo effettivamente interessati.

Per facilitare questo tipo di filtraggio dall'utente (che in Eclipse viene eseguito automaticamente) i messaggi raccolti dal logcat vengono divisi in vari livelli di priorità:

- V - Verbose (priorità più bassa)
- D - Debug
- I - Info (Livello delle classiche stampe tramite `System.out.println()`
- W - Warning
- E - Error (Le eccezioni appartengono a questa categoria)
- F - Fatal
- S - Silent (Priorità più alta: se si sceglie questo filtro non si otterranno stampe)

[3] Universal Android ADB usb compatible drivers: driver universali che ci permettono di collegare uno smartphone Android ad un PC rendendo possibile la comunicazione fra questi. In questo modo è possibile lo scambio di informazioni di vario tipo, fra le quali la possibilità di leggere il logcat.

[4] Opzione all'interno dei settings standard del sistema Android.

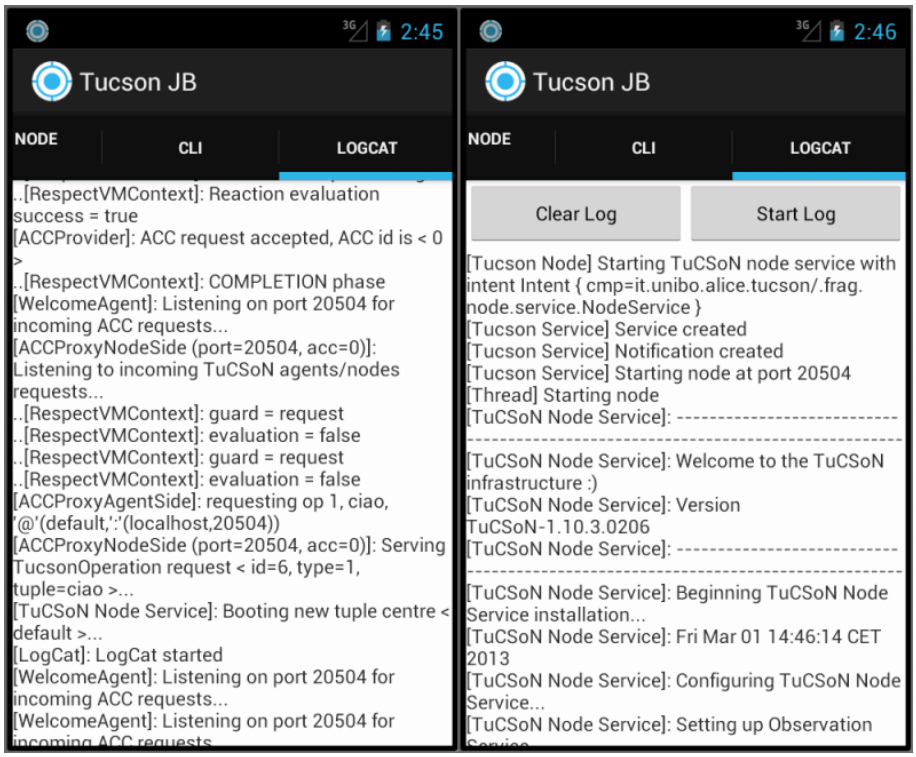

Figura 4.3: Il fragment relativo al logcat. Sulla sinistra possiamo vedere uan richiesta effettuata sul nodo da un CLI, sulla destra come si presenta l'inizio del logcat dopo un avvio del nodo.

Tramite il comando aggiuntivo `-v` possiamo decidere in che modo e con quante informazioni aggiuntive vengono mostrate le informazioni del logcat: il livello, l'orario etc. Nel nostro caso è stato utilizzato sempre `-v raw`, il quale mostra solo il messaggio grezzo, senza metadati aggiunti.

Un comando del tipo `logcat -d -v raw *:W` ad esempio ci mostrerà soltanto le stampe di priorità maggiore ai Warning; il comando `logcat -d -v raw *:S` invece risulterà completamente vuoto (non esistono stampte a priorità maggiore di Silent).

Nella nostra applicazione viene utilizzato il comando `logcat -d -v raw System.out:I System.err:W *:S`, che ha lo scopo di stampare soltanto i messaggi provenienti da stampe effettuate da `System.out` e da errori proveniente da `System.err`, filtrando tutto il resto (motivo per cui è presente `*:S`).

Si potrebbe, manualmente, filtrare anche per package in modo da isolare la nostra applicazione. Si è deciso di lasciare tutte le stampe di sistema in quanto genericamente non utilizzate da altre applicazioni, e per dare il modo di vedere quando e come altre applicazioni interferiscono con l'esecuzione del nodo TuCSoN.

Inoltre tramite il comando `-d` il logcat viene letto e stampato una tantum. Ciò significa che eventuali stampe seguenti non vengono automaticamente aggiunte una volta stampate a schermo. Per questo il fragment contenente il logcat, come mostrato in figura 4.3, contiene due pulsanti; uno per pulire il logcat del sistema (qualora abbiamo troppe informazioni), l'altro per far partire un Thread avente lo scopo di aggiornare ogni secondo il logcat, permettendo un monitoring real-time.

4.4 Il CLI

A questo punto l'app ci fornisce la possibilità di installare un nodo, di spegnerlo, di testarne la funzionalità e la reattività tramite dei comandi di ping e di monitorare le stampe di controllo lanciate dal nodo e dal sistema TuCSoN in generale.

L'unica cosa di cui potremmo aver bisogno ulteriormente è un modo per interagire attivamente con il nodo, lanciando su di esso un qualsiasi comando TuCSoN liberamente. Infatti allo stato attuale l'unico comando che possiamo lanciare attivamente sono i comandi di ping, il che non lascia molte libertà riguardo ad eventuali interventi che si possono voler fare sul nodo.

A questo scopo il pacchetto TuCSoN, nella sua versione tradizionale, fornisce un componente chiamato CLI (Command Line Interpreter), il quale non fa altro che leggere da console dei comandi, tradurli in operazioni TuCSoN ed eseguire queste ultime su un nodo specificato. Esattamente la stessa funzionalità che vorremmo rendere disponibile nella nostra applicazione.

A questo punto ci si scontra con un problema: mentre su pc scrivere su terminale è abbastanza immediato, su Android abbiamo sì una console, ma di default questa non è accessibile all'utente. Esistono vari sistemi per aggirare il problema ed accedervi ugualmente, esistono ad esempio delle applicazioni che simulano un terminal (Terminal Emulator) prendendo informazioni inserite dall'utente e passandole alla console, oppure è possibile accedervi direttamente tramite i driver ADB di cui abbiamo parlato precedentemente utilizzando un PC, ma non esiste un modo per accederci direttamente dal dispositivo stesso.

Visto che utilizzare un'altra applicazione come un Terminal Emulator per scrivere sulla console, per poi collegarci il CLI per poter leggere i comandi, non è la soluzione più efficace, si è deciso di fornire una semplice interfaccia grafica interna all'applicazione, in cui l'utente può scrivere un qualsiasi comando TuCSoN, e poi mandarlo in esecuzione direttamente.

A questo scopo verrà sfruttata la funzionalità del CLI originale per leggere e tradurre le informazioni passate, collegando il CLI alla stringa scritta dall'utente invece che dal terminale, e, qualora il comando sia va-

lido, verrà eseguito il comando sul nodo, notificando all'utente il risultato dell'operazione.

Abbiamo in questo modo un sistema per interagire liberamente con il nodo, utilizzando qualsiasi delle primitive forniteci da TuCSoN.

4.5 Settings in Android

L'applicazione si presenta ora in un buono stato: semplice da utilizzare, sufficientemente completa, dando all'utente disponibilità di effettuare varie operazioni sul nodo.

Tuttavia l'utente è abbastanza limitato nelle scelte.

Può effettuare il ping solo sul proprio nodo, può lanciare il nodo solo sulla porta di default e può utilizzare il CLI solo sul proprio nodo.

Avendo implementato tutte queste funzionalità, ed essendo il *middleware* su Android funzionante, potrebbe essere utile rendere queste funzionalità completamente indipendenti ed utilizzabili separatamente ed a piacere dall'utente.

Per ora infatti la funzionalità di test ed il CLI hanno senso di esistere solo nel caso in cui il nodo stesso sia funzionante: potremmo tuttavia voler utilizzare il CLI su un nodo TuCSoN qualsiasi presente in rete, così come potremmo voler controllare se tale nodo sia disponibile ed in quanto tempo risponde ad eventuali comandi tramite la funzionalità di ping già implementata e disponibile nell'applicazione.

A questo scopo, sfruttando le API Android riguardo i settings, sono state implementate varie opzioni del menu, prima non disponibile.

La gestione dei menu e dei settings, in Android, è automatizzata: il menu dell'applicazione viene specificato in un file xml, indicando quali bottoni e in che ordine devono essere visualizzati nel menu a tendina.

Nella main activity è presente una routine di controllo, che lancia a seconda della scelta una activity associata: ad esempio per la pressione del pulsante help viene lancianto un `Intent` che apre nel browser la pagina ufficiale di Android in APICe, per le patch notes o per le info vengono lanciati dei fragment di popup, per i settings invece viene lanciata una nuova activity.

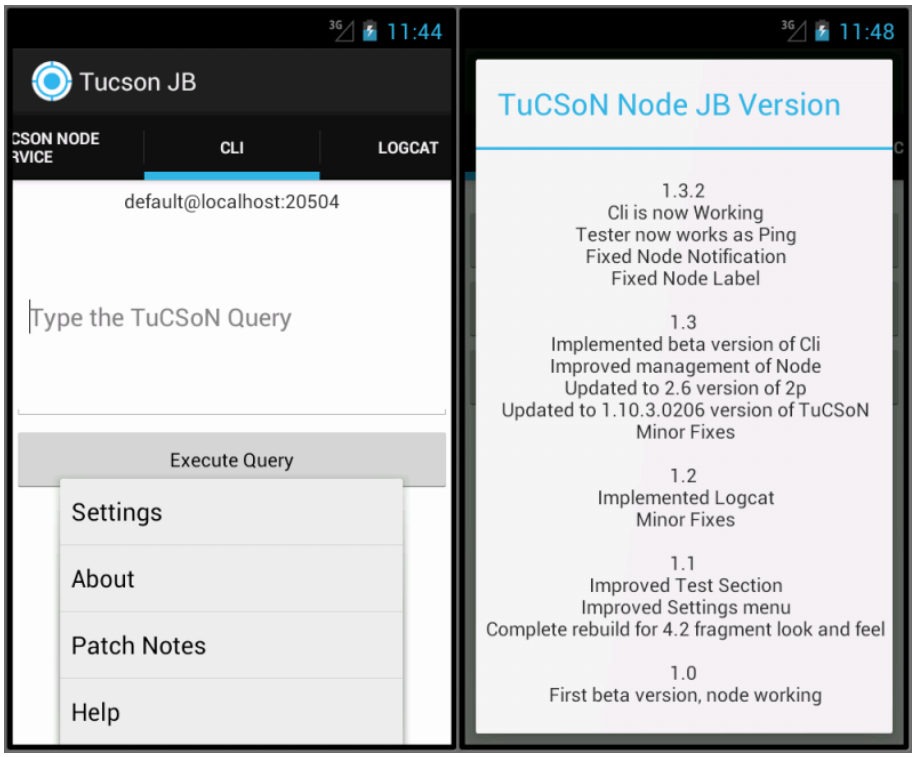

Figura 4.4: Il menu dell'applicazione sulla sinistra, il popup contenente le Patch Notes sulla destra, realizzato tramite fragment.

I settings sono anch'essi automatizzati in Android: tramite le API viene fornita già una Activity predefinita da utilizzare (chiamata `SettingsActivity` compresa di Fragments (`PreferenceFragment`) che disegna il layout delle opzioni seguendo gli standard Android, evitandoci di dover creare il layout per questa activity. Le preferenze vengono invece specificate in un file xml, nel quale possiamo specificare il tipo di preferenza, che dato deve contenere, cosa utilizzare di default etc.

Una volta specificata una preferenza essa rimarrà invariata anche a lanci successivi, rimanendo salvata in memoria (viene infatti consigliato di utilizzare i settings per tutti quei dati che non vengono cambiati spesso), ed il suo valore è accessibile da qualsiasi punto dell'applicazione tramite il `PreferenceManager`.

In questo modo abbiamo potuto aggiungere varie opzioni, come vediamo in figura 4.5: possiamo ora lanciare il nodo in qualsiasi porta (esistente) del sistema, possiamo utilizzare il CLI come semplice agente specificando nelle opzioni ip e porta di un nodo remoto, oppure possiamo effettuare test di ping alla stessa maniera su qualsiasi nodo remoto.

In questo modo abbiamo slegato completamente le varie funzionalità offerte dall'app: possiamo infatti ora utilizzare per esempio il CLI su un

nodo remoto come un semplice agente, mentre contemporaneamente il nodo sul device potrebbe essere spento, allargando in questo modo le funzionalità offerte dall'applicazione.

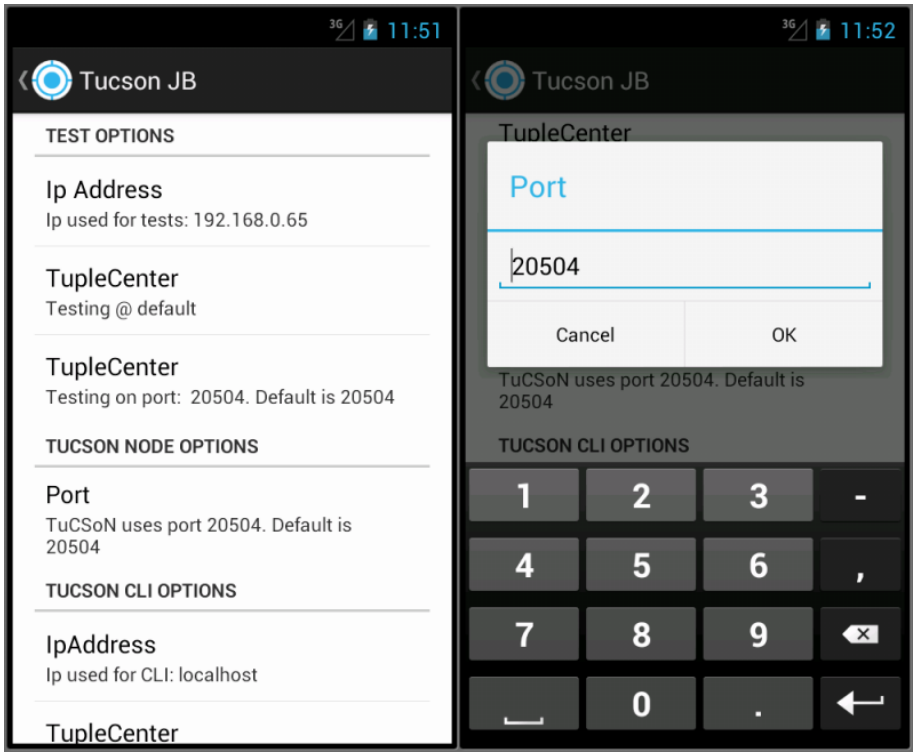

Figura 4.5: Sulla sinistra vediamo le opzioni disponibili nell'applicazione, divise per categorie, mentre sulla destra un esempio del popup di inserimento. Notiamo che nell'esempio della porta, essendo un intero, la tastiera Android in automatico ci fornisce solamente i numeri. È stata comunque implementata una routine di controllo per evitare valori non utilizzabili (come porte protette o non esistenti). Notiamo inoltre che l'`ActionBar` in alto è diventata cliccabile, riportandoci alla home dell'applicazione.

Parte III

Conclusioni

Capitolo 5

L'applicazione

Giunti al risultato ottenuto, possiamo considerarci soddisfatti dell'applicazione ottenuta.

È ora possibile progettare un qualsiasi sistema TuCSoN includendo, ove sensato e possibile, device mobili Android, dando la possibilità al progettista di creare un server mobile, sfruttando la possibilità di lanciare un nodo TuCSoN. Uno scenario del genere può risultare interessante nel caso in cui si voglia creare un sistema con estrema mobilità.

Ad esempio gli smartphone Android sono in grado di generare un hotspot wi-fi: immaginando scenari di emergenza, in cui si debba organizzare e coordinare un lavoro, ed una mancanza totale di infrastruttura a supporto di una rete internet, magari data dall'emergenza stessa, tramite degli smartphone Android (e magari un extender wifi in caso serva coprire un raggio ampio) sarebbe possibile creare un sistema di telecomunicazione semplice, rapido ed efficace.

Ciò non deve tuttavia far pensare che il lavoro svolto sia concluso: sarà neccessario mantenere l'applicazione, aggiornando TuCSoN e tuProlog conseguentemente all'uscita delle nuove versioni, effettuando bugfix ove richiesto e sistemando problemi che inevitabilmente insorgeranno[1]

Dovrà inoltre essere sviluppata ulteriormente l'applicazione conseguentemente agli aggiornamenti del sistema Android, per mantenere un'applicazione usabile, semplice e veloce, come è successo con l'introduzione dei Fragment. Se ad esempio avessimo deciso di intraprendere questo percorso un anno fa, i fragment sarebbero risultati un concetto astratto, non essendo ancora implementati stabilmente nel sistema Android; questo ci avrebbe costretto a creare una applicazione strutturata interamente in activity, dando ad esempio maggiore importanza all'AndroidManifest e agli Intent, che avrebbero sicuramente ricoperto un ruolo centrale nello sviluppo dell'applicazione.

[1] Vedi il passaggio da Java 6 a Java 7.

Non mancano inoltre spunti di lavoro futuri per allargare le funzionalità dell'applicazione, o aggiungerne di nuove, tramite sia estensione dell'applicazione Android sia fornendo nuove API in TuCSoN stesso.

Nei capitoli seguenti vengono proposte alcuni semplici spunti di lavoro su cosa si potrebbe aggiungere alle tecnologie utilizzate, o come si intende mantenere e supportare il lavoro svolto nel futuro prossimo.

Capitolo 6

Supporto Futuro

Come appena detto, un lavoro come quello svolto durante il percorso della tesi non può essere considerato *una tantum* per il tipo di tecnologie utilizzate: esse infatti sono in continuo sviluppo ed in continua crescita. Sarà pertanto necessario, con il passare del tempo, continuare a lavorare, estendere e migliorare l'applicazione creata.

Ad esempio, sebbene la versione di Android presa come base per la progettazione e costruzione dell'applicazione sia relativamente nuova (Android 4.1 Jelly Bean è stato presentato il 27 Giugno 2012, ed è stato rilasciato al pubblico il 9 Luglio 2012), l'uscita di Android 5.0 è annunciata già per la primavera 2013; e sebbene le informazioni disponibili su questa nuova versione siano ancora limitate, è possibile (e probabile) che questa nuova versione richiederà migliorie e lavoro sull'applicazione appena creata allo scopo di mantenerla al passo con la versione più efficiente di Android, garantendo, al contempo, la massima retrocompatibilità possibile con le vecchie versioni di Android.

La difficoltà nello sviluppo per una piattaforma come Android risiede anche nella frammentazione della distribuzione del sistema stesso: la necessità di dare supporto all'ultima versione disponibile si scontra necessariamente con la necessità di dare supporto al numero più ampio possibile di dispositivi, spesso non aggiornati alla versione più recente.

Allo stesso modo sarà necessario aggiornare l'applicazione ad ogni uscita di una nuova versione di TuCSoN o di tuProlog, affinché l'utente abbia sempre le versioni più aggiornate possibili, comunicando allo staff di sviluppo di queste ultime tecnologie in caso parti di codice implementato causino problemi in Android, in modo che possano essere modificate rendendole utilizzabili sia su Android sia su PC.

6.1 Possibili estensioni dell'applicazione

Collegandoci al discorso precedente, oltre all'aggiornamento delle tecnologie utilizzate e conseguente aggiornamento dell'applicazione, sarà possibile, in futuro, estendere le funzionalità dell'applicazione già presenti, o aggiungerne di nuove, come già iniziato a fare all'interno della tesi con l'implementazione del CLI o del LogCat; a differenza dei punti precedentemente descritti questo non dipende dalla versione di Android, TuCSoN, o tuProlog, bensì si tratta di ipotetiche funzionalità non ancora previste né implementate, spesso scaturite dai confronti e dalle discussioni avute durante lo sviluppo dell'applicazione con il team di sviluppo di TuCSoN ma alla fine non state implementate.

Vengono ora presentati alcuni degli spunti più interessanti sorti da tali discussioni, con una breve descrizione.

6.1.1 Divisione in più service del nodo TuCSoN

Durante lo studio dei Service e del loro ciclo di vita ci si è posto il problema riguardo alle interruzioni non previste date dal sistema Android.

Come descritto in precedenza, infatti, Android possiede una lista di priorità, contentente tutti i processi attivi sul sistema, ordinati secondo la loro importanza; questa viene utilizzata per decidere che processi terminare in caso di necessità di risorse, seguendo l'ordine dettato da tale lista.

In Android non si ha quindi mai la certezza che un processo non venga mai terminato dal sistema, in quanto ogni processo è presente nella lista, ed in quanto tale possiede virtualmente la possibiltà di ritrovarsi in cima alla stessa.

Quando ci si è posto il problema dell'implementazione di un server, ci si è resi conto che si aveva la necessità che questo non fosse mai interrotto; a questo scopo si era ipotizzato di dividere il nodo TuCSoN in più service che riflettessero la divisione stessa del nodo, creando cioè un service contenente i vari `ACCProxyNodeSide` e l'`ACCProvider`, un secondo service per il `WelcomeAgent` ed infine un ultimo service per il `NodeManagementAgent`. Si era pensato in questo modo di lanciare in *foreground* solamente il `WelcomeAgent`, rendendo gli altri due service per così dire sacrificabili.

In questo modo si sarebbe ottenuto ad agire in *foreground* un solo service molto leggero a livello di risorse hardware richieste: immaginando uno scenario in cui un sistema Adroid necessita di risorse, avremmo ottenuto l'interruzione di tutti i service che comunicano effettivamente con il nodo, ma vista la scarsità di risorse richieste dal `WelcomeAgent` avremmo probabilmente il nodo ancora funzionante ed attivo per ricevere eventuali richieste in entrata.

Si potrebbe quindi salvare in qualche modo le richieste da eseguire, in modo da effettuarle in un secondo momento quando il sistema Android potrà

rilanciare i vari service, avendo ancora diponibilità di risorse, facendo tornare il nodo effettivamente funzionante.

Il motivo per cui si è deciso di evitare questo tipo di implementazione è sia la relativa leggerezza del sistema TuCSoN, che probabilmente avrebbe occupato più risorse in esecuzione per il passaggio di dati fra service diversi rispetto all'implementazione utilizzata attualmente in cui tutto risiede sullo stesso service, sia perché il trend tecnologico sta portando a dispositivi sempre più potenti dal punto di vista hardware, per cui uno scenario in cui il nodo TuCSoN raggiunge la prima posizione della lista è risultato altamente improbabile.

6.1.2 Salvataggio dati in caso di interruzione del nodo

Dal punto precedente è nata successivamente l'idea di rendere i centri di tuple salvabili e ripristinabili in seguito.

Allo stato attuale TuCSoN non ha memoria: una volta interrotto il nodo, tutti i centri di tuple e le tuple stesse vengono cancellate irrimediabilmente.

Sebbene TuCSoN fornisca delle API che rendono possibile la lettura di tutte le tuple all'interno di uno spazio di tuple, rendendo di fatto possibile il salvataggio dei dati, si è vagliata l'ipotesi di fornire a livello di API TuCSoN la possibilità di generare un file rappresentante lo stato attuale di un nodo TuCSoN, nonché la possibilità di lanciare un nodo fornendo un file dello stesso tipo in modo da ripristinare un nodo ad una situazione qualsiasi precedente.

Questa sorta di backup è ora realizzabile solo manualmente, tramite una lettura manuale di tutte le tuple presenti sfruttando l'API appena menzionata ed un reinserimento successivo una volta rilanciato il nodo.

La sua utilità sarebbe molteplice, tuttavia in primis si potrebbe generare un file di questo tipo nella rara (seppur possibile) situazione in cui il sistema Android decidesse di terminare l'esecuzione del nodo TuCSoN; si potrebbe, in quel caso, effettuare la chiamata dell'API all'interno del metodo `onDestroy()` del service, in modo che all'avvio successivo si possa scegliere se lanciare un nodo nuovo o se caricare un qualche file precedentemente creato, ripristinando il sistema ad una configurazione passata.

6.1.3 Implementazione file config.rsp custom al lancio del nodo

Relativamente al lancio di un nodo TuCSoN, allo stato attuale il nodo viene sempre lanciato tramite il file di configurazione standard, fornito dal .jar di TuCSoN stesso.

Si potrebbe utilizzare la memoria SD, presente in Android, per leggere eventuali file .rsp e fornire all'utente la possibilità di lanciare il nodo con

file di configurazione personalizzati: definendo all'interno delle opzioni il percorso al file da utilizzare.

Ciò richiederebbe l'utilizzo di un `intent` per sfruttare un'applicazione esterna, allo scopo di recuperare il file dalla memoria SD, nonché di garantire i permessi all'applicazione di accedere alla memoria del cellulare, permessi di cui ora non necessitiamo.

6.1.4 Implementazione Inspector

Una nuova funzionalità che sarebbe interessante aggiungere all'applicazione sarebbe l'Inspector.

L'Inspector è un tool fornito dal package standard TuCSoN che permette, tramite una semplice interfaccia grafica, di leggere le tuple presenti in un nodo TuCSoN, modificarle, modificare le reazioni ReSpecT [1], vedere le operazioni da eseguire (come ad esempio una in quando non è ancora presente una tupla all'interno dello spazio di tuple) etc.

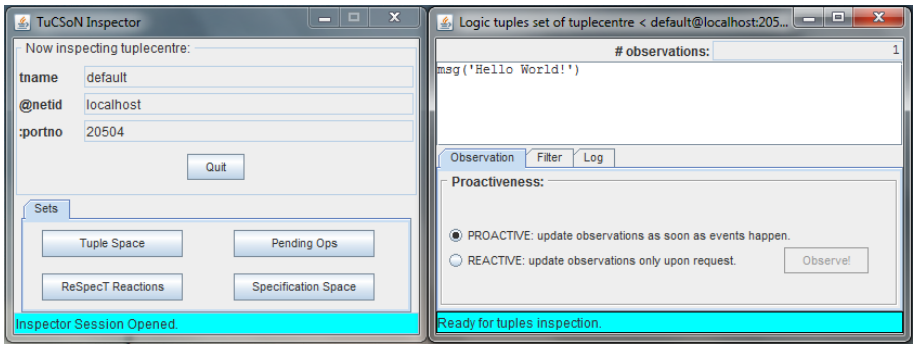

Figura 6.1: Uno screenshot dell'inspector. Sulla destra la schermata ottenuta cliccando tuple space, che mostra le tuple presenti nel centro di tuple `default`.

La difficoltà principale in un eventuale *porting* su Android risiede nella ricostruzione completa della GUI: nella sua implementazione esistente infatti l'Inspector fa ampio utilizzo della libreria `Swing` di java, non presente all'interno di Android, in quanto quest'ultimo utilizza dei file xml per generare le proprie GUI.

6.1.5 Creazione di un Agent stand-alone

Viste le criticità del *porting* di TuCSoN in Android, e visto che nell'applicazione è già presente un CLI liberamente utilizzabile (il quale è a tutti gli effetti un agente TuCSoN) si può ora considerare possibile e completamente realizzabile un sistema TuCSoN completamente mobile: si può considerare

[1] non trattate in questa tesi.

l'ipotesi quindi di progettare sistemi in cui un device Android, qualunque esso sia, ha la funzione di semplice agente TuCSoN.

Si può ad esempio pensare ad un sistema di ordini per un locale: i camerieri che prendono l'ordine possono registrare quest'ultimo, tramite il proprio device Android, in un nodo TuCSoN, rendendo le informazioni instantaneamente disponibili nella cucina o nel bar del locale.

Oppure si può immaginare ad una semplice chat, in cui possono essere disponibili client sia web, sia applicazioni java, sia mobile Android.

A questo punto è facile intuire come la mobilità di un dispositivo Android apra nuovi scenari in cui TuCSoN può risultare utile e facilmente utilizzabile come scheletro per la progettazione di un sistema distribuito.

6.1.6 Aggiunta localizzazione agente TuCSoN

Allo scopo sopra descritto può risultare utile, infine, fornire ad un agente TuCSoN coscienza del fatto che questo è mobile.

Si può immagine un sistema TuCSoN in cui gli agenti, tramite il proprio modulo gps, possono capire quale è il nodo più vicino a loro, non più quindi identificato solamente dal proprio indirizzo IP ma anche dalla propria posizione geografica, allo scopo di creare sistemi topologicamente complessi.

Si pensi ad esempio ad un sistema di autotrasporti, in cui un autobus può avere un dispositivo Android. Un altro dispositivo può in questo modo monitorare la posizione dell'autobus e scrivere, all'interno dei suoi centri di tuple, informazioni come il numero di passeggeri, le fermate, o altre informazioni utili.

In conclusione, è stato raggiunto lo scopo prefissatoci all'inizio di questo percorso: ampliare il numero di device possibili supportati dal *middleware* TuCSoN, utilizzabile come scheletro per sistemi distribuiti anche molto complessi, dandogli la possibilità di usare in detti sistemi anche dispositivi estremamente mobili, quali sono dei semplici smartphone, senza che il sistema stesso risenta di queste differenze, anzi, potendole sfruttare ove opportuno per aggiungere funzionalità al sistema non possibili con l'utilizzo escluso di semplici desktop/laptop.

Parte IV

Appendice

Capitolo 7

Ringraziamenti

Il momento più atteso e tuttavia più complicato di un intero percorso universitario risulta probabilmente essere la scrittura di questo specifico capitolo all'interno della tesi.

O, almeno, così risulta essere per me.

Dopo giorni passati conoscendo nuove persone, imparando nuove cose, notti passate insonni fra codici non funzionanti, dimostrazioni matematiche, libri e dispense, cercare di mettere a fuoco chi ringraziare al completamento del percorso risulta essere più difficile che superare un esame.

Primi fra tutti vorrei ringraziare i docenti della seconda facoltà di Ingegneria dell'Università di Bologna, in particolar modo il professore Andrea Omicini ed il suo assistene Stefano Mariani, che mi hanno aiutato nel compimento di questo lavoro, sopportato le mie numerose email e mi hanno sempre risposto con gentilezza e precisione, guidandomi fino alla fine di questo percorso.

Poi vorrei ringraziare i miei genitori, che per primi hanno creduto in me e al fatto che sarei riuscito ad arrivare in fondo, anche in momenti in cui lo studio era opprimente e gli esami non davano i risultati sperati.

A Enrico, che ha sempre trovato un modo diverso ed interessante per disturbarmi e distrarmi dallo studio, di fatto togliendomi il problema di quando e per quanto fare delle pause dai libri (anche se spesso troppo frequenti e non richieste).

A Rachele, che mi ha sopportato nei numerosi momenti di agitazione ed ansia che inevitabilmente si impossessavano di me con l'avvicinarsi di certi esami, rendendomi di fatto intrattabile ad ogni singola sessione.

Agli amici, che al contrario dei precedenti hanno fatto di tutto per impedirmi di compiere questo percorso, per questo li ringrazio tutti: spesso staccare la spina è più efficace che ore di studio; non posso nominare ad uno ad uno le persone coinvolte, in quanto penso servirebbe un'altra tesi

solo a questo scopo, ma chiunque si è presentato volontario ad un qualsiasi marafone organizzato illegalmente in sala studio è sicuramente compreso nell'elenco, o chi mi ha trascinato fuori casa la sera prima di un esame per bere una birra in centro, o chi mi ha fatto fare tardi in giro per locali e feste in spiaggia in settimane in cui le mie energie sarebbero servite nello studio.

Ed infine a tutte quelle persone che, in un modo o nell'altro, hanno aiutato il percorso di crescita personale che mi ha portato dall'essere un liceale al diventare un ingegnere.

Grazie.

Bibliografia

[1] Stefano Mariani Andrea Omicini. *The TUCSoN Coordination Model & Technology, a guide.*, 1.0.2 edition. TuCSoN official manual.

[2] code.google.com/p/tucson/. TuCSoN on googlecode.

[3] Google Inc. android.com. Official Android Website.

[4] GooGle Inc. *developer.android.com.* Official Android Developer WebSite.

[5] tucson.apice.unibo.it. Official TuCSoN website.

[6] alice.unibo.it/xwiki/bin/view/Tuprolog/. Official tuProlog website.

[7] code.google.com/p/tuprolog. tuProlog on googlecode.

[8] Antonio Pedone. mobile TuCSoN: theoretical and technological requirements for TuCSoN's porting over android mobile devices. Master's thesis, Università di Bologna, sede di Cesena, 2011.

[9] William Enick, Machigar Ongtang, and Patrick McDaniel. Understanding android security. Technical report, Pennsylvania State University, 2012.

[10] mobworld. Memory management in android. Small article regarding memory management in Android, 2010.