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

# *Sperimentazione 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
**Luca Nastro**
**matr. M63000011**

“*Energia e perseveranza ti possono far raggiungere ogni traguardo*”
**Benjamin Franklin**

# Indice

Indice ..... IV
Introduzione ..... 6
Capitolo 1: Ambiente Android ..... 8
1.1 Perché Android ..... 8
1.2 Che cos'è Android ..... 9
1.3 Architettura di Android e la Dalvik Machine ..... 10
1.3.1 Core Libraries e Libraries ..... 11
1.3.2 Application Framework ..... 12
1.3.3 Applications ..... 14
1.4 Architettura delle applicazioni Android ..... 14
1.4.1 Activity ..... 15
1.4.2 I Service ..... 17
1.4.2.1 Location Service ..... 17
1.4.2.2 Sms Service ..... 18
1.4.2.3 Telephony Service ..... 18
1.4.2.4 Sensor Service ..... 19
Capitolo 2: Android GUI Testing ..... 20
2.1 Testing di applicazioni Android ..... 20
2.1.1 Android Testing Framework ..... 21
2.2 Strumenti per il testing Android ..... 22
2.3 Testing Gui Based ..... 23
2.4 Testing Gui Based di applicazioni android ..... 25
Capitolo 3: Android Ripper e Architettura parallela ..... 26
3.1 Android Ripper ..... 26
3.1.1 Il processo di ripping ..... 27
3.1.2 L'output del processo di ripping ..... 28
3.1.3 La tecnica di ripping ..... 29
3.2 Architettura del Ripper ..... 32
3.3 Altri lavori per il testing di applicazioni Android ..... 36
3.3.1 Dynodroid ..... 36
3.3.2 Swiftend ..... 36
3.4 Architettura parallela del Ripper ..... 37
3.4.1 Logica parallela ..... 37
3.4.2 Versione parallela sistematica ..... 38
3.4.3 Versione parallela randomica ..... 39
3.4.4 Regioni critiche e gestione della concorrenza ..... 40
3.5 Ripper su server remoto ..... 41

Capitolo 4: Android Ripper Wep Application ..... 42
4.1 Web App ..... 42
4.2 Diagramma dei casi d'uso ..... 44
4.3 Architettura a livelli dell'intero sistema ..... 45
4.4 Dettagli della Web Application ..... 47
4.4.1 HttpServelt Scanner ..... 47
4.4.2 HttpServelt CopyAvd ..... 48
4.4.3 HttpServelt Upload ..... 50
4.4.4 HttpServelt AppInstall ..... 51
4.4.5 HttpServelt WebMain ..... 52
4.4.6 HttpServelt Execution ..... 53
4.4.7 HttpServelt ParzialResult ..... 54
4.4.8 HttpServelt Stop ..... 55
4.4.9 HttpServelt End ..... 56
4.4.10 HttpServelt Download ..... 57

Capitolo 5: Sperimentazione ..... 58
5.1 Obiettivi ..... 58
5.2 Veriabili e misure ..... 59
5.2.1 Indepent Variable ..... 60
5.2.2 Depent Variable ..... 61
5.2.3 Control Variable ..... 62
5.3 Configurazione sperimentale ..... 63
5.4 Dati e discussione sperimentale. ..... 65
5.4.1 Coperture ..... 65
5.4.2 Tempo di esecuzione ..... 71
5.4.2.1 Numero totale di emulatori ..... 71
5.4.2.2 Tempo al variare delle macchine ..... 74
5.4.2.3 Tempo al variare degli emulatori ..... 82
5.5 Threats to validity ..... 92

Conclusioni ..... 93
Bibliografia ..... 94

# Introduzione

Negli ultimi dieci anni abbiamo assistito ad una nuova rivoluzione tecnologica, quella dei dispositivi mobili. Tutto quello che prima era accessibile solo attraverso un pc, ora può essere raggiunto attraverso tablet o smartphone con diversi sistemi operativi, sui quali sono in esecuzione molteplici applicazioni dotate di una GUI (Graphic User Interface). Per far si che una applicazione mobile sia performante, è necessario sottoporla ad attività di testing così come avviene in generale in qualsiasi contesto di sviluppo software. L'attività di testing possiamo dividerla in testing manuale e testing automatico. Il testing manuale è l'attività di testing più popolare per le interfacce grafiche di queste applicazioni ma risulta lento, noioso e tendente all'errore. Invece, le tecniche per la generazione automatica dei casi di test possono ridurre drasticamente i costi e i tempi legati alla fase di test design.

Prima di procedere, cerchiamo di chiarire meglio cosa intendiamo con la frase "generazione di casi di test" presente nel titolo. In effetti la nostra base di partenza è il Gui Ripper , un sistema di testing automatico di GUI il quale 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.

Quindi nel nostro contesto quando parliamo di generazione automatica di test,

intendiamo la tripla (osservazione-generazione-eplorazione).
L'obiettivo di questo lavoro di tesi è quello di verificare l'ipotesi di aumento dell'efficenza del Gui Ripper, rendendolo parallelo e quindi riducendo i tempi di esecuzione distribuendo il carico di lavoro di testing su diverse macchine.

Questo sistema parallelo è stato reso accessibile anche da remoto attraverso una interfaccia grafica html. L'utente tramite il proprio browser può decidere quante macchine utilzzare, la tecnica (sistematica o randomica), installare la propria applicazione, e avviare il processo per poi ottenere i risultati in termini di copertura.

Nel primo capitolo faremo un introduzione del sistema operativo Android, la sua architettura e la struttura delle sua applicazioni. Nel secondo capitolo parleremo del GUI testing, in particolare del Testing Adnroid, e degli strumenti messi a disposizione dal Framework Android per tale attività
Nel terzo parleremo del GUI Ripper e della sua versione parallela realizzata.
Il quarto capitolo descriverà la nostra Gui Ripper Web Application e quindi del funzionamento del Gui Ripper remoto nelle due versioni, quella Sistematica e 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. Il quinto capitolo fornirà i risultati provenienti dalla sperimentazione effettuata su applicazioni quali Tomdroid, TippyTipper e TicTacToe secondo varie configurazioni.

# Capitolo 1: Ambiente Android

In questo capitolo illustriamo il contesto nel quale Android nasce ed intende affermarsi come ambiente per la realizzazione di applicazioni mobili. Successivamente faremo una panoramica sul software testing, in particolare sul testing Android e sul Gui Testing.

## 1.1 Perchè Android

La rivoluzione dei dispositivi mobili 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 un'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), un'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 Che cos'è Android

Con il termine Android non facciamo riferimento solo ad un Sistemo Operativo per piattaforme mobile, ma ad un complesso framework software composto da strumenti, librerie e tutto ciò di cui hanno bisogno operatori, vendor dei dispositivi e sviluppatori per raggiungere i propri obiettivi. Android, a differenza di altri ambienti, è open. Con questo termine si vogliono intendere diverse cose.

È open in quanto il suo codice è completamente open source quindi è consultabile, migliorabile, ma soprattutto non è necessario pagare nessuna royalty per utilizzarlo sui dispositivi.

È open in quanto sfrutta diverse tecnologie open-source; infatti è basato sul kernel Linux nella versione 2.6.

È open in quanto le librerie e le API utilizzate per la sua realizzazione sono le stesse che possono essere sfruttate per la creazione di nostre applicazioni.

Possiamo anche sostenere che il concetto di "open" è legato anche alla possibilità di utilizzare diversi linguaggi di programmazione, anche se il più usato e supportato è Java.

## 1.3 Architettura di Android e la Dalvik Machine

Abbiamo detto che il linguaggio più utilizzato per lo sviluppo di applicazioni Android è Java, ma sappiamo che un programma Java per essere eseguito necessita di una Virtual Machine (JVM) in grado di interpretare il bytecode (file di estensione .jar) ed eseguirlo su ogni macchina. In realtà la Virtual Machine utilizzata per Android è la Davlik Virtual Machine (DVM) in grado di eseguire codice contenuto all'interno di un file di estensione .dex, ottenuti a partire dal bytecode Java. Perché questa scelta?

Il primo motivo è il risparmio di spazio in quanto un file .dex ha dimensioni nettamente inferiori rispetto ad un .jar.

Il secondo motivo è che la DVM è ottimizzata per macchine con risorse ridotte e quindi risulta ad-hoc per i dispositivi mobili.

Una caratteristica molto importante della DVM è quella di consentire un'efficace esecuzione di più processi contemporaneamente; infatti ciascuna applicazione è in esecuzione nel proprio processo Linux e questo comporta dei vantaggi dal punto di vista delle performance, ma allo stesso tempo porta delle implicazioni dal punto di vista della sicurezza.


Figura 1.1 Kernel Linux

Di seguito possiamo osservare i livelli dell'architettura Android. Tali livelli offrono servizi per il livello superiore offrendo un più alto grado di astrazione. Senza andare nel dettaglio, possiamo affermare che il core vero e proprio di Android è costituito dal livello sopra al Kernel Linux, cioè le native Libraries realizzate in C e C++.

Queste Librerire vengono poi utilizzate da un insieme di componenti di più alto livello che costituiscono l'Application Framwork (AF). Si tratta di un insieme di API e componenti per l'esecuzione di funzionalità ben precise e di fondamentale importanza per le applicazioni. Vediamone alcune:

L'activity manager, il content Provider, il Telephony Manager, il Location Manager, il Notigication Manager ed altre che illustreremo nel detttaglio più avanti

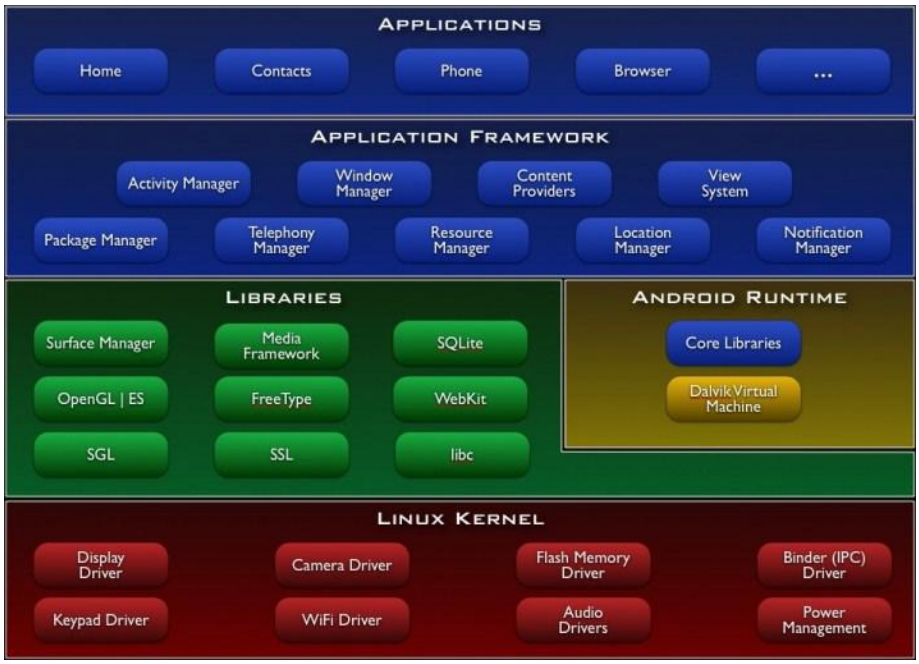



Figura 1.2 Architettura di Android

### 1.3.1 Core Libraries e Libraries

Le Core Libraries insieme alla Davlik Virtual Machine appartengono logicamente all'Andrtoid Runtime cioè una parte dello stack Android dedicata all'esecuzione delle applicazioni.

Le Core Libraries di fatto mettono a disposizione la maggior parte delle funzionalità disponibili nel linguaggio di programmazione Java: strutture dati, utility, librerie per l'accesso ai file, librerie di rete e così via.

Prima di tutto la **Bionic libc**, più piccola e veloce rispetto alla libreria di GNU in linguaggio C perché ottimizzata per dispositivi mobili; poi le **librerie di Utility** come Webkit, Media Framework e SQLite, quelle di **astrazione Hardware** come Graphics, Camera, Bluetooth, GPS, Radio, Wi-Fi etc. ed infine i **server nativi**, come ad esempio Surface Manager e Audio Manager.

Vediamolo in dettaglio

**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.

### 1.3.2 Application Framework

All'Application Manager appartengono 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. Vediamo quali sono:

- Media Framework → librerie per la gestione dei vai formati multimediali;
- FreeType → libreria per la gestione dei font;
- SQLite → libreria per lo storage dei dati (un database embedded);
- WebKit → un browser kit che può essere incluso nelle applicazioni.

Queste librerie sono utilizzabili dagli sviluppatori attraverso l'Application Framework, il quale mette a disposizione degli sviluppatori la possibilità di utilizzare in modo immediato componenti riutilizzabili per lo sviluppo delle proprie applicazioni. Tali componenti, inoltre, sono facilmente sostituibili con implementazioni personalizzate. Tra i più importanti ricordiamo:

- View System → permette di costruire un'applicazione includendo liste, griglie, caselle di testo, bottoni, un browser web embedded e così via;
- Content Provider → permette alle applicazioni di scambiarsi i dati tra loro, condividendoli o accedendo a dati condivisi;
- Resource Manager → permette l'accesso a quelle risorse che non sono codice (strighe, immagini, xml, …);
- Notification Manager → permette alle applicazioni di inviare notifiche nella barra di stato;
- Activity Manager → gestisce il ciclo di vita delle applicazioni e provvede alla visualizzazione delle Activity;
- Location Manager → permette alle applicazioni di ottenere dati sul posizionamento geografico del dispositivo;
- Package Manager → permette la gestione delle applicazioni sul dispositivo.

### 1.3.3 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 [3].

## 1.4 Architettura delle applicazioni Android

Le applicazioni Android sono sviluppate in Java mediante l'utilizzo dell'Android Software Development Kit (SDK) oppure in C/C++ utilizzando il Native Development Kit (NDK). Le applicazioni scritte con l'SDK sono quelle eseguite dalla macchina virtuale Dalvik e consistono in uno o più dei seguenti componenti:

- Activities → un'applicazione che ha un'interfaccia grafica; è quella che viene avviata dall'utente attraverso il lanciatore delle applicazioni (launcher);
- Services → un'applicazione che richiede di rimanere in esecuzione in background, non dotata di interfaccia grafica: un esempio può essere un tool per l'aggiornamento del software o un analizzatore di rete;
- Content Providers → si può pensare ai Content Providers come ad un livello di accesso ad un database; il lavoro di un Content Provider è di gestire l'accesso ai dati da parte delle applicazioni (ad esempio, l'accesso alla lista dei contatti memorizzati sul dispositivo);
- Broadcast Receivers → un'applicazione Android che risponde ad un particolare evento generato dal sistema stesso o da altre applicazioni (ad esempio, l'evento "batteria scarica", la ricezione di un messaggio, …);
- Mentre l'attivazione di un Broadcast Receiver è legata ad un evento, quella delle altre componenti è legata agli Intent. Un Intent è la descrizione astratta di un'operazione da eseguire. Distinguiamo due casi:

- Intent esplicito → permette il passaggio da un'Activity ad un'altra specificando esplicitamente la classe Java dell'Activity che si vuole mandare in esecuzione;
- Intent implicito → non viene specificata l'Activity da mandare in esecuzione, ma vengono forniti i dati da elaborare e/o una descrizione dell'azione da svolgere: sarà il sistema operativo ad inoltrare la richiesta.

### 1.4.1 Activity

Come già anticipato, un'Activity è quel componente di un'applicazione Android che mette a disposizione una schermata con la quale l'utente può interagire per effettuare un'operazione.

Tipicamente un'applicazione Android consta di più Activity legate in qualche modo l'una all'altra; tra di esse, una sarà la prima mostrata all'utente al lancio dell'applicazione e prende il nome di Activity principale. Ogni Activity può avviarne un'altra in base allo scatenarsi di eventi generati dall'utente, dal sistema operativo o dall'hardware. Poiché sullo schermo può esserci una sola Activity alla volta, quando ne viene avviata una nuova, la precedente viene messa in pausa ed il sistema operativo la conserva in uno stack (back stack), mentre la nuova viene portata in primo piano; quando le operazioni sull'Activity corrente sono concluse o viene premuto il pulsante BACK sul dispositivo, quella attualmente in primo piano viene rimossa dallo stack (e distrutta) e la precedente viene riportata in primo piano. L'ActivityManager si occupa della gestione dello stack e della gestione del ciclo di vita delle activity. L'arresto di un'Activity a causa dell'avvio di una nuova viene notificato all'Activity stessa mediante i metodi di callback del suo ciclo di vita (Figura 1.3) che permettono di rispondere in modo adeguato al presentarsi di una transizione di stato dell'Activity (ad esempio, col salvataggio dello stato su memoria di massa).

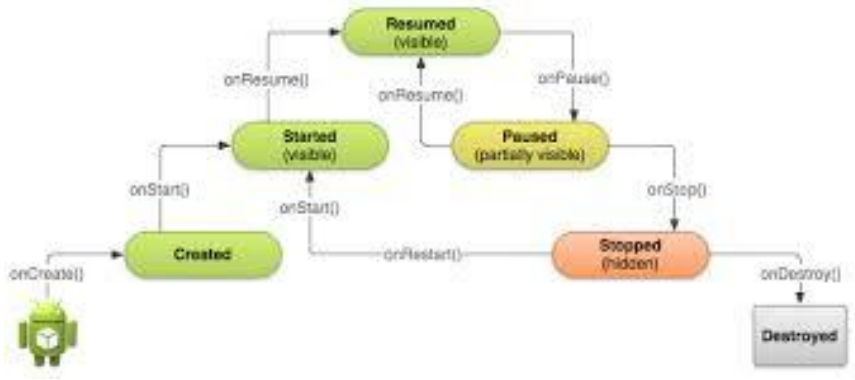



Figura 1.3 ciclo di vita di un Activity

Vediamo nello specifico quali sono gli stati in cui può trovarsi un'Activity:

- Starting → l'Activity è in avvio;
- Running → l'Activity è in primo piano sullo schermo ed ha il focus;
- Paused → un'altra Activity è un primo piano ed ha il focus, ma questa è ancora visibile; un'Activity in questo stato è ancora "viva" (l'oggetto Activity è in memoria insieme al suo stato ed è assegnata al Windows Manager), ma potrebbe essere "uccisa" dal sistema operativo in caso di situazioni di bassa disponibilità di memoria;
- Stopped → l'Activity è in background ma è ancora "viva" (l'oggetto Activity è in memoria insieme al suo stato, ma non è assegnata al Windows Manager); il sistema potrebbe "ucciderla" se ha bisogno di memoria;
- Destroyed → l'Activity è stata terminata;

Ad ogni transizione di stato è associata l'esecuzione di una o più callback sull'oggetto Activity.

- onCreate() → risponde alla creazione della Activity;
- onStart() → l'Activity sta per diventare visibile;
- onResume() → l'Activity è diventata visibile (è nello stato Running);

- onPause() → un'altra Activity sta prendendo il focus (l'Activity corrente sta per entrare nello stato Paused);
- onStop() → l'Activity non è più in primo piano (è nello stato Stopped);
- onDestroy() → l'Activity sta per essere distrutta.

### 1.4.2 I Service

Un servizio (service) è un processo che gira in background che non prevede l'interazione diretta con l'utente; nasce per svolgere attività in modo "invisibile", ovvero senza interfaccia utente. La gestione del servizio e l'interazione con l'utente potrà avvenire tramite un'Activity o un altro servizio.

Un tipico esempio di servizio è il player multimediale di Android, il quale continua la riproduzione anche quando la sua interfaccia grafica (l'Activity che lo gestisce) passa in secondo piano. Quindi, il servizio implementerà il codice necessario alla riproduzione del file multimediale, mentre l'Activity ne permetterà il controllo.

Vediamo di seguito alcuni esempi rilevanti ai fini della trattazione di servizi messi a disposizione dal sistema operativo Android.

#### 1.4.2.1 Location Service

Android mette a disposizione dello sviluppatore un framework per il rilevamento della posizionamento, contenuto nel package android.location. Il componente centrale di questo framework è il servizio di sistema LocationManager il quale mette a disposizione delle API per determinare la posizione ed il bearing del dispositivo (se supportato). Questo componente non può essere istanziato dall'applicazione utente, ma essa può richiederne al sistema operativo un'istanza tramite la chiamata

Context.getSystemService(Context.LOCATION_SERVICE)

Una volta ottenuta questa istanza, si può:

- richiedere la lista dei LocationProvider, ovvero dei fornitori delle informazioni di posizionamento (GPS, rete cellulare, rete wifi, …);

- registrarsi per la ricezione di aggiornamenti da parte dei LocationProvider;
- registrare un Intent da sollevare se il dispositivo si trova presso una certa posizione geografica.

Per l'utilizzo di queste funzionalità, l'applicazione deve ottenere uno dei seguenti permessi: ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION, ACCESS_MOCK_LOCATION

#### 1.4.2.2 Sms Service

Un dispositivo connesso alla rete cellulare ha la possibilità di inviare e ricevere messaggi di testo SMS. Il servizio SMS di Android si occupa della gestione, dell'invio e della recezione di questi messaggi (se il dispositivo prevede tale possibilità).

L'invio dei messaggi avviene tramite i metodi della classe SmsManager, una cui istanza deve essere ottenuta mediante una chiamata al metodo statico SmsManager.getDefault().

La ricezione dei messaggi mediante un BroadcastReceiver che intercetti l'intent android.provider.telephony.SMS_RECEIVED.

Per la gestione degli SMS, l'applicazione deve ottenere uno dei seguenti permessi: RECEIVE_SMS, READ_SMS, SEND_SMS, WRITE_SMS.

#### 1.4.2.3 Telephony Service

Il servizio permette l'accesso alle informazioni sui servizi di telefonia del device (se previsti dal dispositivo), tramite la classe TelephonyManager. Le applicazioni possono utilizzare i metodi di questa classe per determinare lo stato dei servizi di telefonia, sottoscriversi per aggiornamenti relativi al servizio e accedere alle sue

funzionalità. Un'istanza della classe può essere ottenuta tramite la chiamata

Context.getSystemService(Context.TELEPHONY_SERVICE)

L'accesso ai servizi di telefonia è anch'esso soggetto all'acquisizione dei necessari permessi.

#### 1.4.2.4 Sensor Service

Il Sensor Service permette all'applicazione l'accesso ai dati provenienti dai sensori (se presenti sul dispositivo); ciò avviene tramite la classe SensorManager, una cui istanza può essere ottenuta tramite la chiamata

Context.getSystemService(Context.SENSOR_SERVICE)

A differenza degli altri, l'utilizzo di questo servizio non è soggetto all'acquisizione di alcun permesso.

# Capitolo 2: Android GUI Testing

In questo capitolo descriveremo le principali tecniche di GUI Testing per un'applicazione Android. Successivamente illustreremo altri lavori per la misura dell'efficienza. Prima di iniziare diamo solo due definizioni che ci aiuteranno nella lettura del capitolo.

- *Efficienza* di una attività di testing possiamo definirla come:

  (numero di casi di test rilevanti difetto) / (numero di casi di test eseguiti)

- *Efficacia* di una attività di testing possiamo definirla come:

  (numero di difetti scoperti)/(numero di difetti esistenti)

Bisogna però sottolineare che non è possibile misurare l'efficacia dei casi di test in generale in quanto non sappiamo il numero di difetti nel software. Ma possiamo misurare l'efficacia relativa di una test suite rispetto ad un'altra in base al rapporto tra i difetti scoperti.

## 2.1 Testing di applicazioni Android

Quando si sviluppano applicazioni per dispositivi mobili, in particolare dotati di sistema operativo Android, il processo realizzativo non è diverso da quello di una normale applicazione; tuttavia queste applicazioni posseggono una serie di caratteristiche e requisiti peculiari che vanno tenuti in conto in fase di sviluppo e di test.

Questa classe di applicazioni è destinata infatti ad essere eseguita su dispositivi con risorse hardware limitate e display di dimensioni ridotte, dotate di sensori e dispositivi

di comunicazione eterogenei, alimentati a batteria.Tutto questo fa si che requisiti non funzionali come le performance, il consumo di batteria, la connettività, la sicurezza, la user experience e la portabilità su diversi dispositivi diventino fattori importanti di cui tener conto durante lo sviluppo ed il testing dell'applicazione, fattori che ne decretano il successo o l'insuccesso, più delle funzionalità stesse. [7]

Poiché l'attività di testing nel processo di sviluppo software è di sostanziale importanza, Android ci mette a disposizione nel suo SDK tools di supporto che ci permettono di effettuare in modo semplice ed automatizzato il testing a vari livelli: testing di unità, testing funzionale, testing di regressione e testing della GUI (*Graphical User Inteface*).

### 2.1.1 Android Testing Framework

Prima di parlare degli strumenti che l'SDK di Android ci mette a disposizione per realizzare l'attività di testing, facciamo una piccola parentesi sull'emulatore Android.

Dato che è di fatto impossibile avere accesso a tutti i dispositivi Android presenti sul mercato; l'emulatore può essere configurato per simulare la maggior parte delle situazioni. Ovviamente l'emulatore ha i suoi limiti, che possono essere superati da test su dispositivi reali.

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.

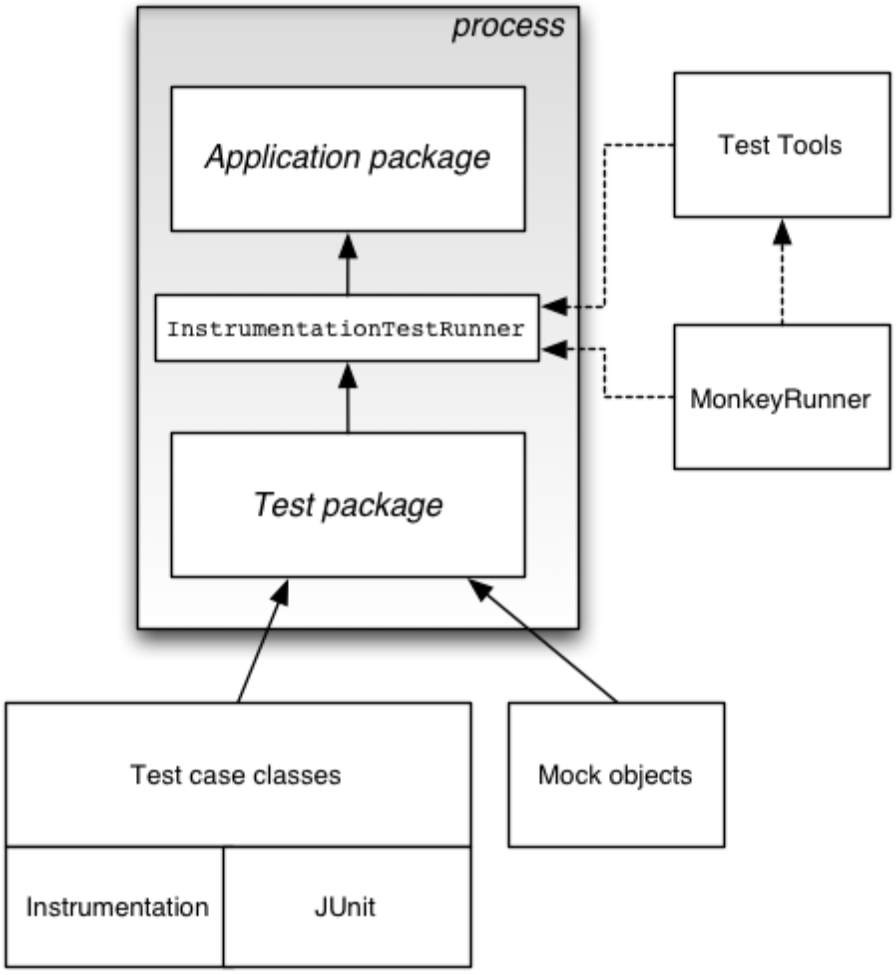



Figura 1.3: Testing Framework di Android

## 2.2 Strumenti per il testing Android

In questo paragrafo illustriamo brevemente alcuni strumenti utili per il testing Android forniti con l'SDK.

- **Robotium**: è divenuto uno standard de-facto. È un'estensione dell'Android Test Framework creato allo scopo di facilitare la scrittura di casi di test black-box automatizzati per le applicazioni Android. Robotium permette di definire e comprendere in modo semplice i casi di test dedicati alle Activity Android. 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;

- **Monkey**: è un programma che viene eseguito sull'emulatore o il device con lo scopo di generare delle sequenze pseudo-casuali di eventi utente e di eventi di sistema. Uno degli utilizzi principali di Monkey è per effettuare lo stress-test delle applicazioni.[8]
- **Monkey Runner**: a differenza di Monkey, è un API che consente la scrittura di programmi in grado di controllare un dispositivo Android dall'esterno;
- **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.

## 2.3 Testing GUI Based

La GUI (*Graphical User Interface*) è il mezzo che l'utente ha per interagire con il sistema software; essa risponde agli eventi generati dall'utente (come ad esempio un click) eseguendo il codice ad essi collegato. Uno dei metodi comunemente utilizzati per rilevare la presenza di difetti in un software è quello di esercitare la sua GUI. A differenza di altri approcci al testing in cui le test-suite sono composte da casi di test che invocano metodi del sistema software e catturano il valore ritornato, l'approccio gui-based prevede componenti in grado di:

- rilevare e riconoscere i componenti della GUI;
- esercitare gli eventi sulla GUI (cliccare con il mouse, …);
- fornire degli input ai componenti della GUI (riempire i campi di testo, …);
- controllare le rappresentazioni della GUI per verificare se sono consistenti con quelle attese. Ciò rende il testing GUI-Based particolarmente difficoltoso e la sua implementazione strettamente dipendente dalla tecnologia utilizzata. D'altro canto però questa tecnica di testing è facilmente eseguibile ed automatizzabile.

Ci sono diversi approcci al testing GUI-Based:

- manuale → basato sulla conoscenza del dominio e dell'applicazione da parte del tester;
- capture-and-replay → basato sulla cattura e sulla riesecuzione delle sessioni utente;
- testing model-based → basato sull'esecuzione di sessioni utente ricavate dal modello della GUI;

Per quanto riguarda quest'ultimo approccio, il modello descrittivo più comunemente utilizzato è quello di macchina a stati. Possiamo definire la GUI come il front-end grafico, gerarchico di un sistema software; essa è composta da un insieme di oggetti grafici chiamati widget, ognuno dei quali ha un insieme di proprietà le quali possono assumere valori discreti a tempo di esecuzione. L'insieme dei widget, le loro proprietà e l'insieme dei propri valori descrivono lo stato della GUI. Gli stati dunque saranno come degli "screenshot" della GUI, mentre le transizioni saranno eventi che ne provocano un cambiamento di stato.
Il problema sta nel come ottenere un diagramma degli stati che descriva adeguatamente la GUI e possa essere utilizzato per generare i casi di test. Ci sono due possibilità:

1. la macchina a stati può essere prodotta in fase di sviluppo dalle specifiche del sistema o a partire dai requisiti, ma può essere poco aderente all'effettiva implementazione dell'interfaccia;
2. la macchina a stati può essere ricostruita per reverse engineering dalla user interface implementata, inferendo il modello a partire dall'esecuzione dell'applicazione e rifinendolo manualmente (se necessario).

Indipendentemente dall'approccio utilizzato, un altro problema sta nel definire gli oracoli per controllare la correttezza del programma rispetto ai casi di test. Guardando alla GUI potrebbe infatti essere difficile rilevare i difetti dell'applicazione sotto

test. Tipicamente una tecnica utilizzata è il crash testing, il quale permette la scoperta degli errori a tempo di esecuzione causati da eccezioni non gestite. Un'altra possibilità consiste nel confrontare lo stato rilevato con quello atteso, ma questo ne presuppone la conoscenza.

## 2.4 Testing GUI Based di applicazioni Android

Gli approcci al testing GUI-Based possono essere divisi in tre categorie:

- tecniche model-based, nelle quali esiste un modello, una descrizione formale dell'applicazione sotto test ed in particolare della GUI; tale modello avrà un livello di dettagli sufficiente alla generazione automatica dei casi di test e dei rispettivi oracoli per la verifica del software;
- tecniche di random testing, nelle quali, in assenza di un modello, l'applicazione viene esercitata in maniera casuale, alla ricerca di eventuali eccezioni non gestite; questa tecnica viene implementata da Monkey, distribuito con l'SDK di Android;
- tecniche crawler-based, anch'esse non basate su un modello preesistente, ma l'esplorazione invece di avvenire in maniera casuale viene effettuata in maniera metodica, seguendo una precisa strategia di navigazione; tale tecnica permette non solo di rilevare eccezioni non gestite (crash testing), ma permette anche di effettuare il reverse engineering del modello il quale potrà essere utilizzato per ulteriori elaborazioni; un approccio simile è utilizzato nell'Android Ripper, lo strumento oggetto di questo lavoro di tesi.

# Capitolo 3: Android Ripper ed architettura parallela

In questo capitolo introduciamo uno strumento automatico per il testing Android, il GUI Ripper. Procederemo poi alla descrizione della sua versione parallela (parte di questo lavoro di tesi), concentrandoci maggiormente sugli aspetti architetturali che implementativi. Infine una breve descrizione di altri due strumenti per il testing Android come Dynodroid e Swiftend

## 3.1 Android Ripper

Android Ripper è uno strumento di automazione del testing configurabile che implementa sia tecniche crawler-based che di random testing; è basato su un approccio black-box e permette l'esplorazione automatica della struttura della GUI attraverso la generazione di input ed eventi non solo a partire dalla sua analisi, ma anche considerando i possibili eventi scatenabili sul dispositivo (rotazione, pressione del tasto back, pressione del tasto menù, …). Tutti gli aspetti dell'esplorazione (strategia, criterio di terminazione, possibili eventi, possibili input e così via) possono essere configurati per emulare determinati comportamenti dell'utente. 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 informazioni per la generazione del modello e di nuovi casi di test (testing mutazionale), sia per la generazione di casi di test jUnit-Android per effettuare il testing di regressione.

### 3.1.1 Il processo di ripping

Osserviamo la figura 3.1. Essa rappresenta l'algoritmo adoperato dal Ripper per l'esplorazione dell'applicazione sotto test. Vediamo come evolve:

- per prima cosa viene analizzato lo stato corrente della GUI (Activity State) descrivendo l'Activity in primo piano in termini dei suoi parametri e dei widget in essa contenuti; i widget a loro volta saranno descritti in termini dei propri attributi;
- una volta descritto lo stato, è possibile ottenere l'elenco delle azioni scatenabili sull'Activity, filtrati in base a criteri forniti da tester o infieriti dal Ripper, in grado di scatenare il passaggio ad una diversa interfaccia (eventi); per ogni elemento dell'elenco generato viene generato 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 generato 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.

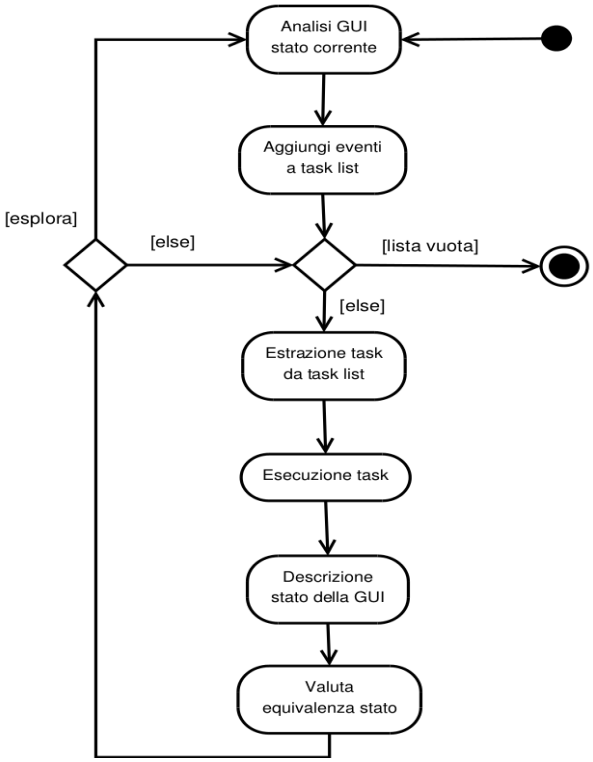



Figura 3.1: Activity Diagram dell' Android Ripper

### 3.1.2 L'output del processo di ripping

L'output del processo di Ripping è costituito da un file contenente l'intera sessione (Session), ovvero un rapporto dell'esecuzione dei singoli task (trace), ed un file contenente tutti gli stati della GUI incontrati durante l'esplorazione.

Il primo file contiene la rappresentazione dell'applicazione sotto test generata dall'esplorazione; nell'attuale implementazione dell' Android Ripper si ottiene un albero detto GUI Tree.

Ogni stato della GUI viene rappresentato nel GUI Tree come nodo di un albero; gli eventi che provocano transizioni di stato sono rappresentati come archi. Ogni percorso che parte dall'interfaccia iniziale e segue le transizioni eventualmente, ma non necessariamente, fino ad un nodo foglia, rappresenta una traccia di esecuzione.

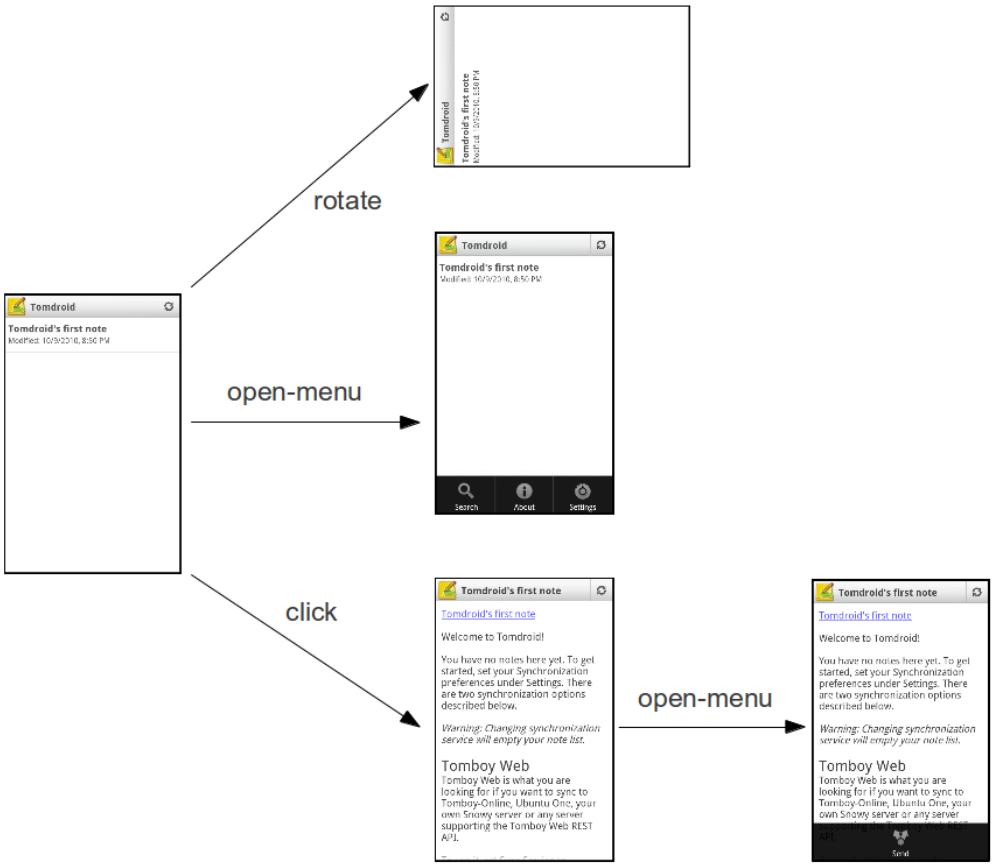



Figura 3.2: Estratto di GUI Tree di Tomdroid

### 3.1.3 Tecnica di ripping

Vediamo come funziona nel dettaglio l'algoritmo di ripping sul quale si basa il nostro strumento di testing automatico. Possiamo 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 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* ← *robot*.getCurrentActivity()
*activityDescription* ← *extractor*.describe(*baseActivity*)
*activityState* ← *abstractor*.abstract(*activityDescription*)
*strategy*.storeVisitedState(*activityState*)
*plan* ← *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'algoritmo 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 nell'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.

**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
```

## 3.2 Architettura del Ripper

Il Ripper è implementato come un progetto Android Test, che fa uso delle librerie del Framework jUnit, ed utilizza per interagire con l'applicazione l'Android Instrumentation ed il Framework Robotium. La classe Engine estende infatti ActivityInstrumentationTestCase2 ed inoltre i componenti dedicati ad astrazione dello stato ed esecuzione dei task fanno uso di Robotium e dell'Android Instrumentation.

Andiamo a considerare uno ad uno i componenti del Ripper, riassumendo brevemente le loro responsabilità.

L'**Engine** (motore) è il controllore centralizzato che racchiude la business logic del ripper. Esso gestisce il processo di ripping in ogni sua fase, supervisionando il flusso di esecuzione e garantendo lo scambio di messaggi fra le componenti. In particolare ci sono al momento due implementazioni di questo componente:

- GuiTreeEngine, che effettua il processo di Ripping utilizzando come modello quello di macchina a stati descritto precedentemente; (GuiTree), estraendo un task alla volta a seconda della strategia adottata dallo scheduler;
- RandomEngine, che permette di eseguire un testing random.

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 dati e fornisce all'Engine, di volta in volta, il task da eseguire. La scelta di una struttura dati piuttosto che un'altra può influenzare la strategia seguita nell'imporre l'ordine di esecuzione dei task. In particolare sono implementate due strategie: esplorazione in profondità (implementata con uno stack di task) ed esplorazione in ampiezza (implementata con una coda di task).

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 e Robotium; il robot è il componente che svolge il ruolo di driver.

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 l'Activity corrente ed i widget presenti a video.

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.

Lo **Strategy** ha due principali responsabilità:

- effettua il confronto fra lo stato corrente dell'applicazione (e precisamente dell'istanza di interfaccia che la rappresenta) e gli stati già visitati in precedenza. In caso di equivalenza lo stato corrente non necessita di ulteriore esplorazione da parte del ripper;
- prende 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. Tali decisioni includono:

- imporre o meno l'esplorazione dell'ultimo stato raggiunto.o imporre l'interruzione;
- la chiusura della sessione quando intervengono particolari condizioni;
- riconoscere se l'esecuzione di un'azione ha effettivamente comportato una transizione di stato o meno.

Queste decisioni vanno a determinare il flusso di esecuzione che sarà imposto al ripper dal motore.

Il **planner** ha la responsabilità di generare il test plan che definisce le modalità di esplorazione dell'applicazione a partire dallo stato 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 seleziona quei widget che, in base a regole prestabilite o definite dal tester, sono ritenuti in grado di innescare una transizione di stato nell'applicazione. Saranno quindi generati dei task che esercitano tali widget scatenando opportuni eventi; tali task andranno a comporre il piano di esplorazione, che sarà poi gestito dallo scheduler secondo le politiche implementate in quel componente.

Il **persistence manager** provvede a tutte le operazioni da e verso le memorie di massa; esso ha la responsabilità di gestire la memorizzazione della sessione.[7][8]

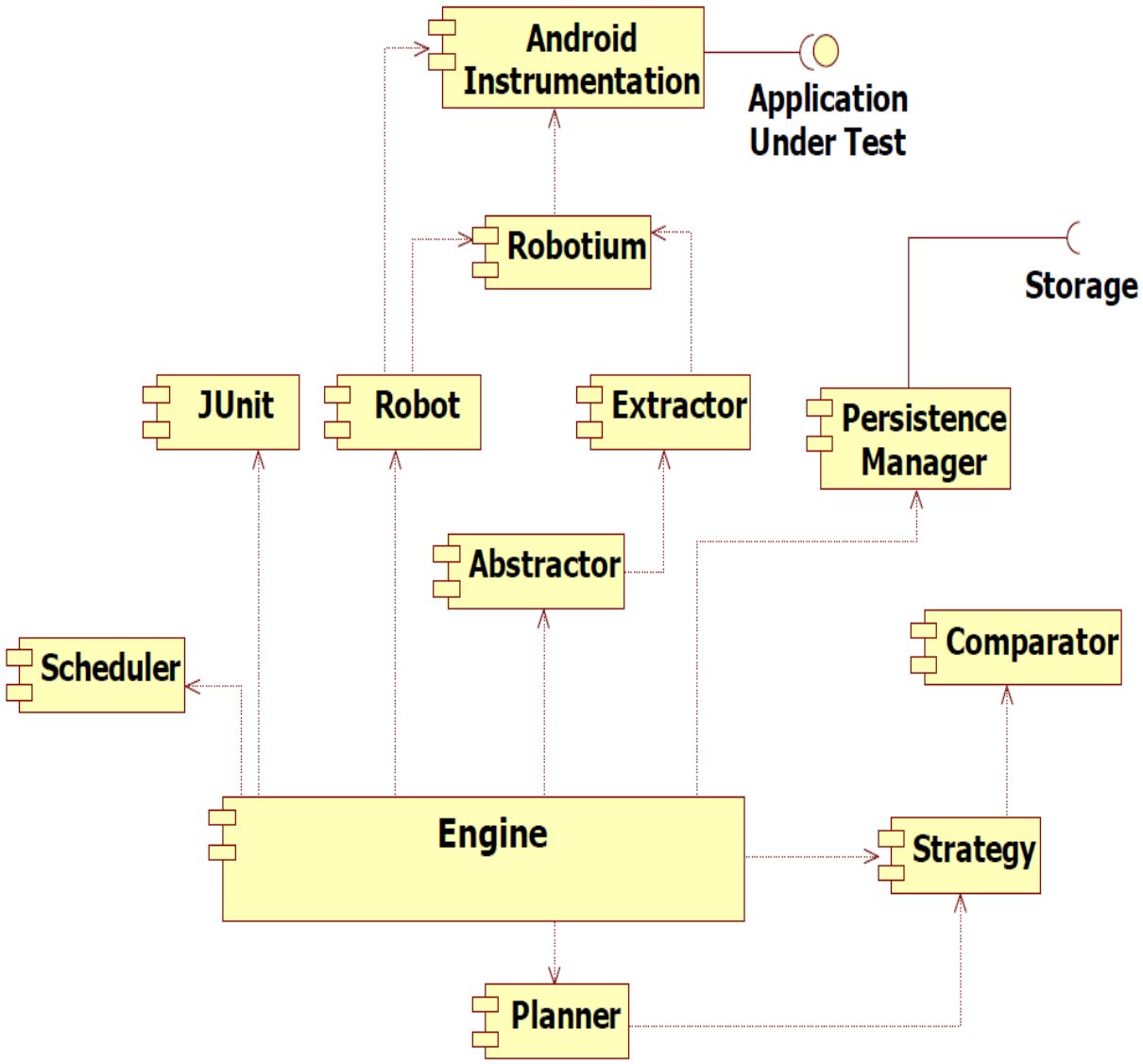



Figura 3.3: Architettura Android Ripper

## 3.3 Altri lavori per il testing di applicazioni Android

In questo paragrafo illustriamo due dei principali lavori, oltre al GUI Ripper, per il testing di applicazioni Android.

### 3.3.1 Dynodroid

Dynodroid[5] è un sistema automatico per la generazione di inputs significativi per applicazioni Android da testare. Il principio fondamentale alla base di Dynodroid è un ciclo "observe-select-execute".

Osservazione di quegli eventi che sono rilevanti per l'applicazione nello stato corrente.

Selezione di uno di questi eventi

Esecuzione degli eventi selezionati per produrre un nuovo stato in cui si ripete questo processo.

Può generare sia inputs di interfaccia utente che inputs di sistema e permette la combinazione tra ingressi umani e ingressi macchina.

Dynodroid è stato confrontato con due strumenti per la generazione degli input: test manuale ed un tool fornito dalla piattaforma Android Noto Come Mokey. E' possibile affermare che Dynodroid può automatizzare in modo significativo le attività di test e generarare sequenze di inputs decisamente più concise di Monkey. (Monkey è un programma in esecuzione sull'emulatore Android o sul dispositivo e consente di generare un flusso pseudo casuale di eventi utenti come click, tocchi o gesti così come anche eventi di sistema, agendo come "stress-test" sull'applicazione in maniera casuale ma ripetibile).

### 3.3.2 Swifthand

E' una tecnica automatica per la generazione di sequenze di input test per applicazioni Android. L'obiettivo è quello di raggiungere una buona copertura del codice in maniera rapida a partire dall'insegnamento e dall'esplorazione della GUI dell'app.

## 3.4 Architettura parallela del Ripper

Questo lavoro di tesi ha come compito quelo di illustrare i risultati ottenuti a seguito della re-ingegnerizzazione dei Gui Ripper di base, per poterlo utilizzare in un contesto parallelo con accesso remoto per utenti che intendone testare la propria applicazione. La re-ingengerizzazione ha comportato la modifica di alcune componenti del ripper di base oltre alla implementazione di altre componenti e servizi che illustriamo in questo paragrafo. I risultati di questo sforzo li presenteremo nel capitolo sulla sperimentazione.

### 3.4.1 Logica parallela

In Figura 3.4 è mostrata una schematizzazione del istema di partenza a singola machina.

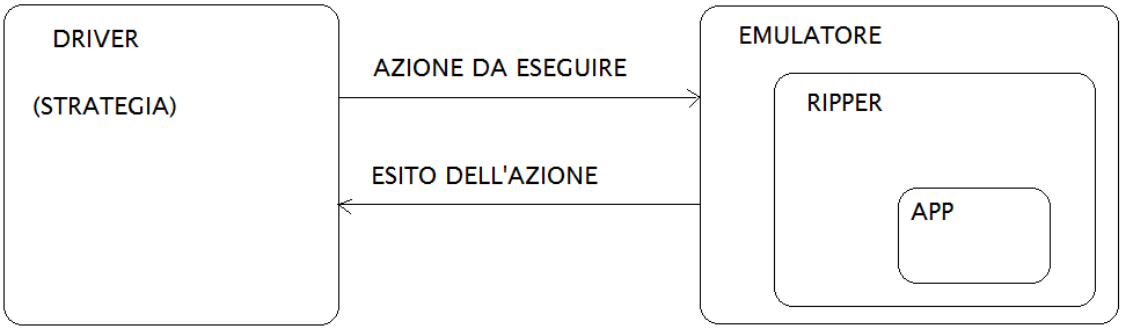



Figura 3.4 Architettura complessiva a singolo emulatore

A partire da questo schema abbiamo progettato la sua versione paralella. Dato che la componente denominata Driver è quella che si occupa di fatto dell'intero processo di ripping (sia in strategia random che sistematica), e che nellla versione base sulla stessa macchiana erano in esecuzione anche gli emularori, abbiamo pensato separare la logica elaborativa, il driver, dalla logica applicativa/operativa. In Figura 3.5 possiamo osservare lo schema del sistema nella versione parallela.

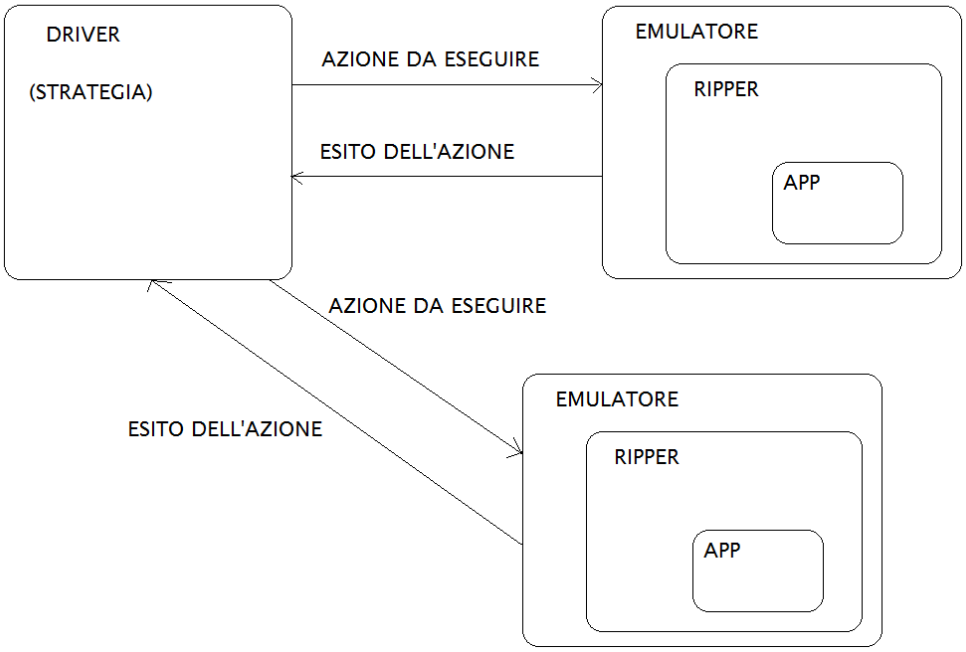



Figura 3.5 Architettura complessiva multiemulatore

### 3.4.2 Versione parallela sistematica

Per realizzare l'esecuzione parallela si è deciso di utilizzare una strategia multithread, implementando una apposita classe chiamata ClientSistThread che estende la classe Thread. Le tre componenti principali del sistema sono le classi SystematicRipper, SystematicDriver e ClientSistThread.

- **SystematicRipper**: rappresenta il main dell'applicazione. Si occupa della preconfigurazione del sistema caricando un file di configurazione (config.properties) definito durante l'interazione utente con la Web App. Invocherà il metodo **startRipping()** che rappresenta di fatto lo start dell'intero processo di ripping. Vengono istanziati a lanciati i vari thread, tanti quanti sono gli emulatori che si vuole funzionino contemporaneamente ai quali verranno passati parametri ed oggetti fondamentali per il processo (Scheduler, Planner, Comparator, ActivityList, DescriptionLoader, TerminationCriterion );
- **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. Il metodo run di qeusta classe non fa altro che invocare 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.

### 3.4.2 Versione parallela randomica

La versione parallela randomica 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.

- **RandomRipper**: rappresenta il main. Così come nella versione sistematica, si occupa di caricare un file di configurazione sempre impostato tramite le interazione utente con la WebApp, e con il metodo **startRipping()** dà il via al processo istanziando tanti thread quanti sono gli emulatori che si vuole funzionino contemporaneamente. Una cosa da notare è che in questo caso data la natura del tutto casuale con cui sono generati i task da eseguire, RandomRipper non istanzia nè Planner nè Scheduler, cosa che veniva fatta nel caso sistematico dove avevamo la necessità di mantenere le stesse istanze di Scheduler e Planner e passarle ai vari thread;
- **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. 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.

### 3.4.3 Regioni critiche e gestione della concorrenza

Data la natura parallela del sistema realizzata con un comportamento multithread, ci siamo trovati difronte a problemi di concorrenza nell'accesso a porzioni di codice che devono ora essere acceduti in mutua esclusione. Si è resa quindi necessaria l'introduzione di regioni critiche mediante l'uso dei semafori.

La prima regione critica si ha in corrispondenza di una fase di bootstrap. Questa fase 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.

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 così 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.

## 3.5 Ripper su server remoto

Il nostro 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.

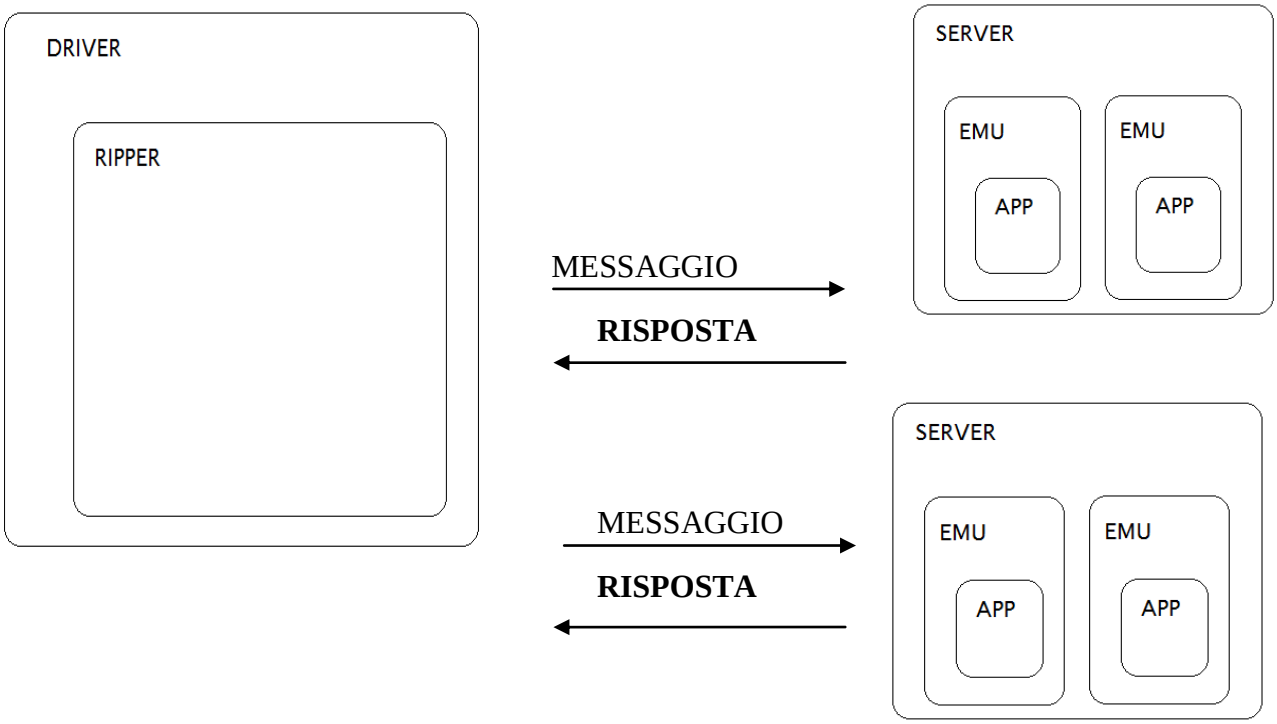



Figura 3.12 Architettura parallela MacchinaDriver MacchinaServer

L'architettura parallela remota funziona sull'attività di scambio di messaggi tra il driver ed i server. I server comandano gli emulatori ed inviano i risultati elaborati al driver.

# Capitolo 4: Android Ripper Web Application

Abbiamo precedentemente descritto il funzionamento del GUI Ripper nella sua versione remota e multitrhead. Per rendere il sistema accessibile ed utilizzabile da chiunque volesse testare la propria applicazione, è stata progettata una Web Application che ci ha consentito di offrire un *front-end* utente per l'interazione con il sistema. In questo capitolo analizziamo nel dettaglio le struttura della web application.

## 4.1 Web Application

Per la realizzazione del nostro sistema di testing remoto, è stata realizzata in prima luogo una versione multithread del GUI Ripper sia nella versione Randomica che nella versione Sistematica. Successivamente abbiamo sviluppato una Web Application che ci ha consentito l'esecuzione del nostro GUI Ripper Multithread su diverse macchine, mantenendo la logica dell'algoritmo su un'unica macchina centrale che di qui in avanti chiameremo Macchina Driver (MD).

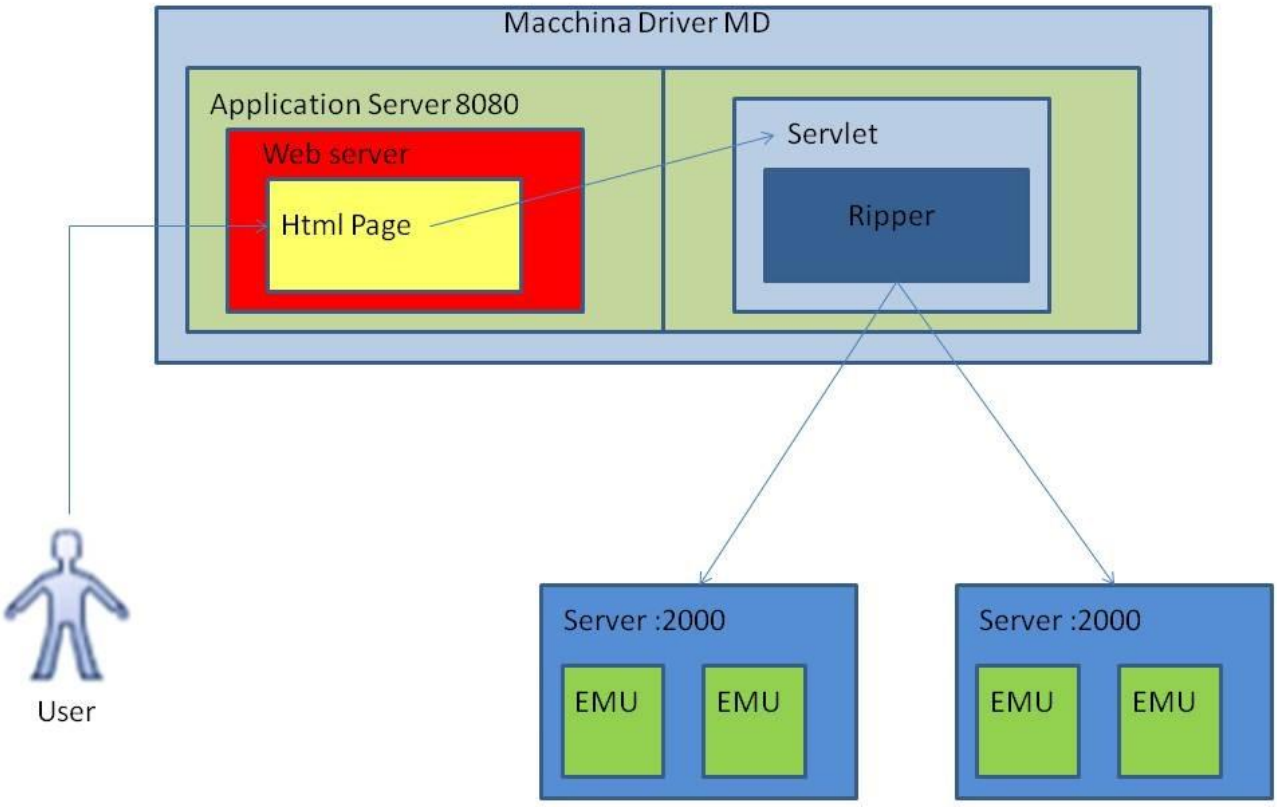



Figura 4.1 Schema generale

Come possiamo vedere in Figura 4.1, sulla MD è in esecuzione un Web Server in ascolto sulla porta 8080. Sulle macchine server (MS) invece, vi è in esecuzione un processo Server in ascolto sulla porta 2000, che fa da proxy per la WebApp. Per la realizzazione della Web Application, la tecnologia utilizzata sono le Servlet in particolare HttpServlet, con supporto Javascript. La scelta di utilizzare le HttpServlet è stata dettata dal contesto di evoluzione del sistema cioè un contesto basato sul paradigma richiesta/risposta. Come vedremo più avanti, le interfacce utente del sistema sono costituite da pagine Html che fanno uso anche di codice Javascript. Ad

ogni azione utente corrisponde una richiesta cioè un messaggio che viene inviato al server. La HttpServlet in ascolto elabora la richiesta, interagendo con il GUI Ripper Multithread, ed invia la risposta all'utente. Nel nostro caso le richieste utente sono incapsulate in messaggi Http.

## 4.2 Schema di interazione

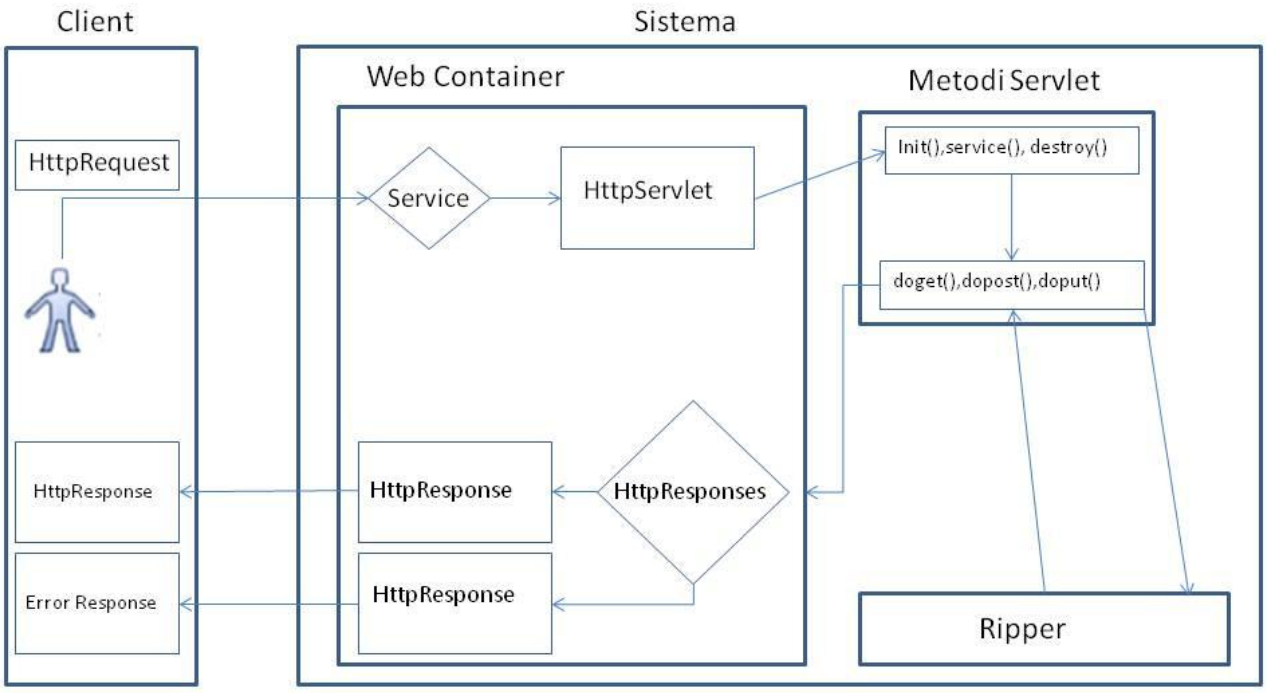



Figura 4.2 Use Case Diagram

Lo schema di interazione in Figura 4.2, ci mostra l'evoluzione della Web App. Per semplicità non sono stati riportati tutti i componenti del sistema in quanto in questa fase si vuole illustrare solo il funzionamento della Web App indipendentemente dall'evoluzione di back-end dell'intero sistema. A partire da questo schema, e facendo riferimento anche al diagramma dei casi d'uso in Figura 4.2, vediamo allora le

interazioni utente:

L'utente accede al servizio tramite l'url della pagina index.html presente sulla MD.

Fa una richiesta di utilizzo del servizio. Se questo non è disponibile, l'utente viene avvertito e potrà riprovare in seguito. Se invece è disponibile, visualizzerà una schermata con la quale potrà scegliere il numero di server da utilizzare ed il metodo di funzionamento (Sistematico o Random). Per ogni server sarà specificato il numero di Emulatori a disposizione. Nel caso venga scelto il modo Random, l'utente potrà inserire il numero di eventi che intende realizzare.

A questo punto può caricare la propria applicazione Android. Il sistema richiede che il formato del file da caricare sia con estensione ".zip".

Al termine dell'upload della propria applicazione, l'utente potrà lanciare l'esecuzione del Ripper.

Durante l'esecuzione l'utente può prelevare i risultati parziali oppure interrompere l'esecuzione. In entrambi i casi avrà la possibilità di ottenere le coperture prodotte fino a quel momento.

## 4.3 Architettura a livelli dell'intero sistema

L'intero sistema può essere visto come un'architettura *multi-tier*. In effetti le componenti software sono suddivise su più livelli logicamente separati ma in qualche modo in comunicazione tra loro.

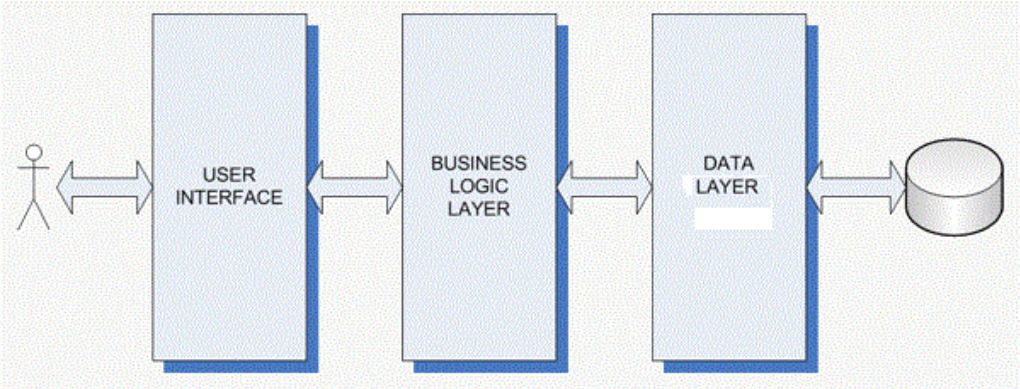



Figura 4.3 3-tier architecture

Logica di presentazione o di interazione (presentation layer).

Con questo livello facciamo riferimento al mezzo utilizzato per rendere fruibile il sistema via web per mezzo di una rete (intranet o internet). Stiamo parlando del web-browser dell'utente (front-end) che fa da GUI per la web-app. Attraverso il web-browesr vengono inviate delle richieste, tramite il protocollo http, al livello 2.

Logica applicativa o di elaborazione (application layer).

Questo livello è costituito dall' application server (back-end). Offre un servizio al livello 1 attraverso le varie HttpServlet che vedremo nel dettaglio più avanti.

Logica dati (data layer).

Nel nostro caso, a questo livello appartiene il GUI Ripper Multithread Remoto, una applicazione che fa da Proxy per quest'ultimo, ed una applicazione deputata alla gestione del trasferimento dei file durante e alla fine di ogni esecuzione del sistema.

## 4.4 Dettagli della Web Application

Illustriamo passo passo come evolve la web application e quali sono le interazioni utente facendo riferimento al diagramma dei casi d'uso visto nel paragrafo precedente. Ovviamente sono state implementate anche delle classi di utilità per le HttpServlet. La Web App è composta dalle seguenti HttpServlet che reagiscono a richieste post http.

### 4.4.1 HttpServlet Scanner

Questa HttpServlet si occupa di verificare se il servizio è disponibile (facendo una scansione tra i processi java in esecuzione), ed in caso affermativo di verificare di quante Macchine Server (MS) dispone il sistema. Tale HttpServlet viene risvegliata accedendo alla seguente pagina index.html.


Figura 4.4 index.html

Quando l'utente clicca su pulsante "invia richiesta", di fatto effettua una richiesta post() http.

La Httpservlet Scanner in ascolto, verifica se il Ripper è in esecuzione controllando la lista dei "*jar"* in esecuzione sulla MD. Se il servizio è disponibile allora ispeziona un file di configurazione nel quale sono stati memorizzati gli indirizzi ip delle MS. A partire da questi ip, tenta una connessione via Socket. Se la funzione *connect()* non restituisce eccezioni, allora l' Httpservlet ritiene che quella macchina server è disponibile. A questo punto la HttpServlet restituisce il risultato:

```
PrinterWriter out=response.getWriter();
response.setcontentType("text/html");
        .
        .
        .
out.println(); //  scrittura del body html sullo Stream verso il Brower del client.
        .
        .
out.close();
```

### 4.4.2 HttpServlet CopyAvd

Questa Httpservlet è di fondamentale importanza in quanto ad ogni nuovo test, è necessario riportare gli emulatori in uno stato noto. Per motivi soprattutto di tempo (inserire una nota che spieghi cosa intendiamo), abbiamo deciso di mantenere degli emulatori preconfigurati (stato noto). Tali emulatori saranno copiati in una directory opportuna ed è su questi emulatori nella nuova directory che sarà eseguita l'installazione e l'esecuzione del test.

Nella Figura 4.5 mostriamo il risultato della HttpServlet Scanner illustrata precedentemente.

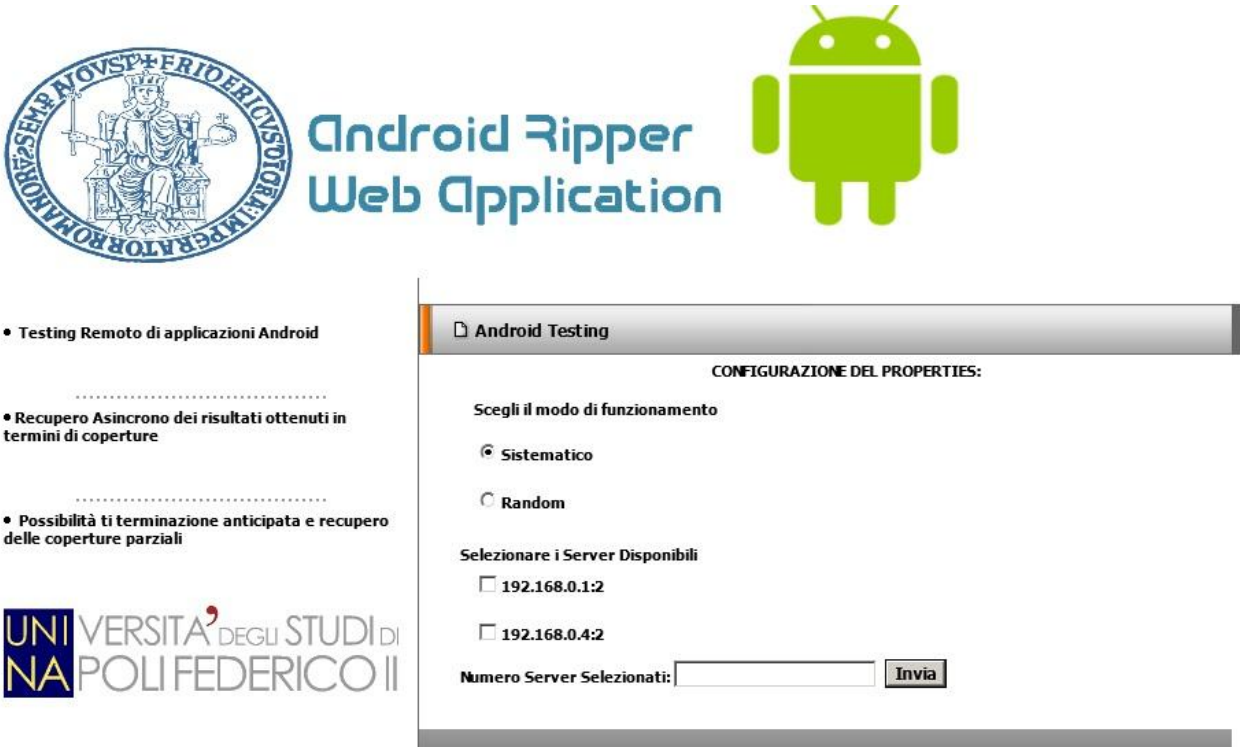



Figura 4.5 pre-CopyAvd

Per poter copiare quindi le avd, l'utente deve selezionare il metodo di funzionamento, nel caso Random stabilire quanti eventi devono essere realizzati, e su quali server. Premendo sul tasto "Invia", viene effettuata una richiesta post http che fa riferimento all'HttpServlet CopyAvd. Questa HttpServelt interagisce con il processo server in esecuzione su ogni MS scelta per il test, inviando un messaggio di Copia. Il processo server esegue la copia ed al termine risponde con l'esito dell'operazione. Se la copia è avvenuta con successo allora CopyAvd scrive sullo stream verso il browser utente il contenuto Html che dovrà essere visualizzato analogamente a quanto visto precedentemente. Il body html in questo caso è costituito da un blocco upload html che l'utente utilizzerà per caricare la su applicazione.

### 4.4.3 HttpServlet Upload

Questa HttpServlet si occupa dell'upload dell'applicazione da testare. In Figura 4.6, mostriamo come l'utente può caricare la sua applicazione.

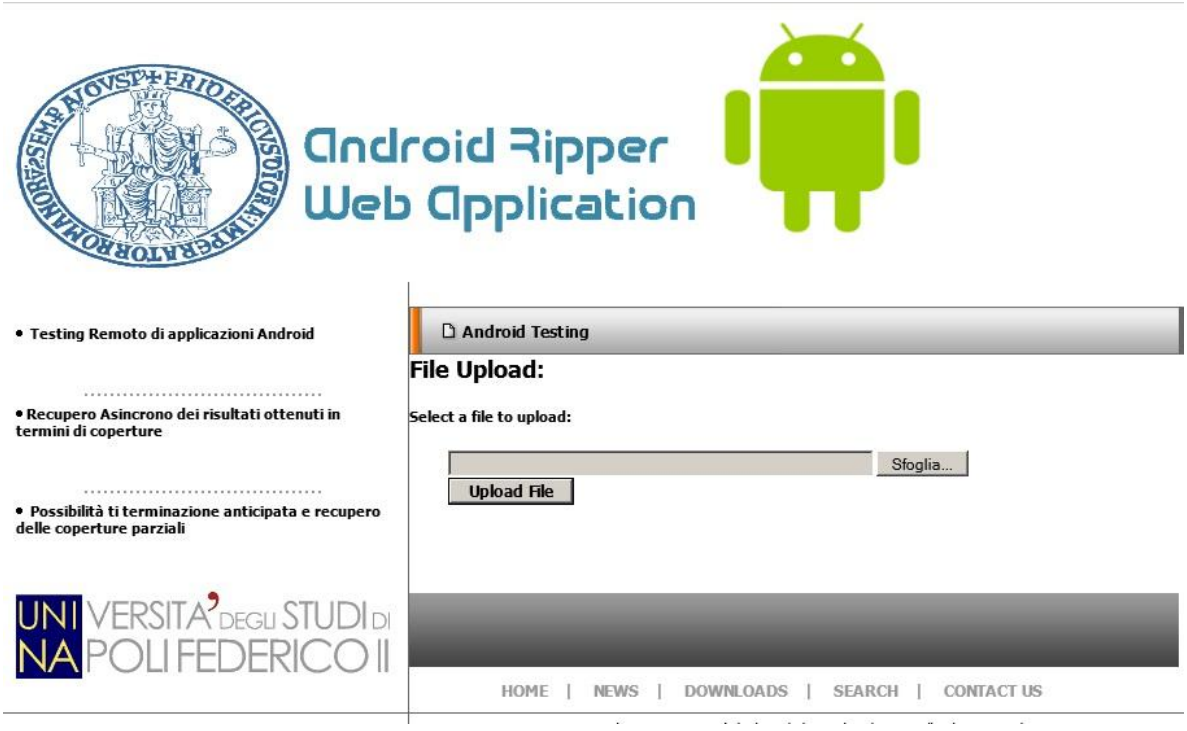
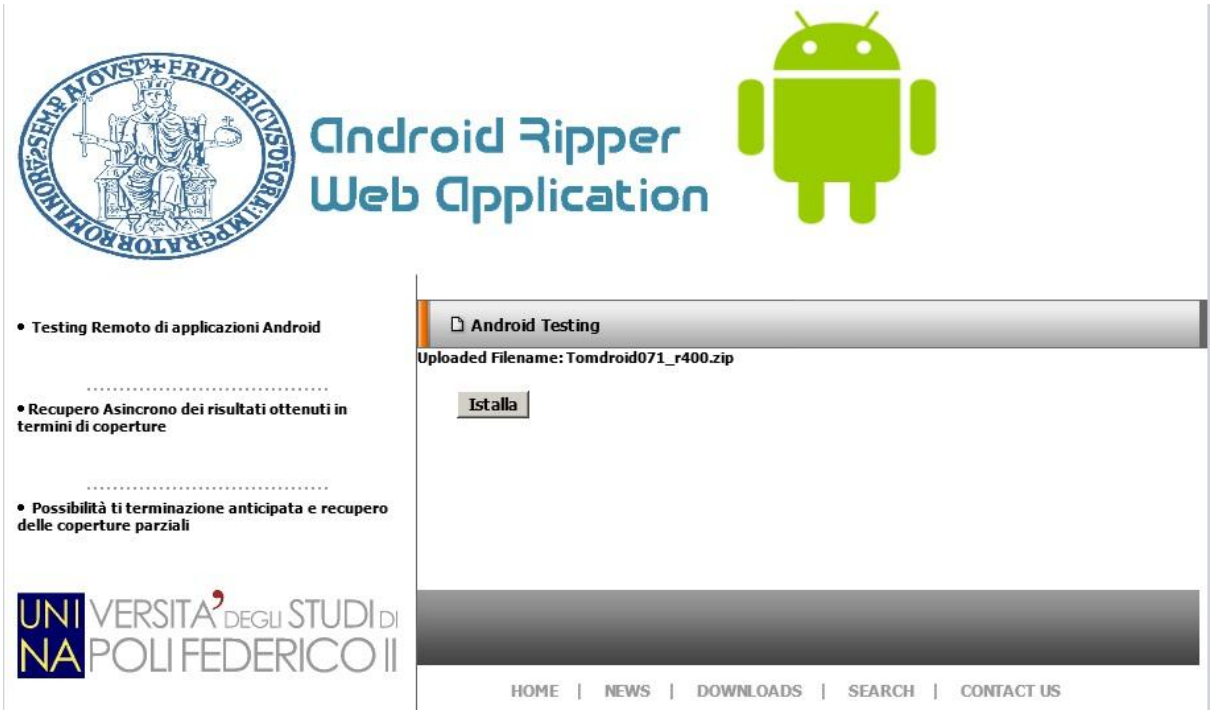



Figura 4.6 Upload

L'utente sceglierà l'applicazione da caricare premendo il tasto "Sfoglia" e premendo sul tasto "Upload File", sarà invocata una richiesta post http la quale scatenerà l'invocazione dell' HttpServlet Upload. Quest'ultima si occuperà del trasferimento dell'applicazione sulle Macchine Server (MS) selezionate.

### 4.4.4 HttpServlet AppInstall

Questa si occupa dell'installazione dell'app su tutti gli emulatori distribuiti. In Figura 4.7 mostriamo come l'utente lancia l'installazione di una applicazione di esempio: Tomdroid. Fino a questo momento sono state create solo delle avd (emulatori android sulle varie macchine).




\ Figura 4.7 AppInstall

La Figura 4.7 ci mostra il risultato dell' HttpServlet precedente. Premendo sul tasto "Installa", l'utente effettuerà una richiesta post http che scatenerà l'invocazione del metodo dopost() dell'HttpServlet AppInstall.

### 4.4.5 HttpServlet WebMain

Questa è la servlet che consente di avviare il processo del Ripper Multithread Remoto, a partire dalla configurazione impostata nel corso delle interazioni dell'utente fino a questo punto premendo sul tasto "Esegui" in Figura 4.8

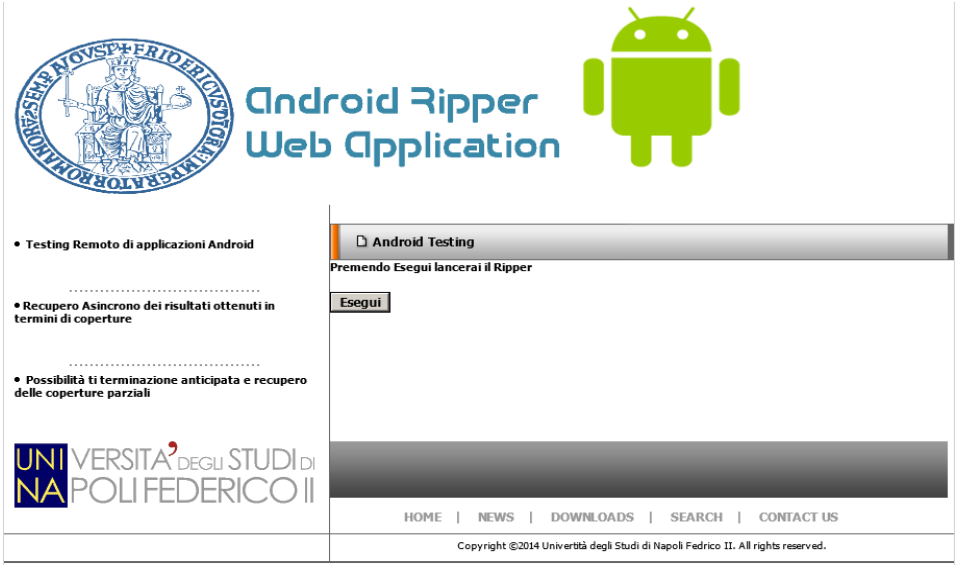



Figura 4.8 Esegui

Una volta premuto su "Esegui", questa HttpServlet visualizza un resoconto della configurazione del sistema (Figura 4.9)


Figura 4.9 Configurazione

### 4.4.6 HttpServlet Execution

Questa è l'HttpServlet che si occupa di controllare lo stato di esecuzione del sistema. Nella Figura 4.10 è possibile notare le opzioni di questa Httpservelt.


Figura 4.10 Execution

Con un refresh periodico della pagina html, consente la visualizzazione del numero di eventi svolti, la possibilità di terminare l'esecuzione e di prelevare risultati parziali.

### 4.4.7 HttpServlet ParzialResult

Questa HttpServlet si occupa del prelievo delle coperture parziali generate su ogni server. Queste coperture devono essere recuperate tutte sulla macchina MD. Per far ciò su ogni MS vi è un processo ad-hoc, deputato al trasferimento di questi file.


Figura 4.11 Execution

La pagina web che viene visualizzata dall'utente dopo l'avvio del sistema è mostrata in Figura 4.11

### 4.4.8 HttpServlet Stop

Questa HttpServlet si occupa della terminazione del sistema. L'utente può farlo premendo sul tasto "Stop" visualizzato nella pagina Html gestita da Execution A seguito di questa azione tutte le coperture saranno trasmesse sulla macchina MD. Il sistema si preoccuperà di autoterminarsi, non prima di aver compresso la cartella dei risultati che l'utente potrà scaricare.


Figura 4.12 Stop

La pagina web visualizzata dall'utente al a seguito dell'avvio del sistema è mostrata in Figura 4.12.

### 4.4.9 HttpServlet End

Questa HttpServlet viene invocata alla fine dell'esecuzione di tutto il sistema. Si occupa del trasferimento dei risultati presenti su tutti le Macchine Server, ed inizia la compressione della cartella che l'utente potrà scaricare.


Figura 4.13 End

Al termine della compressione in formato ".zip" effettua una *servlet redirect* verso l'HttpServlet Download illustrata in seguito.

### 4.4.10 HttpServlet Download

Questa Httpservlet consente all'utente di poter scaricare i risultati preventivamente raccolti in una cartella e poi zippati con il nome scelto dall'utente in fase di configurazione del sistema. L'utente cliccando il tasto "Scarica file" in Figura 4.13, avvierà il download dei risultati sul proprio PC.

Oltre a queste HttpServlet, abbiamo implementato delle classi di servizio per dividere la logica di interazione dalla logica operativa. Inoltre facciamo uso di pagine fogli di stile CSS e pagine Html con diversi contenuti javascript.

# Capitolo 5: Sperimentazione

In questo capitolo analizzeremo gli aspetti sperimentali di questo lavoro, specificando gli obiettivi che ci siamo preposti di raggiungere, le difficoltà incontrate , come sono state risolte e le configurazioni hardware e software utilizzate. Analizzeremo nel dettaglio i risultati sperimentali nelle due modalità di utilizzo del sistema (Random multi-thread remoto e Sistematico multi-thread remoto), facendo dei confronti tra i due approcci.

## 5.1 Obiettivi

Il primo passo attivo di una sperimentazione sono le research question ovvero la base da cui partire in un lavoro di ricerca.
Per ottenere dei buoni risultati, abbiamo provato a definire delle research question in modo chiaro ed accurato. Nel nostro caso partendo da un idea di sviluppo di una architettura parallela per la generazione automatica di casi di test per applicazioni Android, la prima research question che ci siamo posti è:

**RQ 1)** "La copertura ottenuta è funzione del parallelismo?"

Ovviamente non possiamo fermarci solo a questa e quindi per essere più precisi siamo andati oltre e ci siamo posti altre due Research Question:

**RQ 2)** "Quale architettura è più efficiente tra quella esistente del Ripper e quella parallela in termini di tempo di esecuzione?"

Specializzando quest'ultima domanda ci siamo posti altre sottodomande:

**RQ 2.1)** "Il tempo di esecuzione, è funzione del parallelismo?"

Per rispondere a quest'ultima domanda dobbiamo indagare ulteriormente ponendoci altre tre domanda. Ma prima definiamo dei parametri per chiarire meglio le domande:

#ne=numero emulatori

#nm=numero macchine

#nme=(#ne)*(#nm)

Veniamo alle domande:

- **RQ 2.1.1)** "A parità #nme il tempo è lo stesso?" O meglio, le configurazione (1-2) ~ (2-1) ci forniscono lo stesso risultato in termini di tempo?
- **RQ 2.1.2)** "A parità di #ne qual è il legame tra tempo e macchine?"
- **RQ 2.1.3) "**A parità di #nm, qual è il legame tra tempo e macchine?".

Prima di esaminare i dati e di rispondere a queste domande, esplicitiamo le variabili, le misure e le configurazioni adottate per questa sperimentazione in modo da poter dare una lettura chiara dei risultati.

## 5.2 Variabili e misure

Di seguito descriviamo le *independent variables* (o controlled variables), cioè le variabili che sono variate o manipolate durante la sperimentazione, le *dependent variables*, cioè i valori che risultano dalle *independent variables* del nostro problema.

E' necessario valutare e chiarire il rapporto tra le dependent ed independent variables. A tale proposito utilizziamo anche le controlled variables, cioè variabili mantenute

costanti durante le sessioni della sperimentazione.

### 5.2.1 Independent Variables

Per la nostra sperimentazione le variabili indipendenti sono il numero di macchine ed il numero di emulatori per macchina.

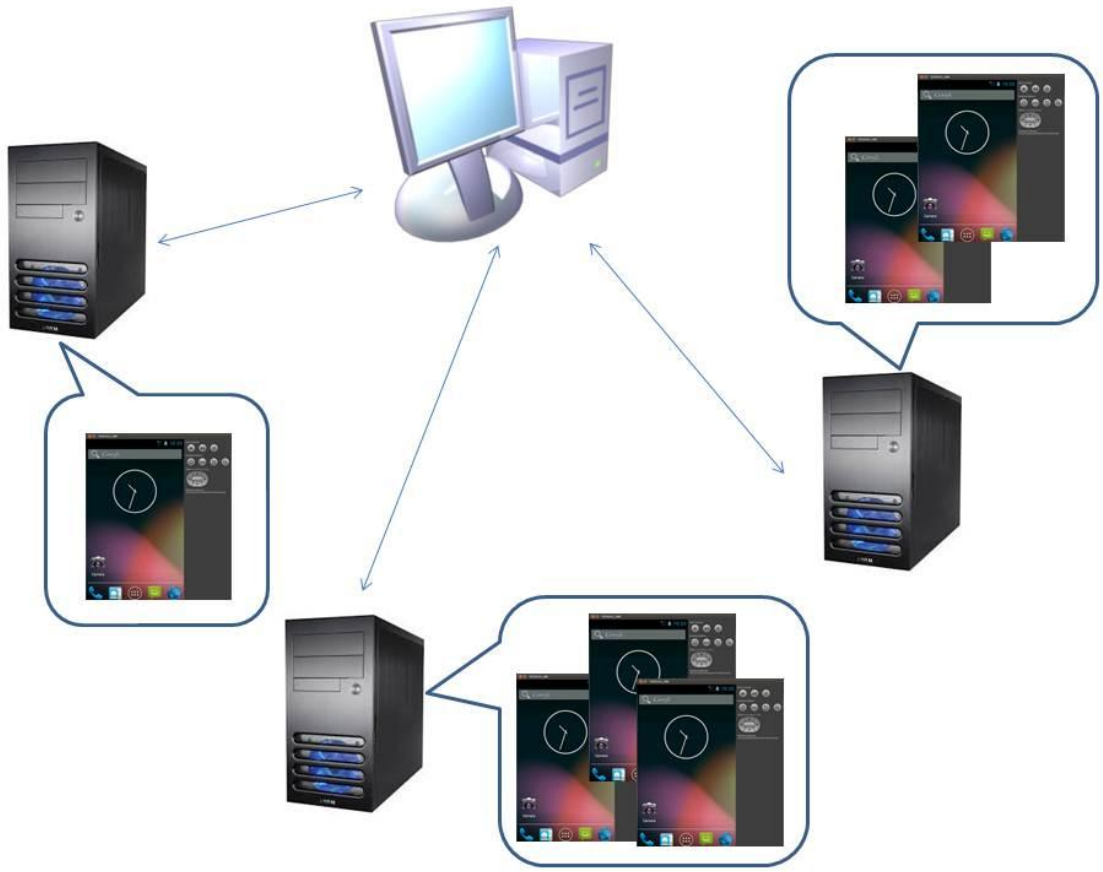


Figura 5.1 Independent Variables

In effetti combinando il numero di macchine con il numero di emulatori per macchina, otteniamo una macro variabile indipendente cioè la configurazione (#macc, # emu/macc).

Le configurazioni provate nella sperimentazione sono:

**conf [ (1,1) ; (2,1) ; (1,2) ; (2,2) ; (6,2) ]**

La configurazione (1,1) è la configurazione del ripper di base.

### 5.2.2 Dependent Variables

Le variabili dipendenti sono:

**- Tempo di esecuzione**

- tempo per macchina;
- tempo totale;
- tempo di avvio.

**- Copertura**

- Copertura nel tempo (nel caso Random è possibile valutare l'andamento del sistema rispetto alle coperture nel tempo ).

**- Numero di eventi (solo per la tecnica randomica)**

In qualche modo le variabili dipendenti sono l'effetto delle variabili indipendenti. Questa affermazione ci porta alle misure di interesse effettuate:

| Sistematico | | Random | |
|---|---|---|---|
| Fisso | - | Fisso | Numero eventi |
| Misuro | - Copertura<br>- Tempo<br>- Eventi | Misuro | - Copertura<br>- Tempo |

### 5.2.3 Controlled Variables

Abbiamo già specificato cosa intendiamo con control variable, vediamo quali sono.

Apps

Le applicazioni da testare nella sperimentazione sono Tomdroid, TippyTipper e TicTacToe. Facciamo solo una piccola osservazione. Prima di poter testare qualsiasi applicazione è necessario installarla su ogni emulatore di ogni macchina. Quindi il tempo di installazione sarà funzione della configurazione scelta (independent variables), del tempo di trasmissione dell' app e del tempo di avvio di ciascun emulatore (bisogna tener presente che se per esempio su una macchina vi sono tre emulatori, l'installazione può avvenire solo un emulatore per volta).

Tecnica

Le due tecniche di esplorazione/generazione dei casi di test sono la tecnica Random e la tecnica Sistematica.
Per il Random è possibile impostare il numero di eventi. Invece, il Sistematico lavora con logica Breadth First Search (BFS), quindi l'esplorazione dei possibili nuovi eventi avverrà in ampiezza in maniera sistematica (Figura 5.2).

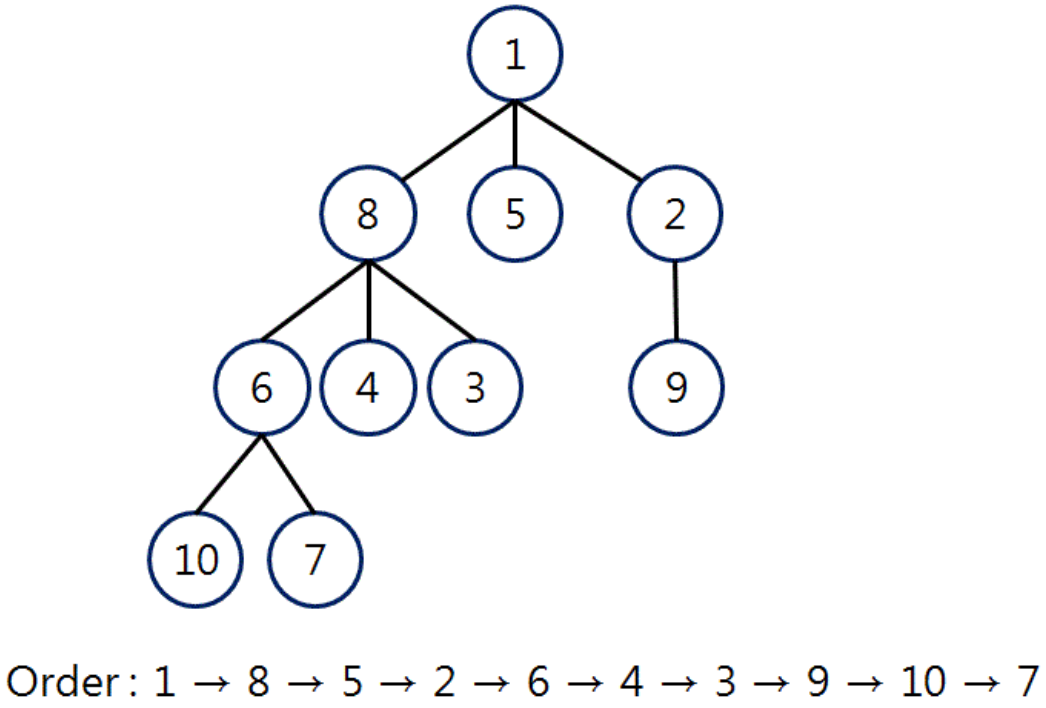



Figura 5.2 Breadth First Search (BFS)

Emulatori

Per ottenere risultati attendibili abbiamo cercato di utilizzare sempre gli stessi emulatori su macchine simili. Gli emulatori utilizzati prevedono come Device un Nexus S(4.0 480x800 dpi), 343 Mb di ram, 200 Mb di internal storage e 64 Mb per Sd card, Android 2.2.3.

## 5.3 Configurazione sperimentale

Per questa sperimentazione sono state utilizzate macchine con diversa capacità di calcolo ma tutte nella stessa rete locale.

La diversità delle macchine influisce sull'esecuzione del test. Per capire facciamo un esempio.

Supponiamo di utilizzare una MD e due MS (MS1 ed MS2) diverse ma entrambe con due emulatori (Figura 5.3). Supponiamo inoltre che la tecnica scelta sia Random e che

siano stati selezionati 8000 eventi.


Figura 5.3 Configurazione 2-2

Il caso ideale è che ciascuno emulatore effettui 2000 eventi e quindi 4000 eventi per MS. Se MS2 è molto più lenta di MS1, bisognerà attendere che MS2 finisca per terminare l'intera esecuzione, lasciando di fatto MS1 infruttuoso per tutto questo tempo.
In questa situazione avremmo una efficienza del sistema non ottimale.

La nostra architettura invece prevede che ciascuna macchina lavori al massimo delle sue potenzialità bilanciando il carico degli eventi a seconda della velocità delle macchine. Alla fine avremo ad esempio che MS1 ha realizzato 5540/8000 con due emulatori, mentre MS2 2460/8000.

## 5.4 Dati e discussione sperimentale

In questo paragrafo presentiamo i risultati della sperimentazione. Per semplicità i dati sono stati organizzati in tabelle e mostrati attraverso dei grafici.

### 5.4.1 Coperture

Iniziamo la discussione rispondendo alla prima Research Question :

**RQ 1)** "La copertura ottenuta è funzione del parallelismo?"

**SISTEMATICO**

| CONFIGURAZIONE - (SERVER,AVD) | COPERTURA (classi-metodi-blocchi-linee) | TEMPO | EVENTI (task) |
|---|---|---|---|
| Tomdroid - (1,1) | 59% - 41% - 31% - 32% | 3h 19 m | 89 |
| Tomdroid - (1,2) | 59% - 41% - 31% - 32% | 1h 45m | 91 |
| Tomdroid - (2,1) | 59% - 41% - 31% - 32% | 1h 11m | 91 |
| Tomdroid - (2,2) | 58% - 41% - 31% - 32% | 0h 49m | 84 |
| Tomdroid - (6,1) | 59% - 41% - 31% - 32% | 0h 33m | 90 |
| Tomdroid - (6,2) | 60% - 42% - 32% - 32% | 0h 16m | 95 |
| | | | |
| TippyTipper - (1,1) | 90% - 65% - 57% - 58% | 1h 02m | 59 |
| TippyTipper - (1,2) | 90% - 65% - 57% - 58% | 0h 45m | 59 |
| TippyTipper - (2,1) | 90% - 65% - 57% - 58% | 0h 43m | 61 |
| TippyTipper - (2,2) | 90% - 65% - 57% - 58% | 0h 32m | 56 |
| TippyTipper - (6,1) | 90% - 65% - 57% - 58% | 0h 19m | 69 |
| TippyTipper - (6,2) | 90% - 65% - 57% - 58% | 0h 14m | 69 |
| | | | |
| TicTacToe - (1,1) | 62% - 34% - 23% - 16% | 0h 19m | 39 |
| TicTacToe - (1,2) | 62% - 34% - 23% - 16% | 0h 13m | 37 |
| TicTacToe - (2,1) | 62% - 34% - 23% - 16% | 0h 11m | 39 |
| TicTacToe - (2,2) | 62% - 34% - 23% - 16% | 0h 08m | 27 |
| TicTacToe - (6,1) | 62% - 34% - 23% - 16% | 0h 07m | 39 |
| TicTacToe - (6,2) | 62% - 34% - 23% - 16% | 0h 04m | 39 |

Figura 5.4 Coperture Tecnica Sistematica

In Figura 5.4, mostriamo le coperture in percentuale relative alle applicazioni utilizzate per la sperimentazione (Tomdroid, TippyTipper e TicTcToe), al variare della configurazione (variabile indipendente), nel caso Sistematico.

Vediamo graficamente l'andamento delle coperture.

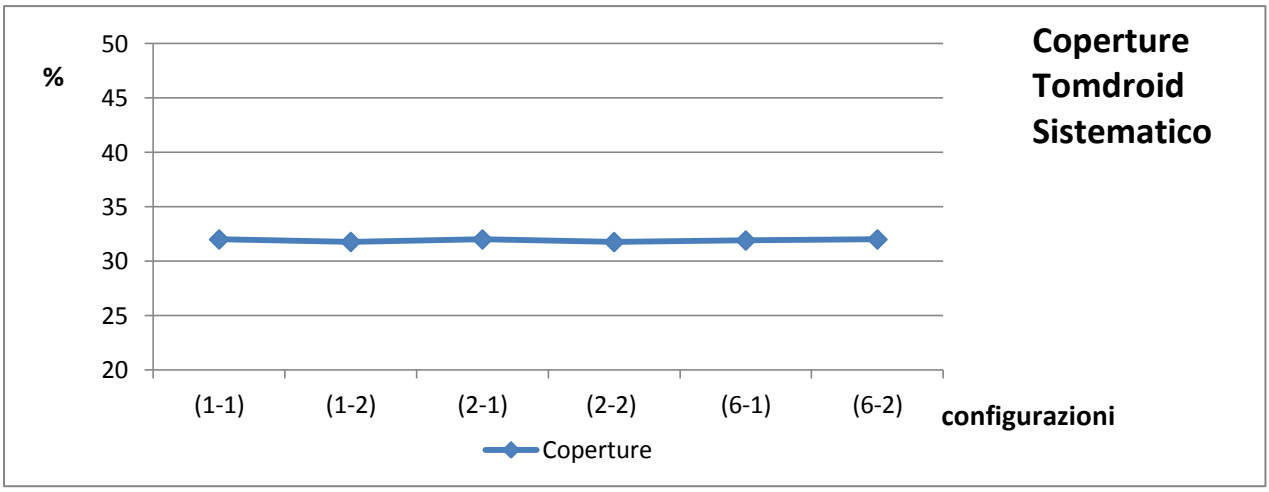



Figura 5.5.1 Andamento Coperture Sistematico Tomdroid

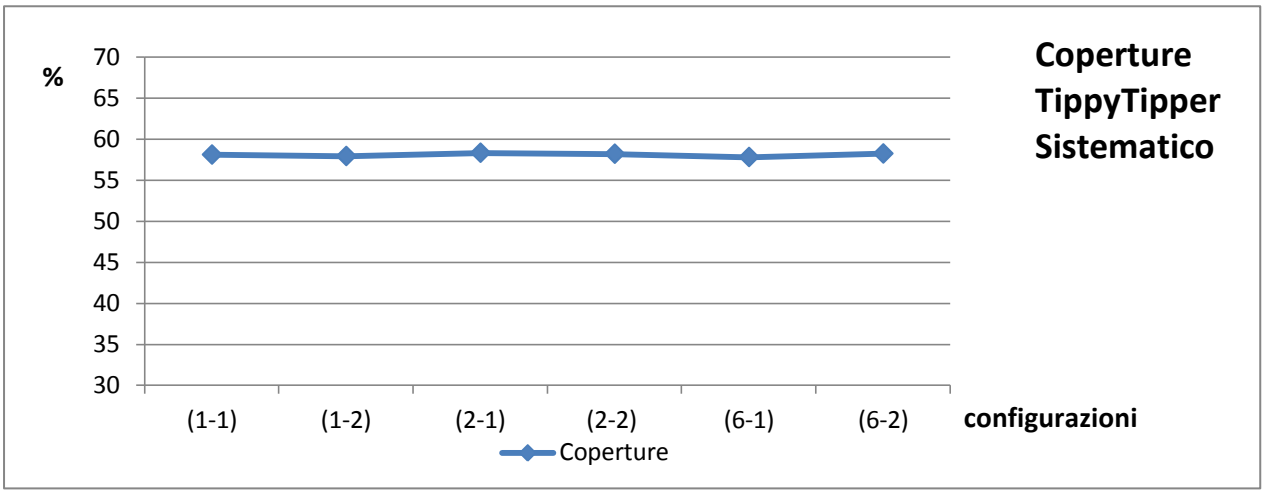



Figura 5.5.2 Andamento Coperture Sistematico TippyTipper

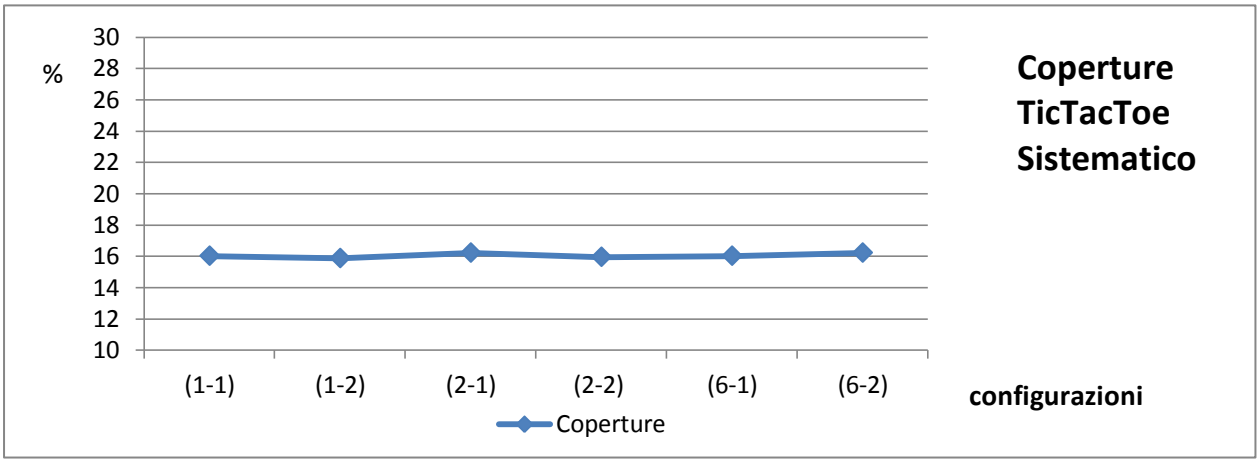



Figura 5.5.3 Andamento Coperture Sistematico TicTacToe

Come possiamo osservare le coperture restano pressoché invariate: al massimo si discostano dello 0,3%. Vediamo nel caso Randomico.

# RANDOM Tomdroid

| CONFIGURAZIONE - (SERVER,AVD) | COPERTURA (classi-metodi-blocchi-linee) | TEMPO | EVENTI |
|---|---|---|---|
| Tomdroid - (1,1) | 66% - 50% - 39% - 40% | 2h 13m | 1000 |
| Tomdroid - (1,2) | 66% - 50% - 41% - 42% | 1h 43 m | 1000 |
| Tomdroid - (2,1) | 66% - 50% - 41% - 42% | 1h 08m | 1000 |
| Tomdroid - (2,2) | 66% - 50% - 39% - 40% | 0h 39m | 1000 |
| Tomdroid - (6,1) | 66% - 50% - 40% - 41% | 0h 25m | 1000 |
| Tomdroid - (6,2) | 66% - 50% - 40% - 41% | 0h 13m | 1000 |
| | | | |
| Tomdroid - (1,1) | 68% - 55% - 47% - 48% | 4h 04m | 2000 |
| Tomdroid - (1,2) | 67% - 54% - 46% - 47% | 2h 34m | 2000 |
| Tomdroid - (2,1) | 68% - 55% - 47% - 48% | 2h 03m | 2000 |
| Tomdroid - (2,2) | 68% - 55% - 47% - 48% | 1h 12m | 2000 |
| Tomdroid - (6,1) | 67% - 54% - 47% - 48% | 0h 47m | 2000 |
| Tomdroid - (6,2) | 67% - 54% - 47% - 48% | 0h 26m | 2000 |
| | | | |
| Tomdroid - (1,1) | 71% - 57% - 49% - 50% | 10h 47m | 5000 |
| Tomdroid - (1,2) | 70% - 57% - 49% - 50% | 5h 26m | 5000 |
| Tomdroid - (2,1) | 71% - 57% - 49% - 51% | 5h 01m | 5000 |
| Tomdroid - (2,2) | 75% - 59% - 52% - 54% | 2h 51m | 5000 |
| Tomdroid - (6,1) | 71% - 58% - 50% - 52% | 1h 55m | 5000 |
| Tomdroid - (6,2) | 71% - 57% - 49% - 51% | 1h 03m | 5000 |
| | | | |
| Tomdroid - (1,1) | 74% - 58% - 53% - 53% | 22h 02m | 10000 |
| Tomdroid - (1,2) | 74% - 58% - 53% - 53% | 11h 08m | 10000 |
| Tomdroid - (2,1) | 74% - 58% - 53% - 53% | 10h 44m | 10000 |
| Tomdroid - (2,2) | 76% - 59% - 53% - 54% | 5h 33m | 10000 |
| Tomdroid - (6,1) | 75% - 57% - 52% - 53% | 4h 07m | 10000 |
| Tomdroid - (6,2) | 75% - 57% - 52% - 53% | 2h 13m | 10000 |
| | | | |
| Tomdroid - (1,1) | 74% - 55% - 52% - 53% | 57h 50m | 30000 |
| Tomdroid - (1,2) | 74% - 56% - 53% - 54% | 33h 52m | 30000 |
| Tomdroid - (2,1) | 74% - 56% - 53% - 54% | 32h 23m | 30000 |
| Tomdroid - (2,2) | 76% - 60% - 54% - 56% | 17h 09m | 30000 |
| Tomdroid - (6,1) | 74% - 55% - 52% - 53% | 12h 26m | 30000 |
| Tomdroid - (6,2) | 75% - 58% - 53% - 54% | 6h 34m | 30000 |

# RANDOM TippyTipper

| | | | |
|---|---|---|---|
| TippyTipper - (1,1) | 90% - 69% - 63% - 65% | 2h 01m | 1000 |
| TippyTipper - (1,2) | 90% - 69% - 64% - 66% | 1h 13m | 1000 |
| TippyTipper - (2,1) | 90% - 69% - 64% - 66% | 0h 58m | 1000 |
| TippyTipper - (2,2) | 90% - 69% - 64% - 66% | 0h 33m | 1000 |
| TippyTipper - (6,1) | 90% - 69% - 63% - 65% | 0h 23m | 1000 |
| TippyTipper - (6,2) | 90% - 69% - 63% - 65% | 0h 17m | 1000 |
| TippyTipper - (1,1) | 98% - 89% - 89% - 88% | 3h 58m | 2000 |
| TippyTipper - (1,2) | 98% - 89% - 89% - 88% | 2h 26m | 2000 |
| TippyTipper - (2,1) | 98% - 89% - 89% - 88% | 2h 04m | 2000 |
| TippyTipper - (2,2) | 98% - 89% - 89% - 88% | 1h 04m | 2000 |
| TippyTipper - (6,1) | 98% - 89% - 89% - 88% | 0h 43m | 2000 |
| TippyTipper - (6,2) | 98% - 89% - 89% - 88% | 0h 25m | 2000 |
| TippyTipper - (1,1) | 98% - 89% - 89% - 88% | 10h 48m | 5000 |
| TippyTipper - (1,2) | 98% - 89% - 89% - 88% | 5h 36m | 5000 |
| TippyTipper - (2,1) | 98% - 89% - 89% - 88% | 4h 52m | 5000 |
| TippyTipper - (2,2) | 98% - 89% - 89% - 88% | 2h 53m | 5000 |
| TippyTipper - (6,1) | 98% - 89% - 89% - 88% | 1h 48m | 5000 |
| TippyTipper - (6,2) | 98% - 89% - 89% - 88% | 1h 02m | 5000 |
| TippyTipper - (1,1) | 98% - 89% - 87% - 88% | 18h 58m | 10000 |
| TippyTipper - (1,2) | 98% - 89% - 87% - 88% | 10h 36m | 10000 |
| TippyTipper - (2,1) | 98% - 89% - 87% - 88% | 9h 49m | 10000 |
| TippyTipper - (2,2) | 98% - 89% - 89% - 88% | 5h 41m | 10000 |
| TippyTipper - (6,1) | 98% - 89% - 88% - 88% | 3h 36m | 10000 |
| TippyTipper - (6,2) | 98% - 89% - 88% - 88% | 1h 57m | 10000 |
| TippyTipper - (1,1) | 98% - 89% - 90% - 89% | 53h 22m | 30000 |
| TippyTipper - (1,2) | 98% - 89% - 90% - 89% | 31h 17m | 30000 |
| TippyTipper - (2,1) | 98% - 89% - 90% - 89% | 30h 10m | 30000 |
| TippyTipper - (2,2) | 98% - 89% - 90% - 89% | 15h 17m | 30000 |
| TippyTipper - (6,1) | 98% - 89% - 90% - 89% | 11h 29m | 30000 |
| TippyTipper - (6,2) | 98% - 89% - 90% - 89% | 5h 44m | 30000 |

# RANDOM TicTacToe

| | | | |
|---|---|---|---|
| TicTacToe - (1,1) | 62% - 38% - 25% - 19% | 3h 12m | 1000 |
| TicTacToe - (1,2) | 62% - 38% - 25% - 19% | 1h 37m | 1000 |
| TicTacToe - (2,1) | 62% - 38% - 25% - 19% | 1h 25m | 1000 |
| TicTacToe - (2,2) | 62% - 38% - 25% - 19% | 0h 39m | 1000 |
| TicTacToe - (6,1) | 62% - 38% - 25% - 19% | 0h 26m | 1000 |
| TicTacToe - (6,2) | 62% - 38% - 25% - 19% | 0h 15m | 1000 |

| | | | |
|---|---|---|---|
| TicTacToe - (1,1) | 62% - 38% - 25% - 19% | 7h 05m | 2000 |
| TicTacToe - (1,2) | 62% - 38% - 25% - 19% | 3h 41m | 2000 |
| TicTacToe - (2,1) | 62% - 38% - 25% - 19% | 3h 21 | 2000 |
| TicTacToe - (2,2) | 62% - 38% - 25% - 19% | 1h 16m | 2000 |
| TicTacToe - (6,1) | 62% - 38% - 25% - 19% | 0h 58m | 2000 |
| TicTacToe - (6,2) | 62% - 38% - 25% - 19% | 0h 33m | 2000 |
| | | | |
| TicTacToe - (1,1) | 62% - 38% - 25% - 19% | 13h 47m | 5000 |
| TicTacToe - (1,2) | 62% - 38% - 25% - 19% | 6h 57m | 5000 |
| TicTacToe - (2,1) | 62% - 38% - 25% - 19% | 6h 35m | 5000 |
| TicTacToe - (2,2) | 62% - 38% - 25% - 19% | 2h 58m | 5000 |
| TicTacToe - (6,1) | 62% - 38% - 25% - 19% | 2h 05m | 5000 |
| TicTacToe - (6,2) | 62% - 38% - 25% - 19% | 1h 29m | 5000 |
| | | | |
| TicTacToe - (1,1) | 62% - 38% - 25% - 19% | 28h 12m | 10000 |
| TicTacToe - (1,2) | 62% - 38% - 25% - 19% | 13h 45m | 10000 |
| TicTacToe - (2,1) | 62% - 38% - 25% - 19% | 13h 23m | 10000 |
| TicTacToe - (2,2) | 62% - 38% - 25% - 19% | 5h 57m | 10000 |
| TicTacToe - (6,1) | 62% - 38% - 25% - 19% | 4h 23m | 10000 |
| TicTacToe - (6,2) | 62% - 38% - 25% - 19% | 2h 11m | 10000 |
| | | | |
| TicTacToe - (1,1) | 62% - 38% - 25% - 19% | 80h 57m | 30000 |
| TicTacToe - (1,2) | 62% - 38% - 25% - 19% | 43h 22m | 30000 |
| TicTacToe - (2,1) | 62% - 38% - 25% - 19% | 42h 12m | 30000 |
| TicTacToe - (2,2) | 62% - 38% - 25% - 19% | 16h 39m | 30000 |
| TicTacToe - (6,1) | 62% - 38% - 25% - 19% | 11h 12m | 30000 |
| TicTacToe - (6,2) | 62% - 38% - 25% - 19% | 5h 24m | 30000 |

Figura 5.6 Coperture totali

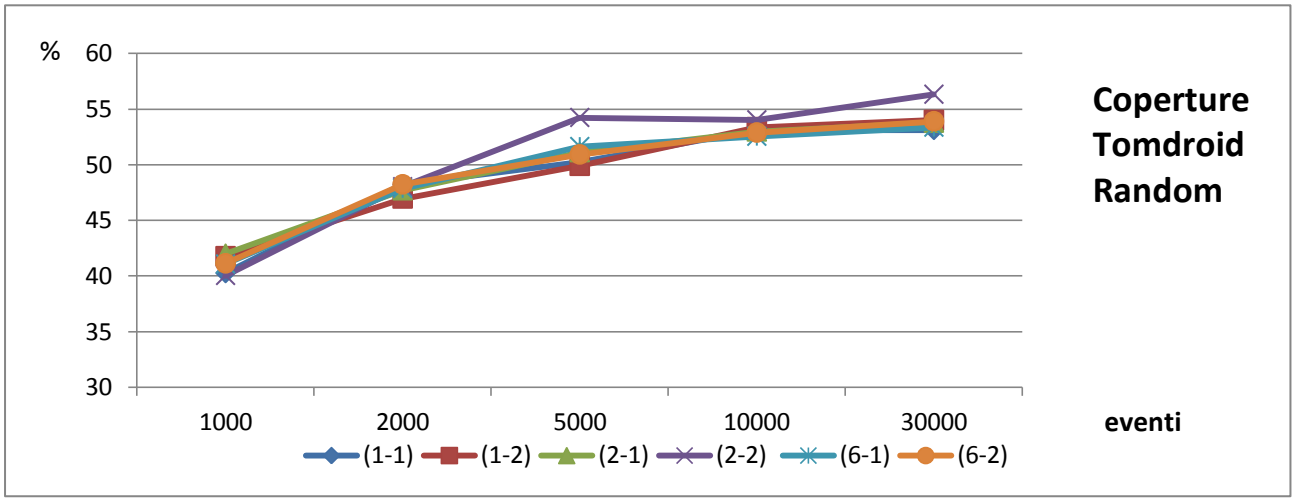



Figura 5.7 Andamento Coperture Random Tomdroid

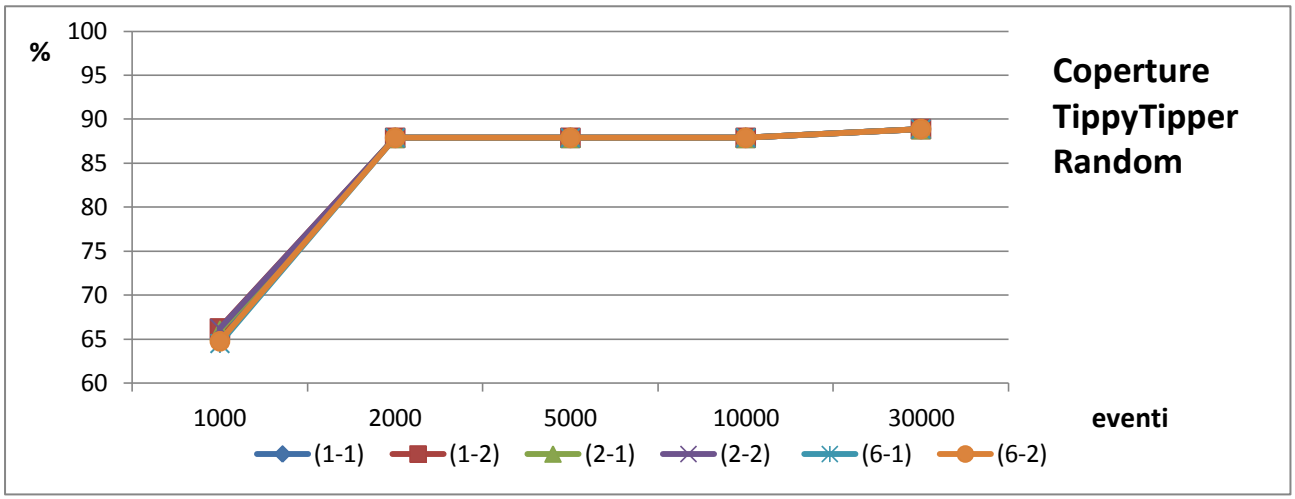



Figura 5.8 Andamento Coperture Random TippyTipper

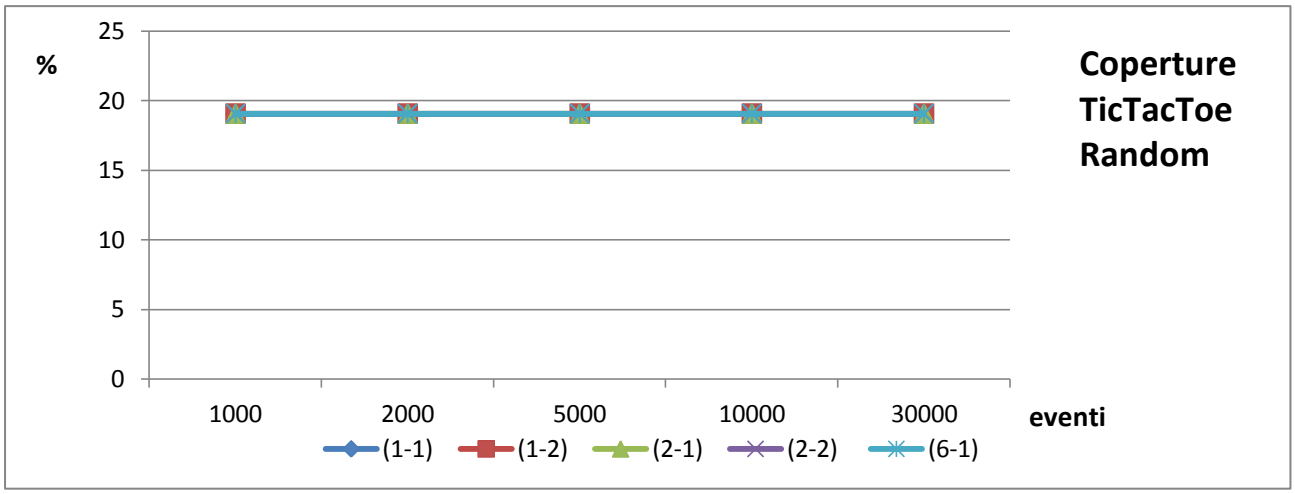



Figura 5.9 Andamento Coperture Random TicTacToe

Dato che la sperimentazione nel caso Random è stata fatta per un numero crescente di eventi (1000, 2000, 5000, 10000, 30000), in un unico grafico riportiamo l'andamento al variale delle configurazioni a parità di numero di eventi.

### 5.4.2 Tempo di esecuzione.

Dobbiamo rispondere alla seconda Research Question:

**RQ 2.1)** "Il tempo è funzione del parallelismo?"

Come già accennato in precedenza per poter rispondere a questa domanda, dobbiamo rispondere prima ad altre tre domande:

- **RQ 2.1.1)** "A parità #nme il tempo è lo stesso?"
- **RQ 2.1.2)** "A parità di #ne qual è il legame tra tempo e macchine?"
- **RQ 2.1.3) "**A parità di #nm, qual è il legame tra tempo e macchine?".

### 5.4.2.1 Numero totale di emulatori

Per rispondere a alla RQ 2.1.1, abbiamo verificato che se consideriamo lo stesso numero di emulatori totali ma ottenuti con due configurazioni diverse, il tempo di esecuzione rimane lo stesso. Meglio ancora, bisogna capire se avere una macchina con due emulatori in esecuzione è più vantaggioso di avere invece due macchine ciascuna con un solo emulatore in esecuzione, e viceversa.

Vediamo cosa accade nel caso Sistematico

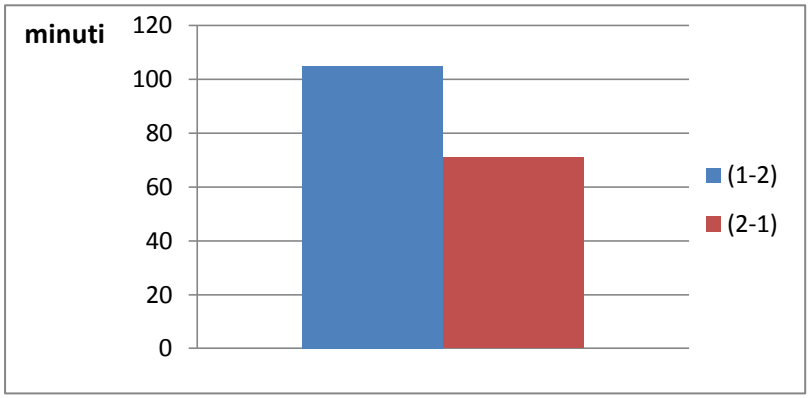



Figura 5.10 Tempo Tomdroid caso Sistematico

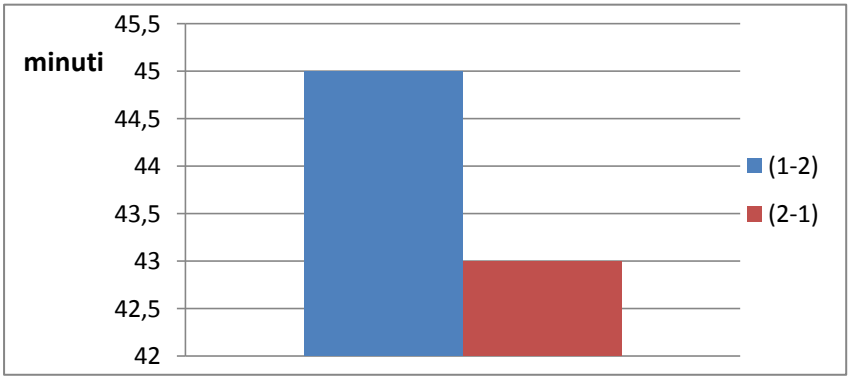



Figura 5.11 Tempo TippyTipper caso Sistematico

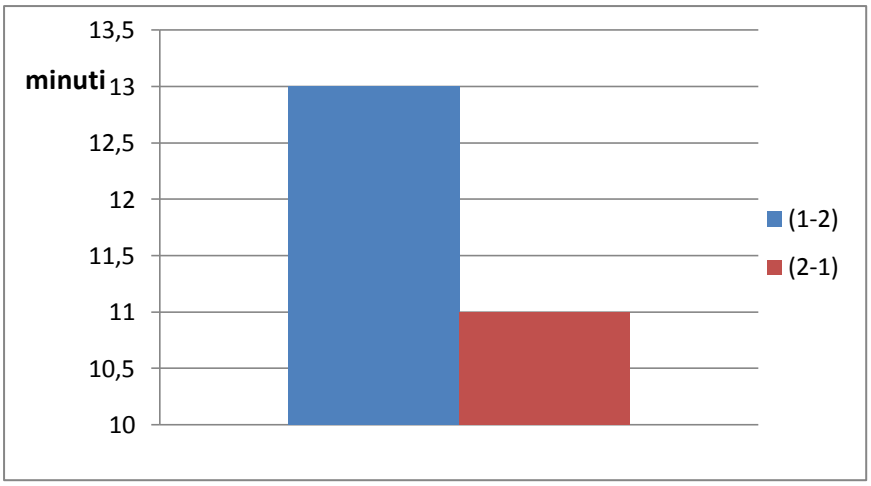



Figura 5.12 Tempo TicTacToe caso Sistematico

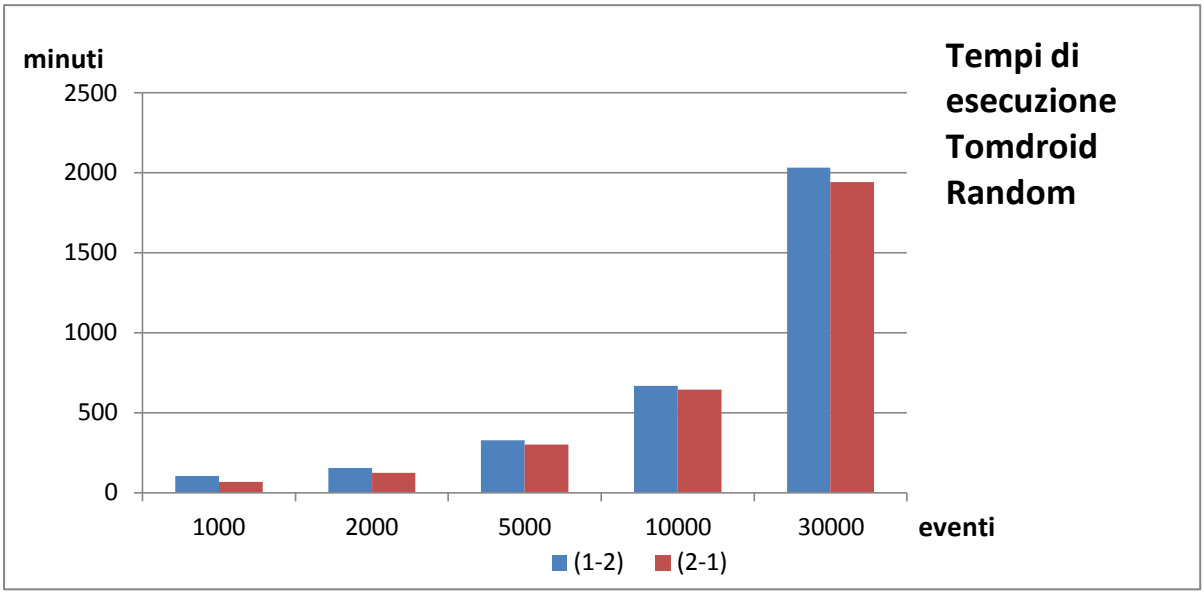



Figura 5.13 Tempo Tomdroid caso Random

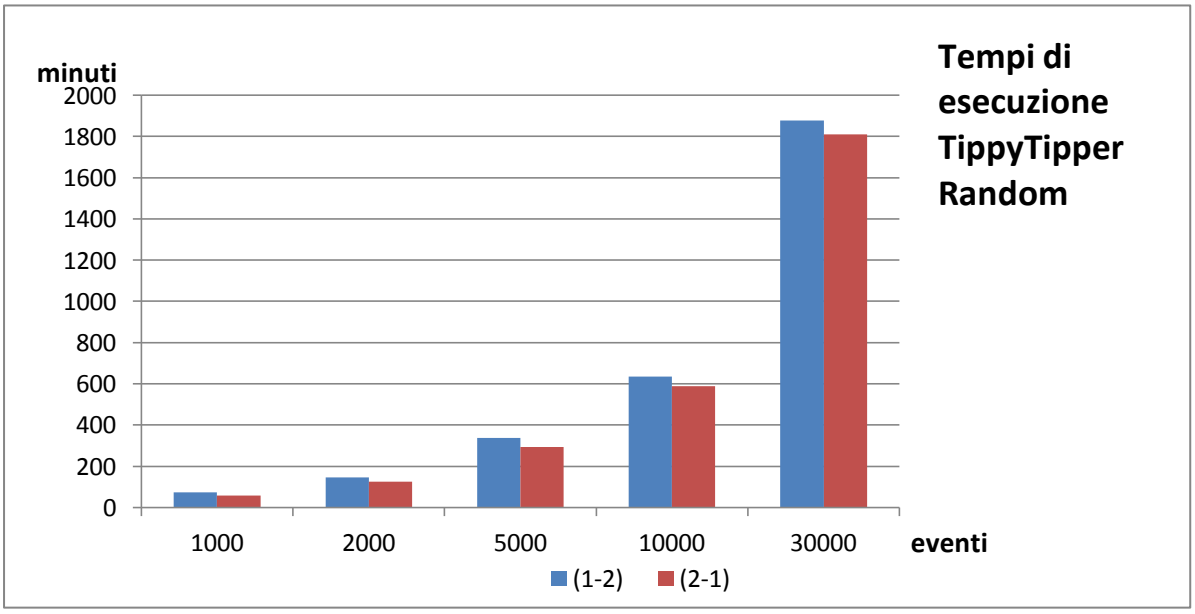



Figura 5.14 Tempo TippyTipper caso Random

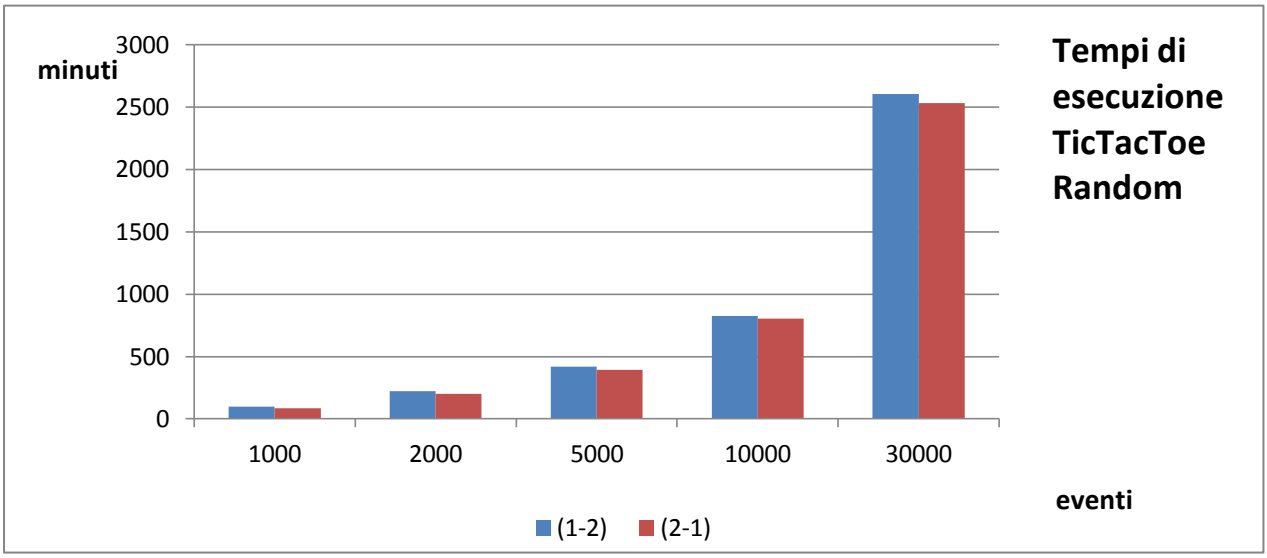



Figura 5.15 Tempo TicTacToe caso Random

Così come possiamo osservare dalle Figure dalla 5.10 alla 5.15, notiamo che la configurazione (1-2) è più lenta di quella (2-1). Nello specifico nel caso Random la media delle differenze in percentuale è 18.5% per Tomdroid, del 13.6% per TippiTipper e del 6.9% per TicTacToe; nel caso Sistematico invece la differenza è del 47.8% per Tomdroid, del 4.6% per TippyTipper e del 18.1% per TicTacToe . Ciò può dipendere dal fatto che ogni emulatore nella sua configurazione minima ha bisogno di 350 Mb di ram, richiede almeno 200Mb di Internal Storage e 64 Mb di Sd card. Quindi per una macchina[1] risulta più onerosa la gestione di due emulatori invece di uno soltanto. Inoltre su ogni macchina server abbiamo realizzato un componente che fa da proxy, che è il tramite tra il driver e l'emulatore quindi un altro motivo di ritardo è dovuto al carico di lavoro che deve svolgere questo componente.

Quindi la risposta alla domanda RS 2.2.1 è:

**RA 2.2.1)** "Il tempo di esecuzione non è lo stesso a parità di numero di emulatori totali".

[1] Intel Core i5-3330 3.0Ghz, 4Gb Ram, S.O. Windows 7 a 64 bit

### 5.4.2.2 Tempo al variare delle macchine

Vediamo cosa succede se fissiamo il numero di emulatori per macchina e facciamo variare il numero delle macchine. Sarebbe ideale osservare un andamento lineare.

Vediamo il caso randomico.

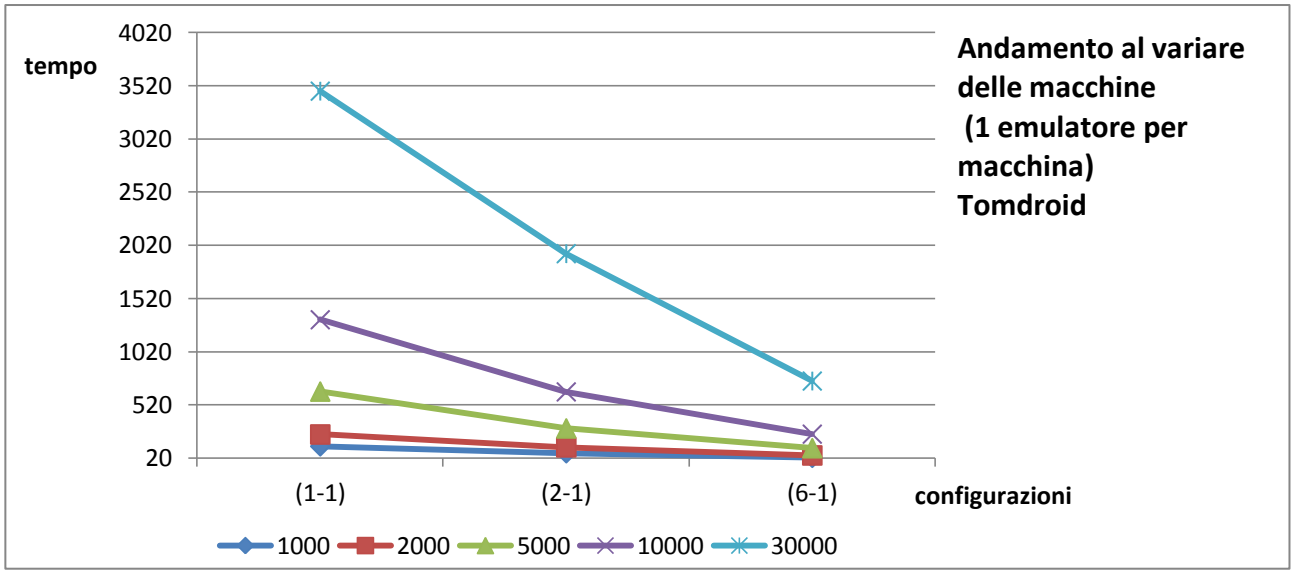



Figura 5.16.1 Tempo al variare delle macchine Tomdroid

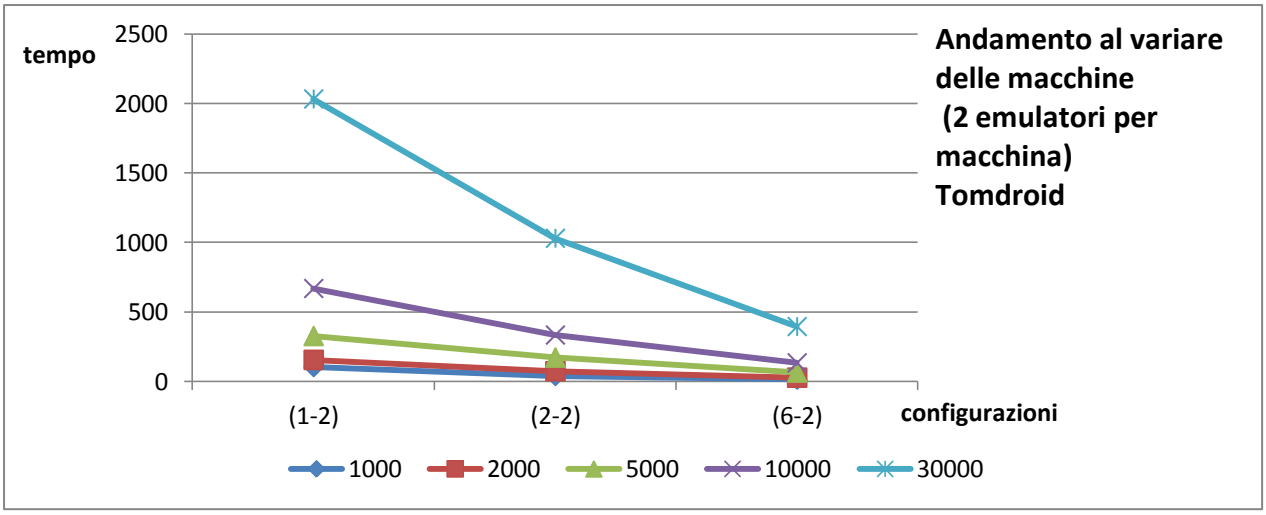



Figura 5.16.2 Tempo al variare delle macchine Tomdroid

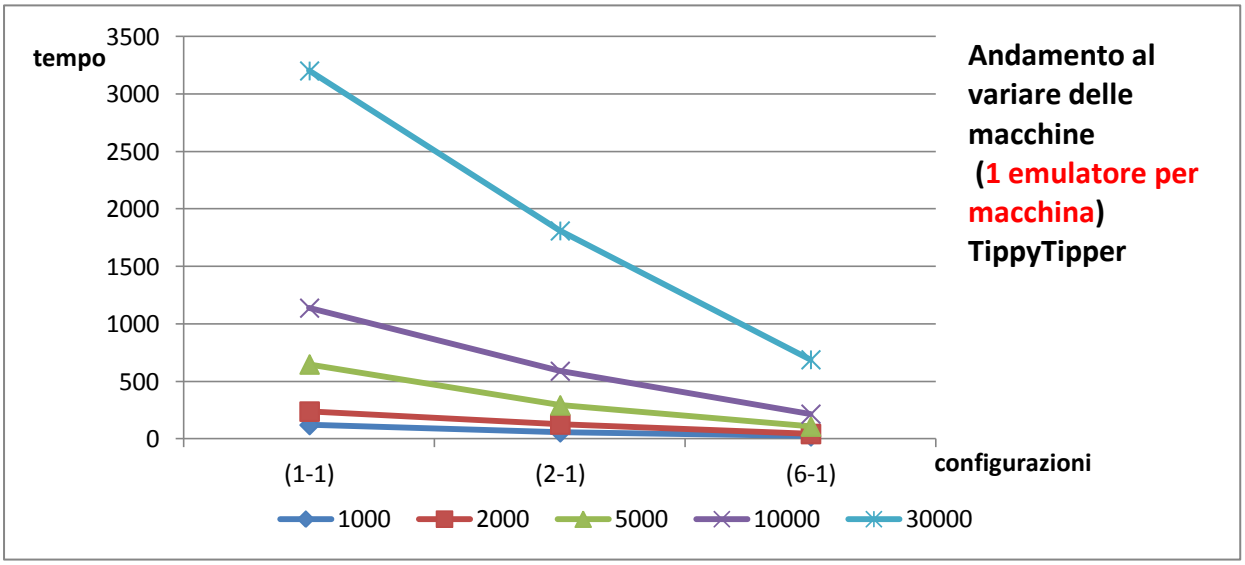



Figura 5.16.3 Tempo al variare delle macchine TippyTipper

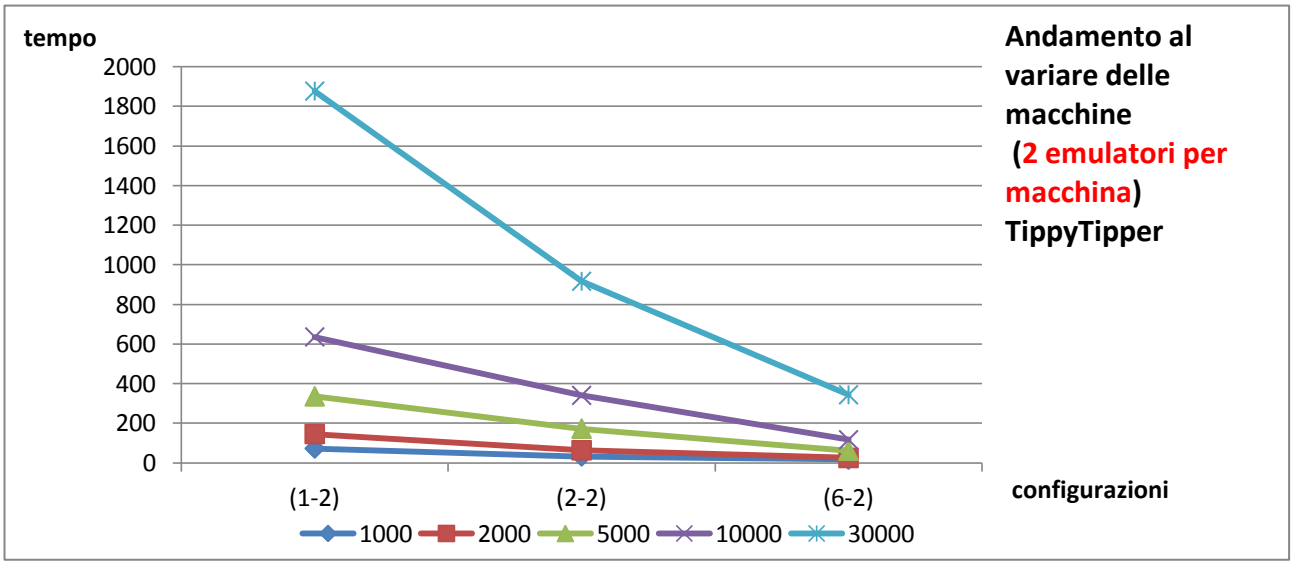



Figura 5.16.4 Tempo al variare delle macchine TippyTipper

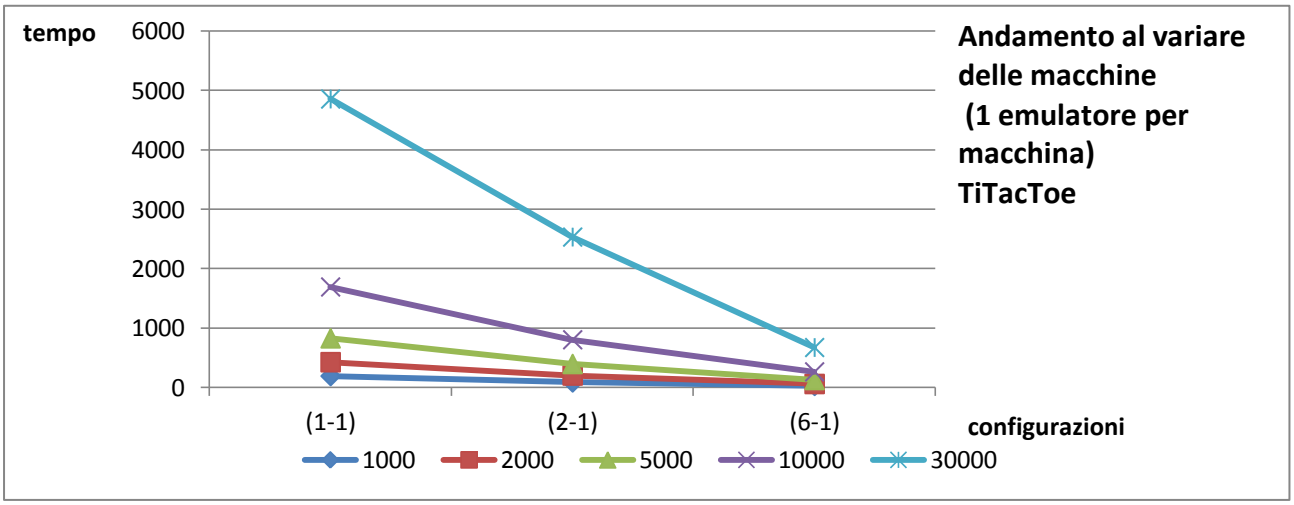



Figura 5.16.5 Tempo al variare delle macchine TiTacToe

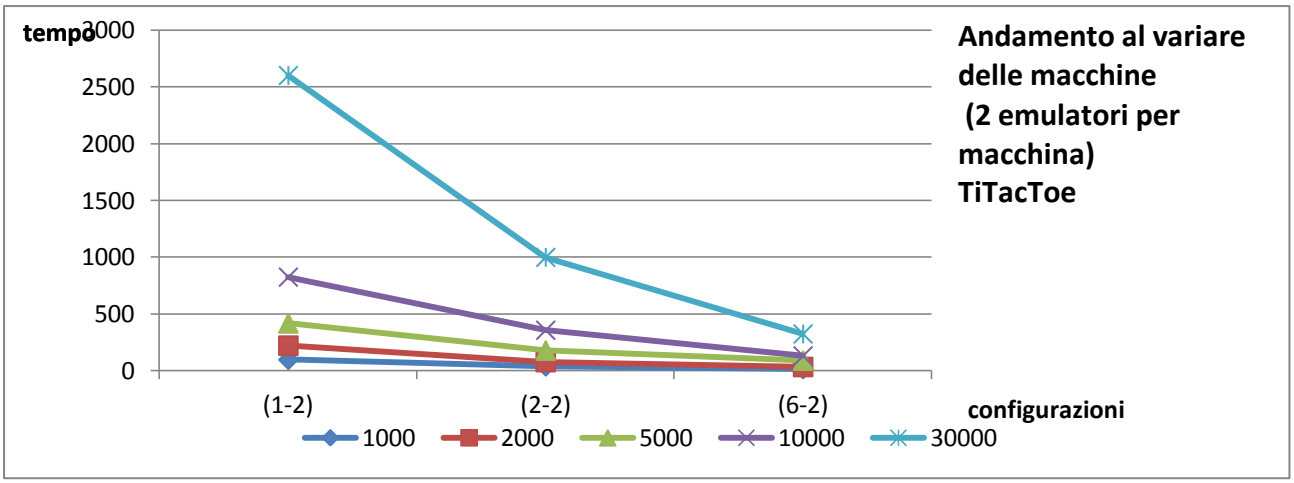



Figura 5.16.6 Tempo al variare delle macchine TiTacToe

Vediamo il caso Sistematico.

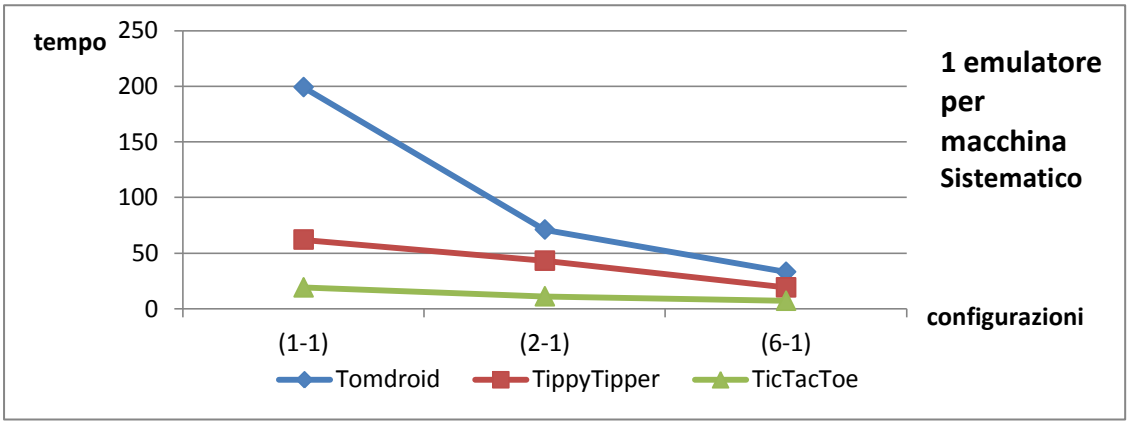



Figura 5.16.7 Tempo al variare delle macchine 1 emulatore

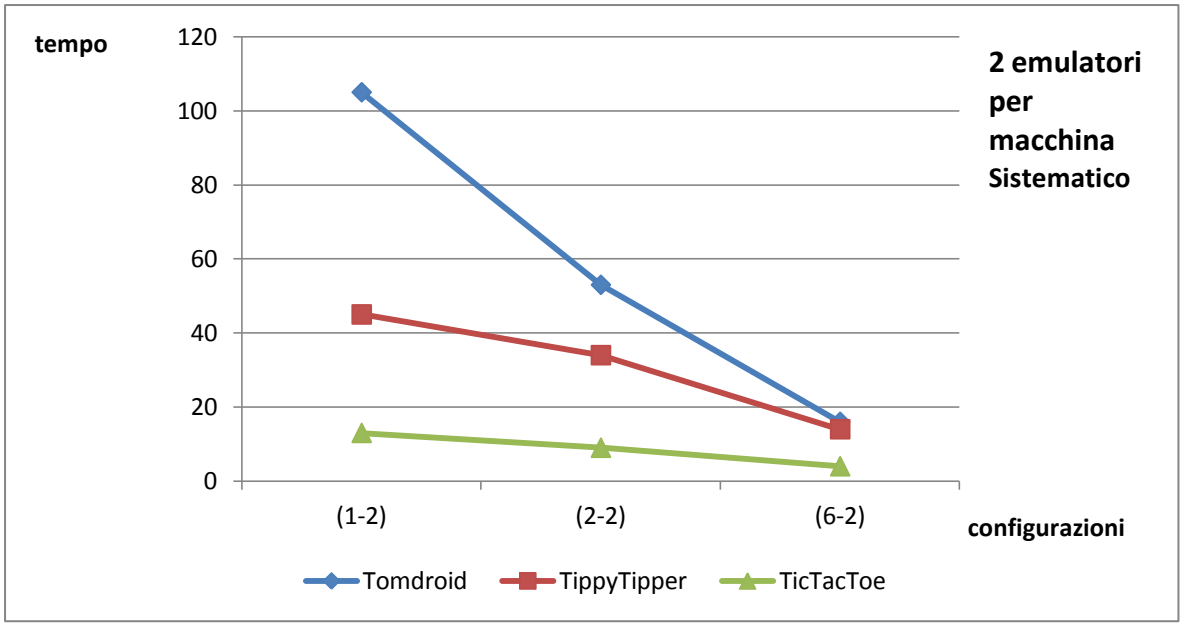



Figura 5.16.8 Tempo al variare delle macchine 2 emulatori

A partire proprio da questi risultati, possiamo senza dubbio affermare che il sistema parallelo così come è stato realizzato in ogni sua configurazione, consente una significativa riduzione dei tempi di esecuzione.

Tale risultato risulta particolarmente evidente nel caso Random con un numero elevato di eventi da effettuare; infatti nel caso peggiore con la configurazione più favorevole (6-2), il sistema ha impiegato 6 ore e 34 minuti circa a fronte delle 57 ore e 50 minuti circa della configurazione di base (1-1). Quindi è stato **12 volte** circa più veloce del caso base.

Volendo determinare una relazione matematica, dobbiamo considerare **#ne** (numero emulatori), #**nm** (numero macchine), T_tot (tempo totale).

Idealmente avremmo voluto avere una relazione del tipo:

$$T\_tot = t(1\text{-}1)/(\#ne * \#nm) \qquad a)$$

In realtà possiamo ipotizzare che la relazione sia del tipo:

$$T\_tot = k*[t(1\text{-}1)/(\#ne * \#nm)] \qquad b)$$

La presenza di questo **k** è legato a dei ritardi inevitabili causati in primo luogo dalla comunicazione di rete tra gli emulatori e il Driver. Inoltre, come abbiamo già affermato in precedenza, sia la macchina server che il proxy (che lavora su essa), devono gestire il doppio del lavoro (il proxy scambia il doppio dei messaggi tra l'emulatore ed il driver). Calcolando **k** possiamo osservare in Figura 5.17.1 caso Random e 5.17.2 i valori calcolati nel caso Sistematico.

| **Tomdroid** | **(1-2)** | **(2-1)** | **(2-2)** | **(6-1)** | **(6-2)** | **eventi** |
|---|---|---|---|---|---|---|
| k | 1,55 | 1,02 | 1,06 | 1,12 | 1,18 | 1000 |
| | 1,27 | 1,01 | 1,18 | 1,16 | 1,29 | 2000 |
| | 1,01 | 0,9 | 1,06 | 1,07 | 1,17 | 5000 |
| | 1,01 | 0,975 | 1 | 1,12 | 1,21 | 10000 |
| | 1,17 | 1,12 | 1,18 | 1,29 | 1,36 | 30000 |
| **TippyTipper** | **(1-2)** | **(2-1)** | **(2-2)** | **(6-1)** | **(6-2)** | **eventi** |
| k | 1,21 | 0,96 | 1,09 | 1,14 | 1,68 | 1000 |
| | 1,23 | 1,06 | 1,08 | 1,09 | 1,26 | 2000 |
| | 1,04 | 0,9 | 1,07 | 0 | 1,15 | 5000 |
| | 1,118 | 1,03 | 1,2 | 1,14 | 1,24 | 10000 |
| | 1,17 | 1,13 | 1,146 | 1,291 | 1,29 | 30000 |
| **TicTacToe** | **(1-2)** | **(2-1)** | **(2-2)** | **(6-1)** | **(6-2)** | **eventi** |
| k | 1,012 | 0,88 | 0,82 | 0,81 | 0,94 | 1000 |
| | 1,04 | 0,94 | 0,72 | 0,82 | 0,94 | 2000 |
| | 1,01 | 0,95 | 0,86 | 0,91 | 1,28 | 5000 |
| | 0,975 | 0,95 | 0,85 | 0,93 | 0,93 | 10000 |
| | 1,072 | 1,043 | 0,82 | 0,83 | 0,82 | 30000 |

Figura 5.17.1 valori di k caso Random

| **Tomdroid** | **(1-2)** | **(2-1)** | **(2-2)** | **(6-1)** | **(6-2)** |
|---|---|---|---|---|---|
| k | 1,055 | 0,714 | 1,07 | 1 | 0,97 |
| **TippyTipper** | **(1-2)** | **(2-1)** | **(2-2)** | **(6-1)** | **(6-2)** |
| k | 1,46 | 1,39 | 2,78 | 1,84 | 2,71 |
| **TicTacToe** | **(1-2)** | **(2-1)** | **(2-2)** | **(6-1)** | **(6-2)** |
| k | 1,37 | 1,16 | 1,9 | 2,23 | 2,55 |

Figura 5.17.2 valori di k caso Sistematico

Di seguito mostriamo dei grafici a dispersione per valutare l'andamento di **k** e capire se la relazione **b)** può essere considerata di tipo lineare o meno.

Vediamo prima il caso Random al variare delle macchine mantenendo fissi gli emulatori.

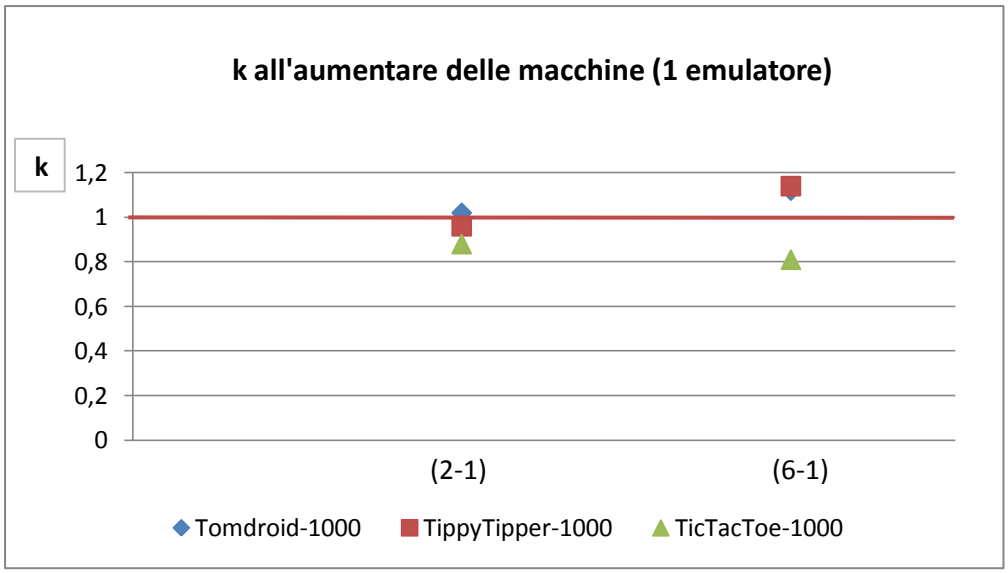



Figura 5.18.1 valori di k (Random 1000 eventi 1 emulatore)

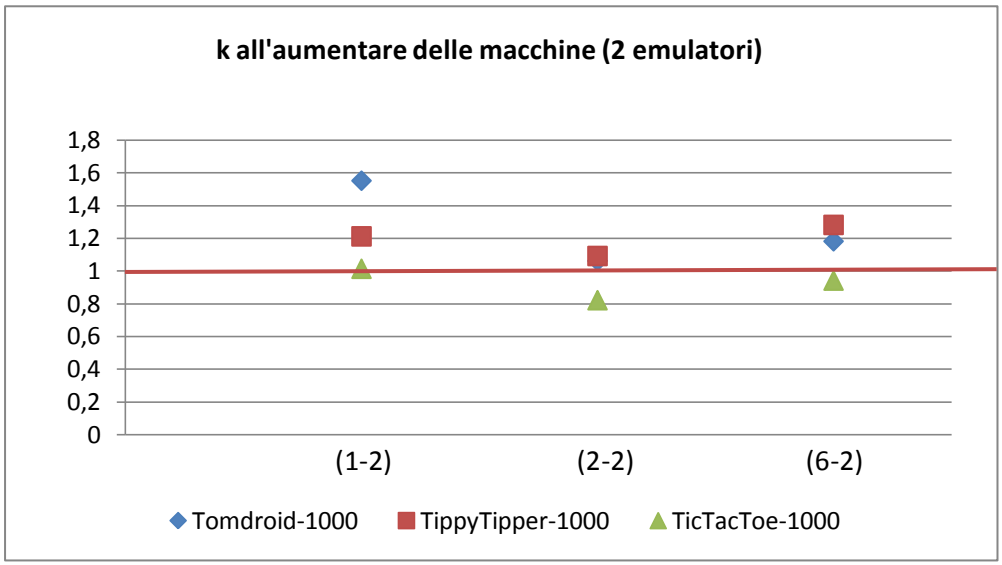



Figura 5.18.2 valori di k (Random 1000 eventi 2 emulatori)

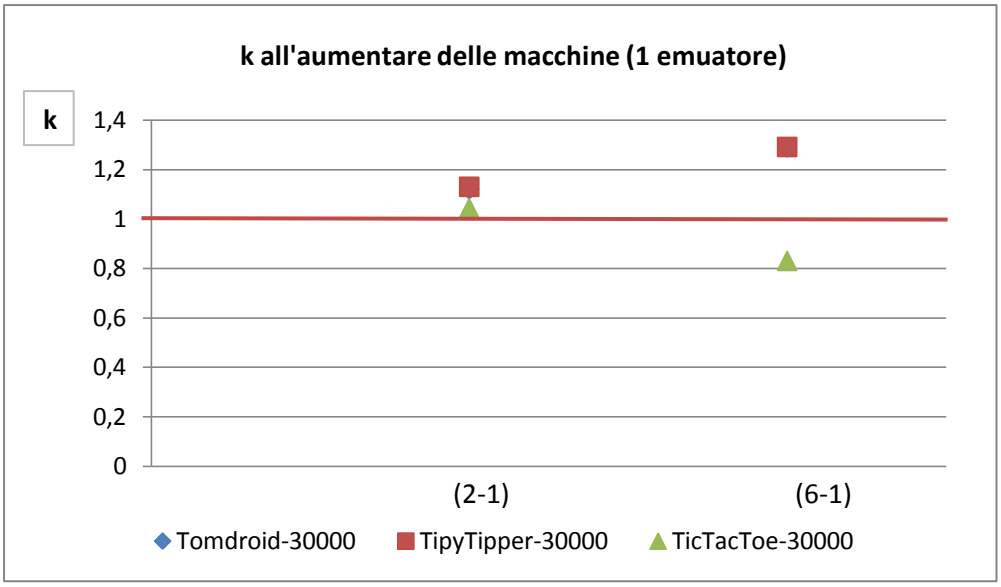



Figura 5.18.2 valori di k (Random 30000 eventi 1 emulatore)

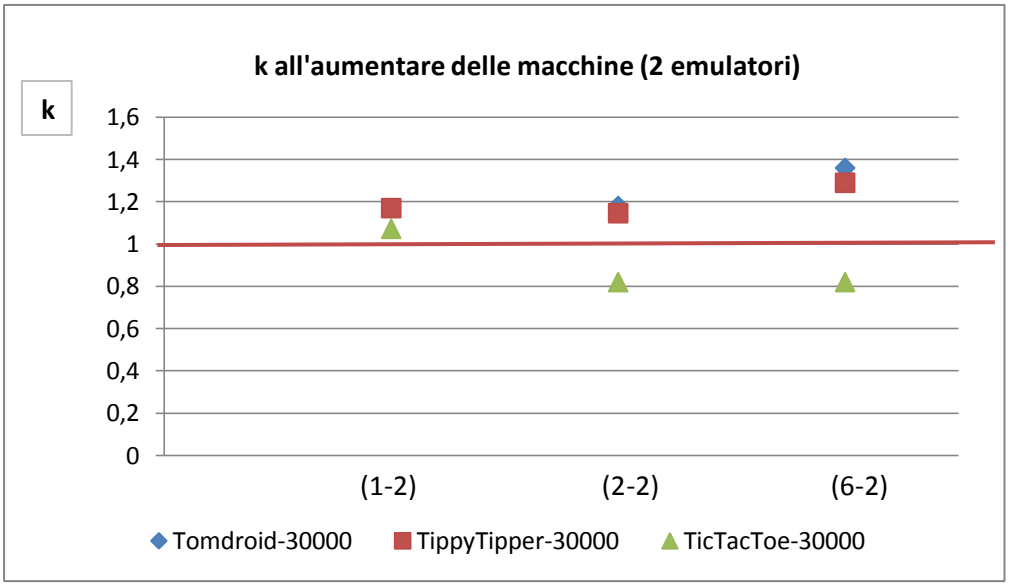



Figura 5.18.3 valori di k (Random 30000 eventi 2 emulatori)

Avremmo voluto osservare un andamento costante del parametro k. In realtà non possiamo affermare né che la relazione **b)** sia lineare né possiamo escluderlo.

Vediamo ora il caso Sistematico sempre al variare delle macchine mantenendo fisso il numero di emulatori:

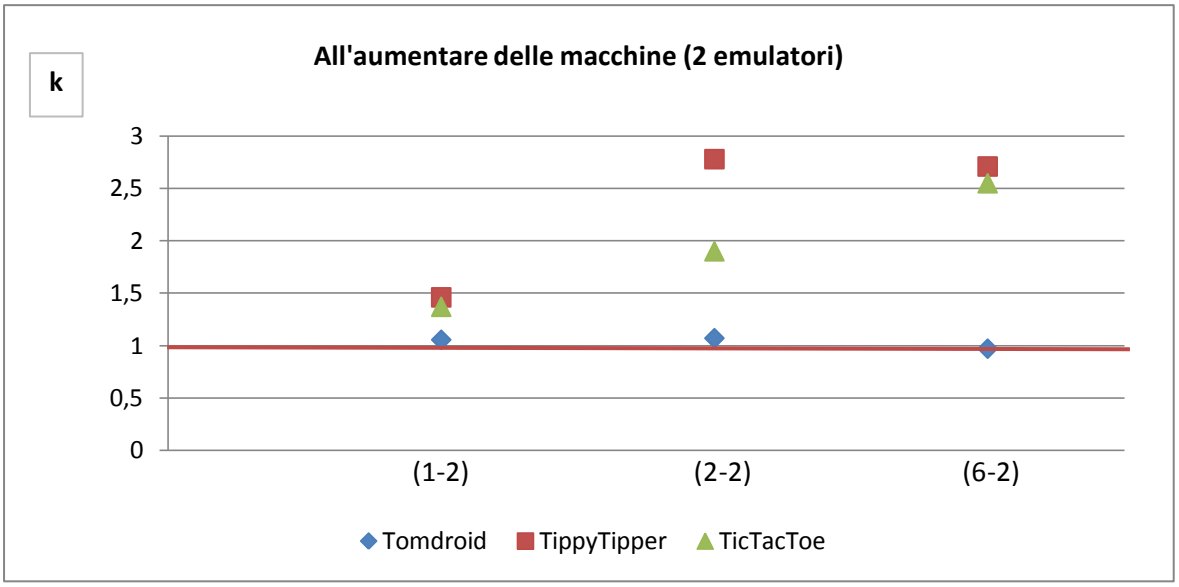



Figura 5.18.4 valori di k (Sistematico con 2 emulatori per amcchina)

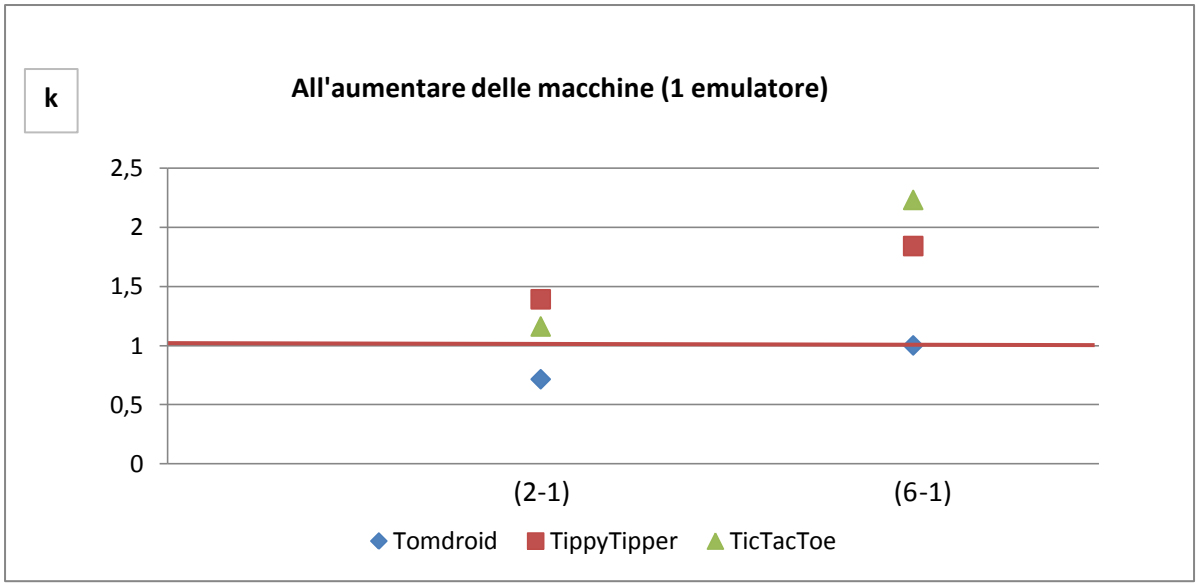



Figura 5.18.5 valori di k (Sistematico con un emulatore per macchina)

Anche in questo non possiamo affermare né che la relazione **b)** sia lineare né possiamo escluderlo.

### 5.4.2.3 Tempo al variare degli emulatori

Di seguito, vediamo cosa accade se invece di far variare le macchine facciamo variare gli emulatori. Consideriamo l'approccio con tecnica randomica.

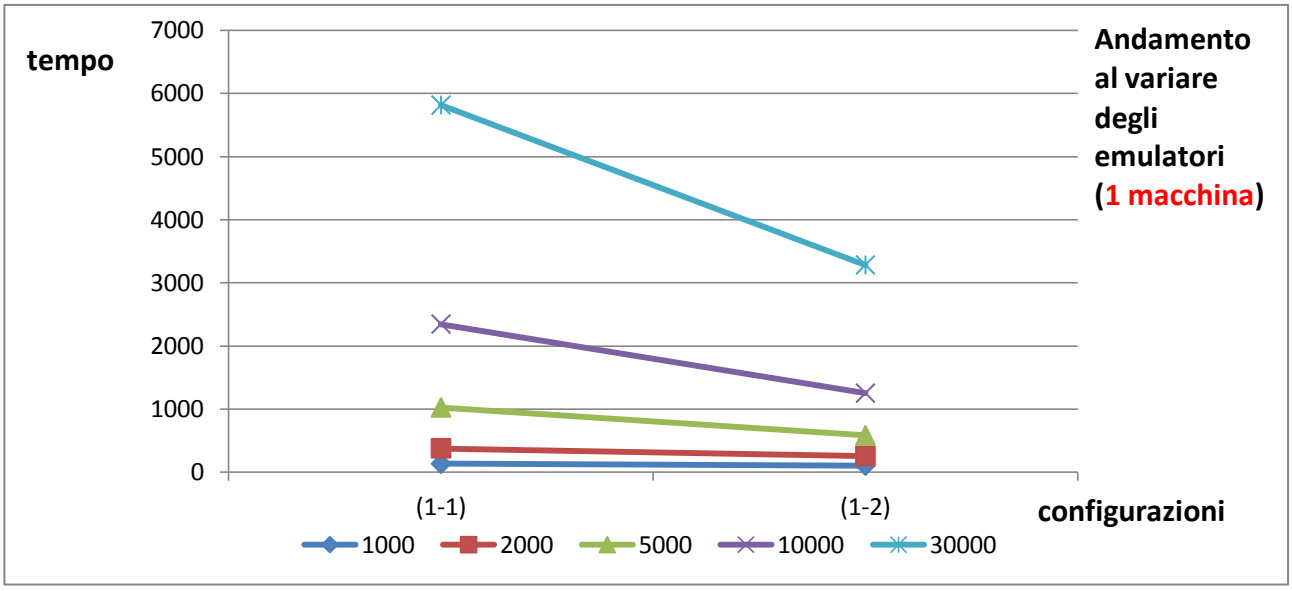



Figura 5.19.1 Tempo al variare degli emulatori Tomdroid

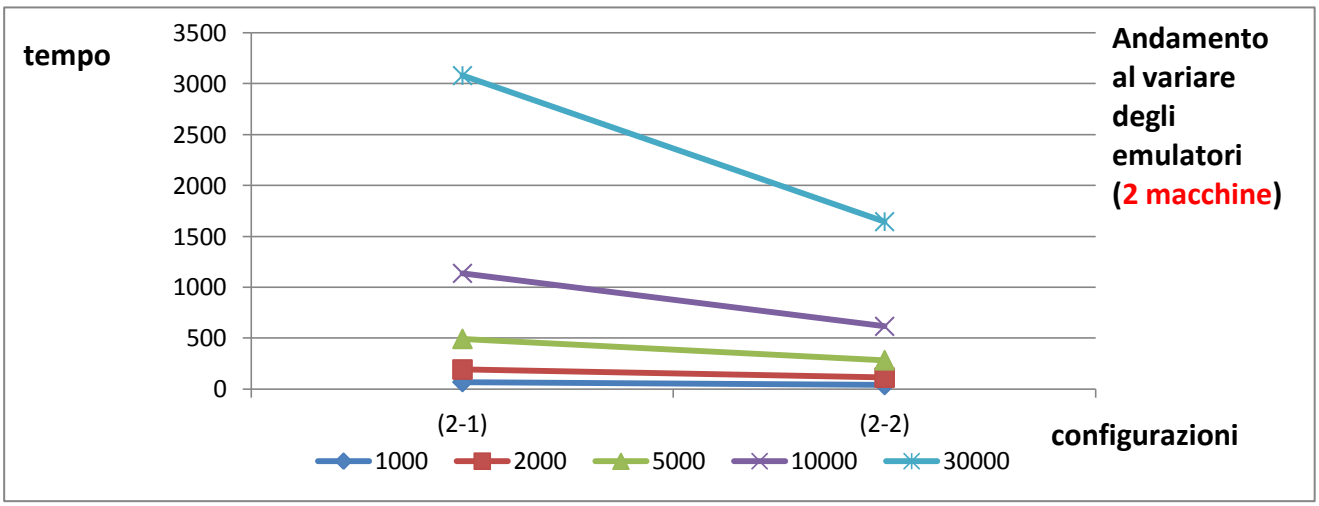



Figura 5.19.2 Tempo al variare degli emulatori Tomdroid

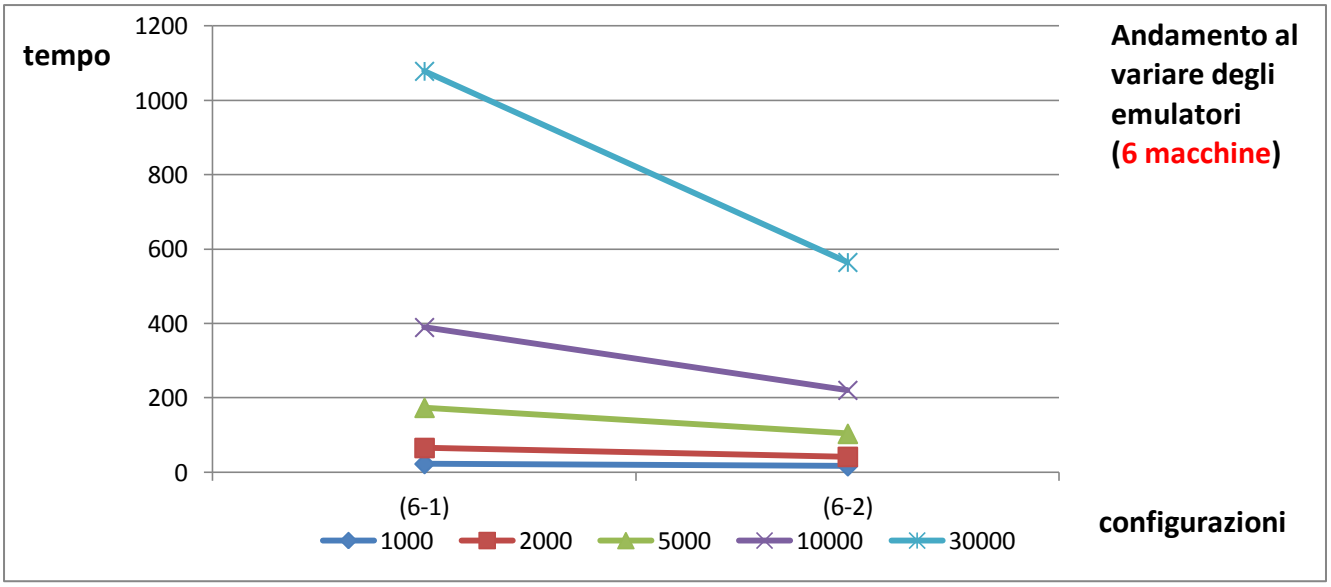



Figura 5.19.3 Tempo al variare degli emulatori Tomdroid

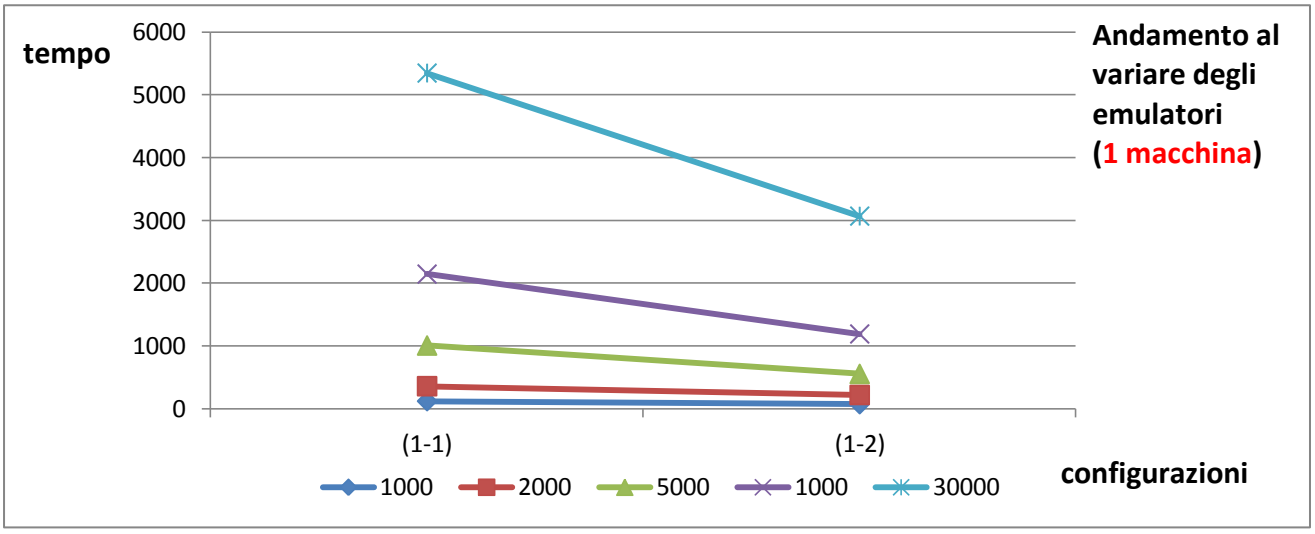



Figura 5.19.4 Tempo al variare degli emulatori TippyTipper

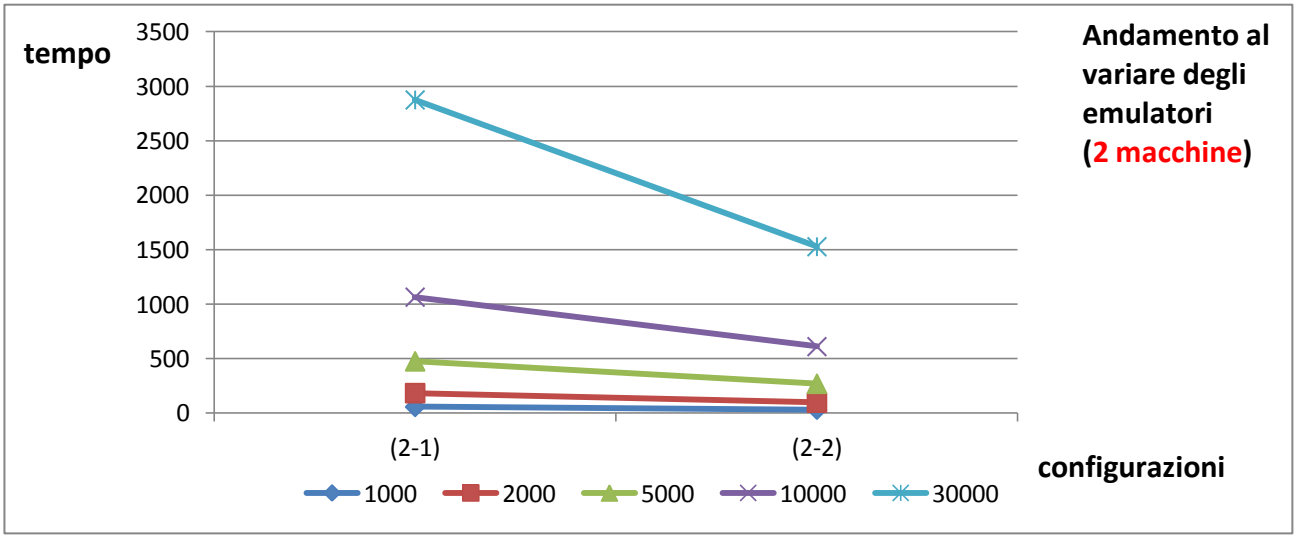



Figura 5.19.5 Tempo al variare degli emulatori TippyTipper

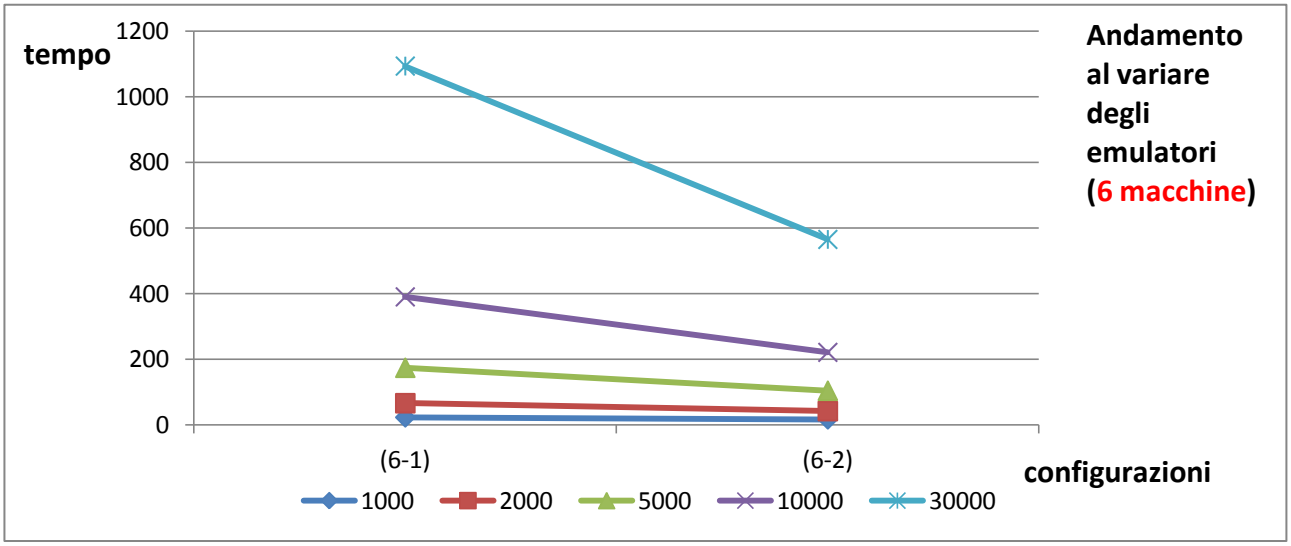



Figura 5.19.6 Tempo al variare degli emulatori TippyTipper

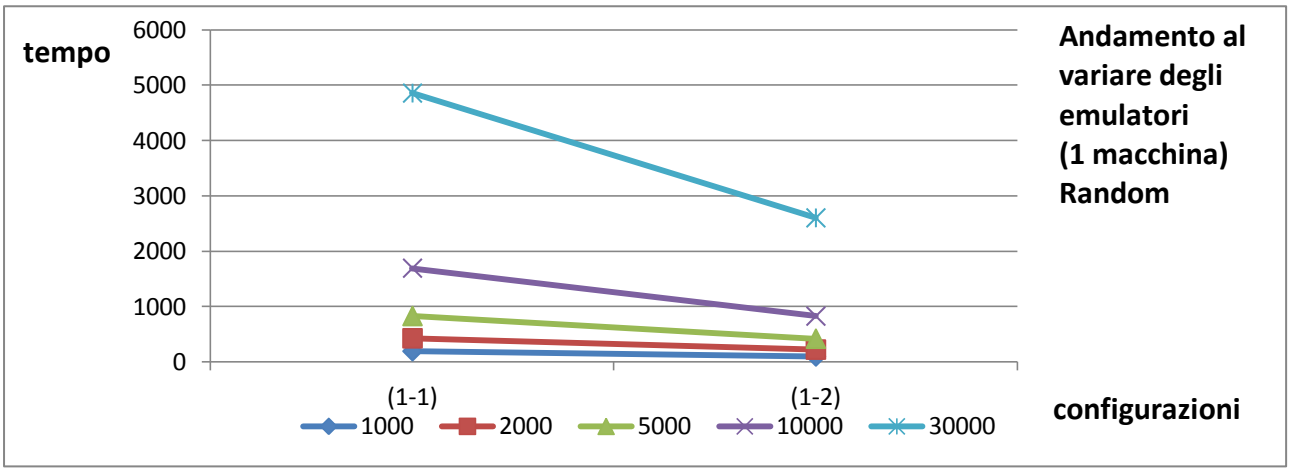



Figura 5.19.7 Tempo al variare degli emulatori TitTacToe

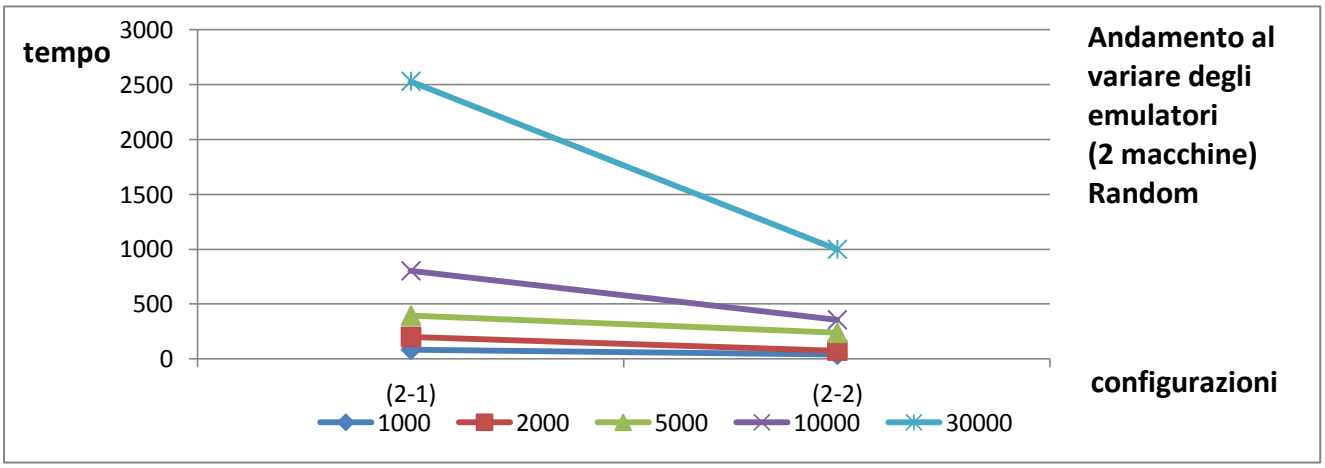



Figura 5.19.8 Tempo al variare degli emulatori TitTacToe

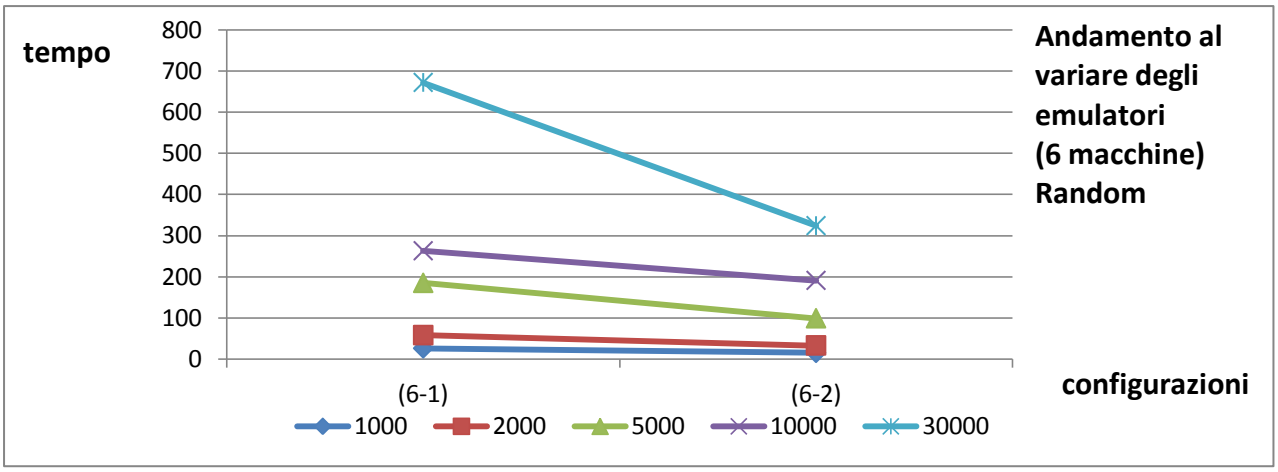



Figura 5.19.9 Tempo al variare degli emulatori TicTacToe

Vediamo cosa succede nel caso Sistematico.

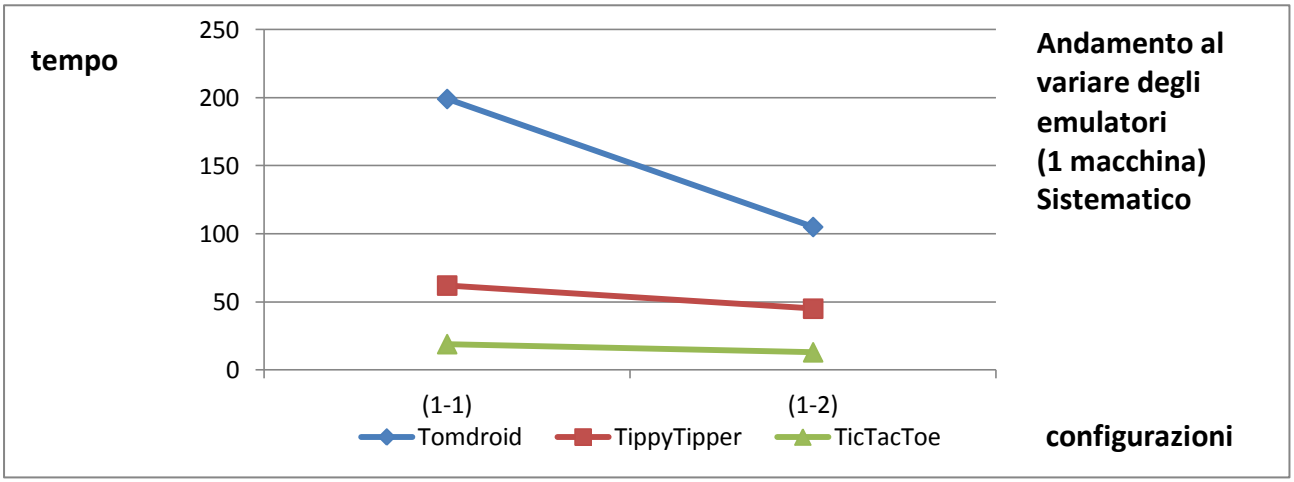



Figura 5.19.10 Tempo al variare degli emulatori Tomdroid

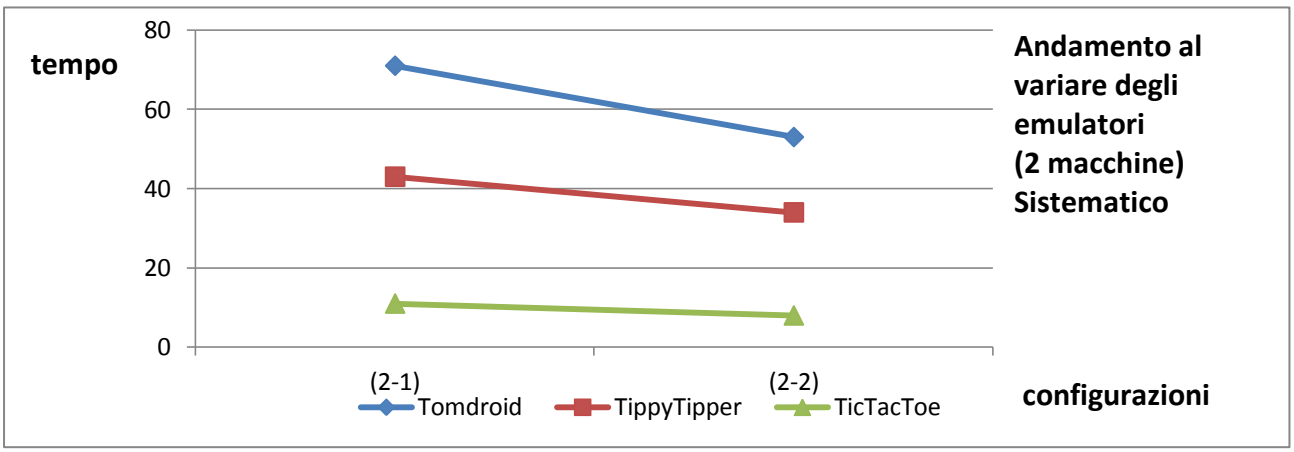



Figura 5. 19.11 Tempo al variare degli emulatori TippyTipper

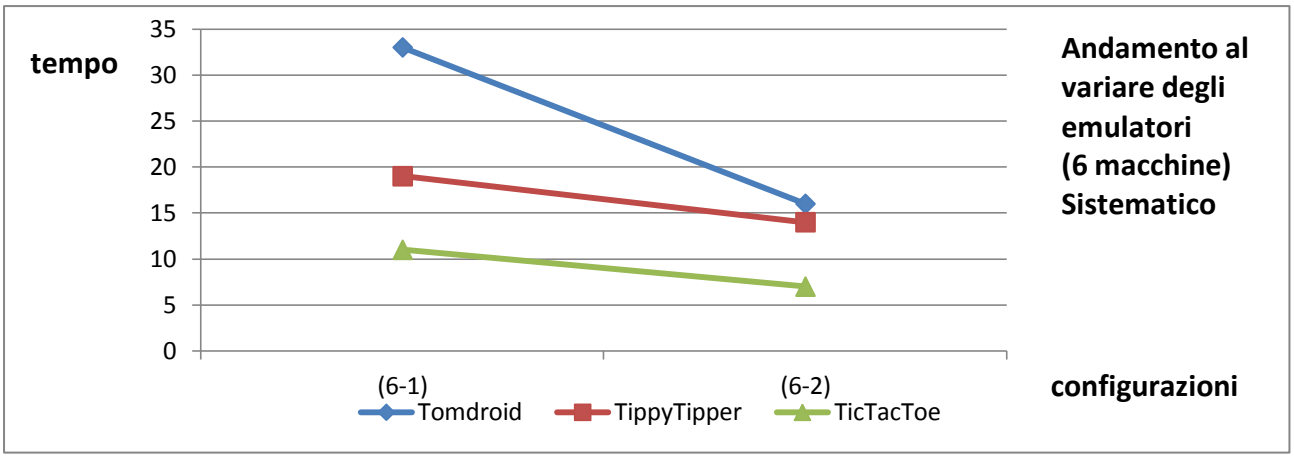



Figura 5.19.12 Tempo al variare degli emulatori TicTacToe

Anche in questo caso possiamo affermare che il sistema parallelo, consente una significativa riduzione dei tempi di esecuzione.

Ora a partire dalle tabelle di Figura 5.17.1 e 5.17.2. vediamo qual' è l'andamento del parametro **k** in questo caso però, al variare degli emulatori. Iniziamo dal caso Random.

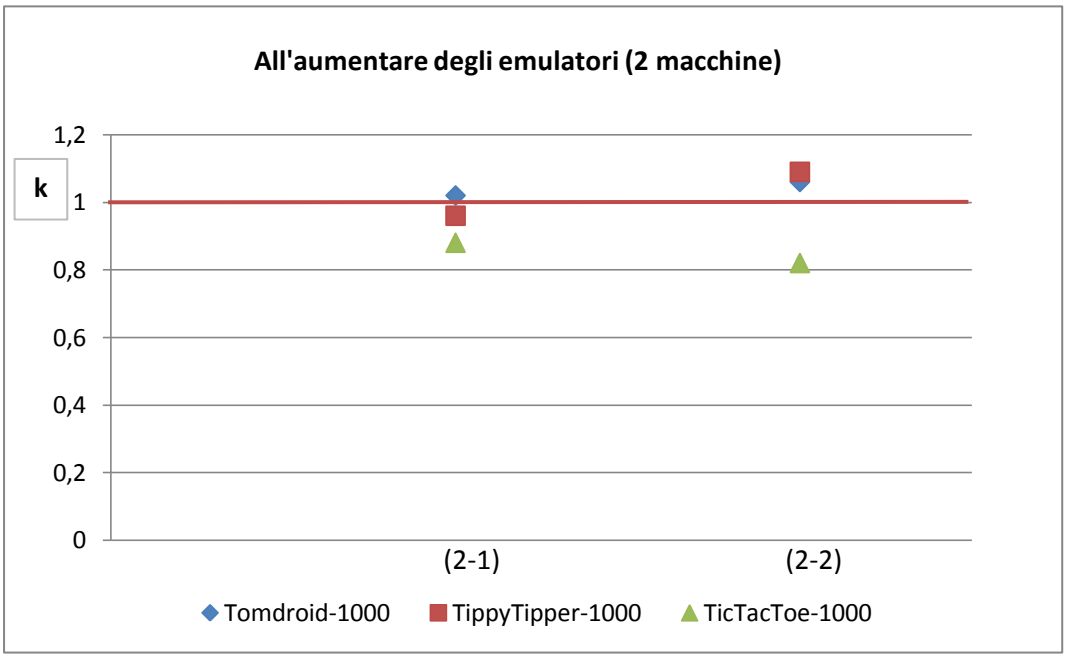



Figura 5.19.13 k al variare degli emulatori (1000 eventi )

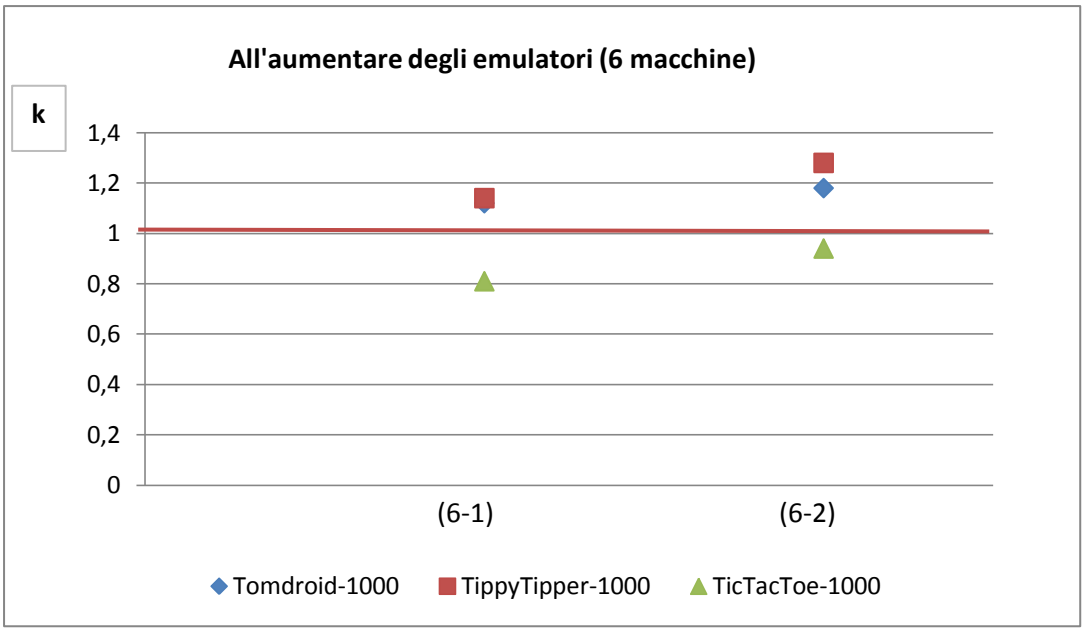



Figura 5.19.14 k al variare degli emulatori (1000 eventi)

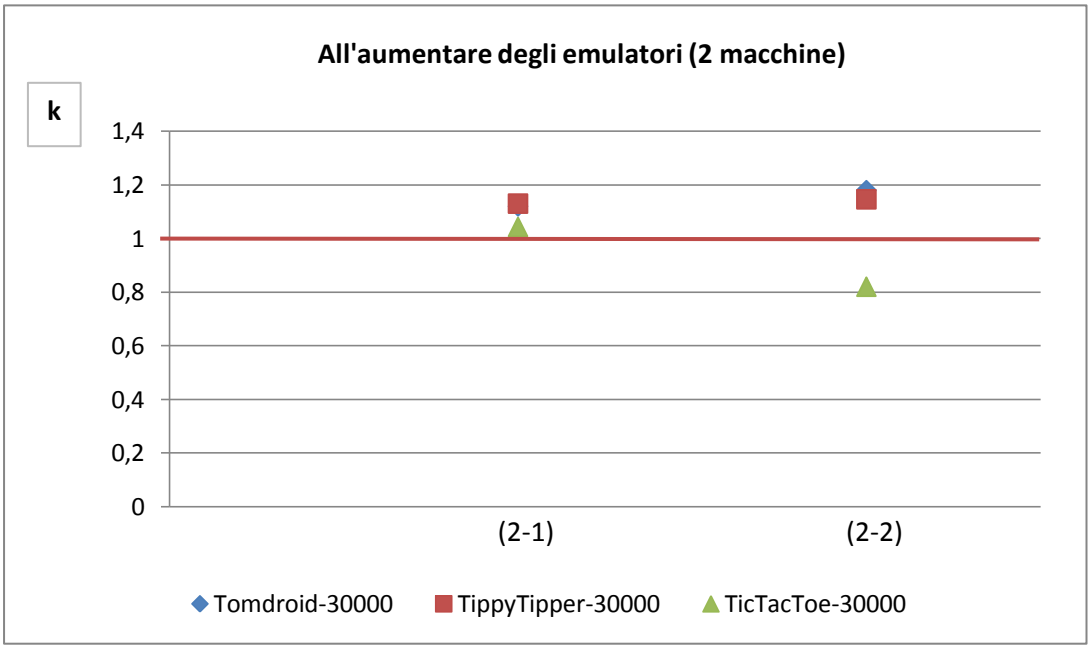



Figura 5.19.15 k al variare degli emulatori (3000 eventi)

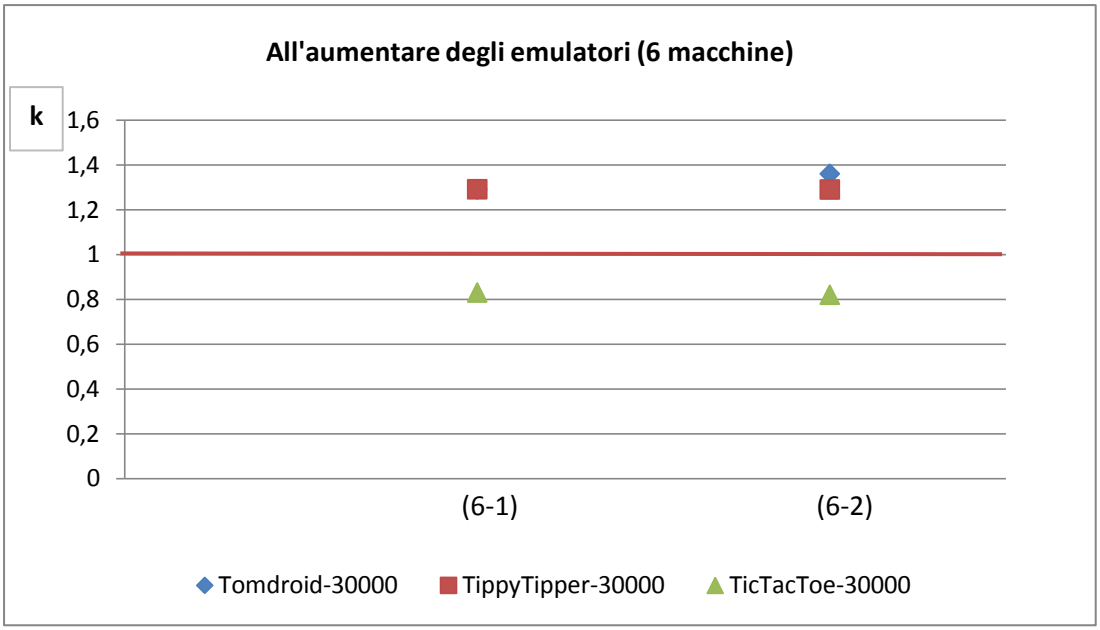



Figura 5.19.15 k al variare degli emulatori (3000 eventi)

Vediamo anche il caso Sistematico.

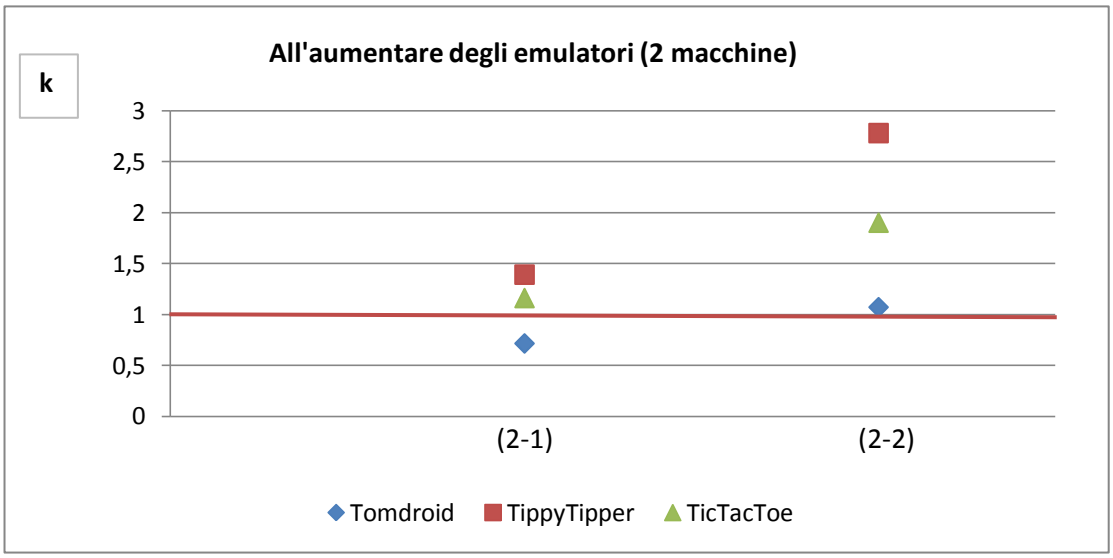



Figura 5.19.16 k al variare degli emulatori

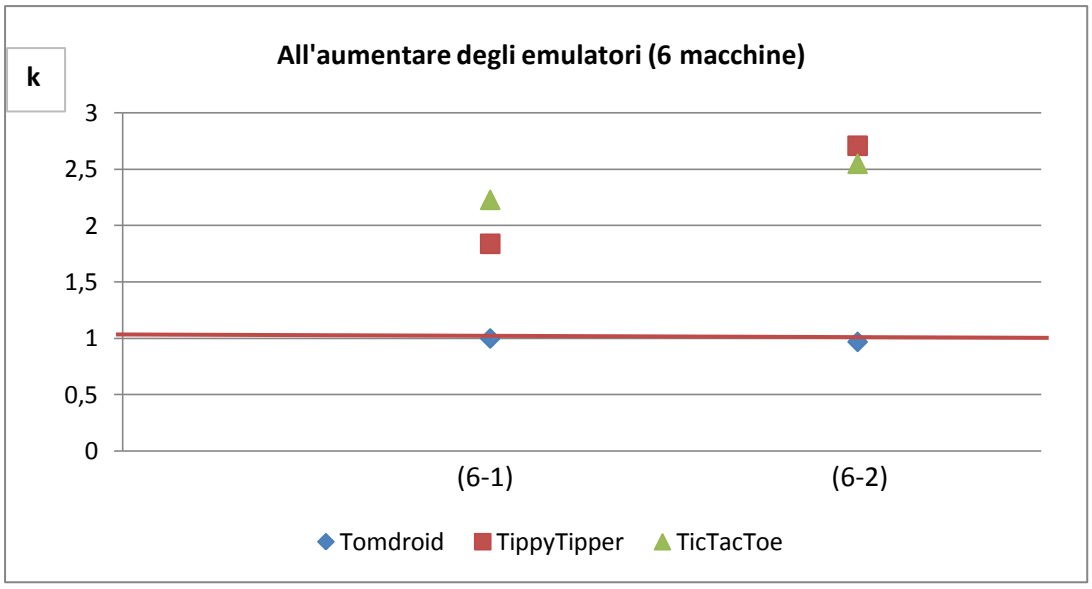



Figura 5.19.16 k al variare degli emulatori

Anche in questo caso (valutazione di k al variare degli emulatori sia con tecnica random che sistematica) non possiamo affermare né che la relazione **b)** sia lineare né possiamo escluderlo.

Possiamo però fare una osservazione. I valori di **k** si discostano maggiormente dal **k** ideale quando andiamo a considerare il doppio degli emulatori per macchina. Ciò a conferma di quanto abbiamo già osservato quando abbiamo provato a dare una risposta alla domanda **RQ 2.2.1).**

Le prove sperimentali sono state realizzate solo con un numero di macchine per noi significativo (1- 2 - 6) in quanto ci interessava valutare il guadagno che riuscivamo ad ottenere con il nostro sistema parallelo. Il numero massimo di emulatori per macchina (2 emulatori), è stato dettato da vincoli tecnologici, cioè dalle risorse che richiede ogni emulatore. Di seguito vediamo quali sono stati i guadagni in percentuale ottenuti nel passaggio da una configurazione all'altra.

| Tomdroid 2 emulatori | 1-2;2-2 | 2-2;6-2 | 1-2;6-2 | eventi |
|---|---|---|---|---|
| guadagno % | 62% | 67% | 87% | 1000 |
| | 53% | 64% | 83% | 2000 |
| | 48% | 63% | 81% | 5000 |
| | 50% | 61% | 81% | 10000 |
| | 51% | 62% | 81% | 30000 |

| Tomdroid 1 emulatore | 1-1;2-1 | 2-1;6-1 | 1-1;6-1 | eventi |
|---|---|---|---|---|
| guadagno % | 48% | 63% | 81% | 1000 |
| | 49% | 62% | 81% | 2000 |
| | 52% | 62% | 82% | 5000 |
| | 51% | 62% | 81% | 10000 |
| | 46% | 62% | 79% | 30000 |

Figura 5.19 guadagno %

| TippyTipper 1 emulatore | 1-1;2-1 | 2-1;6-1 | 1-1;6-1 | eventi |
|---|---|---|---|---|
| guadagno % | 52% | 60% | 81% | 1000 |
| | 48% | 66% | 82% | 2000 |
| | 53% | 63% | 83% | 5000 |
| | 49% | 63% | 81% | 10000 |
| | 45% | 62% | 80% | 30000 |

| TippyTipper 2 emulatori | 1-2;2-2 | 2-2;6-2 | 1-2;6-2 | eventi |
|---|---|---|---|---|
| guadagno % | 55% | 49% | 77% | 1000 |
| | 57% | 61% | 83% | 2000 |
| | 50% | 64% | 82% | 5000 |
| | 49% | 65% | 82% | 10000 |
| | 52% | 63% | 82% | 30000 |

Figura 5.20 guadagno %

| TicTacToe 1 emulatore | 1-1;2-1 | 2-1;6-1 | 1-1;6-1 | eventi |
|---|---|---|---|---|
| guadagno % | 56% | 70% | 86% | 1000 |
| | 53% | 71% | 86% | 2000 |
| | 52% | 68% | 85% | 5000 |
| | 53% | 67% | 84% | 10000 |
| | 48% | 73% | 86% | 30000 |

| TicTacToe 2 emulatori | 1-2;2-2 | 2-2;6-2 | 1-2;6-2 | eventi |
|---|---|---|---|---|
| guadagno % | 60% | 62% | 85% | 1000 |
| | 66% | 57% | 85% | 2000 |
| | 57% | 50% | 79% | 5000 |
| | 57% | 63% | 84% | 10000 |
| | 62% | 68% | 88% | 30000 |

Figura 5.20 guadagno %

| Applicazione | 1-1;2-1 | 2-1;6-1 | 1-1;6-1 |
|---|---|---|---|
| Tomdroid | 64% | 54% | 83% |
| TippyTipper | 31% | 54% | 70% |
| TicTacToe | 43% | 36% | 63% |
| | 1-2;2-2 | 2-2;6-2 | 1-2;6-2 |
| Tomdroid | 50% | 70% | 85% |
| TippyTipper | 25% | 60% | 70% |
| TicTacToe | 31% | 56% | 70% |

Figura 5.21 guadagno % caso Sistematico

**Random**

| | Tomdroid | | | TippyTipper | | | TicTacToe | | |
|---|---|---|---|---|---|---|---|---|---|
| eventi | (1-1) | (1-2) | guadagno | (1-1) | (1-2) | guadagno | (1-1) | (1-2) | guadagno |
| 1000 | 133 | 103 | 23% | 121 | 73 | 40% | 192 | 97 | 49% |
| 2000 | 244 | 154 | 37% | 238 | 146 | 39% | 425 | 221 | 48% |
| 5000 | 647 | 326 | 49% | 648 | 336 | 48% | 827 | 417 | 50% |
| 10000 | 1322 | 668 | 49% | 1138 | 636 | 44% | 1692 | 825 | 51% |
| 30000 | 3470 | 2032 | 41% | 3202 | 1877 | 41% | 4857 | 2602 | 46% |

Figura 5.22 guadagno tra (1,1) ed (1,2) caso Random

**Sistematico**

| Tomdroid | | | TippyTipper | | | TicTacToe | | |
|---|---|---|---|---|---|---|---|---|
| (1-1) | (1-2) | guadagno | (1-1) | (1-2) | guadagno | (1-1) | (1-2) | guadagno |
| 199 | 105 | 47% | 62 | 45 | 28% | 19 | 13 | 32% |

Figura 5.23 guadagno tra (1,1) ed (1,2) caso Sistematico

Dalle Figure 5.19, 5.20, 5.21, 5.22 e 5.23 confermiamo il guadagno temporale ipotizzato.

## 5.5 Threats to validity

In questo paragrafo facciamo alcune considerazioni che ci portano ad affermare che i risultati ottenuti non possono essere considerati come certi ma solo come congetture.

La prima considerazione è che nel caso della tecnica randomica abbiamo utilizzato un solo seed di casualità. Per ottenere dei risultati più attendibili sarebbe stato necessario eseguire più test con seed diversi. Nel caso della tecnica Sistematica dobbiamo fare una precisazione. Alcune prove prelimari dimostrano una dipendenza dal tempo (in termini di *race condition*) mostrando una differenza di coperture ottenute e tempi di esecuzione. Quindi anche in questo caso abbiamo la necessità di effettuare altre prove per poter essere di attendibili.

La seconda considerazione è legata al numero di macchine utilizzate. Le configurazioni sperimentali utilizzate prevedono l'utilizzo di 1, 2 e 6 macchine. Per ottenere dei risultati maggiormente attendibili avremmo dovuto utilizzare anche 3,4,5 macchine. Sarebbe opportuno effettuare anche test con un numero di macchine superiore a 6 in quanto abbiamo osservato un miglioramento dei tempi utilizzando fino a 6 macchine. Ciò non vuol dire che con un numero superiore di macchine non si ottengano risultati ancora migliori.

La terza considerazione è legata questa volta al numero di emulatori per macchina

(max 2). Questo vincolo è un vincolo tecnologico in quanto, avere sulle nostre macchine un numero superiore di emulatori, avrebbe causato una riduzione delle performance dei sistema. Ricordiamo che ogni emulatore Android richiede delle risorse che vanno inevitabilmente ad appesantire la macchina su cui questi sono in esecuzione.

La quarta considerazione riguarda la diversità delle macchine utilizzate per la sperimentazione. Avere macchine con diversa capacità hardware fa si che la macchine più potente lavori di più di quella meno potente in quanto il sistema bilancia il carico di lavoro tra le macchine. Per avere risultati più attendibili avremmo dovuto utilizzare macchine con uguale configurazione hardware.

Un'altra considerazione è legata alle applicazioni. Nella sperimentazione abbiamo utilizzato solo tre applicazioni, questo fatto può minare la natura delle congetture fatte. Andrebbero eseguiti altri test con un numero elevato di applicazioni per poter quindi considerare come affermazioni le nostre congetture.

# Conclusioni

Con questo lavoro di tesi volevamo dimostrare i vantaggi della nostra architettura parallela per la generazione automatica di casi di test per applicazioni android. Vediamo i risultati. Il risultato più importante che abbiamo dimostrato è la riduzione dei tempi di esecuzione.

Detto ciò , volevamo capire se la copertura totale fosse funzione del parallelismo introdotto. Dai risultati sperimentali possiamo dire che la copertura non è funzione del parallelismo ed inoltre il tempo di esecuzione, a parità di numero di emulatori, non è funzione del parallelismo ( 1-2 è diverso da 2-1).

Inoltre, volevamo arrivare alla conclusione ideale che all'aumentare delle risorse (macchine ed emulatori) il tempo si riducesse con un andamento lineare. Dalle prove fatte, osservando anche gli andamenti del parametro k, per ora possiamo solo ipotizzare che il legame tra tempo e macchine/emulatori non sia di tipo lineare. Per averne la certezza andrebbero effettuati ulteriori test che tengano conto di un numero maggiore di macchine, emulatori ed applicazioni, e di un seed diverso nel caso randomico.

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

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

[5] 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.

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

[7] 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.

[8] 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.

# Ringraziamenti

*"Quando t'accorgi che stai guardando lontano, guarda ancora più lontano"*

**Robert Baden-Powell**

Devo in primo luogo ringraziare me stesso e la mia perseveranza. Ero arrivato lontano con la laurea triennale ed ho guardato ancora più lontano… fino ad arrivare ad oggi.

Finalmente termino il mio cammino universitario, non senza qualche rammarico, ma con la consapevolezza di aver raggiunto un grande traguardo.

In questo cammino mi è stata molto vicina tutta la famiglia, compresi zii e parenti vari, ansiosi di vedermi realizzato.

Mi è stata vicina la mia futura moglie, che come me ha raggiunto un grande obiettivo.

Ringrazio il Professore Porfirio Tramontana e l'Ing. Nicola Amatucci per la loro disponibilità.

E ringrazio anche Giuseppe, un grande compagno di studi, senza il quale forse oggi non sarei ancora laureato.

A tutti grazie di vero cuore.