UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale in Ingegneria Informatica

Realizzazione di tecniche parallele di generazione automatica di casi di test per applicazioni Android

Anno Accademico 2013/2014

relatore

Ch.mo Prof. Porfirio Tramontana

correlatore
Ing. Nicola Amatucci

candidato
Giuseppe Di Maio
matr. M63000026

Alla mia famiglia e ai miei nonni

Indice

Introduzione

Smartphones e tablets dotati di GUI (Graphic User Interfaces) stanno diventando sempre più popolari. Centinaia di migliaia di applicazioni specializzate, dette apps, sono disponibili per tali piattaforme mobili. Il testing manuale è la tecnica più popolare per testare le interfacce grafiche di tali applicazioni ma risulta noioso e propenso all'errore. Un' alternativa all'approccio manuale è rappresentata dal testing automatico. Le tecniche per la generazione automatica dei casi di test possono ridurre drasticamente i costi e i tempi legati alla fase di test design.[11] Le tecniche di GUI Testing possono dividersi in tre categorie:

- Tecniche Model Based, in cui esiste una descrizione dell'applicazione sotto test sufficiente alla generazione automatica dei Test Case per la verifica del software;

- Tecniche Random Testing, in assenza di un modello, l'applicazione viene esercitata in maniera casuale alla ricerca di eventuali eccezioni non gestite;

- Tecniche Model Learning, in cui l'applicazione viene esplorata senza una conoscenza pregressa della sua struttura seguendo una strategia di navigazione.

Questo lavoro presenta una tecnica di GUI testing automatico basata su di un ripper che automaticamente esplora l'interfaccia grafica con l'obiettivo di mettere in esercizio l'applicazione e rivelare crash a run-time. Inoltre il ripper costruisce un GUI model ed una test suite eseguibile basata sul framework JUnit.

Scopo del lavoro di tesi è quello di aumentare l'efficienza di questo strumento rendendolo parallelo e funzionante da remoto. La parallelizzazione permette di aumentare il numero di macchine che svolgono contemporaneamente il lavoro di testing ottendendo in questo modo una notevole riduzione dei tempi necessari allo svolgimento dell'intero processo. Il funzionamento da remoto invece, permette ad un utente qualsiasi di accedere allo strumento di testing automatico, installare la propria applicazione, e avviare il processo per poi ottenere i risultati in termini di copertura.

La tesi è cosi strutturata: un primo capitolo che tratterà le generalità del sistema Android e relative problematiche legate al testing. Un secondo capitolo introdurrà diversi strumenti per il testing automatico per poi descrivere brevemente il funzionamento del GUI ripper.
Il terzo capitolo entra nel vivo del nostro lavoro di tesi in quanto descriverà l'algoritmo per la parallelizzazione e per il funzionamento da remoto sia per strategia sistematica che per quella randomica. Il quarto capitolo descriverà la web application tramite la quale l'utente potrà accedere al sistema e, scegliendo fra varie opzioni, avrà modo di testare la propria applicazione. Infine saranno presentate le considerazioni conclusive sull'architettura realizzata, i possibili sviluppi futuri ed un appendice contente l' "How-To" ovvero una guida per l'installazione dell'intero sistema sulle macchine desiderate.

Capitolo 1: Android e testing di applicazioni

In questo capitolo forniremo un panoramica generale sul sistema Android descrivendone le caretteristiche ed illustrando il contesto nel quale nasce ed intende affermarsi. Successivamente verranno introdotti i concetti principali relativi alla tematica del testing di applicazioni Android.

1.1 Perchè Android

La rivoluzione dei dispositivi mobili nell'ultimo decennio, ha cambiato il concetto di fruibilità di quei contenuti e servizi che prima erano accessibili solo da un PC. In questo contesto i principali costruttori di dispositivi mobili hanno messo a disposizione degli sviluppatori i propri sistemi operativi, ciascuno con il proprio ambiente di sviluppo, i propri tool ed il proprio linguaggio di programmazione. Nessuno di questi però si è affermato come standard. Per esempio, per sviluppare una applicazione per iphone è necessario disporre di un sistema operativo Mac OS X, oltre che la conoscenza di linguaggio Objective-C. Questo tipo di "imposizione" (S.O. - linguaggio) ha portato alcuni produttori di cellulari a perdere una consistente fetta di mercato come ad esempio la Nokia con S.O. Symbian e linguaggio simil C++.

In realtà, il concetto da sottolineare è che per sviluppare un applicazione per un particolare dispositivo mobile, a seconda del sistema operativo è necessario acquisire la conoscenza di un ambiente, di una piattaforma e di un linguaggio. Esiste quindi la necessità di una standardizzazione verso la quale si sono diretti Google e la Open Handset Alliance (OHA), una organizzazione composta dalle principali aziende mondiali di telefonia come

Motorola, Samsung e Sony-Ericsson, dalle principali compagnie di telefonia come Vodafone e T-mobile, da costruttori hardware come Intel e Texas Instruments, oltre che ovviamente da Google.

Tra i tanti sistemi operativi presenti, uno di cui vale la pena parlare è Android. Tutto il mondo Android ruota attorno alla parola open source: infatti si ha libero accesso a tutti gli strumenti utilizzati dagli sviluppatori stessi di Android e quindi il potenziale delle applicazioni sviluppate da terzi non è soggetto a limitazioni. Inoltre, chiunque può sviluppare per Android senza dover acquisire software o licenze particolari. [1] [2]

1.2 Il sistema Android

Android, è ,una piattaforma di esecuzione gratuita per dispositivi mobili, creata nel 2003 da Andy Rubin e acquistata da Google nel 2005. Va fatta da subito una precisazione sul termine piattaforma, spesso abusato: una piattaforma è una base software/hardware sulla quale vengono eseguite o sviluppate applicazioni. E' importante quindi distinguere le piattaforme di esecuzione, cioè quelle dedicate all'esecuzione di programmi, comunemente note come sistema operativo, da quelle di sviluppo, cioè quelle utilizzate per sviluppare programmi. I due concetti non sono comunque così distanti: una piattaforma di sviluppo solitamente è anche di esecuzione, ma non necessariamente una piattaforma di esecuzione è anche di sviluppo. Nel mercato sono attualmente disponibili molte piattaforme di esecuzione, come Windows Phone, iOS, Symbian, LiMo, Bada, BlackBerry OS: perchè allora molti sviluppatori scelgono proprio Android? Quando un programmatore decide di sviluppare un'app, deve anche scegliere su quale piattaforma questa potrà essere eseguita. Si ritrova quindi davanti a due strade:

Scegliere un linguaggio come Java, per Android, che è conosciuto, diffuso e molto utilizzato ma che dà forse troppe libertà.

Scegliere un linguaggio meno flessibile ma che permetta di concentrarsi di più sul programma e meno sull'interfaccia, in quanto questa sarà gestita dal sistema operativo. Anche qui si arriva ad un bivio: il C# di Windows Phone oppure l'Objective-C di Apple.

Il linguaggio Java, rispetto al C, gestisce automaticamente il ciclo di vita di un'applicazione e anche l'allocazione della memoria, rendendo più semplice lo sviluppo. Android inoltre è completamente open source, quindi rende possible reperire, modificare e ridistribuire il codice sorgente adattandolo ad ogni dispositivo e creandone la propria versione personalizzata ed ottimizzata. E' anche il sistema operativo mobile più diffuso, ha una gestione di grafica e audio di alta qualità, le applicazioni native e di terze parti sono di uguale importanza e possono essere aggiunte o rimosse senza alcun vincolo. La scelta di Android non porta tuttavia solo vantaggi dal punto di vista informatico, ma anche dal punto di vista economico. Pubblicare applicazioni sullo store di Android (Google Play), ha un costo di iscrizione di 25$ ed è possibile programmare l'app con tutti i sistemi operativi per desktop più diffusi: Windows, OS X e Linux. Scegliere iOS invece comporta un costo 4 volte superiore per l'iscrizione all'AppStore e serve necessariamente un computer Mac per programmare, anche questo molto costoso.

1.3 L'architettura di Android

Il sistema operativo Android è basato su kernel Linux e consiste in una struttura formata da vari livelli o layer, ognuno dei quali fornisce al livello superiore un'astrazione del sistema sottostante. I layer principali sono quattro:

- Linux Kernel
- Libraries
- Application Framework
- Applications

Ciò si può vedere dall'immagine:

Figura 1.1: Architettura di Android

1.3.1 Linux Kernel

Il sistema è basato su Kernel Linux, inizialmente 2.6 poi 3.X, che costituisce il livello di astrazione di tutto l'hardware sottostante: include cioè i driver per la gestione del WiFi, Bluetooth, GPS, fotocamera, touchscreen, audio, USB driver, ecc. Ogni produttore di smartphone dovrà modificare in primo luogo questa parte per renderlo compatibile con l'hardware scelto. Anche la creazione di una custom ROM spesso ha bisogno di modifiche a questo livello, in quanto spesso vengono effettuati dei porting delle ultime versioni di Android su dispositivi che non riceverebbero più aggiornamenti uffciali. L'astrazione hardware permette ai livelli superiori e agli sviluppatori una programmazione ad alto livello senza preoccuparsi del tipo di hardware che monta il telefono. A differenza di un kernel Linux standard, in quello di Android sono stati aggiunti ulteriori moduli per la miglior adattabilità a dispositivi mobili, come:

- Binder (IPC) Driver: driver dedicato che permette la comunicazione tra processi con un costo computazionale minore e quindi un un più basso consumo di batteria
- Low Memory Killer: modulo che si occupa di terminare i processi in mododa liberare spazio nella memoria centrale per soddisfare le richieste di altri processi. La terminazione avviene sulla base di un sistema di ranking che assegna dei

punteggi in base all'importanza dei processi. Il processo init2 non può essere terminato.

- Android Debug Bridge: strumento che permette di gestire in maniera versatile un'istanza dell'emulatore o di un dispositivo reale.
- RAM Console e Log devices: modulo per agevolare il debugging. Android fornisce infatti la possibilità di memorizzare messaggi di log generati dal kernel in un buffer RAM, in modo da poter essere osservati dagli sviluppatori per trovare eventuali bug.
- Ashmem: sistema di memoria condiviso anonimo (Anonimous Shared Memory) che definisce interfacce che permettono ai processi di condividere zone di memoria attraverso un nome.
- Power Management: sezione progettata per permettere alla CPU di adattare il proprio funzionamento per non consumare energia se nessuna applicazione o servizio ne fa richiesta.

1.3.2 Libraries

Il livello superiore riguarda le librerie C/C++ che vengono utilizzate da vari component del sistema. Tra esse troviamo:

- **Il Surface Manager**: modulo che gestisce le View, cioè i componenti di un'interfaccia grafica.
- **Il Media Framework**: si occupa dei Codec per l'acquisizione e riproduzione di contenuti audio e video.
- **SQLite**: il Database Management System (DBMS) utilizzato da Android. E' un DBMS relazionale piccolo ed effciente messo a disposizione dello sviluppatore per la memorizzazione dei dati nelle varie applicazioni sviluppate.
- **FreeType**: un motore di rendering dei font.
- **LibWebCore**: un browser-engine basato su WebKit, open source, che può essere integrato in qualunque applicazione sotto forma di finestra browser.

- **SGL e OpenGL ES**: librerie per gestire rispettivamente grafica 2D e 3D.
- **SSL**: libreria per il Secure Socket Layer.
- **LibC**: implementazione della libreria di sistema standard C, ottimizzata per dispositivi basati su versioni embedded di Linux.

1.3.3 Android Runtime

In questo livello, di grande rilievo c'è l'ambiente di Runtime, costituito dalla Core Library e la Dalvik Virtual Machine: insieme costituiscono l'ambiente di sviluppo per Android. Le Core Libraries includono buona parte delle funzionalità fornite dalle librerie standard di Java a cui sono state aggiunte librerie specifiche di Android. La Dalvik Virtual Machine (DVM) è invece un'evoluzione della Java Virtual Machine sviluppata da Google in cooperazione con altri marchi noti come ad esempio nVidia e Intel, in modo da lavorare su dispositivi poco performanti come i cellulari. Bisogna tuttavia sottolineare che con gli ultimi modelli di smartphone si sono raggiunte delle prestazioni veramente elevate (processori superiori a 2 GHz e memorie RAM da 2 GB), paragonabili a dei notebook di fascia medio-bassa. Vediamo più nel dettaglio come è organizzata la Dalvik Virtual Machine. Per poterlo capire dobbiamo dapprima chiarire il concetto di macchina virtuale: una VM è un software che può essere un sistema operativo, un emulatore o una virtualizzazione hardware completa.

Ci possono essere due tipi di macchine virtuali: System VM e Process VM. La prima supporta l'esecuzione di un sistema operativo completo mentre la seconda supporta solo l'esecuzione di processi singoli. Rappresentandole con dei layer si troverebbe che la System VM è al di sopra della Process VM.

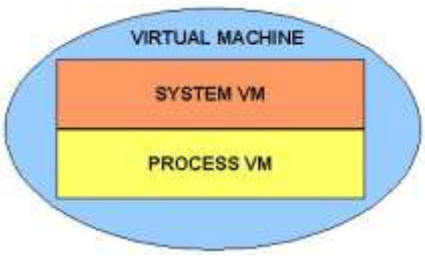

Figura 1.2: System e Process Machine

E' utile inoltre chiarire la differenza tra Stack Based VM e Registered-Based VM. La prima è orientata, come dice il nome, all'uso dello stack mentre la seconda è orientata all'uso dei registri. Se inizialmente, per facilità di implementazione, si utilizzava la Stack Based, ora invece si preferisce la Registered-Based in quanto riduce di molto il numero di istruzioni.

La DVM è una Process Virtual Machine che adotta proprio l'approccio Registered Based, a differenza della JVM che utilizza l'approccio più vecchio, quello Stack Based. E' quindi veloce in esecuzione ma leggermente più lenta nella creazione del bytecode. Ad ogni applicazione viene associato un processo che gira all'interno di una DVM ad esso dedicata, ed il sistema gestisce più macchine virtuali contemporaneamente per rendere possibile il multitasking. Il bytecode costituisce l'insieme delle operazioni che la macchina dovrà eseguire e viene scritto una sola volta. Quello della DVM deriva da quello della JVM: viene creato da un file .class un file .dex (Dalvik EXecutable), che non è altro che una modifica del bytecode della JVM a cui vengono tolte le parti ripetute più volte. Da alcune versioni di Android inoltre (in particolare dalla 2.2), la DVM implementa un Just in Time Compiler (JIT), che attraverso il riconoscimento di alcune parti di codice Java, esegue una traduzione in parti di codice nativo C o C++ rendendo l'esecuzione molto più veloce. Riepilogando, i motivi per cui viene adottata la DVM sono: minor spazio occupato dal bytecode, maggior velocità in esecuzione rispetto ad una Stack Based VM.

Sempre relativamente al kernel, quello di Linux di Android è un sistema multi-utente nel quale ogni applicazione è un utente differente. Il sistema infatti crea uno user ID distinto per ogni applicazione e imposta i permessi dei file dell'applicazione stessa in modo che solo quell'ID possa accedervi. Inoltre, ogni applicazione sul telefono viene lanciata in un processo Linux a sè stante all'interno della propria istanza della JVM: questa architettura a sandbox garantisce la stabilità del telefono nel caso in cui qualche applicazione crei problemi. Tutto questo non limita però lapossibilità di interazione tra i processi: permette anzi loro di condividere lo stesso user ID e la stessa JVM in modo da preservare la coerenza delle risorse di sistema.

1.3.4 Application Framework

In questo livello è possibile rintracciare i gestori e le applicazioni di base del sistema. In questo modo gli sviluppatori possono concentrarsi nella risoluzione di problem non ancora affrontati, avendo sempre a propria disposizione il lavoro già svolto da altri. Tra i vari gestori presenti troviamo:

- **Activity Manager**: gestisce tutto il ciclo di vita delle Activity. Le Activity sono entità associate ad una schermata dell'applicazione. Il compito dell' ActivityManager è quindi quello di gestire le varie Activity sul display del terminale e di organizzarle in uno stack in base all'ordine di visualizzazione sullo schermo. I concetti di Activity e ciclo di vita dell'Activity saranno esposti in modo più esaustivo successivamente.
- **Content Providers**: gestiscono la condivisione di informazioni tra i vari processiattivi.
- **Window Manager**: gestisce le finestre relative a differenti applicazioni.
- **Il Telephony Manager**: gestisce le funzioni base del telefono quali chiamate ed SMS.
- **Resource Manager**: gestisce tutte le informazioni relative ad una applicazione (file di configurazione, file di definizione dei layout, immagini utilizzate, ecc).
- **Package Manager**: gestisce i processi di installazione e rimozione delle applicazioni dal sistema.
- **Location Managaer**: mette a disposizione dello sviluppatore una serie di API che si occupano della localizzazione (tramite GPS, rete cellulare o WiFi).
- **System View**: gestisce l'insieme degli elementi grafici utilizzati nella costruzione dell'interfaccia verso l'utente (bottoni, griglie, text boxes, ecc. . .).
- **Notification Manager**: gestisce le informazioni di notifica tra il dispositivo e l'utente. Le notifiche possono consistere in vari eventi: la comparsa di un'icona nella barra di notifica, l'accensione del LED del dispositivo, l'attivazione della

retroilluminazione del display, la riproduzione di suoni o vibrazioni.

1.3.5 Applications

Al livello più alto risiedono le applicazioni utente. Le funzionalità base del sistema, come per esempio il telefono, il calendario, la rubrica, non sono altro che applicazioni scritte in Java che girano ognuna nella propria DVM. E' bene notare che Android non differenzia le applicazioni di terze parti da quelle già incluse di default, infatti garantisce gli stessi privilegi a entrambe le categorie [4].

1.4 Testing di applicazioni

Testare un software significa eseguirlo con l'intento di trovarvi un malfunzionamento.[5] Per malfunzionamento si intende la manifesta incapacità del sistema, o di una sua componente, a svolgere la funziona richiesta nel rispetto di specifici vincoli prestazionali. Le operazioni della fase di test si possono dividere in test di convalida, il cui scopo è verificare che il software implementi esattamente le funzionalità richieste, ed in test di verifica, il cui scopo è la ricerca di difetti. È soprattutto in quest'ultimo caso che l'automazione può giocare un ruolo fondamentale; il test di convalida, invece, deve basarsi su di un piano e su procedure approvate dal cliente. Le principali tipologie di test sono:

- **Regression Test**: è la ri-esecuzione (di un sottoinsieme) dei test già condotti su una versione precedente del software. Si applica in seguito ad un intervento di manutenzione su di un software esistente, per il quale esiste già un piano di test. Il tester vuole assicurarsi che la correzione di un bug individuato in una fase di test precedente, non abbia di fatto introdotto qualche nuovo problema.

- **Crash Test**: il crash test si occupa di individuare quei comportamenti inaccettabili (crash di applicazione, crash di sistema, interfaccia che non risponde ai comandi, etc...) che non possono essere ritenuti accettabili anche se non sono

necessariamente presenti nelle specifiche dei vincoli di sistema. L'obiettivo è ricercare eccezioni o errori a runtime che interrompano l'esecuzione.

- **Smoke Test**: Una varietà del crash testing, nella quale l'applicazione viene esplorata e navigata il più possibile, cercando di causare un crash Tipicamente, può essere eseguito durante il naturale ciclo di sviluppo dell'applicazione. Ad esempio, un ciclo di smoke testing può essere eseguito durante la notte. Originariamente utilizzato per il testing di componenti hardware, è molto diffuso nell'ambito dei cicli di sviluppo agili, nei quali una versione integrata e testabile del software dovrebbe essere molto spesso disponibile.

1.4.1 Testing Automation

Con il termine Testing Automation intendiamo l'insieme delle tecniche e delle tecnologie che consentono di automatizzare (anche parzialmente) alcune attività del Processo di Testing. Il framework JUnit [6] è un esempio di test manager: il tester estende le classi messe a disposizione dal framework per creare un ambiente di test integrato. Ogni test case viene implementato sotto forma di metodo o di oggetto, ed un test runner si occupa della loro esecuzione. L'esito di un test case viene definito dichiarando una serie di asserzioni, ossia condizioni che se non verificate provocano l'interruzione dell'esecuzione del test case e la registrazione del fallimento dello stesso. Il framework Selenium [7], specializzato per le applicazioni web, si occupa della creazione di casi di test. In particolare, il tool Selenium IDE [8] fornisce un ambiente integrato per procedure di capture/playback, inclusa la possibilità di modificare a mano gli script prodotti prima di riprodurli o di esportarli sottoforma di classi JUnit.

1.5 Testing di applicazioni Android

Utilizzare tecniche sviluppate per altri sistemi allo scopo di testare applicazioni Android richiederebbe un considerevole lavoro di adattamento dei modelli e delle strategie adottati

per poter tenere in conto le peculiarità della piattaforma. Di conseguenza, è necessario sviluppare nuove tecniche di reverse engineering e crawling della GUI e nuovi strumenti per l'analisi dei risultati della sessione di crawling. Le transizioni di stato dell'interfaccia utente di un'applicazione Android sono guidate da eventi. Gli eventi possono essere eventi originati dall'utente, tramite l'interazione con i widget che compongono la GUI, oppure eventi associati ai messaggi di interrupt inviati da uno dei sensori o dispositivi di comunicazione di cui è dotato lo smartphone su cui l'applicazione è in esecuzione. Android mette a disposizione apposite classi che, estendendo quelle di JUnit, automatizzano le procedure necessarie ad interfacciarsi col componente, permettendo al tester di concentrarsi unicamente sul compito di definire le operazioni e le condizioni che compongono i casi di test.

1.5.1 Android Testing Framework

Android è dotato di framework interno per il test delle applicazioni basato su JUnit, il cui schema di principio è descritto in figura 1.3. I casi di test JUnit sono semplici metodi Java, organizzati in classi di test, contenute in package di test che compongono il progetto di test. Il framework viene esteso da classi specifiche per i vari tipi di component dell'application framework in maniera tale da permettere di svolgere operazioni quali la creazione di stub ed il controllo del ciclo di vita del componente. Mentre in JUnit si utilizza direttamente un test runner per l'esecuzione dei casi di test, in Android è necessario impiegare appositi tool per caricare i package del progetto di test e l'applicazione sotto test; quindi, un test runner specific per Android (InstrumentationTestRunner) viene controllato dal tool per l'esecuzione dei test.

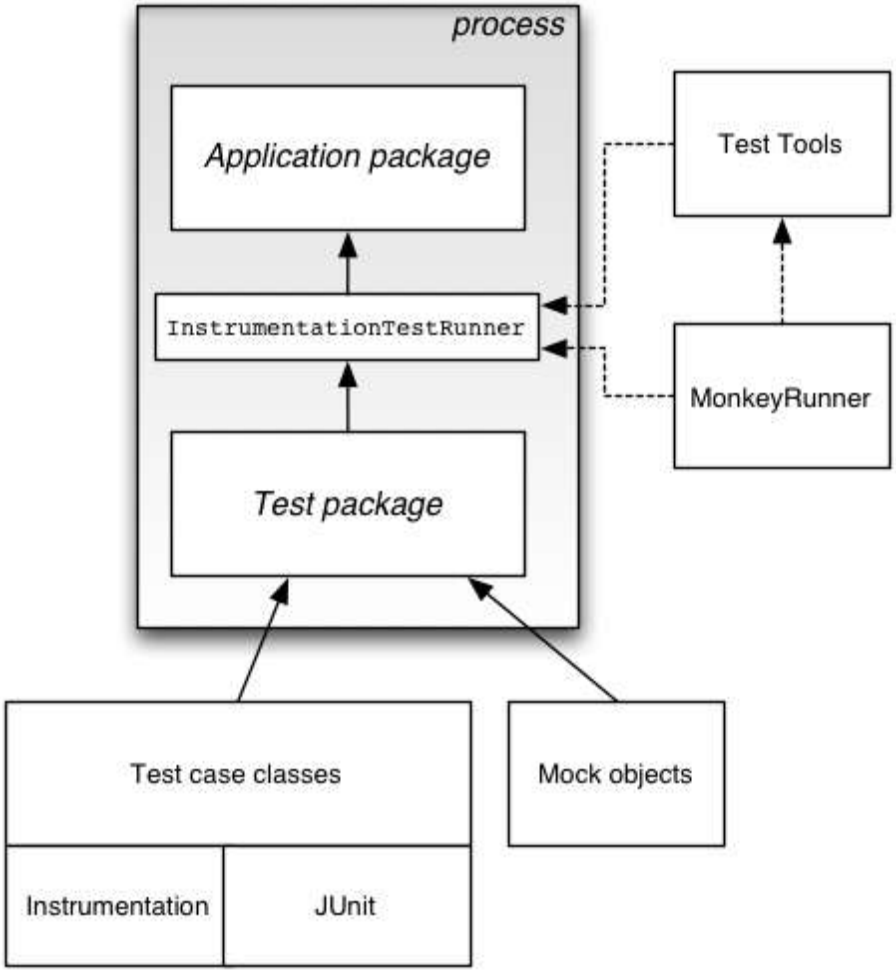

Figura 1.3: Testing Framework di Android

1.5.2 Monkey e Monkey Runner

Monkey è un'utility interna fornita con l'android SDK, che è in grado di generare eventi utente pseudocasuali su una qualsiasi interfaccia, registrando gli eventuali crash. Monkeyrunner, a differenza di monkey, è un API che consente la scrittura di programmi in grado di controllare un dispositivo Android dall'esterno.

1.5.3 L'instrumentation

L'instrumentation di Android è un insieme di chiamate di sistema grazie alle quali è possibile controllare i componenti del framework Android indipendentemente dal loro normale ciclo di vita. Il programmatore Android implementa tali metodi, specificando il codice che dovrà essere eseguito alla loro chiamata, ma non può invocarli a piacimento:

questo viene fatto dal framework. In fase di test, vorremmo invece essere in grado di invocare tali metodi per forzare il componente nello stato desiderato, lo stato il cui comportamento deve essere verificato. Questo è possibile grazie all'instrumentation: ad esempio, con il metodo getActivity() è possibile ottenere un riferimento all'activity in esecuzione ed eseguire uno qualsiasi dei metodi che essa definisce. Un altro modo in cui l'instrumentation permette di controllare l'esecuzione di un programma è attraverso la classe TouchUtils4 che mette a disposizione una serie di metodi che simulano l'interazione dell'utente con l'applicazione sotto test tramite un touch screen. Come rappresentato in figura 1.3, l'instrumentation viene attivata attraverso il test runner di Android e possono accedervi tutti i test case che estendono InstrumentationTestCase, ed in particolare ActivityInstrumentationTestCase2.

1.5.4 Robotium

Robotium è un framework a supporto del testing di unità delle Activity che estende e potenzia Junit. In particolare consente al tester di concentrarsi sulla logica del test case, lasciando a Robotium il compito di eseguire le particolari interazioni richieste di volta in volta attraverso l'instrumentation. Il funzionamento di Robotium è tutto basato sull'utilizzo di un oggetto denominato "Solo". Tramite l'oggetto Solo è possibile interrogare e modificare i widget della UI, eventualmente anche senza conoscerne l'identificativo.

1.5.5 Emma

Emma è uno strumento che permette di misurare la copertura del codice in ambito Java. Ci permette di conoscere quante e quali parti del codice sorgente siano state effettivamente attraversate durante l'esecuzione di un test, permettendo di ottenere una misura quantitativa della bontà della test-suite utilizzata. Genera report e metriche, anche in formato HTML e risulta molto utile per il testing White Box.

Capitolo 2: Tecniche di GUI Testing e Android Ripper

In questo capitolo descriveremo le principali tecniche di GUI Testing per un'applicazione Android. Il testing dell'interfaccia grafica rappresenta infatti l'approccio principale per testare un prodotto software di questo tipo. Successivamente verrà presentata una tecnica che prevede l'utilizzo di un crawler per l'esecuzione di crash testing basato sulla GUI e per l'estrazione di un modello della stessa.

2.1 Testing GUI Based

Un'interfaccia grafica è composta da una serie di schermate a loro volta costituite da oggetti (widget) con i quali l'utente finale può interagire al fine di controllare l'applicazione. Ogni interazione valida genera particolari messaggi, detti eventi, che vengono raccolti da componenti in ascolto, generalmente esterni alla GUI. L'elaborazione degli eventi genera un cambiamento di stato nella schermata visualizzata e/o il passaggio ad una nuova schermata. Le difficoltà principali legate al testing della GUI sono da ritrovarsi nella sua natura dinamica. Infatti l'interfaccia grafica dovrà essere esercitata dal tester cercando di riprodurre lo stesso comportamento dell'utilizzatore finale. L'automatizzazione di un simile procedimento richiede l'utilizzo di un software che riproduca tali azioni. Inoltre è necessario possedere un modello formale del funzionamento dinamico della GUI per poter decidere se il comportamento osservato corrisponde a quello atteso.

Le tecniche di GUI testing possono dividersi in tre categorie:

- Tecniche Model Based, in cui esiste una descrizione dell'applicazione sotto test sufficiente alla generazione automatica dei Test Case per la verifica del software;

- Tecniche Random Testing, in assenza di un modello, l'applicazione viene esercitata in maniera casuale alla ricerca di eventuali eccezioni non gestite;

- Tecniche Model Learning, in cui l'applicazione viene esplorata senza una conoscenza pregressa della sua struttura seguendo una strategia di navigazione.

2.2 Android Ripper

Android Ripper è uno strumento che permette di realizzare testing automatico implementando sia tecniche Random che Crawler-Based. Esso è in grado di esplorare la struttura della GUI mediante la generazione di input ed eventi. Durante l'esplorazione viene tenuta traccia di tutte le eccezioni non gestite (crash) e della sequenza di eventi che le ha generate. Al termine dell'esplorazione in output al processo di ripping si avrà tutto il necessario non solo per ricostruire i suoi dettagli, ma anche il per la generazione del modello e di nuovi casi ti test (testing mutazionale), sia per la generazione di casi di test jUnit-Android per effettuare il testing di regressione.

2.2.1 Il processo di ripping

Analizziamo i passi fondamentali che caratterizzano il processo di ripping:

- Inizialmente si analizza lo stato corrente della GUI. Lo stato della GUI è rappresentato dall'activity corrente descritta in termini dei suoi parametri e dei widget in essa contenuti; i widget saranno a loro volta descritti in termini di uno o più attributi. Sarà importante decidere quanti e quali di questi attributi e parametri considerare come indicatori dello stato dell'Activity, ed in particolare del

cambiamento di stato.

- A partire dallo stato descritto è possible ottenere la lista degli eventi scatenabili sull'activity, ovvero quelle azioni che permettono di passare da un'interfaccia ad un'altra. Per ogni elemento della lista generata viene pordotto un task contenente una successione ordinata di input ed eventi, i quali vengono aggiunti ad un piano di test (TaskList).
- Se il piano di test contiene task da eseguire, ne viene estratto uno secondo la strategia configurata e si passa alla sua esecuzione, altrimenti il processo termina rendendo (eventualmente) disponibili i suoi output.
- Dopo l'esecuzione del task, viene fornito un rapporto di esecuzione (trace) e lo stato della GUI raggiunto viene descritto e confrontato con gli stati visitati; se lo stato è già stato visitato, si passa all'estrazione del task successivo, altrimenti si analizza lo stato corrente in cerca di nuovi task da generare.

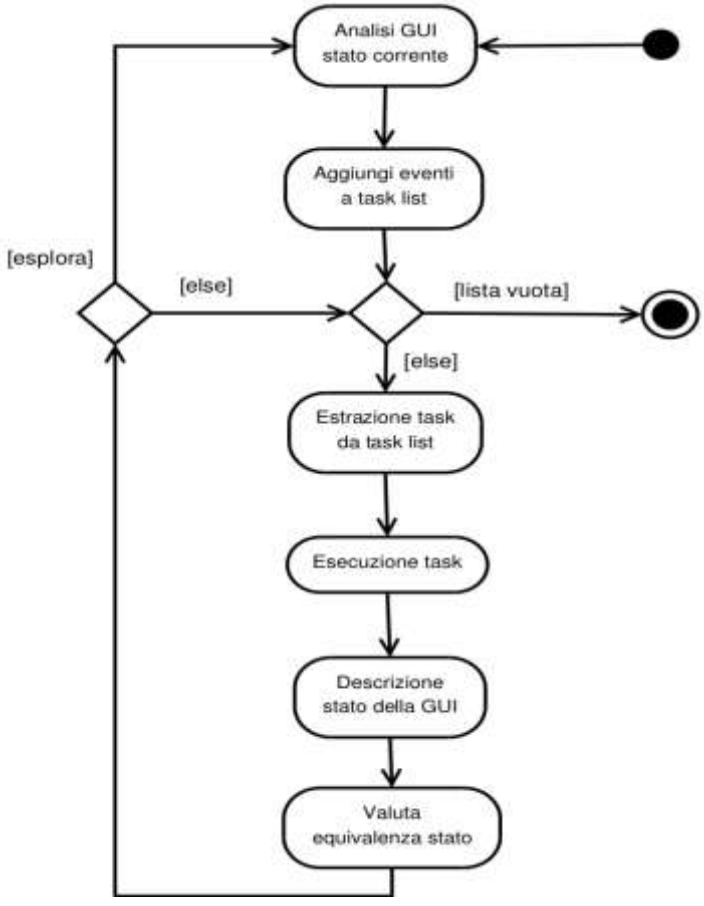

Figura 2.1: Activity Diagram dell' Android Ripper

L'output finale del processo di ripping sarà rappresentato da due file: uno contente un rapporto dell'esecuzione dei singoli task ed un altro contenente tutti gli stati attraversati durante l'esplorazione della GUI. A partire dal primo file prodotto sarà possibile ottenere una rappresentazione dell'applicazione sotto forma di GUI Tree. Nel GUI Tree, ogni stato S (activity) viene rappresentato come nodo di un albero, collegato tramite un arco orientato a tutti gli stati che è possibile raggiungere esercitando opportunamente i widget dell'applicazione mentre essa si trova in S. Gli archi rappresentano, quindi, transizioni di stato.[10]

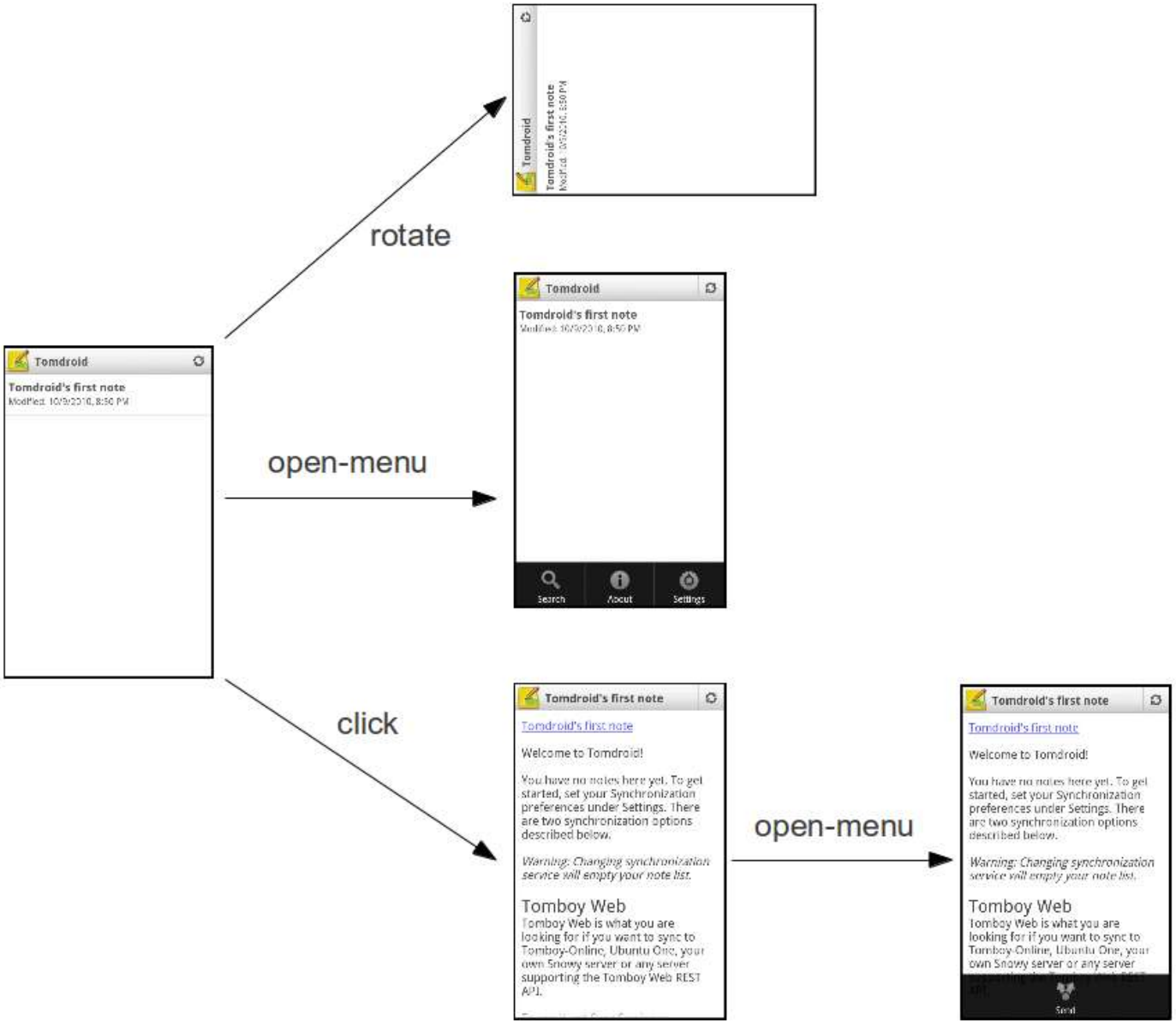

Figura 2.2: Estratto di GUI Tree di Tomdroid

2.2.2 Il modello dell'applicazione

Per capire come funziona l'algoritmo si rende necessario introdurre i concetti principali sui quail esso si basa. Essi sono:

- **Activity State**: rappresenta l'Activity correntemente visualizzata (running activity), in termini dei suoi parametri e dei widget in essa contenuti; i widget saranno a loro volta descritti in termini di uno o più attributi.
- **Evento**: un azione utente eseguita su uno dei widget della GUI in grado di determinare il passaggio da una schermata ad un'altra.
- **Input**: un'interazione con la GUI dell'applicazione sotto test che non provoca cambiamenti di stato. In pratica, tutte le interazioni che non sono eventi,saranno considerate di input.
- **Azione**: la sequenza ordinata di un evento e di tutti gli input che lo precedono. Essa rappresenta le operazioni necessarie ad indurre un passaggio di stato nella GUI dell'applicazione sotto test.
- **Task**: coppia (Azione, GUI State) che rappresenta un azione eseguita sull'interfaccia.
- **Plan**: è l'insieme dei task che descrivono le possibili azioni con le quali è possibile interagire per innescare ulteriori transizioni e proseguire l'esplorazione.
- **Trace**: un rapporto sull'esecuzione di un task, corrispondente ad una traccia di esecuzione del GUI Tree. Per ogni azione componente il task, il trace dovrà contenere lo stato dell'Activity all'inizio, la sequenza degli input, l'evento e lo stato dell'Activity alla fine della transizione.
- **Sessione**: l'insieme di tutte le operazioni eseguite dal ripper, cominciando con la prima inizializzazione fino alla generazione del file di output.

2.2.3 Le componenti del Ripper

Saranno adesso introdotti i componenti dell'applicazione, accennando a quelle che sono le loro principali responsabilità:

- **Engine**: L'Engine è il controllore centralizzato che racchiude la business logic del ripper. Esso gestisce il crawling in ogni sua fase, supervisionando il flusso di esecuzione e garantendo lo scambio di messaggi fra le componenti.
- **Scheduler**: Lo scheduler (o dispatcher) è il componente che decide l'ordine di esecuzione dei task generati per l'esplorazione dell'applicazione sotto test. Esso si occupa di memorizzare i task in attesa di essere eseguiti in un'opportuna struttura datie fornisce all'Engine, di volta in volta, il task da eseguire.
- **Robot**: Il robot è il componente che si occupa dell'interfacciamento con l'applicazione, ed esegue i task pianificati per esplorarla, sfruttando la classe Instrumentation messa a disposizione dal Testing Framework di Android. Esso agisce riproducendo le interazioni che un utente reale eserciterebbe per svolgere il compito descritto nel task da eseguire.
- **Extractor**: L'extractor ha la responsabilità di estrarre le informazioni che determinano lo stato dell'applicazione, ed in particolare l'aspetto dell'interfaccia grafica dell'Activity attiva nel momento in cui il componente svolge il suo compito. A seguito dell'elaborazione, l'extractor mette a disposizione del Ripper una Activity Description, una descrizione dell'istanza di interfaccia corrente. Questa descrizione sarà una collezione di riferimenti ad oggetti del framework Android presenti nel contesto della JVM. Esempi di oggetti estratti da questo componente sono la Activity corrente ed i widget presenti a video.
- **Abstractor**: L'abstractor crea e fornisce al ripper un modello esportabile dell'interfaccia correntemente visualizzata dall'applicazione, senza fare riferimento agli oggetti effettivamente istanziati nella JVM, al fine di rendere possibile la costruzione di un modello dell'applicazione la cui validità si estenda oltre la durata della singola sessione di ripping.

- **Strategy**: Lo Strategy si occupa di due compiti principali:

    - confrontare lo stato corrente dell'applicazione e gli stati già visitati in precedenza. In caso di equivalenza lo stato corrente non necessita di ulteriore esplorazione da parte del ripper.

    - prendere le decisioni che guidano il flusso di esecuzione del ripper in base al risultato del confronto fra stati, ai dati forniti dall'abstractor, alla descrizione dell'ultimo task eseguito ed eventualmente ad ulteriori informazioni interne al componente, quali il tempo di esecuzione del software ed il livello di profondità raggiunto.
- **Planner**: Il Planner ha il compito di generare il il plan che definisce le modalità di eplorazione dell'applicazione a partire dall'activity corrente. I nuovi task saranno generati in base all'analisi del risultato dell'esecuzione del task che ha portato l'applicazione in questo stato, e solo se di tale stato è stata richiesta l'esplorazione da parte dello strategy. Il planner esamina la struttura dell'istanza di interfaccia visualizzata e selezionaquei widget che, in base a regole prestabilite o definite dal tester, sono ritenuti in grado di innescare una transizione di stato nell'applicazione.
- **Persistence Manager**: Il persistence manager provvede a tutte le operazioni da e verso le memorie di massa (lo storage interno o la scheda SD del dispositivo) come l'apertura, la chiusura, la copia e la cancellazione di file.[9]

L'architettura complessiva dell'Android Ripper è schematizzata nell'immagine seguente:

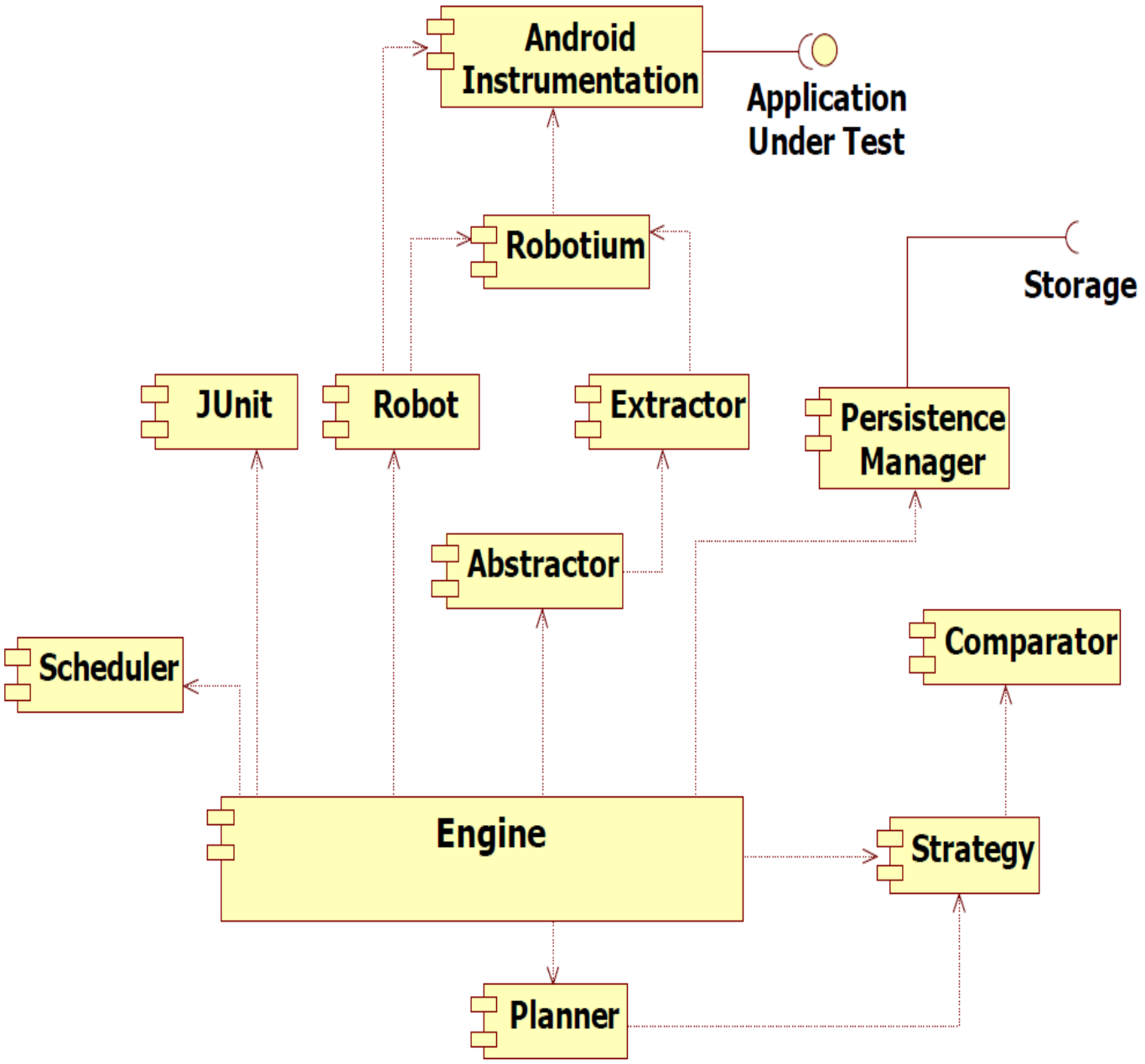

Figura 2.3: Architettura Android Ripper

2.2.4 Tecnica di ripping

Vediamo come funziona nel dettaglio l'algoritmo di ripping sul quale si basa il nostro strumento di testing automatico. Possimo distinguere le seguenti fasi:

Inizializzazione: è la fase durante la quale le varie componenti del ripper vengono istanziate ed inizializzate.

Setup: è la fase in cui il ripper si interfaccia all'applicazione da esplorare, ne esamina lo stato iniziale e popola la lista dei task da eseguire, inizialmente vuota. Lo stato dell'activity iniziale, opportunamente elaborato, viene memorizzato nella lista degli stati visitati per successivi confronti, ed inviato al Planner per la compilazione di un piano di esplorazione contenente l'elenco dei primi task generati, che andranno successivamente eseguiti nella fase di esplorazione.

Algoritmo Setup

Input: $androidApplication$ = the application under test
$robot$.bindApplication($androidApplication$)
$baseActivity \leftarrow robot$.getCurrentActivity()
$activityDescription \leftarrow extractor$.describe($baseActivity$)
$activityState \leftarrow abstractor$.abstract($activityDescription$)
$strategy$.storeVisitedState($activityState$)
$plan \leftarrow planner$.createPlan($activityState$)
$scheduler$.addPlan($plan$)

Esplorazione: tale fase tratta, in successione, i task estratti dallo scheduler, eseguendoli, elaborandone i risultati e generando eventualmente nuovi task. L'algorimto è il seguente: inizialmente, l'Extractor e l'Abstractor forniscono una descrizione dell'activity corrente, ovvero quella ottenuta al termine dell'esecuzione del task. Tale descrizione viene poi inviata allo Strategy per effettuarne la comparazione con gli stati visitati in precedenza; se lo stato corrente non equivale a nessuno di quelli presenti nella activity list, allora dovrà essere aggiunto. A questo punto, lo Strategy individua se una transizione ha effettivamente avuto luogo alla fine del task eseguito: ad esempio, la pressione di un elemento di menu

disattivato non produce alcun risultato. In questo caso, le operazioni riportate di seguito non vengono eseguite e l'esecuzione riprende dall'inizio del ciclo con l'estrazione di un nuovo task. In caso contrario, l'esecuzione prosegue con la generazione del trace che descrive l'esecuzione del task appena terminato. Esso viene poi passato al Persistence Manager per essere memorizzato su disco. L'insieme di questi trace costituirà l'output della sessione, dal quale sarà in seguito possibile estrarre un modello a stati dell'applicazione sotto test. Lo Strategy dovrà ora decidere se lo stato corrente è passibile di ulteriore esplorazione. Nel caso più semplice, tale decisione equivale all'esito del confronto appena effettuato: lo stato verrà esplorato se e solo se non è mai stato esplorato in precedenza. Infine, si controlla se uno dei criteri di terminazione è verificato. In tal caso, la sessione viene chiusa e si esce dal ciclo di iterazione. Altrimenti si procede con l'esecuzione di un nuovo task, se disponibile.[9]

Algoritmo Exploration

```
while scheduler.hasMoreTasks() do
  task ← scheduler.getNextTask()
  strategy.setCurrentTask(task)
  robot.process(task)
  currentActivity ← robot.getCurrentActivity()
  activityDescription ← extractor.describe(currentActivity)
  activityState ← abstractor.abstract(activityDescription)
  isStateNew ← strategy.compareState(activityState)
  if isStateNew then
    strategy.storeVisitedState(activityState)
  end if
  if strategy.transitionOccurred() then
    trace ← abstractor.createTrace (task, activityState)
    persistence.addTrace(trace)
    if strategy.explorationNeeded() then
      plan ← planner.createPlan(activityState)
      scheduler.addPlan(plan)
    end if
    if strategy.sessionTermination() then
      close the session
      break the loop
    end if
  end if
end while
```

Capitolo 3: Ripper parallelo e funzionamento da remoto

Nel capitolo seguente si entra nel vivo del lavoro di tesi in quanto verrano introdotti gli algoritmi e le modifiche apportate al ripper originale che, partendo da un'architettura a singola macchina con un solo emulatore, hanno permesso di realizzare una struttura parallela composta da diverse macchine, ognuna delle quali provvista di più emulatori. Successivamente verrà illustrato il meccanismo per il funzionamento remoto grazie al quale si è reso possibile utilizzare la funzione di testing da una macchina remota.

Prima di addentraci nella descrizione degli algoritmi vi sono alcune premesse che è importante fare. Il Ripper descritto nel capitolo precedente consiste nella sua versione "sistematica". Esiste un'ulteriore modalità di funzionamento, detta "random" che prevede che i task, e quindi gli eventi, vengano generati in maniera del tutto casuale, in base ad uno specifico seed. Entrambe le tipologie di ripper sono state oggetto di studio.

3.1 Introduzione del parallelismo nel Ripper

Come detto precedentemente, primo punto del lavoro di tesi è stato il passaggio da una struttura a singola macchina con singolo emulatore a singola macchina multiemulatore.

La Figura 3.1 mostra l'architettura di partenza dell'intero sistema mentre la Figura 3.2 mostra il risultato ottenuto. La componente denominata "Driver" è quella che si occupa di realizzare l'intero processo di ripping con strategia di tipo sistematica o randomica. Quest'ultima invierà all'emulatore/i le azioni da eseguire e riceverà l'esito prodotto.

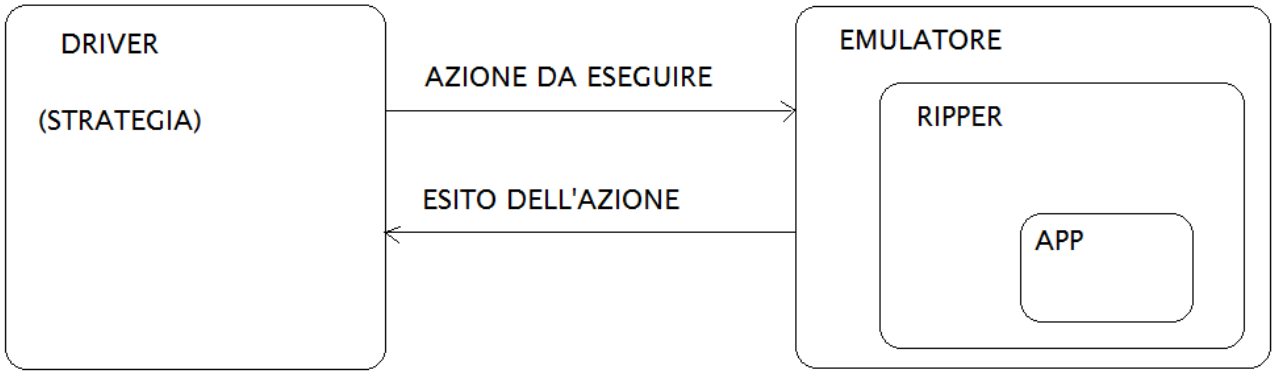

Figura 3.1: Architettura complessiva a singolo emulatore

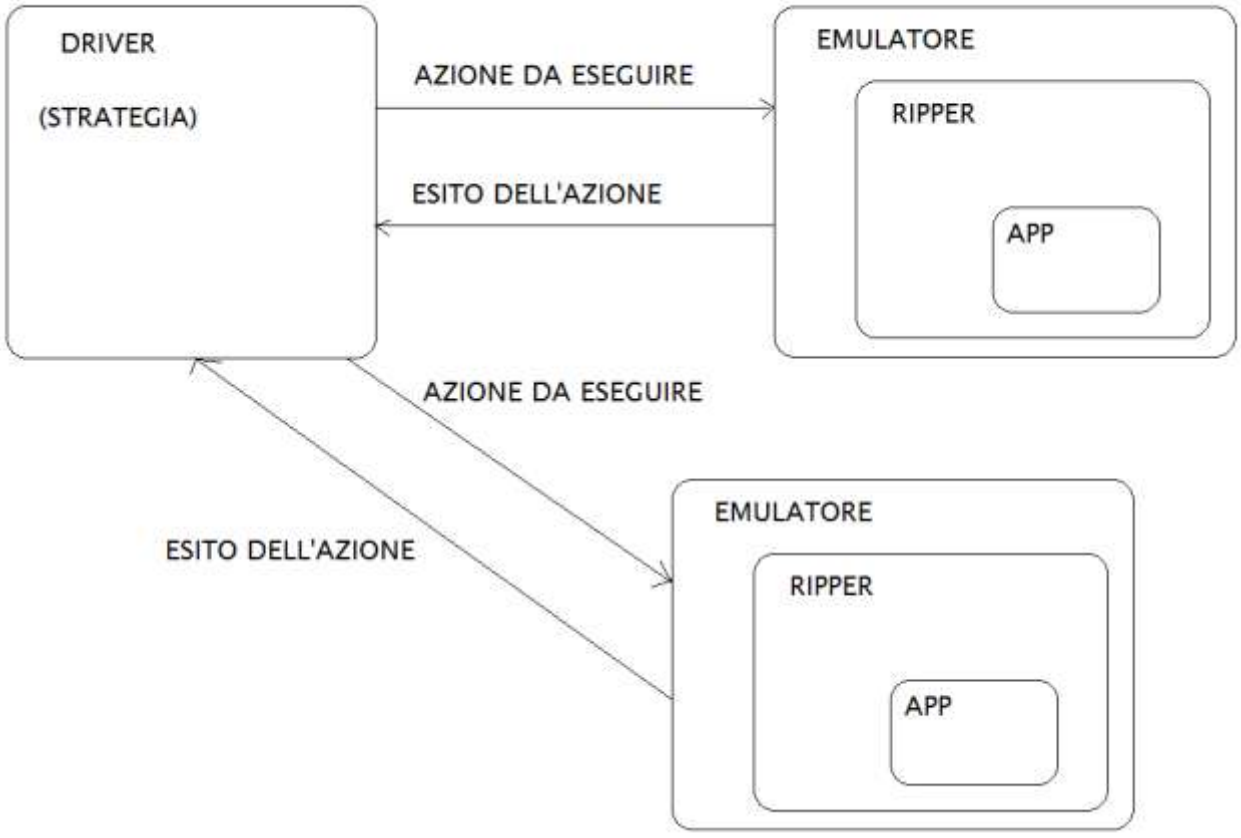

Figura 3.2: Architettura complessiva multiemulatore

Per realizzare l'esecuzione parallela si è deciso di utilizzare una strategia multithread. Analizzeremo i passaggi eseguiti per la realizzazione di tale obiettivo nelle due strategie previste.

3.2 Parallelismo del ripper sistematico

Il multithreading è stato implementato mediante l'introduzione di una classe chiamata ClientSistThread che estende la classe Thread. Il funzionamento del sistema può essere ricondotto a tre classi principali: SystematicRipper, SystematicDriver e ClientSistThread appartenenti al package **it.unina.android.ripper.systematic.driver**. Vediamo nel dettaglio di cosa si occupano queste tre classi:

SystematicRipper: rappresenta il main dell'applicazione. Primo compito di tale classe è quello di istanziare gli oggetti Scheduler, Planner, Comparator, ActivityList, DescriptionLoader, TerminationCriterion. Tali oggetti verrano istanziati un'unica volta all'interno del costruttore e passati a tutti i thread:

```
public SystematicRipper()
{
super();
this.scheduler = new BreadthScheduler();
 if (PLANNER.equals("ConfigurationBasedPlanner"))
this.planner = new ConfigurationBasedPlanner();
else
this.planner = new HandlerBasedPlanner();
this.comparator = new GenericComparator(
GenericComparatorConfiguration.Factory.getCustomWidgetSimpleComparator() );
this.descriptionLoader = new XMLDescriptionLoader();
this.statesList = new ActivityStateList(this.comparator);
this.terminationCriterion = new
EmptyActivityStateListTerminationCriterion(this.scheduler.getTaskList());
this.ripperOutput= new XMLRipperOutput();}
```

Figura 3.3:Costruttore di SystematicRipper

La tecnica di scheduling scelta è l' esplorazione in ampiezza della GUI (breadth-first) il cui funzionamento è esemplificato dall'immagine seguente:

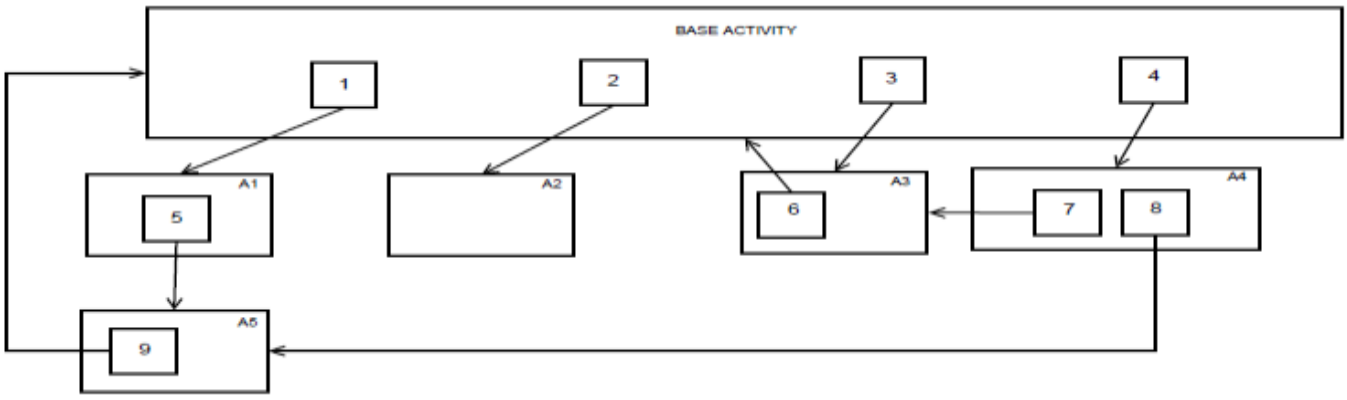

Figura 3.4:Tecnica di esplorazione breadth-first

I riquadri numerati rappresentano i widget sui quali è possibile scatenare un evento che induce una transizione di stato. La freccia indica lo stato stato in cui l'applicazione si trova dopo la transizione. Il procedimento inizia esaminando la base activity ed eseguendo i quattro task che esercitano i widget in essa presenti. L'esecuzione del primo task porta alla scoperta di un nuovo stato, denotato in figura con A1, rappresentato da una nuova activity contenente un widget. L'esplorazione di questo stato tramite l'iniezioni di eventi su tale widget avverrà solo al quinto task, dopo che l'esplorazione della base activity è stata completata. L'intero processo porta alla fine, alla scoperta dello stato A5 il quale viene scoperto all'ottavo task, partendo allo stato A4.

Successivamente viene invocato il metodo **startRipping()** che ha il compito di caricare la configurazione del sistema a partire da un file config.properties. Infine vengono istanziati a lanciati i vari thread, tanti quanti sono gli emulatori che si vuole funzionino contemporaneamente. Tali thread saranno istanze della classe ClientSistThread.

ClientSistThread: permette di realizzare il multithreading estendendo Thread. Il suo costruttore riceve dalla classe SystematicRipper gli oggetti Scheduler, Planner, Comparator, ActivityList, DescriptionLoader, TerminationCriterion e li passa all'oggetto SystematicDriver instanziato al suo interno:

```
public ClientSistThread(String ip,int ID,String avdName,int modo,Scheduler sche, Planner
plan, IComparator comp, IDescriptionLoader desc, ActivityStateList statelist,
TerminationCriterion terminator, RipperOutput ripout, String covPath){
      super();
      this.AvdName=avdName;
      this.IP_server=ip;
      this.id=ID;
      this.coveragePath=covPath;
      this.mod=modo;
      systDriver=new SystematicDriver(sche,plan,comp,desc,statelist,terminator,ripout);
      }
```

Figura 3.5:Costruttore di ClientSistThread

Il metodo run di questa classe non fa altro che richiamare il metodo startRipping sull'oggetto SystematicDriver creato.

SystematicDriver: esegue il processo di ripping vero e proprio. Il suo costruttore riceve

dalla classe ClientSistThread gli oggetti Scheduler, Planner, Comparator, ActivityList, DescriptionLoader, TerminationCriterion:

```
public SystematicDriver(Scheduler sche, Planner plan, IComparator comp,
IDescriptionLoader desc, ActivityStateList statelist, TerminationCriterion terminator,
RipperOutput ripout)
      {
      super();

      sem=new Semaphore(0);
      this.scheduler = sche;
      this.planner = plan;
      this.comparator = comp;
      this.descriptionLoader = desc;
      this.statesList = statelist;
      this.terminationCriterion = terminator;
      this.ripperOutput= ripout;   }
```

Figura 3.6:Costruttore di SystematicDriver

Come gia anticipato, tale classe permette di realizzare l'algoritmo di ripping per mezzo della funzione **public void rippingLoop(String AVD_NAMEs, int EMULATOR_PORTs, int port_localhosts, String coverage_path)** la quale a sua volta, richiama alcune funzioni che individuano le diverse fasi del processo. Una prima fase, detta di startup prevede l'avvio dell'emulatore con relativa attivazione del service e avvio del caso di test; una fase di bootstrap che permette di ottenere la descrizione dell'activity di partenza dell'app sotto test e la aggiunge alla state list. Successivamente viene creato il piano di esecuzione tramite la funzione plan ed i task generati vengono assegnati allo scheduler. Lo scheduler estrae un task dalla tasklist e lo esegue dopodichè viene aggiornato lo stato con l'activity corrente e quest'ultima è confrontata con gli elementi della state list: se l'activity non era gia stata inserita nella state list allora viene posta al suo interno. In quest'ultimo caso viene realizzato il piano di esplorazione a partire dall'ultima activity inserita e la task list generata viene assegnata allo scheduler. Tale procedimento termina quando la task list viene completamente svuotata.

L'immagine 3.7 mostra il diagramma delle classi relative al package it.unina.android.ripper.systematic.driver:

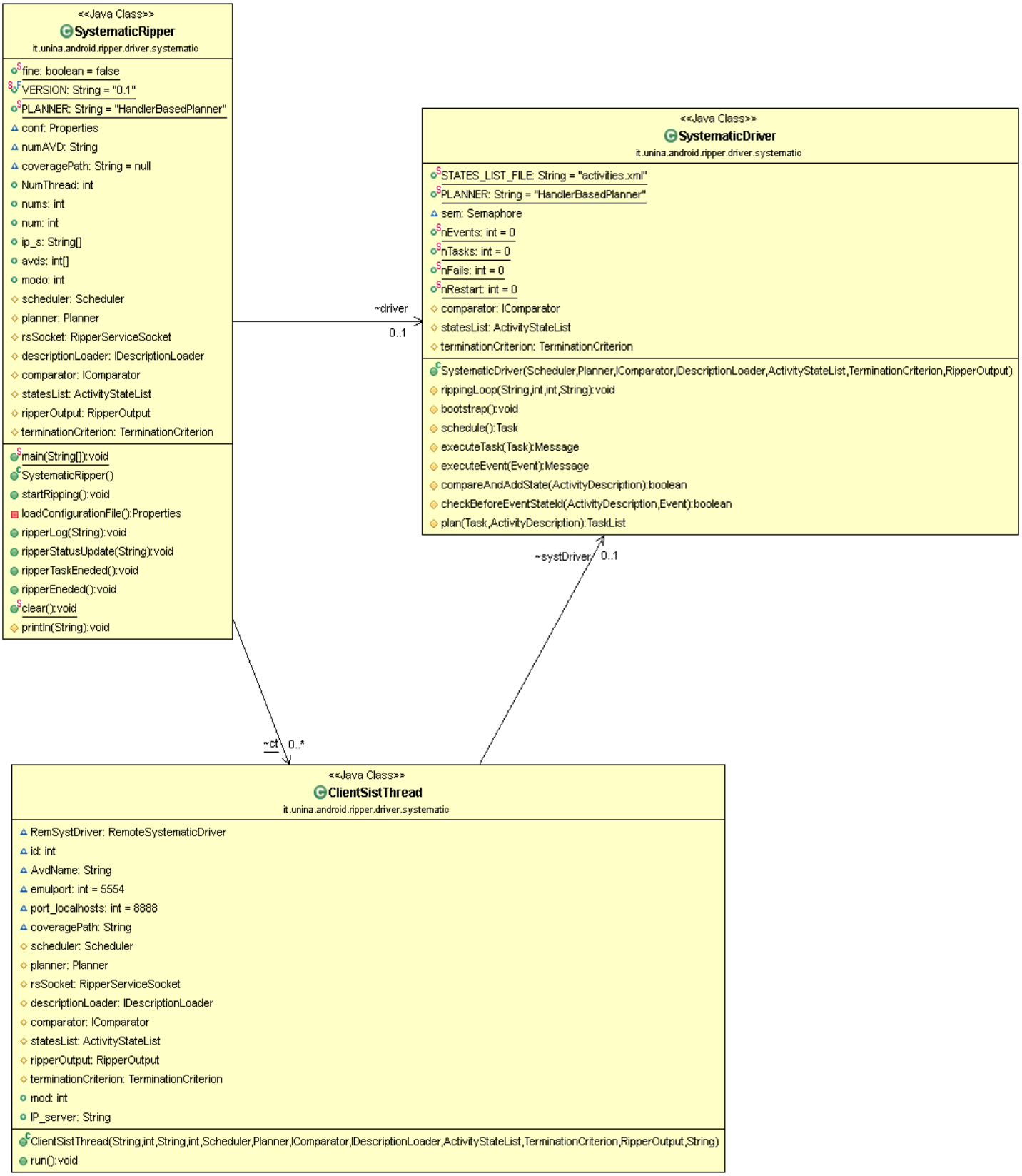

Figura 3.7:Class diagram di it.unina.android.ripper.systematic.driver

3.2.1 Realizzazione di una regione critica

L'introduzione del parallelismo mediante l'utilizzo di thread comporta inevitabilmente alcuni problemi legati alla gestione della concorrenza. Vi saranno infatti alcune porzioni di codice che dovranno essere accedute in mutua esclusione. Per risolvere questo problema è stato necessario introdurre delle regioni critiche mediante l'utilizzo di semafori.

La prima regione critica si ha in corrispondenza dell'operazione di bootstrap. Come abbiamo precedentemente detto, tale operazione permette di ottenere la descrizione dell'activity di partenza e di creare il piano di esplorazione a partire da essa. Tale operazione deve essere eseguita solamente da uno dei vari thread. La regione critica è stata cosi realizzata:

```
boolean bootstrap = false;
sem.acquire(0);
       if (bootstrap == false)
           {
              this.bootstrap();
              bootstrap = true;
           }

Task t = this.schedule();

sem.release(0);
```

Figura 3.8:Regione critica per l'operazione di bootstrap

La variabile booleana bootstrap inizialmente è posta a false. Il primo thread che sopraggiunge acquisisce il semaforo, esegue la funzione di bootstrap e pone a true la variabile booleana per poi estrarre il primo task da eseguire. Se nel frattempo arriva un altro thread, questo resterà fuori dalla regione critica fino a quando il primo thread avrà rilasciato il semaforo. A questo punto la variabile booleana sarà true e si eviterà di eseguire nuovamente l'operazione di bootstrap.

Una seconda regione critica è stata realizzata per la gestione di una risorsa condivisa quale è la state list, ovvero quella lista contenente al suo interno tutti gli stati, e quindi le activity visitate durante il processo di ripping. In particolare, la funzione **protected boolean compareAndAddState(ActivityDescription activity)** verifica che un'activity visitata

non sia presente nella state list e, in quest ultimo caso, la pone al suo interno. La necessità di dover accedere alla state list in mutua esclusione è dovuta al fatto che se cosi non fosse, due thread che scoprono contemporaneamente un nuovo stato non ancora visitato e quindi non ancora inserito nella state list, vorrebbero entrambi porre l'activity scoperta al suo interno. La soluzione è stata cosi implementata:

```
sem.acquire(0);
if ( compareAndAddState( getCurrentDescriptionAsActivityDescription() ) )

  {
  TaskList plannedTasks = plan(t, getLastActivityDescription());
      scheduler.addTasks(plannedTasks);
                                  }
sem.release(0);
```

Figura 3.9:Regione critica per comparazione ed aggiunta di uno stato

Se due thread scoprono contemporaneamente la stessa activity, il primo che acquisisce il semaforo effettua la comparazione con gli elementi della state list e, se l'activity non era stata ancora scoperta la aggiunge al suo interno. Quando il primo thread rilascia il semaforo, il secondo accede alla regione critica ma, essendo l'activity gia stata inserita dal thread precedente, si eviterà che questa venga reinserita nella state list e di conseguenza non verrà creato il piano di esplorazione sulla stessa activity.

3.3 Parallelismo del ripper random

Il ripper nella sua versione random risulta essere molto più elementare. Anche in questo caso, per realizzare il parallelismo si è utilizzata una tecnica multithreading introducendo la classe ClientRandThread che estende Thread. Il funzionamento del sistema può essere ricondotto a tre classi principali: RandomRipper, RandomDriver e ClientRandThread appartenenti al package **it.unina.android.ripper.random.driver**. Vediamo nel dettaglio di cosa si occupano queste tre classi:

RandomRipper: rappresenta il main dell'applicazione. Analogamente al caso sistematico tale classe si occupa di di caricare la configurazione del sistema a partire da un file

config.properties mediante la funzione **startRipping()** ma non istanzia nè Planner nè Scheduler. Nel caso sistematico avevamo infatti la necessità di mantere le stesse istanze di Scheduler e Planner e passarle ai vari thread. Nel caso random, data la natura del tutto casuale con cui sono generati i task da eseguire, questo non è necessario e dunque tali oggetti saranno istanziati dai vari thread. Infine vengono creati a lanciati tanti thread quanti sono gli emulatori che si vuole funzionino contemporaneamente. Tali thread saranno istanze della classe ClientRandThread.

ClientRandThread: permette di realizzare il multithreading estendendo Thread. Il suo costruttore riceve dalla classe RamdomRipper una serie di attributi come l'id del thread, il nome dell'avd, ecc, e si occupa di istanziare un oggetto di tipo RandomDriver al quale passerà il suo id:

```
public ClientDriver(String ip,int ID,String avdName,int modo,String
covPath, String repFile)
{
super();
this.AvdName=avdName;
this.IP_server=ip;
this.id=ID;
this.coveragePath=covPath;
RandomDriver=new RandDriver(id);
this.report=repFile;
this.start();
}
```

Figura 3.10:Costruttore di ClientRandThread

Il metodo run di questa classe agisce richiamando il metodo startRipping sull'oggetto RandomDriver creato.

RandomDriver: esegue il processo di ripping nella versione random. Si occupa inoltre di istanziare planner e scheduler. Quest'ultimo agirà secondo una strategia randomica. Il costruttore della classe è il seguente:

```
public RandomDriver(int idd)
	{
	super();
	this.id=idd;
	planner = new HandlerBasedPlanner();
	scheduler = new DebugRandomScheduler(RANDOM_SEED);
	descriptionLoader = new XMLDescriptionLoader();

	}
```

Figura 3.12:Costruttore di RandomDriver

Richiamando la funzione **public void rippingLoop()** si lancia il processo di ripping che avrà fine quando il numero di eventi prefissato sarà raggiunto. La variabile **RANDOM_SEED** rappresenta il seme di casualità, ovvero quel numero casuale che determina la sequenza degli eventi da eseguire. Inoltre essendo il procedimento del tutto random, non vi sarà bisogno di mantenere una lista degli stati visitati e di conseguenza verrano a mancare tutte le considerazioni precedentemente fatte per il caso sistematico e relative alla gestione delle concorrenza (regioni critiche, semafori, ecc.). In figura 3.11 è possibile osservare il diagramma delle classi per il package **it.unina.android.ripper.random.driver.**

Dopo aver visto come sia stato possibile realizzare il parallelismo su singola macchina, vedremo i passaggi che hanno portato alla realizzazione di un sistema basato su paradigma client-server. Cambiando l'architettura complessiva del sistema è stato possibile ottenere una struttura composta da più macchine con più emulatori.

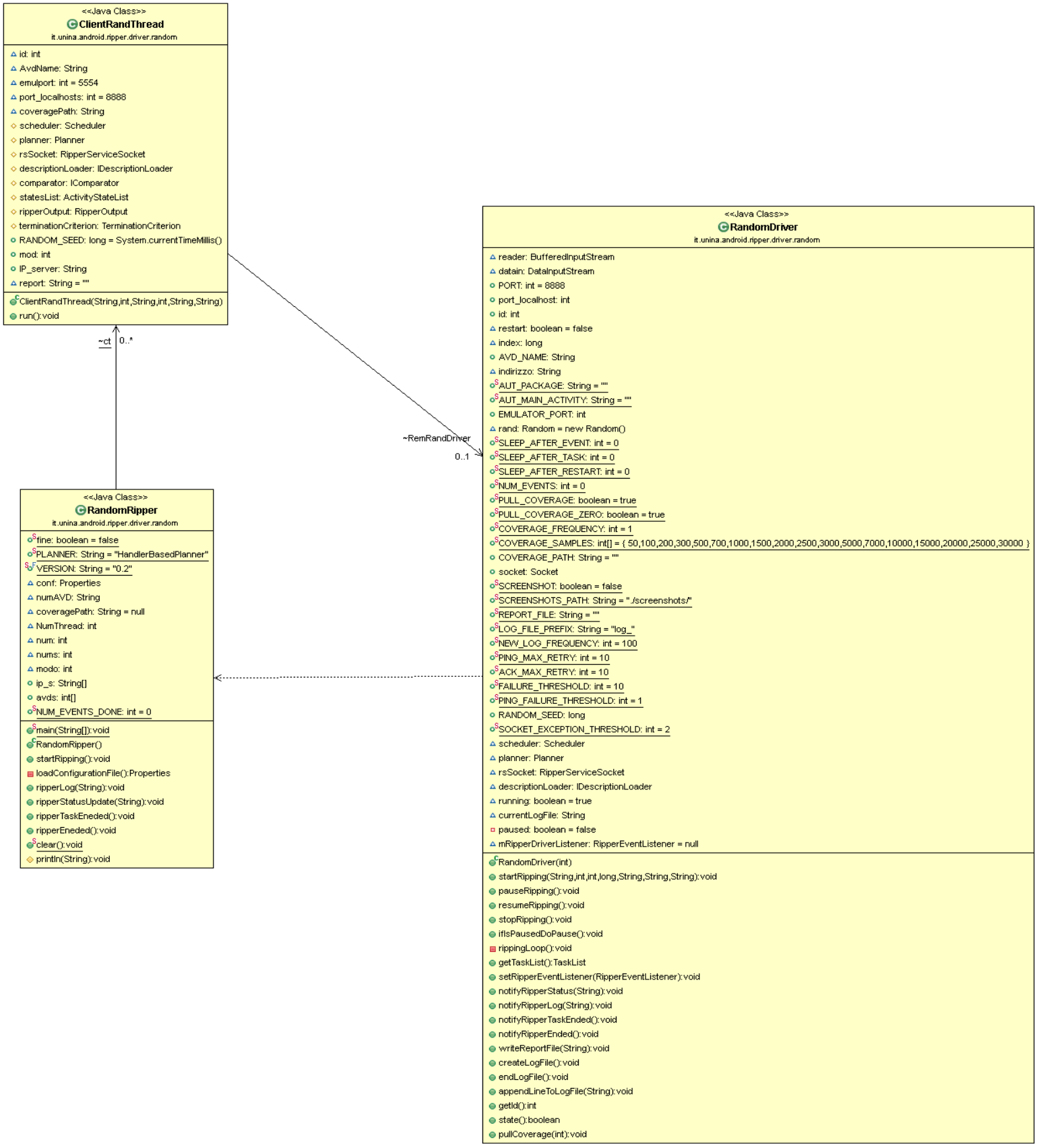

Figura 3.11:Class diagram di it.unina.android.ripper.random.driver

3.4 Ripper su server remoto

L'architettura che si è deciso di realizzare per aumentare l'efficienza del sistema prevede due entità principali: l'entità sulla quale gira il ripper e che dovrà essere unica, che chiameremo "driver" e una o più macchine che indicheremo con il termine "server" sulle quali verranno spostati gli emulatori con relative app da testare. La figura 3.12 mostra il passaggio da un'architettura all'altra:

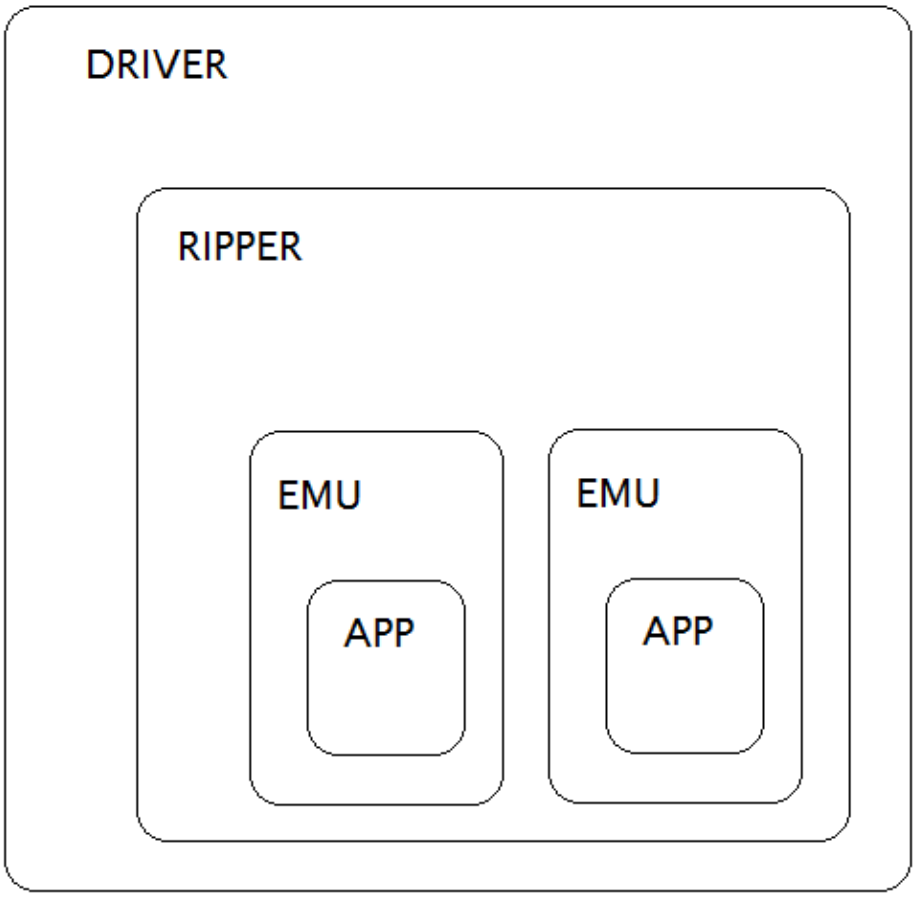

Figura 3.12.a:Architettura a singola macchina

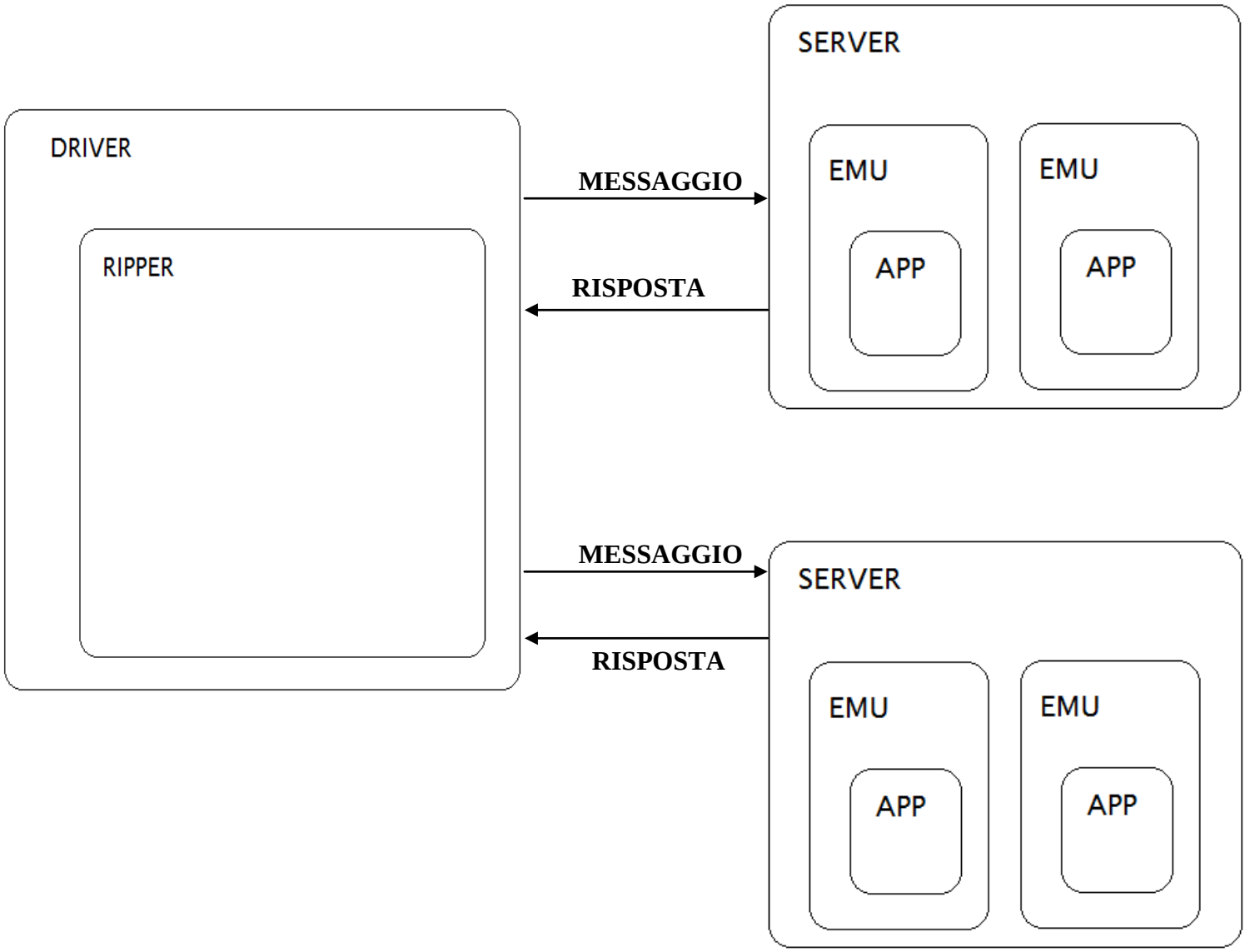

Figura 3.12.b:Architettura driver-server

La nuova architettura funzionerà sulla base di un'attività di scambio di messaggi tra il driver e i vari server. Questi ultimi saranno in grado di comandare gli emulatori a loro disposizione e di fornire gli esiti delle relative elaborazioni al driver che avanzerà nel suo processo di ripping, indipendemente dalla modalità di ripping scelta (sistematica o random).

3.4.1 Funzionamento della macchina driver

Come abbiamo anticipato la macchina denominata come "driver" è quella che mantiene il processo di ripping. Il funzionamento è analogo a quello esplicato nel paragrafo 3.2 in quanto vengono mantenute le classi SystematicRipper/RandomRipper e ClientSistThread/ClientRandThread nel caso sistematico/random. L'introduzione delle classi RemoteSystematicDriver nel caso sistematico e RemoteRandomDriver nel caso random, ha permesso di realizzare il funzionamento dell'algoritmo su macchine remote. Vediamo come agisce questo meccanismo inizialmente lato driver. La differenza sostanziale sta nel meccanismo di comunicazione tra driver ed emulatori. Questi ultimi si troveranno dislocati sulle macchine server e pertanto, il driver andrà a dialogare con esse mediante lo scambio di messaggi inviati su socket alla porta 2000. Le funzioni che realizzano lo scambio di messaggi appartengono alla classe RemoteActions e sono elencate in figura 3.13. Ad ognuna di queste funzioni è associato un messaggio inviato sottoforma di stringa alle macchine server. Per ogni comando inviato vi sarà un'azione corrispondente eseguita dalle macchine server. A titolo dimostrativo vengono riportate due funzioni della classe RemoteActions:

```
public static void startEmulator_Remote(int emuport,Socket sock,String Avd,int
emuport_remot)
	{
	try{
		String s=new String("avvia,"+emuport+","+Avd+","+emuport_remot);
		sendMessageInBytes(s,sock);
	}catch(IOException e){e.printStackTrace();}
	sleepSeconds(ANDROID_RIPPER_SERVICE_WAIT_SECONDS);

	}
```

La funzione startEmulator_Remote permette di inviare un messaggio del tipo "avvia" alle macchine server, concatenando ad esso informazioni come il porto dell'emulatore, il nome dell'Avd e il porto sul quale eseguire le redirezione. Quando la macchina server ricerverà il messaggio lo scomporrà nelle varie parti e sarà in grado di avviare l'emulatore col nome specificato sul porto indicato.

```
public static void KillEmul_Remote(int emuport,Socket sock,String Avd){
      try{


            String s=new String("kill,"+emuport+","+Avd);
            sendMessageInBytes(s,sock);


      }catch(IOException e){e.printStackTrace();}
      }
```

La funzione killEmul_Remote permette di inviare un messaggio del tipo “kill” alle macchine server, comandando la chiusura dell’emulatore con nome e porto specificato nei parametric della funzione.

L’algoritmo di ripping sarà lo stesso utilizzato per il funzionamento a singola macchina con la differenza che le funzioni che prima andavano a comandare direttamente gli emulatori saranno sostituite dalle funzioni remote indicate.

Figura 3.13:Class diagram per RemoteActions

3.4.2 Funzionamento della macchina server

Il funzionamento delle macchine server è racchiuso in tre classi appartenenti al package **it.unina.android.ripper.server:** AndroidRipperServer, AndroidRipperThread, UtilityRipperServer. Vediamo nel dettaglio i ruoli svolti da queste tre classi:

AndroidRipperServer: rappresenta il main dell'applicazione lato server. Tale classe non fa altro che mettersi in ascolto di richieste di connessione sulla porta 2000 tramite la classe ServerSocket. Per ogni nuova richiesta viene creato un thread del tipo AndroidRipperThread passando come argomento la socket restituita mendiante la funzione **accept()**.

UtilityRipperServer: contiene diverse funzioni che vengono utilizzate prima, durante e dopo il processo di ripping. Tra queste ricordiamo **public static String DirCopy(String avd_path)** che permette di copiare le avd nella cartella di base di Android, **public static String appInstall(int num_avd,int port,String OUTPUT_FOLDER,String appName),** che permette di installare l'applicazione **appName** sulle varie Avds presenti, **protected static void replaceStringsInFile(String filePath,boolean verso, String appPackage, String appMainActivity)** che permette di modificare l'AndroidManifest.xml e la classe Configuration.java di AndroidRipper aggiornandoli col nome del package e della main activity dell'applicazione da installare, ecc.

ServerRipperThread: permette di realizzare la comunicazione tra driver ed emulatori presenti sulla macchina server. Estende la classe Thread e ad ogni sua istanza corrisponde corrisponde un dato emulatore identificabile da un nome ed un porto sul quale è attivo.
Il costruttore per tale classe è il seguente:

```
AndroidRipperThread(Socket s){
this.socket=s;
this.running=true;
start();}
```

Figura 3.14:Costruttore di AndroidRipperThread

Esso riceve la socket dalla classe AndroidRipperServer e imposta a **true** la variabile booleana **running.** Il metodo run di tale classe contiene due cicli **while** innestati:

```
public void  run(){
int count=0;
while(running ){
      try{

                  while(true){ . . . . . .
```

Il metodo run termina quando la variabile running è posta a **false.** Il ciclo interno è sempre attivo. Quest'ultimo non fa altro che ciclare all'infinito in attesa di connessioni; ogni volta che riceve una nuova richiesta di connessione da parte di un client, genera un thread che si occuperà di quella specifica richiesta, si rimetterà in attesa di un'altra richiesta e ripeterà l'operazione. Nel nostro caso le richieste di connessione vengono inviate dalla macchina driver per permettere la ricezione dei messaggi che vengono inviati dal driver stesso sulla socket 2000. Il server non conosce lo stato di esecuzione del processo di ripping ma si limita semplicemente ad agire in base al messaggio ricevuto. Tale considerazione giustifica l'esistenza di un loop infinito dal quale si uscirà mediante il comando **break** richiamato secondo necessità. Vediamo quali sono i principali messaggi inviati dal driver e a quali azioni corrispondono:

Messaggio	**Azioni eseguite**	**Funzioni chiamate**
Copy	Copia le avd nella cartella android home	UtilityRipperServer.DirCopy UtilityRipperServer.updateFile
Install	Estrae l'applicazione in formato zip Legge il nome dell'applicazione e la installa sugli emulatori selezionati	UtilityRipperServer.getAppName UtilityRipperServer.unZip UtilityRipperServer.appInstall
avvia	Avvia l'emulatore sul primo porto	Actions.startEmulatorNoSnapshotSave
servizio	Avvia il service sull'emulatore indicato	Actions.startAndroidRipperService
to_emulator	Esegue il redirect sulla porta 8888	Actions.sendMessageToEmualtor(emuport, "redir add tcp:"+port+":"+8888);

crea	Avvia il caso di test sull emulatore indicato	Actions.startAndroidRipper

PING	Richiede una connessione verso l'emulatore, gli invia un messaggio di ping e inoltra la risposta al driver, tipicamente un messaggio di PONG	sock.connect()
		sock.ping();
		this.sendMessage(m,sock.getIndex());

DSC	Invia un messaggio di DESCRIBE all'emulatore ed inoltra la risposta al driver	Message message=sock.describe(); this.sendMessage(message, sock.getIndex());
EVT	Invia un messaggio di EVENT all'emulatore ed inoltra la risposta al driver	sock.sendMessage(event); sock.readMessage(3000);
pull_cov	Crea il file di copertura associate	Actions.pullCoverage
COVER	Invia un messaggio di COVERAGE all'emulatore ed inoltra la risposta al driver	sock.sendMessage(msg); sock.readMessage(3000);
home	Riproduce la pressione del tasto home sull'emulatore	Actions.sendHomeKey
killall	Chiude tutti i processi sull emulatore	Actions.killAll
preleva	Esegue il prelievo dei risultati parziali	UtilityRipperServer.transferFile
restart	Riavvia l'emulatore specificato	Actions.sendMessageToEmualtor(emuport, "kill")
END	Invia un messaggio di END all'emulatore ed esegue la disconnessione della socket	sock.sendMessage(end); sock.disconnect();
kill	Chiude l'emulatore sulla porta specificata	Actions.sendMessageToEmualtor(emuport, "kill")
close	Trasmette i files di copertura alla macchina driver e chiude l'esecuzione	UtilityRipperServer.transferFile

Figura 3.15:Tabella dei messaggi con azioni corrispondenti

Messaggi quali "close" o "kill" nel caso sistematico permettono di settare la variabile

running pari a **false**, causando la terminazione del **while** esterno successivamente al verificarsi di un **break** nel ciclo **while** interno.

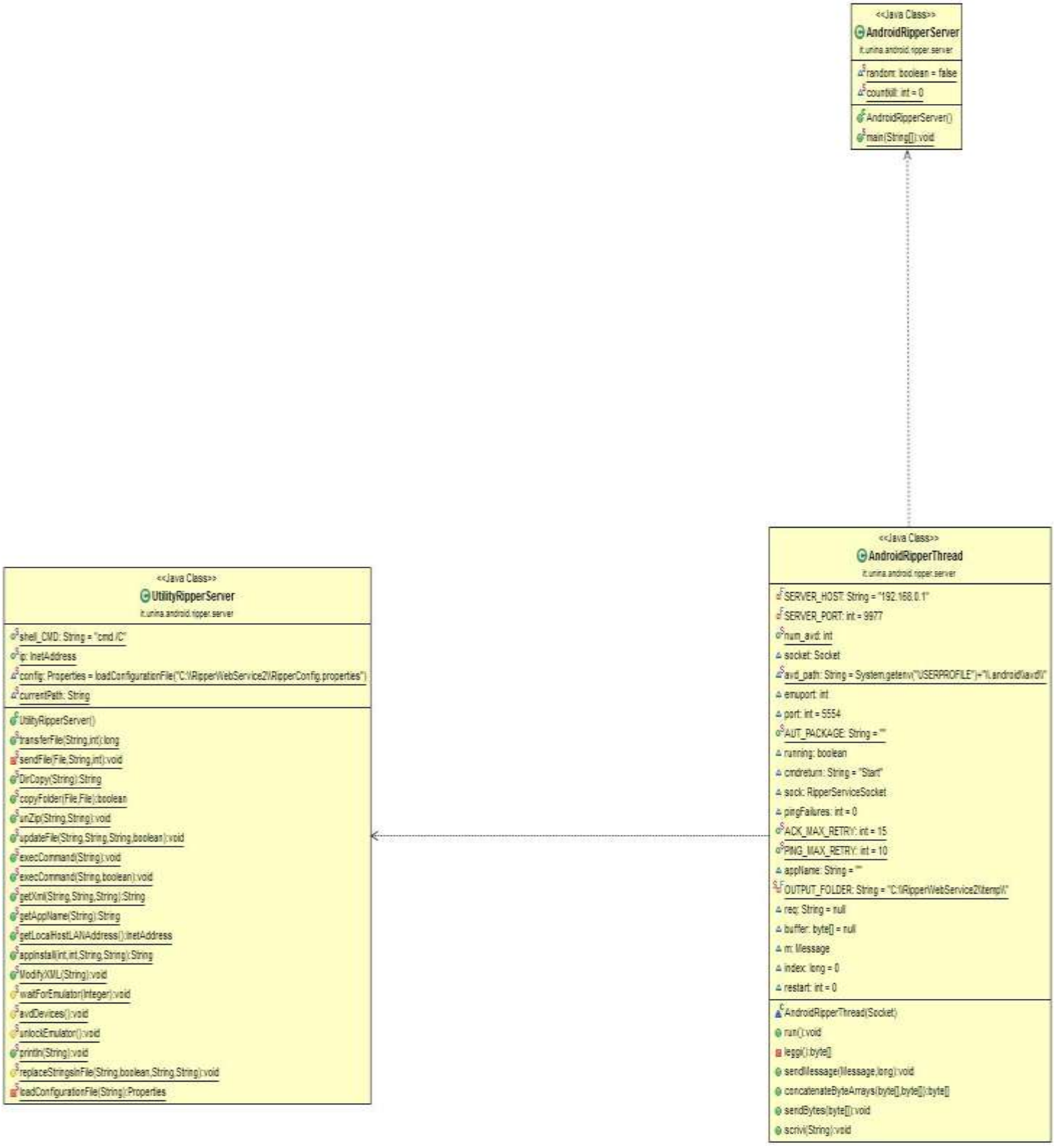

Figura 3.16:Class diagram di it.unina.android.ripper.server

Capitolo 4: Realizzazione di una web application per il Ripper

In questo capitolo tratteremo la web application realizzata che permetterà al generico utente di collegarsi al sito web dell'applicazione, di scegliere quali e quanti server utilizzare, installare la propria applicazione, lanciare il processo di ripping e prelevare i risultati parziali, annullare il processo o attendere la fine di quest'ultimo per poi scaricare i risultati finali sotto forma di file .ec. Le tecnlogie utilizzate sono html, servlet e Apache Tomcat 6.0 come Web Server.

4.1 Architettura di una web application

Nello sviluppo delle web application è possibile descrivere l' architettura del sistema scegliendo fra i molteplici paradigmi a disposizione. Fra questi trova una maggiore applicazione, la cosiddetta "architettura a tre livelli". Quest' ultima prevede che un sistema software sia decomposto in tre livelli distinti ma in comunicazione tra loro secondo un'opportuna gerarchia. I tre livelli in questione sono i seguenti:

Presentation layer: è il livello di presentazione, il cui compito è quello di interagire direttamente con l'utente del sistem, acquisire i dati in input immessi da quest ultimo e visualizzare i risultati dell'elaborazione effettuata dal sistema stesso. Esso, in pratica, definisce la GUI dell'applicazione;

Application processing layer: è il livello in corrispondenza del quale si trova la "business-logic" dell'applicazione e quindi tutti i moduli software che implementano le

funzionalità che il sistema mette a disposizione. In sostanza, è il centro dell'elaborazione dei dati in cui avvengono tutte le computazioni;

Data management layer: è il livello che si occupa della gestione della persistenza e dell'accesso ai dati, per cui è tipicamente caratterizzato da un DBMS.

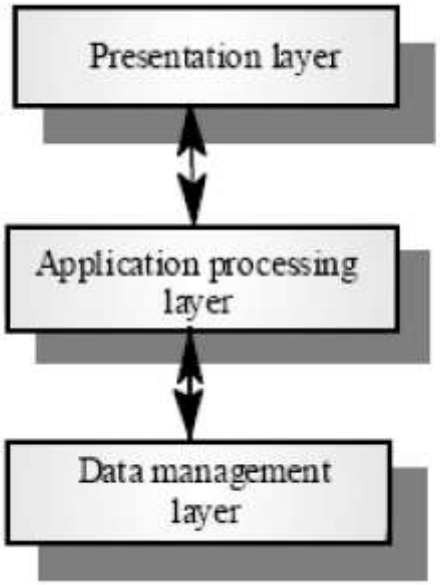

Figura 4.1: architettura software a livelli

Sviluppando un'applicazione secondo questa architettura, ognu livello è indipendente dagli altri, per cui la modifica di uno di essi non ha effetto sui restanti. Tuttavia è prevista la comunicazione fra loro e lo scambio di informazioni. Un tipico scenario di funzionamento del sistema puo essere il seguente: un utente utilizza l'applicazione, interagendo direttamente con la GUI e fornisce quindi al Presentation layer, i dati su cui andrà eseguita l'elaborazione. Il Presentatio layer, acquisiti i dati in input, li trasferisce all'Applicatin processing layer che esegui su di essi una determinata computazione.
Durante l'elaborazione, la business logic puo prevedere la memorizzazione persistente dei dati oppure la necessità di acquisire ulteriori dati gia memorizzati. In questo caso, c è l'interazione con il Data management layer, il quale memorizza i dati che gli vengono passati dal livello superiore, oppure recupera da un Data Source i dati richiesti e li

trasmette alla bussines logic. Al termine dell'elaborazione i risultati vengono passati al Presentation layer che li visualizza in una certa forma all'utente finale.[13]

4.2 Apache Tomcat

Esiste una gran quantità di application server, la scelta deve essere fatta tenendo conto dei linguaggi utilizzati per scrivere le pagine Web che compongono l'applicazione. Nel nostro caso la scelta è ricaduta su Tomcat di Apache Software Foundation3. Tomcat è un ottimo web container open source che fornisce una piattaforma per applicazioni web sviluppate attraverso il linguaggio Java. Oltre che alla funzionalità di web server tradizionale (offerta da Apache) implementa le specifiche JSP e Servlet di Sun Microsystems. Tomcat quindi è un Servlet Container ed un JSP Engine allo stesso tempo: un motore che è in grado di eseguire lato server applicazioni web basate sulla tecnologia J2EE e costituite da componenti Servlet e da pagine JSP.

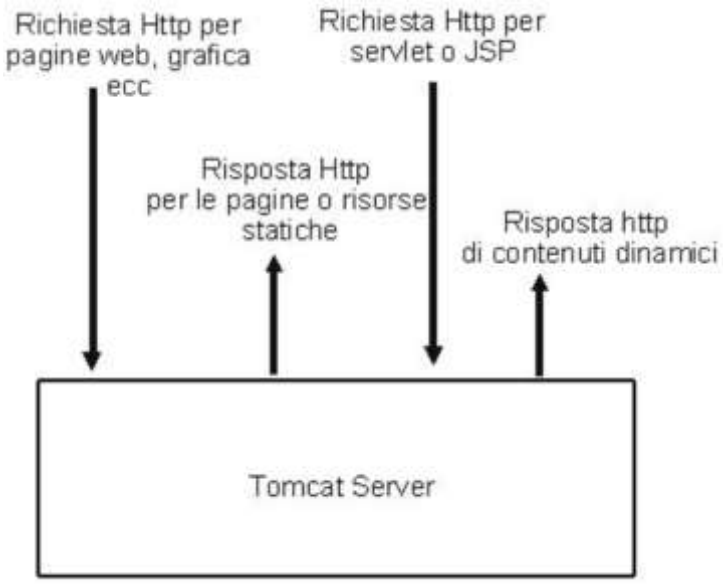

Figura 4.2: Tomcat standalone

Tomcat è inoltre interamente scritto in java e può essere eseguito su una qualsiasi architettura su cui si installata un JVM.[12]

4.3 La Web application Android Ripper Web Application

La web application realizzata è stata denominata Android Ripper Web Application e permette ad un generico utente di accedere al sito web d'interesse, scegliere fra i server disponibili quelli su cui far girare il ripper, caricare la proprio app da testare e lanciare il processo di ripping nella strategia desiderata. Analizzeremo nel dettaglio i files e le classi introdotte per la realizzazione di tale web app:

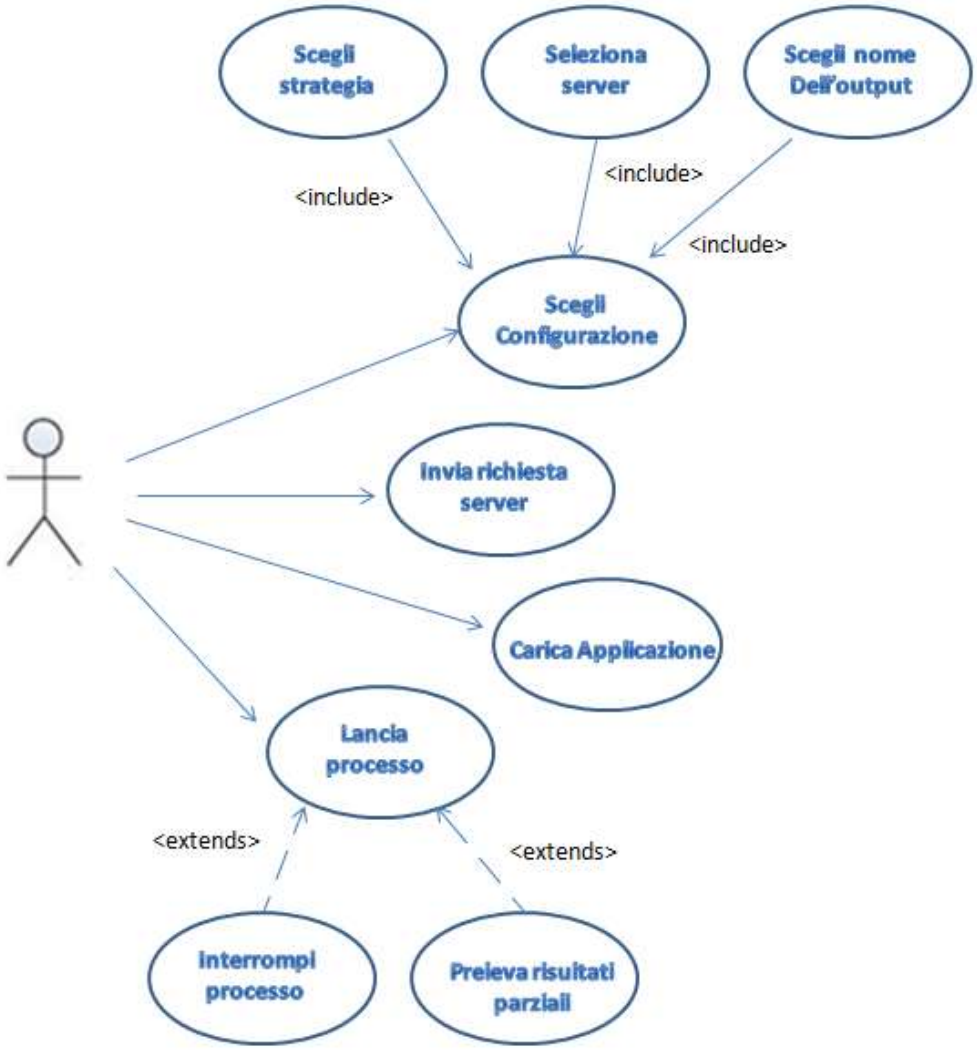

Figura 4.3: Use Case Diagram della web app

index.html: rappresenta la home page della web app. Permette tramite il clic del pulsante "Invia richiesta server" di accedere alla servlet Scanner.java che permetterà a sua volta di stabilire i parametri principali per il processo di ripping:

Figura 4.4: home della web app

Scanner.java: tale servlet legge da file di testo l'ip della macchina driver e gli ip delle macchine server, questi ultimi nel formato ip:numeroavd. Per verificare quali server siano disponibili, la servlet esegue una richiesta di connessione tramite socket all'ip sotto test, se si ha un eccezione allora il server non è disponibile, in caso contrario l'ip sarà visualizzato a video e potrà essere selezionato dall'utente. Inoltre la servlet presenta un form che permetterà di selezionare la strategia di ripping, sistematico o random; in quest'ultimo caso sarà necessario specificare il numero di eventi da eseguire. Dovranno essere selezionati i server su cui eseguire il processo fra quelli disponibili e infine il nome del file di output prodotto. Cliccando su invia si viene indirizzati alla servlet CopyAvd:

Figura 4.5: Scelta dei parametri per il ripper

CopyAvd.java: questa servlet permette di copiare le avd di base nella cartella Android home di ogni macchina server. Tale operazione è necessaria in quanto permette di ottenere delle avd vergini all'inizio di ogni processo di ripping. Le avd di base sono salvate in una cartella denominata BasicAvds. Si è preferito operare in questo modo in quanto esso risulta più pratico e veloce rispetto alla creazione di nuove avd ad ogni nuovo test. Una volta terminata l'operazione di copia l'app rimanda alla servlet UploadServlet:

Figura 4.6: Copia in Corso

UploadServlet.java: permette all'utente di caricare l'app da testare. Affinchè si possa procedere, l'applicazione dovrà essere compressa in formato .zip.

Figura 4.7: Upload dell'applicazione

Una volta eseguita tale operazione, il file verrà inviato alla macchina driver e da questa smistata alle varie macchine server.

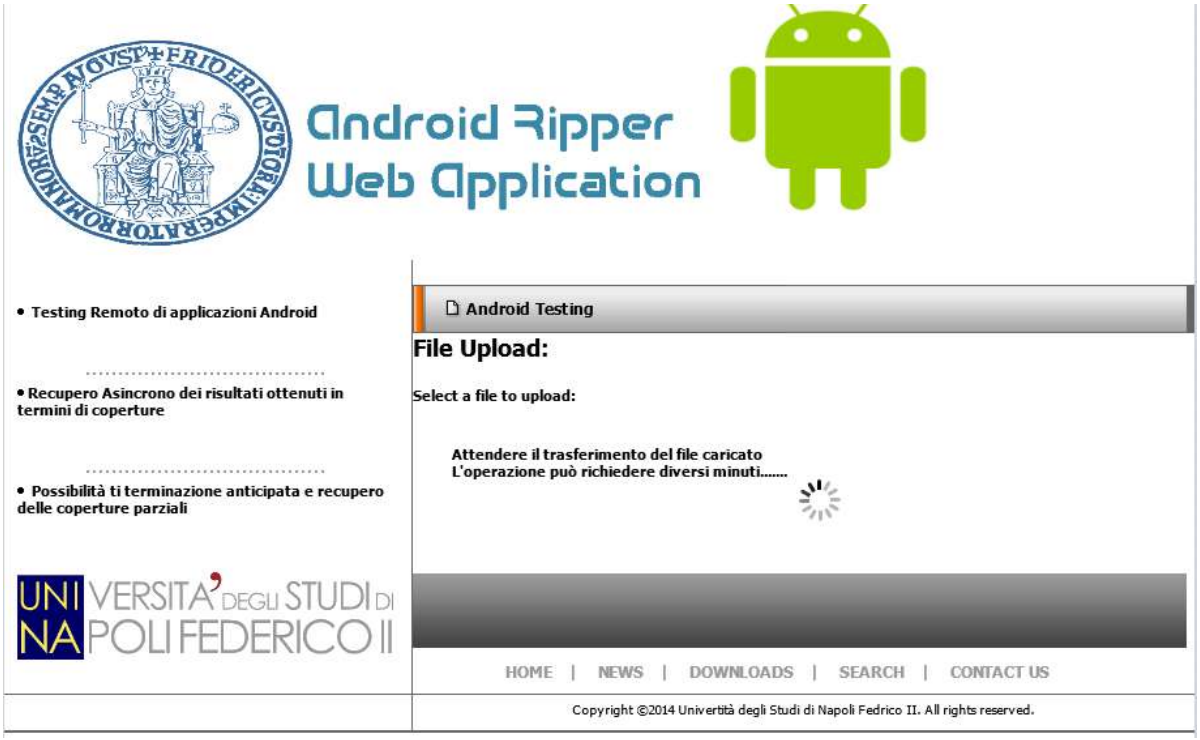

Figura 4.8: Trasferimento dell'applicazione

Terminata tale operazione si verrà indirizzati verso la servlet AppInstall

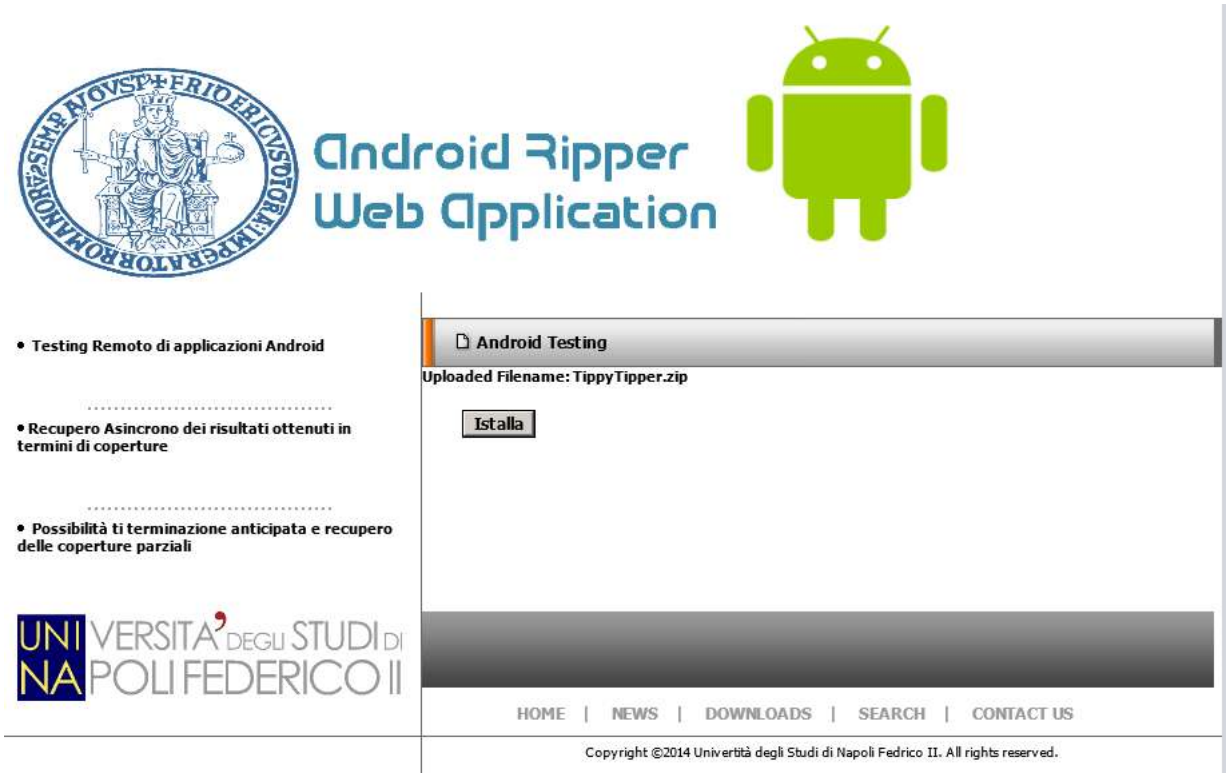

Figura 4.9: App pronta ad essere installata

AppInstall.java: è la servlet che permette di installare l'applicazione da testare sugli emulatori. Per prima cosa si ottiene il nome dell'applicazione a partire dal file .zip ricevuto. Quest'ultimo viene estratto in modo da ottenere la cartella contenente l'applicazione da testare. Si accede all'AndroidManifest.xml dell'applicazione e si leggono i valori del app package e della main activity; questi ultimi verrano inseriti nell'AndroidManifest.xml e nella classe Configuration.java dell'AndroidRipper. Successivamente si esegue l'installazione vera e propria sui vari emulatori. Quando questa termina viene salvato lo snapshot dell'emulatore e ripristinati nell'AndroidManifest.xml e Configuration.java dell'AndroidRipper predisponendosi per nuove e successive installazioni.

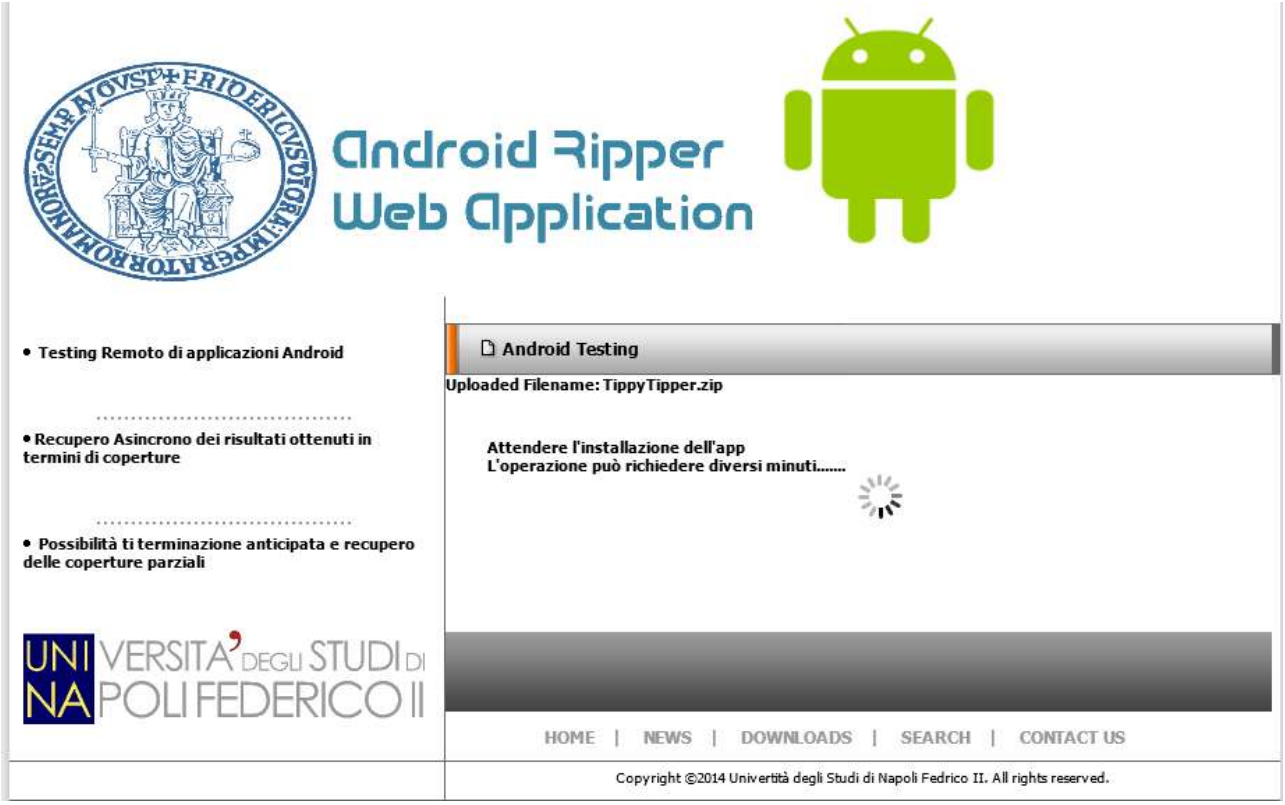

Figura 4.10: Installazione in corso

Terminata la fase di installazione si potrà lanciare il ripper premendo sul tasto esegui rimandando alla servlet WebMain:

Figura 4.11: Lancio del ripper

WebMain.java: è la servlet che riceve tutti i parametri scelti nei passaggi fra le varie servlet ed ha il compito fondamentale di andare a scriver il file di configurazione config.properties dal quale il driver attinge per configurare l'intero processo: numero di server, ip dei server scelti, nome del package, nome del app, numero eventi, ecc., inoltre viene lanciato il processo di ripping richiamando il main del driver, sia esso sistematico o random.

Figura 4.12: Riepilogo del file di configurazione

Premendo ok sulla finestra si viene indirizzati alla servlet Execution.

Execution.java: è la servlet che mostra i progressi del processo di ripping. In particolare evidenzia il numero di eventi correntemente eseguiti. Per fare ciò si legge la variabile nEvents del driver, la quale viene mostrata a video e aggiornata tramite refresh della pagina ogni dieci secondi.

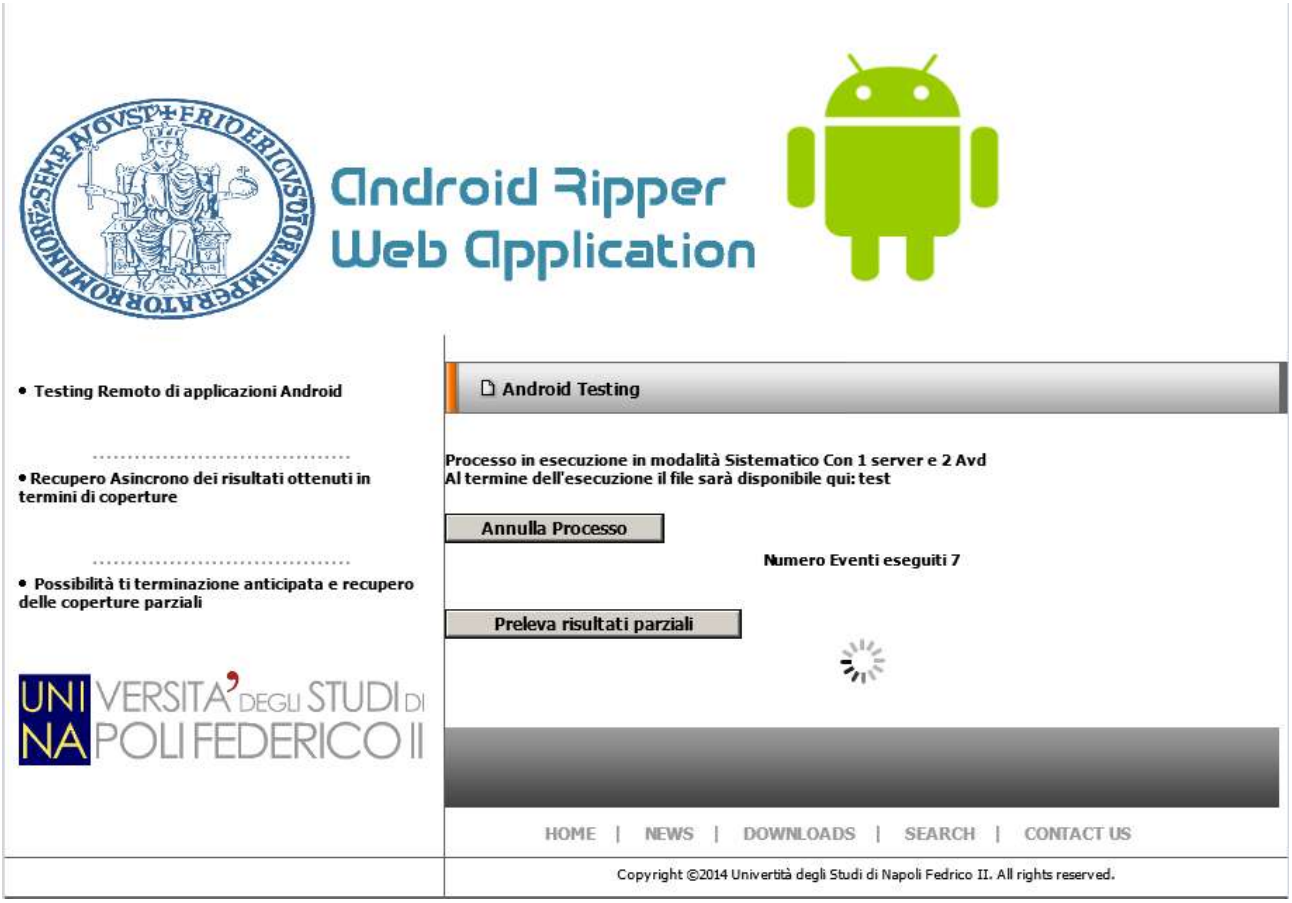

Figura 4.13: Esecuzione in corso

A partire da questa servlet, mediante la pressione di due pulsanti è possibile prelevare i risultati parziali o annullare il processo di ripping. Lasciando terminare l'esecuzione si giunge alla servlet End.

Stop.java: a tale servlet si giunge premendo il tasto annulla processo dalla servlet Execution.java. Tale servlet permette di annullare il processo di ripping in corso fermando

l'esecuzione del driver, chiudendo gli emulatori e indirizzando alla servlet Fine per ottenere il file di output finale.

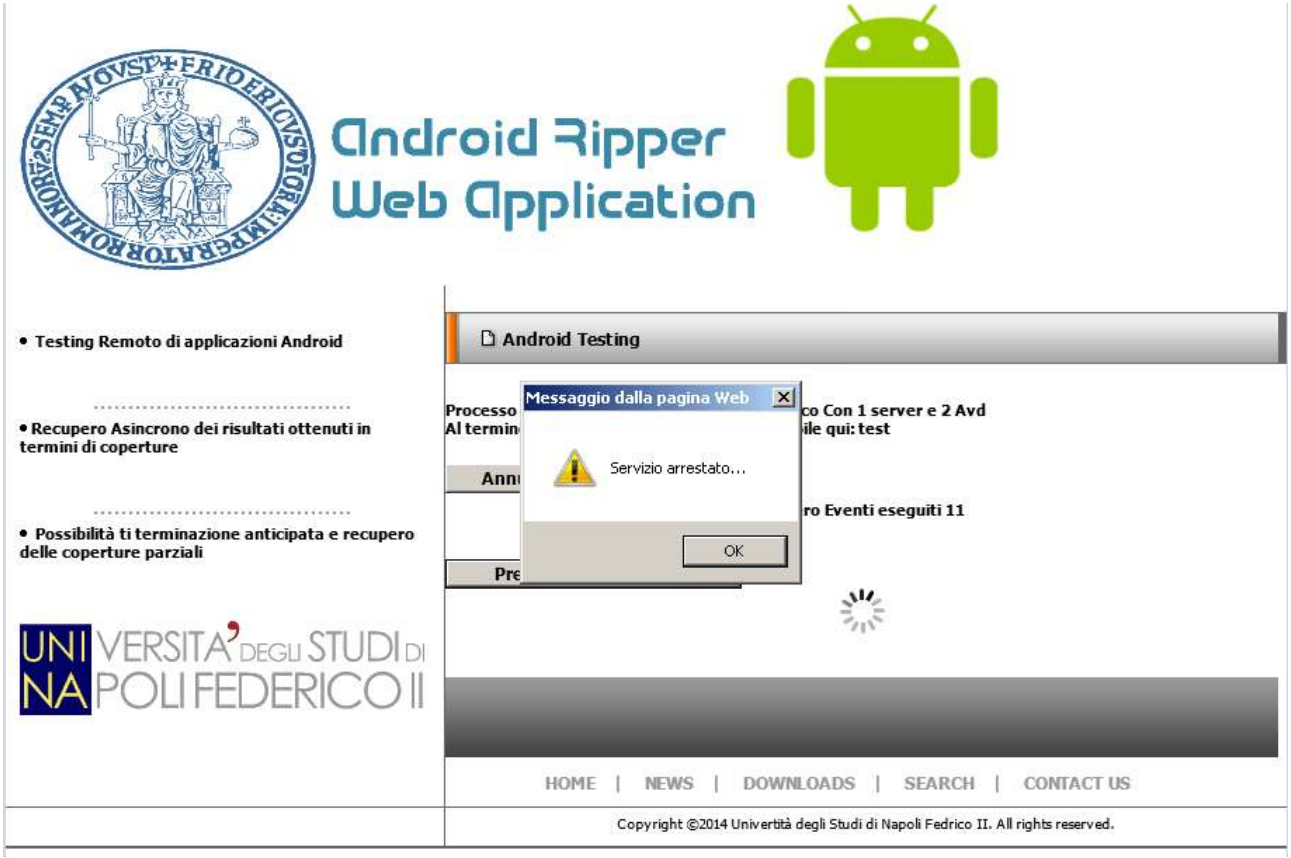

Figura 4.14: Annullamento processo

ParzialResult.java: permette di prelevare i risultati parziali, ovvero i files .ec fino a quel momento prodotti, lasciando inalterato il processo di ripping che continuerà la propria esecuzione

Fine.java: rappresenta la servlet conclusiva del processo: una volta che il test si conclude si viene indirizzati a tale servlet la quale eseguirà la compressione dei file .ec ottenuti in un file .zip denominato con il nome indicato nel form iniziale.

Download.java: è la servlet finale a partire dalla quale l'utente può scaricare sul proprio

pc, in formato .zip il risultato finale. Tale output conterrà tutti i file .ec prodotti durante il processo di ripping

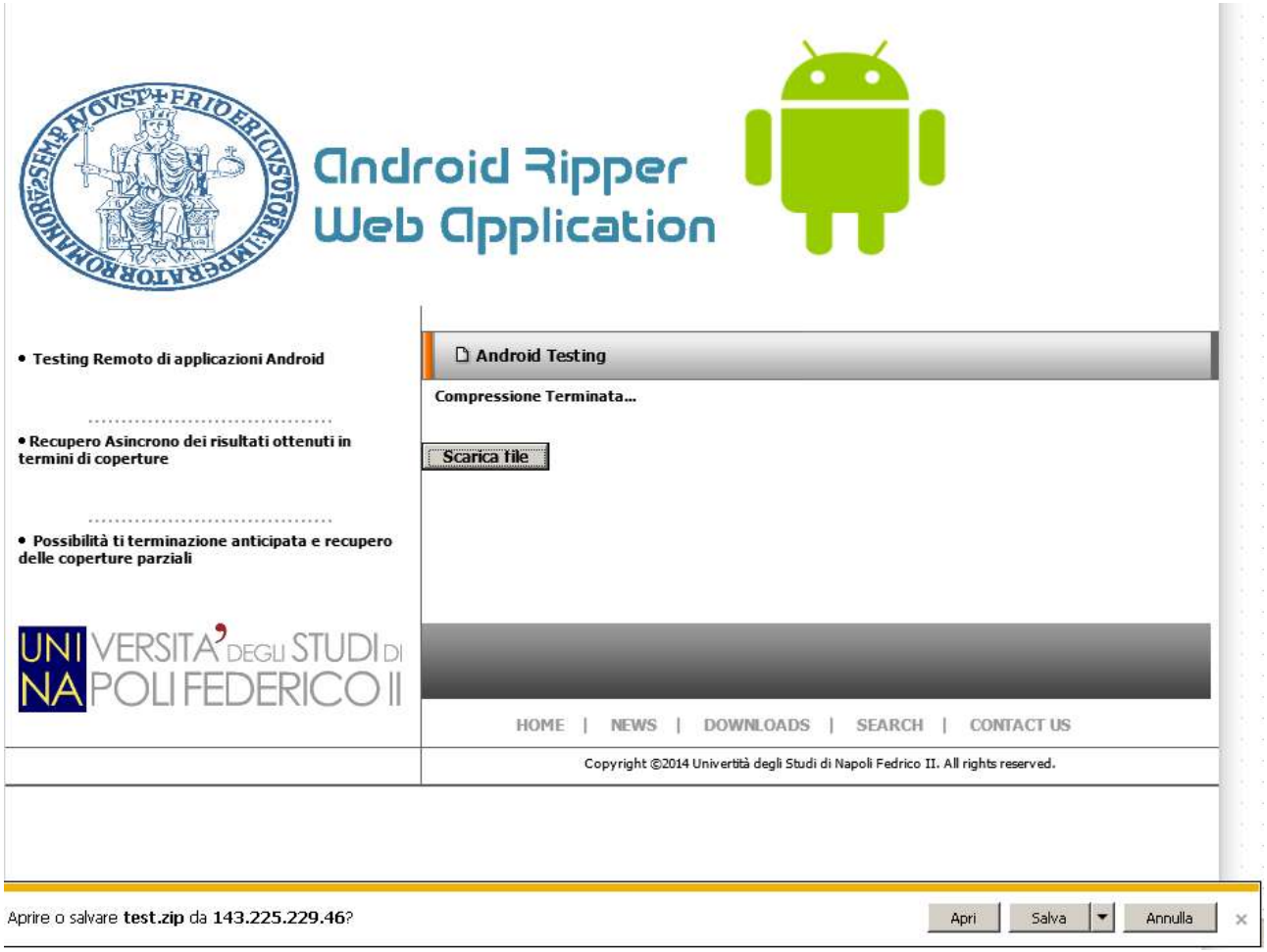

Figura 4.15: Download risultato

Oltre alle Servlet descritte, sono state realizzate altre classi di uso generale: UtilityClass e Thread slave.

UtilityClass: classe all'interno della quale sono implementati diversi metodi richiamati dalle varie servlet, come **static boolean zipDir(String zipFileName, String dir)** che permette di zippare la cartella dei risultati finali, **public void printProperties(String numAvd,String numServer,ArrayList <String> server,String mod,String event)** che

permette di stampare a video il file di configurazione, **byte[] leggi(final Socket so)**, che permette di leggere il messaggio in arrivo.

ThreadSlave: classe che estende Thread e si occupa di gestire la comunicazione tra Server e web application in modo da sincronizzare le varie fasi del processo.

4.3.1 Trasferimento di files tra le entità

Abbiamo visto che, affinché tutto possa funzionare correttamente, è necessario che driver e server si scambino tra loro dei files. Tale compito è svolto dalle classi del package **it.unina.android.ripper.filetransfer:**

ServerTransfer: è un main composto da un server in ascolto di richieste di connessione sulla porta 9977 e si occupa di lanciare il metodo run della classe ReceiverManager.

ReceiverManager: estende Thread ed il suo metodo run si occupa di intercettare i files in arrivo sulla socket specificata. Viene poi effettuato un controllo: se il file ha estensione .zip viene posto nella cartella temp dalla quale è poi prelevato ed inviato a tutte le macchine server; se il file ha estensione .ec allora abbiamo un file di copertura prodotto dal processo di ripping e la sua destinazione sarà la cartella coverage. Infine, alla terminazione dell'intero processo, tutti i file di copertura delle varie macchine server verranno inviate alla macchina driver che si occuperà di raggrupparli in un'unica cartella e zipparla per renderla poi disponibile mediante download. La funzione che si occupa di trasferire i files è **public static long transferFile(String server_host,int server_port)** e viene più volta invocata sia nelle servlet della web application sia dalla classe ServerThread sulle macchine server.

In figura 4.15 è invece rappresentato il class diagram del package **it.unina.android.ripper.filetransfer**

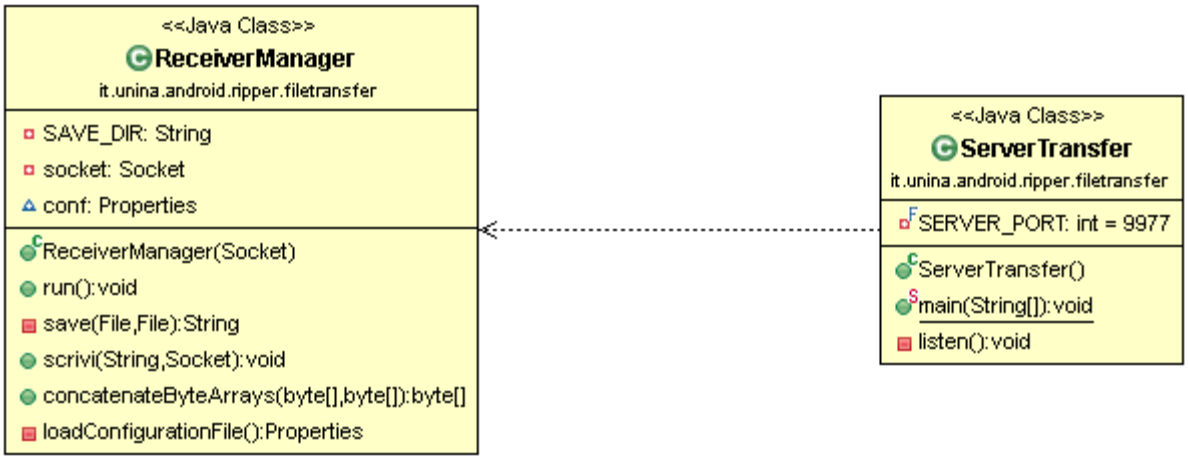

Figura 4.16: class diagram del package it.unina.android.ripper.filetransfer

4.4 Gestione delle eccezioni

Vediamo come sono gestiti i casi particolari come cadute di connessioni, interruzioni da parte dell'utente, ecc.

- **Nessun server disponibile**: quando l'utente avvia il processo inviando una richiesta per la visualizzazione dei server disponibili potrebbe capitare che nessuno di questi sia attivo. In quest'ultimo caso viene mostrato un alert con la comunicazione "Nessun server disponibile, riprovare più tardi" e l'utente è reindirizzato alla pagina iniziale.
- **Crash dell'App**: il crash dell'app è gestito dal driver. In particolare quando ciò si verifica, viene segnalato il crash e il processo continua la sua esecuzione: il sistematico semplicemente si chiude e parte dalla traccia successiva (l'emulatore si riavvia); il caso random prova a riavviare l'esplorazione, se non ci riesce riparte da zero riavviando l'emulatore.
- **Caduta della connessione Client-Server**: la caduta della connessione client-server, una volta avviato il processo di ripping, non da problemi in termini di esecuzione in quanto quest'ultima è indipendente da essa. Tuttavia sarebbe

impossibile per l'utente recuperare la pagina web dell'esecuzione e di conseguenza non potrebbe ne annullare ne terminare il processo e scaricare il file di output. Tale situazione non è ancora stata pienamente gestita ma si è pensato di mostrare a video, durante la fase di esecuzione, l'indirizzo web al quale l'utente potrà scaricare l'output prodotto. Sarà premura dell' utente salvare tale indirizzo in modo da accedervi successivamente in caso di caduta della connessione.

- **Caduta della connessione Server-Server:** in caso di caduta di un server, l'esecuzione del processo di ripping continuerà tranquillamente con un server in meno.
- **Interruzione utente**: l'utente può decidere di interrompere in qualsiasi momento l'esecuzione del processo cliccando sul tasto "Annulla processo" a partire dalla pagina dell'esecuzione in corso. Tale azione comporterà l'interruzione del processo di ripping, la chiusura degli emulatori sulle macchine server e la produzione dei risultati parziali prodotto fino a quel momento che saranno pronti per il download. In alternativa l'utente può prelevare i risultati parziali cliccando sul tasto "Preleva risultati parziali" ma ciò non comporterà l'annullamento del processo.

Inoltre al termine di ogni esecuzione i server sono automaticamente predisposti per accettare una nuova richiesta da parte dell'utente.

Conclusioni e Sviluppi Futuri

Il lavoro di tesi prodotto ha permesso di ottenere un'architettura complessiva più efficiente mediante il funzionamento parallelo e remoto del processo di ripping. Inoltre la realizzazione della web application rende accessibile il sistema ad un qualsiasi utente. Per quanto riguarda gli sviluppi futuri è prevista la possibilità di rendere l'applicazione multiutente. Inoltre si è pensato di implementare in futuro, un sistema che tramite invio automatico di mail possa inviare all'utente, precedentemente registrato, l'output finale in formato .zip; in questo modo ci sarebbe un'ulteriore soluzione alla problematica della caduta della connessione client-server. Infine è preventivata la possibilità di eseguire il processo di test a partire da un esperimento già iniziato (resume).

Appendice 1: How-To installazione del sistema

In quest'appendice vedremo i passi da eseguire per configurare le macchine sulle quali installare il sistema, sia per quel che riguarda la macchina driver, sia per quel che riguarda le varie macchine server.

A1.1 Requisiti

- Windows XP o superiore
- JDK v.1.6.x con x<=43 (scaricabile qui)
- Android sdk v.18 o superiore (scaricabile qui)
- Librerie per Android 2.3.3 o superiore (scaricabile tramite Android SDK Manager)
- Ant v.1.8.2 o superiore (scaricabile qui)

A1.2 Operazioni preliminari

Tali operazioni devono essere eseguite su ogni macchina, sia driver che server e consistono nel settaggio di alcune variabili d'ambiente:

- ANDROID_HOME deve essere settata come path di Android sdk (es: C:\Android\android-sdk)
- JAVA_HOME deve essere settata come path di Java jdk (es: C:\Java\jdk1.7.0_25)
- La variabile d'ambiente PATH deve contenere i seguenti percorsi:

  - cartella bin di ant (es: C:\apache-ant-1.9.0\bin)
  - cartella bin di Java jdk (es: C:\Java\jdk1.7.0_25\bin)
  - cartella platform-tools di Android (es: C:\Android\android-sdk/platform-tools)

- cartella tools di Android (es: C:\Android\android-sdk\tools)
- cartella contenente il file aapt.exe.

Un esempio completo è:

- ANDROID_HOME="C:\Android\android-sdk"
- JAVA_HOME="C:\Java\jdk1.7.0_25"
- PATH="%PATH%;C:\Android\android-sdk/platform-tools;C:\Android\android-sdk\tools;C:\Android\android-sdk\buildtools\17.0.0;C:\Ant\bin;C:\Java\jdk1.7.0_25\bin".

A1.3 Operazioni macchina driver

Elenchiamo le operazioni sa eseguire sulla macchina driver. Quest'ultima dovrà essere unica:

- Installare Tomcat 6 (scaricabile qui) e impostare relativa username e password.
- Estrarre le cartelle RipperWebService2 e RipperTesting fornite dal pacchetto di installazione in formato .zip in C:/
- Aprire il file RipperConfig.properties in C:/RipperWebService2/ e modificare l'ip-driver con l'ip della macchina driver

```
1 ip_driver=143.225.229.234
2 ip_server=143.225.229.234
3 aut_package=com.android.project
4 aut_main_activity=com.android.project.TicTacToe
5
6
```

Figura A.1 Modifica ip_driver

- Lanciare Tomcat da riga di comando

```
C:\Program Files\Apache Software Foundation\Tomcat 6.0\bin>tomcat6
ott 17, 2014 4:34:48 PM org.apache.catalina.core.AprLifecycleListener init
INFO: Loaded APR based Apache Tomcat Native library 1.1.30 using APR version 1.4
.8.
ott 17, 2014 4:34:48 PM org.apache.catalina.core.AprLifecycleListener init
INFO: APR capabilities: IPv6 [true], sendfile [true], accept filters [false], ra
ndom [true].
ott 17, 2014 4:34:50 PM org.apache.catalina.core.AprLifecycleListener initialize
SSL
INFO: OpenSSL successfully initialized with version OpenSSL 1.0.1g 7 Apr 2014
ott 17, 2014 4:34:50 PM org.apache.coyote.http11.Http11AprProtocol init
INFO: Initializing Coyote HTTP/1.1 on http-8080
ott 17, 2014 4:34:50 PM org.apache.coyote.ajp.AjpAprProtocol init
INFO: Initializing Coyote AJP/1.3 on ajp-8009
ott 17, 2014 4:34:50 PM org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 3128 ms
ott 17, 2014 4:34:51 PM org.apache.catalina.core.StandardService start
INFO: Starting service Catalina
ott 17, 2014 4:34:51 PM org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/6.0.41
ott 17, 2014 4:34:51 PM org.apache.catalina.startup.HostConfig deployDescriptor
INFO: Deploying configuration descriptor host-manager.xml
ott 17, 2014 4:34:51 PM org.apache.catalina.startup.HostConfig deployDescriptor
INFO: Deploying configuration descriptor manager.xml
ott 17, 2014 4:34:52 PM org.apache.catalina.startup.HostConfig deployWAR
INFO: Deploying web application archive RipperWebService2.war
ott 17, 2014 4:34:53 PM org.apache.catalina.startup.HostConfig deployDirectory
INFO: Deploying web application directory docs
ott 17, 2014 4:34:53 PM org.apache.catalina.startup.HostConfig deployDirectory
INFO: Deploying web application directory examples
ott 17, 2014 4:34:53 PM org.apache.catalina.startup.HostConfig deployDirectory
INFO: Deploying web application directory ROOT
ott 17, 2014 4:34:53 PM org.apache.coyote.http11.Http11AprProtocol start
INFO: Starting Coyote HTTP/1.1 on http-8080
ott 17, 2014 4:34:53 PM org.apache.coyote.ajp.AjpAprProtocol start
INFO: Starting Coyote AJP/1.3 on ajp-8009
ott 17, 2014 4:34:53 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 2930 ms
```

Figura A.2 Avvio di Tomcat 6

- Effettuare il deploy della WebApp RipperWebService2.war presente in C:/RipperWebService2/ utilizzando il manager di Tomcat 6

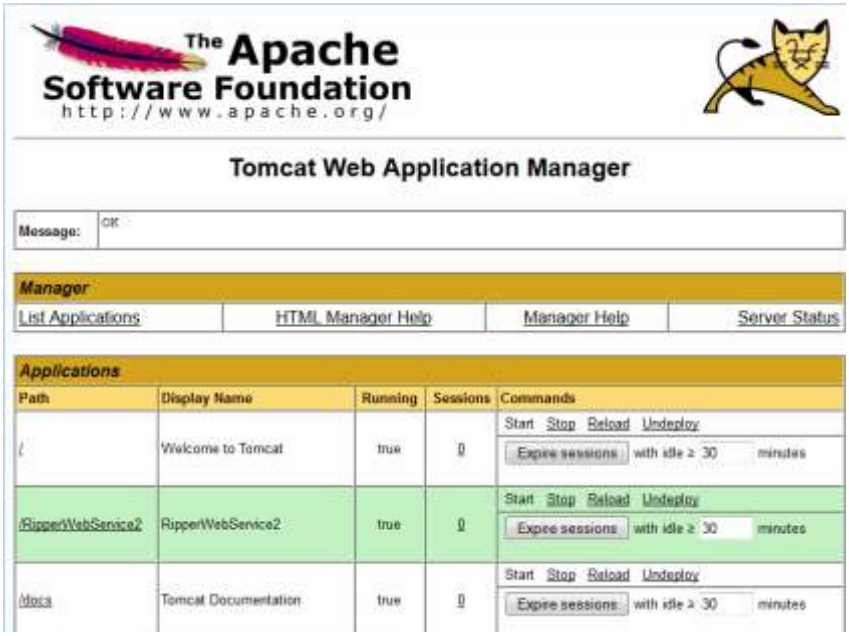

Figura A.3 Tomcat Manager

- Disattivare Windows Firewall
- Condividere le cartelle RipperWebService2 e RipperTesting nella rete locale
- Scrivere nel file IpServerConfig.txt gli indirizzi della macchine server di cui si dispone nella forma ip:numAvd

Figura A.4 IpConfig.txt

- Avviare il jar ServerTransfer.jar presente nella cartella C:/RipperWebService2/

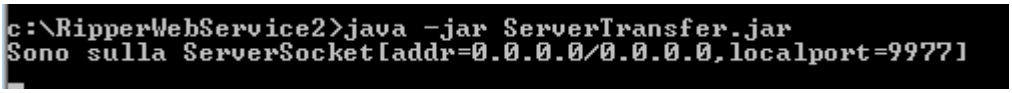

Figura A.5 Avvio di ServerTransfer.jar

A1.4 Operazioni macchina server

Elenchiamo le operazioni da eseguire sulla macchina server generica. In questo caso il numero di macchine è a discrezione dell'installatore:

- Estrarre la cartella RipperWebService2 fornita dal pacchetto di installazione in formato .zip in C:/ e condividerla
- Avviare AVD Manager per la creazione delle avd:
  - i nomi delle avd devono seguire questo schema: avd_ripper_2_3_i con i(3.....n);
  - le caratteristiche delle avd da creare sono: Device: Nexus S(480X800 hdpi); Sd card size=64 ; selezionare il flag "Snapshot";
  - una volta create, lanciare ogni avd con la sola casella "Save snapshot" selezionata;
  - Spostare le avd create ed i rispettivi files avd_ripper_2_3_i.ini dalla cartella di creazione (es.C:\Users\.....\.android\avd) nella cartella C:/RipperWebService2/

BasicAvds;

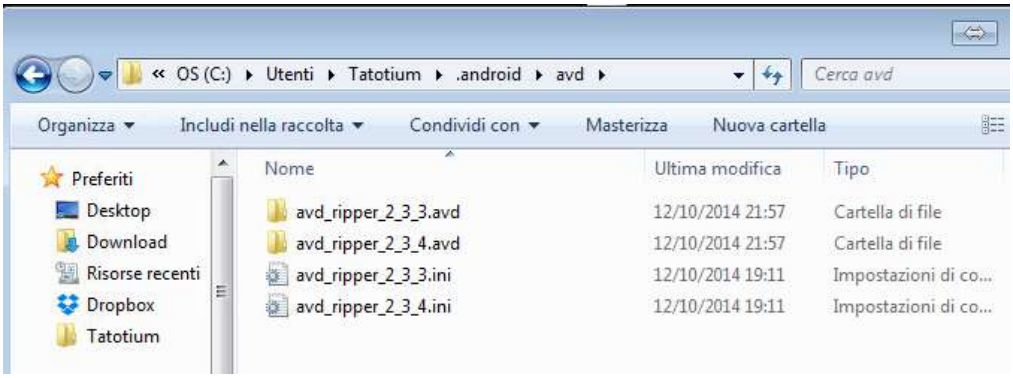

Figura A.6 Cartella delle Avd

- Nella Cartella C:/RipperWebService2/BasicAvds cambiare manualmente i path delle avd presenti nei file: avd_ripper_2_3_i.ini; avd_ripper_2_3_i/hardware-qemu.ini; avd_ripper_2_3_i/snapshots.img.default-boot.ini sostituendo tutte le occorrenze di C:\Users\...\.android\avd con C:/RipperWebService2/BasicAvds;

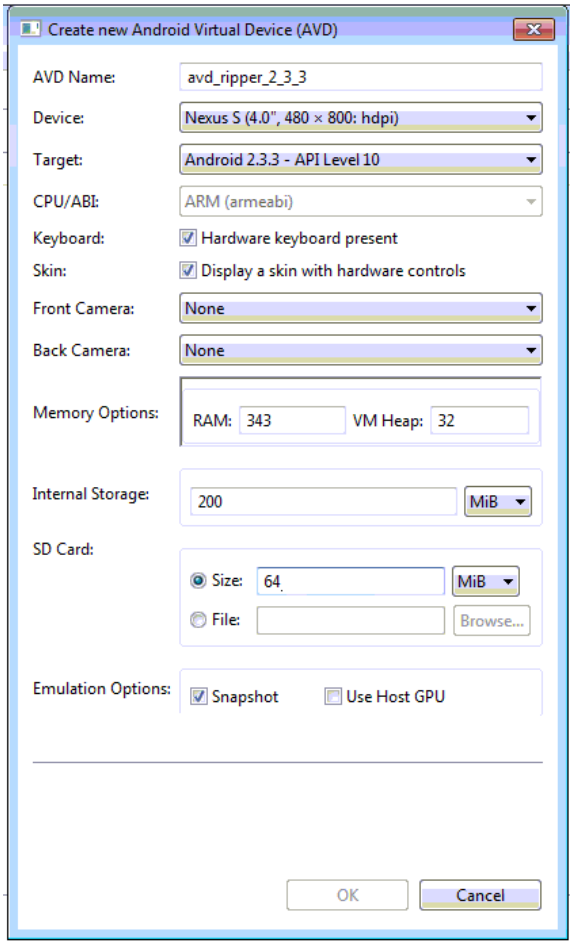

Figura A.7 Creazione Avd

- Aprire il file RipperConfig.properties in C:/RipperWebService2/ e modificare l'ip_driver con l'ip della macchina driver e l' ip_server con l'ip della macchina server;

```
ip_driver=143.225.229.234
ip_server=143.225.229.247
aut_package=net.mandaria.tippytipper
aut_main_activity=net.mandaria.tippytipper.activities.TippyTipper
```

Figura A.8 Modifica ip_server

- Avviare i jar ServerTransfer.jar e RipperServer.jar presenti nella cartella C:/RipperWebService2/;

```
C:\RipperWebService2>java -jar ripperserver.jar
Avvio server...
```

Figura A.9 Avvio RipperServer.jar

- Disattivare Windows Firewall;

A1.5 Operazioni Client

Affinchè l'utente possa utilizzare il sistema dovrà:

- connettersi alla pagina web http://"ip_driver"/html.index
- selezionare i parametri per il ripper (strategia, server, nome dell'output)
- caricare l'app da installare in formato .zip

Bibliografia

[1] Carli M. . Android guida per lo sviluppatore. Apogeo, 2010.

[2] Tapas Kumar Kundu, Kolin Paul . Android on Mobile Devices: An Energy Perspective. Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, pp 2421 – 2426, June 29 2010-July 1 2010

[3] Aravind MacHiry, Rohan Tahiliani, Mayur Naik. Dynodroid: An Input Generation System for Android Apps, ESEC/FSE 2013 Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp 224-234.

[4] Marco Boemo. Customizzazione di Android. Capitolo 2. Tesi di dott. Università degli studi di Bologna, 2011-2012.

[5] Roger S. Pressman, "Software Engineering: A pratictioner's approach", 5th edition - McGraw Hill, 2001

[6] JUnit Testing Framework: http://www.junit.org/

[7] SeleniumWeb Application Testing System: http://seleniumhq.org/

[8] Selenium IDE: http://seleniumhq.org/projects/ide/

[9] Nicola Amaucci. Gui testing automatico di applicazioni Android tramite emulazioni di input ed eventi provenienti da sensori. Tesi di laurea Università degli studi di Napoli Federico II, 2011-2012.

[10] Salvatore De Carmine. Tecniche di crawling per il testing automatico di applicazioni Android.. Tesi di laurea Università degli studi di Napoli Federico II, 2010-2011.

[11]Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning, Wontae Choi, George Necula, Koushik Sen, EECS Department University of California, Berkeley

[12] Mattia Colombari. Un sistema per la pubblicazione automatica via web di video lezioni. Tesi di laurea in Informatica Università degli studi di Trento 2008/2009.

[13] Paolo Patierno. Framework MVC per lo sviluppo di Web Application: JavaServerFaces e Struts. Tesi di laurea Università degli studi di Napoli Federico II, 2004-2005.