

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

Attività progettuale di Sistemi Mobili M

Attività progettuale di Sistemi Distribuiti M

Applicazione per la catalogazione geolocalizzata dei segnali telefonici su piattaforma Android

Progetto di:

Davide Montanari

Relatore:

Prof. Ing. Paolo Bellavista

Anno Accademico 2010/2011

Indice generale

Introduzione.....	1
1. Android.....	3
1.1 Componenti di un'applicazione Android.....	4
1.1.1 Activity.....	5
1.1.2 Service.....	7
1.1.3 Content Provider.....	8
1.1.4 Broadcast Receiver.....	8
1.2 Sensori hardware nella piattaforma Android.....	9
1.2.1 Sistema di coordinate in SensorEvent API.....	10
1.2.2 Sensore di accelerazione.....	10
1.2.3 Sensore di campo magnetico.....	11
1.2.4 Giroscopio.....	11
1.2.5 Sensore di luminosità.....	12
1.2.6 Sensore di prossimità.....	12
1.2.7 Sensore vettore di rotazione.....	12
1.2.8 Sensore di orientamento.....	13
2. Progetto su piattaforma Android.....	14
2.1 Casi d'uso.....	14
2.1.1 Primo caso d'uso: dispositivo con sufficienti risorse hardware.....	14
2.1.2 Secondo caso d'uso: dispositivo con limitate risorse hardware.....	15
2.2 Implementazione.....	16
2.2.1 Acquisizione delle informazioni sul segnale telefonico.....	17
2.2.2 Acquisizione delle informazioni di locazione.....	22
2.2.3 Acquisizione delle informazioni di luminosità.....	24
2.2.4 Generazione del report.....	26
2.2.5 Trasferimento del report tramite Bluetooth.....	30
3. Valutazione delle performance.....	35
3.1 Configurazione e modalità di test.....	35
3.2 Acquisizione della potenza del segnale telefonico.....	36
3.3 Acquisizione delle informazioni di luminosità ambientale.....	37
3.4 Acquisizione delle informazioni di locazione.....	39
3.4.1 Uso della rete telefonica cellulare.....	39
3.4.2 Uso del GPS.....	40
3.5 Trasferimento di report singolo tramite connessione Bluetooth.....	42
3.5.1 Uso del client Bluetooth.....	42
3.5.2 Uso del server Bluetooth.....	44
3.6 Trasferimento di report multiplo tramite connessione Bluetooth.....	45
3.6.1 Uso del client Bluetooth.....	46
3.6.2 Uso del server Bluetooth.....	47
4. Gestione ed elaborazione dei report lato server.....	50
4.1 J2EE Container: JBoss.....	50
4.2 Enterprise JavaBeans.....	51
4.2.1 Tipi di Enterprise JavaBean.....	52
4.2.1.1 Session Bean.....	52
4.2.1.1.1 Stateful Session Bean.....	52
4.2.1.1.2 Stateless Session Bean.....	52
4.2.1.1.3 Singleton Session Bean.....	52
4.2.1.2 Message Driven Bean.....	53
4.2.1.3 Entity Bean.....	53
4.3 RESTful Web Services.....	53

4.3.1 RESTful design vs SOAP-based design.....	55
5. Progetto enterprise su JBoss AS.....	56
5.1 Caso d'uso.....	57
5.2 Implementazione.....	57
5.2.1 Upload del report lato server.....	60
5.2.2 Decompressione dell'archivio per report multipli.....	62
5.2.3 Salvataggio del report su database e gestione della persistenza.....	63
5.2.4 Estrazione dei dati dal supporto di persistenza.....	67
5.2.5 Localizzazione tramite Google Maps Web Services RESTful API.....	69
5.2.6 Interfaccia utente via Web.....	72
6. Valutazione delle performance lato server.....	75
6.1 Configurazione e modalità di test.....	75
6.2 Carico reale in localhost.....	76
6.3 Carico di picco in localhost.....	78
6.4 Carico reale in rete locale.....	80
6.5 Carico di picco in rete locale.....	83

Introduzione

In questi ultimi anni il panorama dei sistemi operativi (o stack software) per sistemi mobili sta accelerando la propria crescita ed evoluzione grazie alla esponenziale diffusione di dispositivi PDA quali smartphone e tablet. A causa delle differenze che li collocano in un piano diverso rispetto ai tradizionali Personal Computer portatili (notebook e netbook) e la sempre crescente potenza computazionale e dotazione hardware/software, la progettazione di tali sistemi ha portato ad una particolare sensibilizzazione riguardo alcune tematiche fondamentali: il consumo energetico, la connettività permanente, la sicurezza e la gestione del contesto (facente uso dei sistemi di posizionamento e di localizzazione). In particolare questi dispositivi possono avere “coscienza” di tutto quello che hanno attorno e quindi possono essere un ottimo strumento per valutare alcune situazioni, dipendentemente dal contesto in cui operano. L'unione di questo approccio insieme alla ricca dotazione di sensori installata all'interno dei PDA e smartphone di ultima generazione danno la possibilità agli sviluppatori di poter creare applicazioni ricche sia dal punto di vista dell'innovazione che dal punto di vista dell'utilità e dell'esperienza utente. La conseguenza è la nascita di nuovi paradigmi di utilizzo del proprio device per l'utente finale: l'utilizzo prevalente come aggregatore sociale, come console per videogiochi, come aggregatore di notizie dal mondo, come dispositivo per attuare collaborazione sociale, ecc. In tutti questi contesti sono molto importanti i contributi dati dai vari moduli presenti: la rete mobile, Wi-Fi e il GPS per la geolocalizzazione, i sensori di orientamento e giroscopio per aumentare l'efficacia dell'esperienza utente, il sensore di prossimità e di luminosità per diminuire il consumo energetico del dispositivo, l'antenna GSM per le comunicazioni vocali, il sensore di campo magnetico per gestire l'orientamento, il modulo Bluetooth per le comunicazioni tra dispositivi a corto raggio, ecc. Il lavoro presentato in questo documento unisce tutte le principali caratteristiche di un device moderno per sfruttarle all'interno di un contesto di collaborazione sociale: la catalogazione geolocalizzata del segnale telefonico suddiviso per operatore e per tecnologia radio. In questo modo, utilizzando la piattaforma Android per lo sviluppo, è stata creata un'applicazione capace di estrarre le informazioni dal contesto di acquisizione (luminosità, potenza e tipo di segnale mobile, locazione) e di inviare tali dati, tramite Bluetooth, ad un altro dispositivo con maggiori capacità computazionali e che possiede la medesima applicazione (in caso che il device acquisente, ad esempio, non possieda un contratto per la rete mobile o non possieda il modulo dedicato) oppure di inviarli direttamente ad un server remoto dedicato all'elaborazione, all'aggregazione e alla visualizzazione dei report inviati dai singoli utilizzatori. Il seguente documento guida progressivamente l'utente nella comprensione del lavoro svolto. In particolare il primo capitolo mette in luce alcune caratteristiche dello stack Android, descrivendo alcuni componenti fondamentali necessari per lo sviluppo. Successivamente vengono descritti alcuni dei sensori supportati nativamente dalle librerie Android e presenti nella quasi totalità di dispositivi oggi in commercio: sensore di accelerazione, campo magnetico, giroscopio, luminosità, prossimità, vettore di rotazione e orientamento. Il secondo capitolo si concentra maggiormente sulla progettazione dell'applicazione. Vengono mostrati alcuni scenari d'uso reali (tramite l'uso di dispositivi con sufficienti risorse hardware e dispositivi con limitate risorse hardware) e descritti i dettagli di implementazione assieme al relativo codice utilizzato nello sviluppo. Il terzo capitolo invece si focalizza sull'acquisizione delle prestazioni in termini di consumo di CPU e di memoria RAM. Questi due indici sono molto importanti per poi valutare l'impatto che avrà l'applicazione dal punto di vista energetico. Quindi si sono analizzati i singoli casi: acquisizione del segnale telefonico, della luminosità ambientale, della locazione e dell'utilizzo della tecnologia Bluetooth per l'invio dei report a dispositivi connessi all'interno della stessa Personal Area Network.

Il quarto capitolo invece introduce il progetto lato server. Qui vengono illustrate brevemente le tecnologie enterprise utilizzate per la progettazione e la realizzazione dell'applicazione J2EE incaricata a gestire i report. Il quinto capitolo mostra tutta la fase implementativa, definendo il caso d'uso e le modalità di utilizzo per l'utente finale. Quindi viene mostrata ogni singola funzionalità che l'applicazione offre, presentando il codice relativo e motivando ogni scelta presa. Il sesto e ultimo capitolo invece si occupa di test sulle performance dell'applicazione Web/enterprise. Simulando un numero variabile di client in uno scenario locale e remoto all'interno di configurazioni che riproducono casi reali e casi estremi di utilizzo, sono stati ricavati dati importanti per identificare i limiti dell'architettura evidenziando, per ogni test, i valori di latenza per il campionamento delle richieste e i valori relativi allo throughput: dati rilevanti in caso di tuning dell'applicazione.

1. Android

Android è uno stack software open-source per dispositivi mobili che comprende un sistema operativo, un middleware ed alcune applicazioni chiave. Dal 2005 lo sviluppo è a carico di Google Inc., in collaborazione con alcuni membri della Open Handset Alliance. La sua natura open-source permette di effettuare uno studio approfondito e quindi capirne i punti di forza e di debolezza. Lo stack è composto da applicazioni Java in esecuzione all'interno di un framework Java-based orientato agli oggetti, al di sopra delle librerie di sistema in esecuzione all'interno della Dalvik Virtual Machine (una Java Virtual Machine modificata), caratterizzata da una compilazione di tipo JIT che traduce il bytecode nel codice macchina nativo in fase di runtime. Le librerie scritte in C sono implementate per realizzare i seguenti componenti di sistema: surface manager, OpenCore media framework, database relazionale SQLite, OpenGL ES 2.0 3D graphics API, WebKit layout engine, SGL graphics engine, SSL e Bionic libc. Di seguito viene mostrato lo stack architetturale dalla parte più device-dependent fino ad arrivare allo strato più astratto costituito dalle applicazioni:

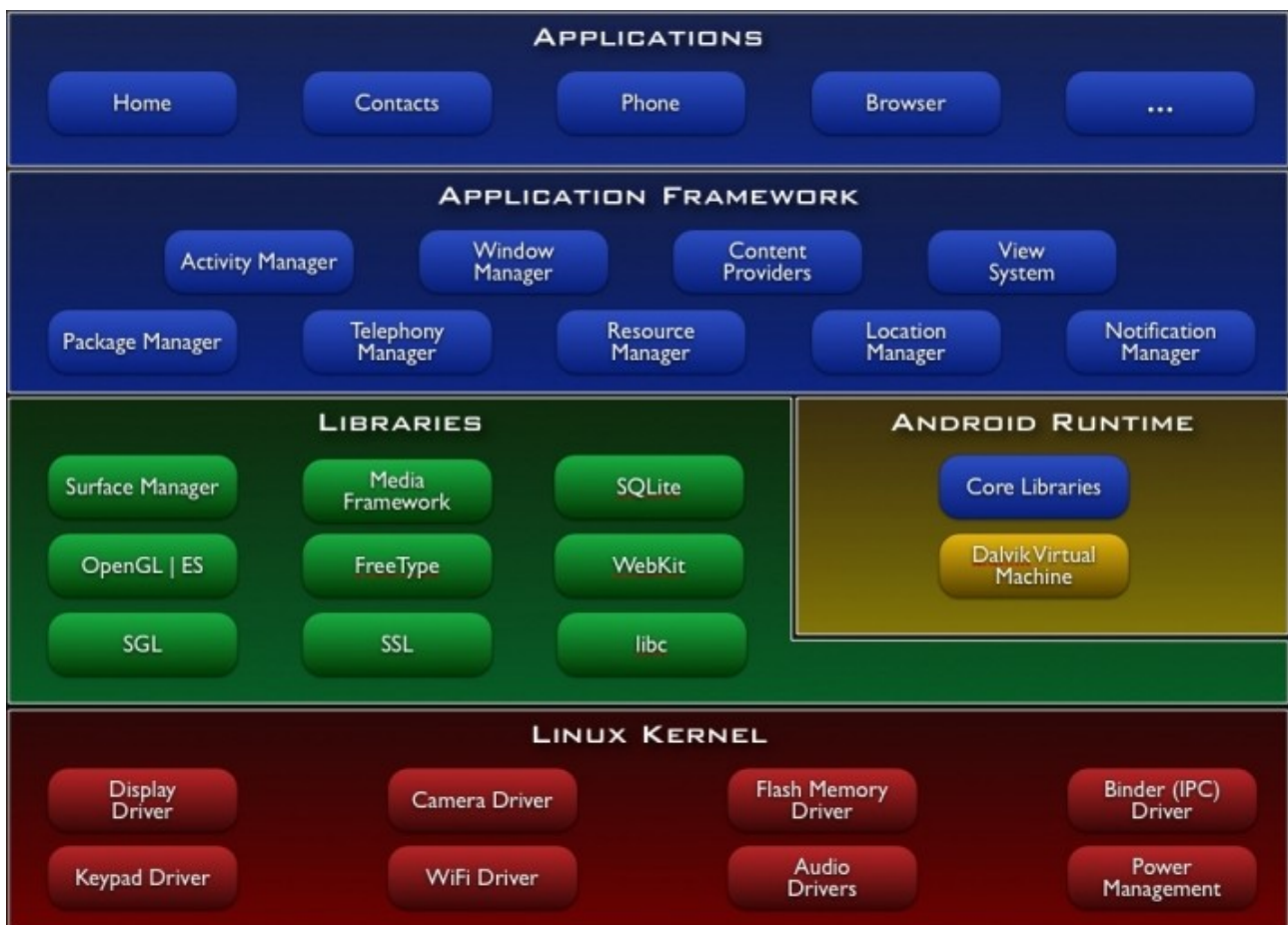


Illustrazione 1: Stack architetturale della piattaforma Android

Tutte le funzionalità di power management e i driver per la visualizzazione, riproduzione audio, comunicazione e accesso alle varie periferiche di I/O vengono gestite dal Kernel Linux. La principale piattaforma hardware utilizzata è ARM, dato l'ottimo rapporto potenza/consumo che la vede adatta nell'uso all'interno di sistemi utilizzati principalmente in mobilità.



Illustrazione 2: Un esempio di dispositivo Android - "Nexus S" progettato da Google e Samsung

Per sviluppare un'applicazione Android è necessaria la conoscenza del linguaggio Java per sfruttare le API fornite dall'SDK. In aggiunta è disponibile un NDK (Native Development Kit) che ha lo scopo di fornire al programmatore le API in C per gestire le componenti fondamentali dell'architettura, utili in particolar modo in casi di porzioni di codice critiche per le performance. Tuttavia il codice scritto in modo nativo viene eseguito all'interno di una Virtual Machine, quindi un side-effect non evitabile è comunque l'incremento della complessità di scrittura dell'applicazione. Per quanto riguarda il packaging, tutti i componenti scritti in linguaggio nativo vengono impacchettati all'interno dello stesso file ".apk" (Android Package) che contiene l'applicazione principale.

1.1 Componenti di un'applicazione Android

Di seguito verranno illustrate le componenti fondamentali e principali per lo sviluppo di un'applicazione Android: Activity, Service, Content Provider e Broadcast Receiver. Esse interagiscono per ottenere tutte le risorse necessarie per l'esecuzione. A supporto viene inoltre redatto un file Manifest in XML che contiene la descrizione dei componenti registrati, occupandosi di effettuare altre operazioni, tra le quali:

- Identificare i permessi richiesti dall'applicazione, come l'accesso ad internet o l'accesso in lettura alla rubrica dell'utente.

- Dichiarare la versione minima delle API richieste dall'applicazione.
- Esporre requisiti hardware/software aspettati o richiesti, come la fotocamera, i servizi Bluetooth o uno schermo multitouch.
- L'elenco delle librerie aggiuntive esterne necessarie, come le librerie di Google Maps.
- Dichiarare le caratteristiche funzionali dei componenti.
- Ecc., ecc.

1.1.1 Activity

L'Activity è un componente applicativo che si presenta all'utente come una qualsiasi schermata dove è possibile effettuare una qualche operazione, come chiamare al telefono, scattare una foto, inviare un'e-mail o consultare una mappa. Ogni Activity è posta dentro ad una finestra (che tipicamente copre tutto lo schermo), dove il programmatore può disegnare l'interfaccia utente. Quindi un'applicazione di solito è costituita da diverse Activity debolmente accoppiate tra loro. Tipicamente, all'interno di un'applicazione, è presente sempre un'Activity principale che viene presentata all'utente al primo lancio dell'applicazione. Ogni Activity può poi, a runtime, invocare un'altra in modo tale da poter creare un ecosistema formato da diverse operazioni: l'applicazione. Al lancio di una nuova Activity, quella in esecuzione precedentemente viene fermata e il sistema la salva all'interno di una struttura a pila (il "back stack"); quando una nuova Activity viene messa in esecuzione, ne viene fatto il push all'interno del "back stack" e ne viene dato il relativo focus all'utente. Il "back stack" utilizza una politica LIFO per l'accodamento e, in questa maniera, quando l'utente decide di lasciare l'Activity corrente e preme il tasto "Back" sul terminale, l'Activity corrente viene rimossa dallo stack (e distrutta) e la precedente Activity viene ripristinata. Se un'Activity viene fermata a causa dell'inizio di una nuova, il cambiamento di stato viene notificato attraverso i corrispettivi metodi di callback implementati appositamente. Esiste una larga varietà di metodi di callback che un'Activity può ricevere a causa di un cambiamento interno del proprio stato – quando il sistema la crea, la ferma, la ripristina o la distrugge – e ogni callback esegue un lavoro specifico appropriato in base alla transizione che si verifica. Per esempio, all'atto di interruzione di una Activity, il metodo di callback specifico rilascia tutte le risorse "pesanti", come ad esempio le connessioni di rete a database. Quando essa viene ripristinata, il metodo apposito ri-acquisisce le risorse necessarie e ripristina le azioni che erano state interrotte prima dello stop. Queste transizioni di stato fanno parte del ciclo di vita di ogni Activity.

La figura seguente mostra i vari stati raggiungibili da un'Activity e i metodi di callback invocati per effettuare le transizioni da uno stato ad un altro. Si noti il fatto che, se un'altra applicazione che condivide la stessa macchina virtuale necessitasse di un quantitativo di memoria centrale maggiore di quella disponibile in un determinato momento e tutte le Activity di una data applicazione fossero nello stato di "Pause" o nello stato di "Stop", in automatico il sistema può decidere di uccidere il processo corrente e liberare tutta la memoria utilizzata da esso. Questo tipo di ottimizzazione permette di evitare l'utilizzo di un task manager manuale per la chiusura delle applicazioni, semplificando da una parte il ruolo dell'utente che viene sollevato da tale responsabilità, ma dall'altra parte provocando un'occupazione alta e costante di memoria di sistema e un utilizzo non ottimizzato di CPU per mantenere le applicazioni in background per un tempo deciso in modo non deterministico.

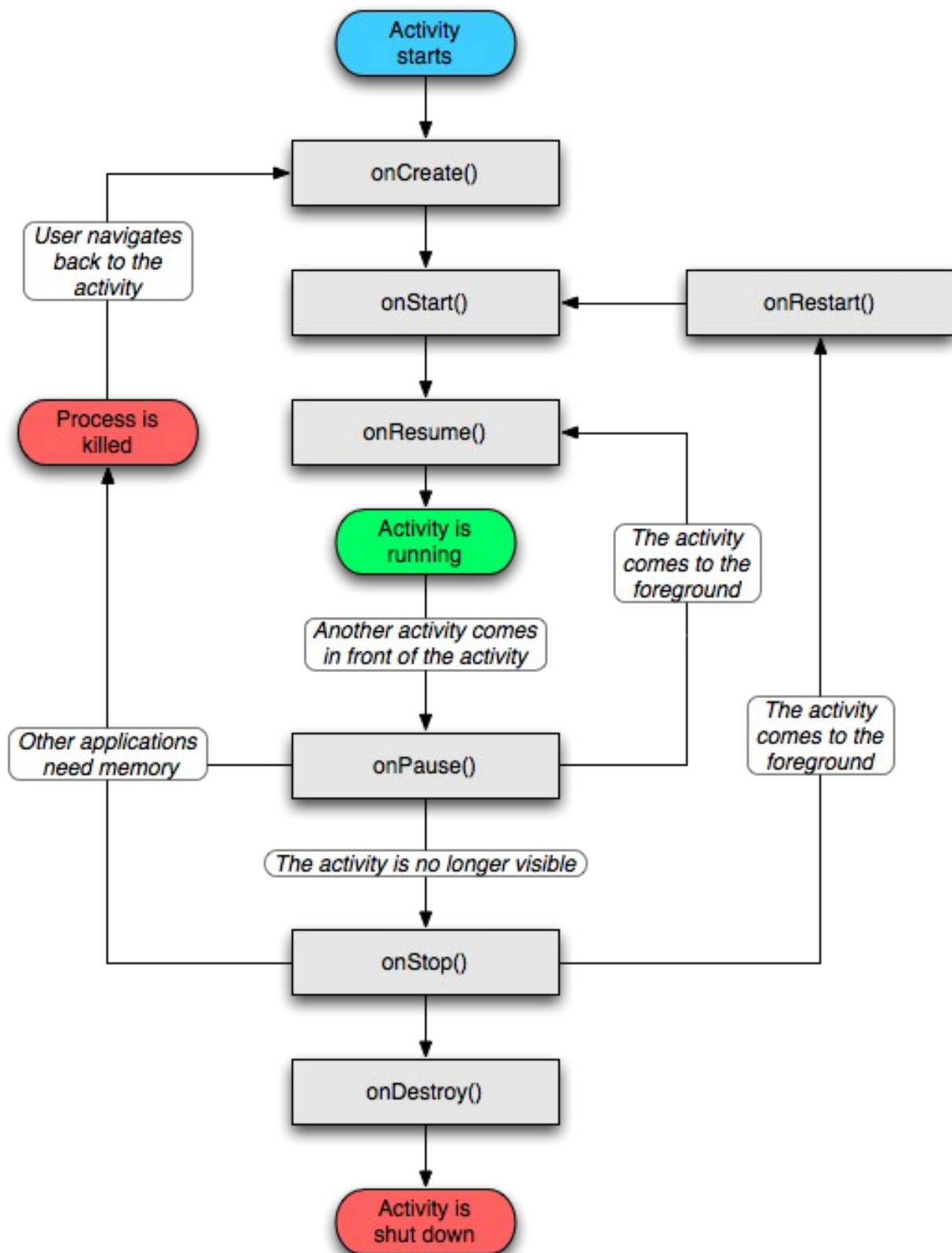


Illustrazione 3: Ciclo di vita di un'Activity su piattaforma Android

1.1.2 Service

Un Service è un componente applicativo atto ad eseguire operazioni in background con un tempo di esecuzione molto lungo, senza fornire alcuna interfaccia utente. Se un altro componente applicativo attiva un Service, quest'ultimo inizia la sua esecuzione in background, continuandola anche se l'utente passasse ad un'altra applicazione. In aggiunta, un componente può collegarsi ad un Service con cui vuole interagire e quindi attuare una comunicazione inter-processo (IPC): per esempio, un Service può fornire funzionalità di gestione di connessioni di rete, riprodurre musica, fare I/O di file su memoria di massa o interagire con un Content Provider, tutto questo in background.

Un Service può essere di due tipi:

- **Started:** Un Service è "started" quando un componente dell'applicazione (come un'Activity) lo fa partire chiamando il metodo "startService()". Una volta partito, un Service può girare in background per un tempo indefinito e può continuare anche se il componente che l'ha messo in esecuzione è stato distrutto. Di solito uno "started Service" esegue una singola operazione e non ritorna alcun valore al chiamante. Per esempio, può scaricare o caricare un file all'interno di una rete: quando l'operazione è conclusa, il Service si ferma in modo autonomo.
- **Bound:** Un Service si dice "bound" quando un componente dell'applicazione si lega ad esso chiamando il metodo "bindService()". Un Service di tipo "bound" offre un'interfaccia client-server che permette ai componenti di interagire con esso, inviare richieste, ottenere risultati, il tutto con la possibilità di farlo attraverso l'utilizzo della comunicazione inter-processo (IPC). Un Service "bound" esegue fino a quando un componente rimane legato ad esso: è possibile legare più componenti in una volta ad un solo Service e solo quando tutti si slegano (unbind), il Service viene distrutto. Esso quindi non esegue in background in modo indefinito. In definitiva, un Service "bound" funge da server in un'interfaccia client-server.

Anche se le due tipologie di Service sono state descritte in modo separato, esse possono entrare in esecuzione in entrambe le modalità – possono essere "started" (per eseguire in un tempo indefinito) e al contempo permettere anche il binding da parte di altri componenti. Il tutto dipende da quale metodo di callback viene implementato all'interno del componente: onStartCommand() permette di avere componenti in modalità "started" mentre onBind() permette il binding.

Senza considerare il modo con cui il Service è stato avviato ("started", "bound" o entrambi), qualsiasi componente dell'applicazione può utilizzarlo (anche da un'applicazione separata) nella stessa maniera in cui un componente può usare un'Activity: inizializzandolo con un Intent. Tuttavia, per ragioni di sicurezza, è possibile dichiarare il Service come privato (nel file Manifest) e bloccare le richieste di accesso da parte di altre applicazioni. A runtime un Service esegue all'interno del thread principale del processo che lo ospita, quindi non crea un proprio thread di esecuzione e non gira in un processo separato (a meno che non sia specificato diversamente). Questo significa che, se viene avviato un Service che implementa delle operazioni con alto carico computazionale per la CPU o con operazioni bloccanti (come la riproduzione di un MP3 o operazioni intensive di rete), è buona norma creare un nuovo thread all'interno del Service atto a compiere queste operazioni: usando un thread separato si riducono così i rischi di "Application Not Responding (ANR)" e il thread principale dell'applicazione può così rimanere dedicato alle operazioni di interazione tra l'utente e le varie Activity.

1.1.3 Content Provider

I Content Provider memorizzano e recuperano dati dalla memoria e li rendono accessibili a tutte le applicazioni, eventualmente condividendoli: non esiste un'area comune di storage alla quale tutti i package Android possono avere accesso.

Android viene rilasciato con una varietà di Content Provider per tipi di dati comuni (audio, video, immagini, informazioni di contatto personali, etc.) e una lista parziale è presente all'interno del package "android.provider".

Per rendere pubblici i propri dati ci sono due possibilità: creare un Content Provider personalizzato (sottoclasse di ContentProvider) oppure aggiungere i dati ad un Content Provider esistente (se ne esiste uno che riesce a controllare gli stessi dati di riferimento e se si possiedono i permessi di scrittura).

1.1.4 Broadcast Receiver

Il Broadcast Receiver è un componente sensibile agli eventi di broadcast lanciati dal sistema. Il sistema genera molti messaggi di broadcast (per esempio un messaggio che annuncia che lo schermo è stato spento, che la carica della batteria è bassa o che è stata appena scattata una foto) e tale funzionalità è svolta anche dalle applicazioni (per esempio per notificare al resto del sistema che dei dati sono stati scaricati e sono disponibili per essere usati). I Broadcast Receiver (come i Service) non mostrano un'interfaccia utente, ma possono creare una status bar per avvertire che un determinato evento è in corso. Più comunemente può essere assimilabile ad un "gateway" per gli altri componenti, utilizzato per eseguire solo una quantità minima di lavoro durante l'esecuzione: ad esempio, potrebbe inizializzare un Service che esegue a sua volta un job basato sull'evento notificato dal particolare Broadcast Receiver che l'ha invocato.

In definitiva, un aspetto unico del design di Android è dato dal fatto che qualsiasi applicazione può azionare un componente di una qualsiasi altra applicazione. Per esempio, se si vuole scattare una foto con la fotocamera di sistema, molto probabilmente nel sistema esisterà già un'altra applicazione adatta allo scopo e qualsiasi applicazione potrà utilizzare tale componente, invece di creare su necessità (manualmente) un'Activity adatta alla cattura della foto: non è obbligatorio incorporare o collegare il codice dall'applicazione della videocamera, è necessario solo avviare l'Activity presente all'interno dell'applicazione che permette di catturare una foto. Al completamento, la foto viene ritornata all'applicazione che ne ha fatto richiesta, con la possibilità di usarla all'interno di elaborazioni specifiche. Come effetto finale per l'utente, il tutto fa sembrare che la fotocamera usata diventi parte integrante dell'applicazione.

Altra caratteristica fondamentale nell'architettura: quando il sistema avvia un componente, avvia il processo per quella particolare applicazione (se non è già in esecuzione) ed istanzia le classi necessarie per il componente. Ad esempio, se un'applicazione avvia un'Activity dedicata alla cattura delle foto all'interno dell'applicazione della fotocamera, quell'Activity esegue nel processo che appartiene all'applicazione della fotocamera, non all'interno del processo esterno che ne ha fatto richiesta. Tuttavia, al contrario della maggior parte degli altri sistemi, le applicazioni Android non hanno un singolo "entry point" (ad esempio, non esiste alcuna funzione "main()") ma, da un certo punto di vista, è l'utente ad essere considerato il vero entry point per l'esecuzione di applicazioni.

1.2 Sensori hardware nella piattaforma Android

Una particolarità che rende interessante l'utilizzo di un device Android è la moltitudine di sensori utilizzabili e supportati dalle API native (tramite le classi *Sensor* e *SensorEvent*). Per ragioni legate alla diversità dei modelli dei vari produttori, tali sensori non sono disponibili nella loro totalità all'interno di qualunque dispositivo ma comunque rappresentano un grande valore aggiunto che permette allo sviluppatore di poter creare applicazioni che possono fare uso di informazioni relative al contesto nel quale l'utilizzatore si trova in un preciso momento. I sensori disponibili e utilizzabili all'interno delle API ricavano valori di: accelerazione, prossimità, temperatura, gravità, orientamento, luce, accelerazione lineare, campo magnetico, pressione, umidità relativa e rotazione. Tuttavia queste informazioni riescono a trovare posto in applicazioni ad alto carattere innovativo se associate ad altri tipi di funzionalità offerte da moduli presenti nei device mobili più moderni, anche se non sono considerati strettamente dei sensori: la geo-localizzazione basata o meno su hardware addizionale, le informazioni sonore ricavate dal microfono interno (ad esempio, per il riconoscimento vocale), le informazioni ricavate dalla fotocamera (si pensi al riconoscimento facciale) e le informazioni relative alla rete telefonica mobile a cui si è connessi. Il mix di queste informazioni aiuta il programmatore nella creazione di applicazioni orientate al calcolo pervasivo, tema nel quale si sta ponendo molta attenzione viste le opportunità offerte dalla tecnologia attuale. Le API utili per l'estrazione e la manipolazione dei dati dei sensori sono state implementate, così come sono definite attualmente, dalla versione 3 dell'SDK (Android 1.6). Alcune proprietà all'atto dell'acquisizione sono comuni all'interno della classe *SensorEvent* e comprendono:

- **public int accuracy:** permette di impostare la precisione del sensore nella fase di acquisizione. Tale valore viene preimpostato tramite alcune costanti definite dalle API, in particolare:
 - *SensorEvent.SENSOR_STATUS_ACCURACY_HIGH* (costante 3) per avere un livello alto di precisione nell'acquisizione dei dati;
 - *SensorEvent.SENSOR_STATUS_ACCURACY_MEDIUM* (costante 2) per un livello medio di precisione (altri dati provenienti dal contesto di rilevazione possono aiutare nell'avere una lettura più accurata dei dati);
 - *SensorEvent.SENSOR_STATUS_ACCURACY_LOW* (costante 1) per un livello basso di precisione. In questo caso è necessaria una calibrazione con i dati provenienti dal resto dell'ambiente per avere una rilevazione approssimata a quella reale.
- **public Sensor sensor:** permette di identificare il sensore che ha generato un nuovo valore, in modo tale da poter ricavare successivamente le informazioni campionate. Questo dato è noto ed è possibile utilizzarlo tramite la classe *Sensor* presente nel package *android.hardware*. Le costanti definite sono:

<i>Sensor.TYPE_ACCELEROMETER,</i>	<i>Sensor.TYPE_AMBIENT_TEMPERATURE,</i>
<i>Sensor.TYPE_GRAVITY,</i>	<i>Sensor.TYPE_GYROSCOPE,</i>
<i>Sensor.TYPE_LINEAR_ACCELERATION,</i>	<i>Sensor.TYPE_MAGNETIC_FIELD,</i>
<i>Sensor.TYPE_PRESSURE,</i>	<i>Sensor.TYPE_PROXIMITY,</i>
<i>Sensor.TYPE_RELATIVE_UMIDITY,</i>	<i>Sensor.TYPE_ROTATION_VECTOR.</i>
- **public long timestamp:** permette di ricavare il tempo, misurato in nanosecondi, in cui è avvenuto l'evento di rilevazione del dato (e quindi anche la distanza dall'ultima rilevazione).
- **public final float[] values:** contiene i valori, contenuti in un array, ricavati da un determinato sensore. La lunghezza dell'array e il contenuto informativo dei valori dipende

dal tipo di sensore monitorato, e quindi questa struttura è completamente generica e utilizzabile da tutti i sensori.

1.2.1 Sistema di coordinate in SensorEvent API

Il sistema di riferimento utilizzato dalle API *SensorEvent* di Android per il posizionamento del device è definito come segue:

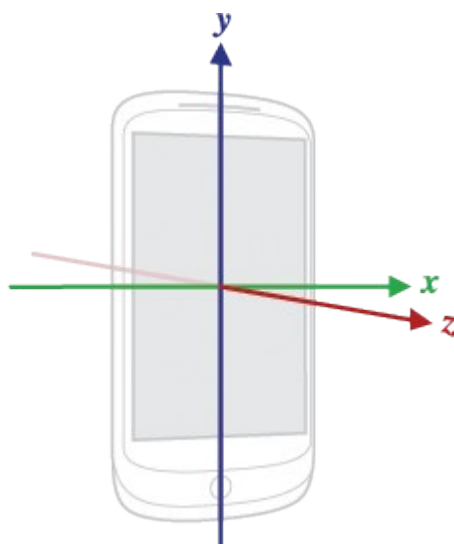


Illustrazione 4: Sistema di coordinate utilizzato da SensorEvent API

Il sistema di coordinate viene definito relativamente allo schermo del telefono nel caso sia orientato nella sua posizione predefinita (come da figura). In ogni modo, gli assi non vengono scambiati quando varia l'orientamento dello schermo del dispositivo. L'asse X è orizzontale e punta verso destra, l'asse Y è verticale e punta verso l'alto e l'asse Z punta verso l'esterno del lato frontale dello schermo. In questo modo, i valori delle coordinate che si trovano lungo l'asse che si prolunga dietro lo schermo assumono valori negativi di Z. Si noti che il sistema di coordinate è differente rispetto a quello utilizzato dalle API 2D di Android, dove l'origine degli assi è posto nell'angolo in alto a sinistra dello schermo.

Di seguito si illustreranno alcuni dei principali sensori presenti all'interno dei dispositivi disponibili in commercio.

1.2.2 Sensore di accelerazione

Il sensore di accelerazione, come dice il nome stesso, permette di misurare il valore di accelerazione applicato sul dispositivo, rilevato sui 3 assi. Tutti i valori sono misurati, secondo le unità SI, in m/s^2 e comprendono (all'interno del vettore dei valori acquisiti):

- values[0]: accelerazione negativa (Gx) sull'asse X
- values[1]: accelerazione negativa (Gy) sull'asse Y
- values[2]: accelerazione negativa (Gz) sull'asse Z

In definitiva, il sensore misura le forze applicate sul dispositivo, utilizzando la relazione

$$\mathbf{Ad} = - \Sigma \mathbf{Fs} / m$$

dove:

- **Ad**: accelerazione applicata al dispositivo;
- **Fs**: forze applicate al sensore di accelerazione
- **m**: massa del dispositivo

In particolare, la forza di gravità influenza sempre questa misura di accelerazione, quindi per avere il valore finale è necessario sottrarla dalla formula vista sopra:

$$Ad = -g - \Sigma Fs / m$$

dove **g** è la forza di gravità.

Per questo motivo, quando ad esempio il dispositivo è situato sopra una superficie piana (e quindi non sta accelerando), l'accelerometro misura la magnitudine di **g**, pari a 9,81 m/s². In modo analogo, quando il dispositivo viene posto in caduta libera e quindi ha un'accelerazione verso terra pari a 9,81 m/s², l'accelerometro rileva una magnitudine di 0 m/s². Operativamente, per misurare la reale accelerazione del dispositivo, il contributo della forza di gravità va eliminato. Questo può essere fatto applicando un filtro passa alto alla rilevazione. In modo analogo, per isolare il valore della forza di gravità, è possibile utilizzare un filtro passa basso al valore rilevato.

1.2.3 Sensore di campo magnetico

Questo sensore viene utilizzato per misurare il valore di campo magnetico nei 3 assi X, Y e Z. Tutti i valori sono in micro-Tesla (μT) e possono variare all'interno dell'intervallo definito da due costanti predefinite all'interno della classe *SensorManager*:

- *SensorManager.MAGNETIC_FIELD_EARTH_MAX*: valore massimo di campo magnetico rilevabile sul pianeta Terra
- *SensorManager.MAGNETIC_FIELD_EARTH_MIN*: valore minimo di campo magnetico rilevabile sul pianeta Terra

1.2.4 Giroscopio

Il giroscopio misura il regime di rotazione (misurato in radianti/secondo) intorno agli assi X, Y, Z locali al dispositivo. Il sistema di coordinate è lo stesso utilizzato per il sensore di accelerazione e la rotazione viene definita positiva se viene fatta in senso orario. I valori in uscita riportati dal sensore sono:

- `values[0]`: velocità angolare intorno all'asse X
- `values[1]`: velocità angolare intorno all'asse Y
- `values[2]`: velocità angolare intorno all'asse Z

In generale l'output derivato dal sensore misura la variazione degli angoli dovuta alla rotazione all'interno di un preciso lasso di tempo. La misurazione comunque è sempre soggetta ad alcuni errori dovuti principalmente al rumore e all'offset del sensore: per effettuare la compensazione generalmente si utilizzano informazioni provenienti da altri sensori in modo da rilevare un valore sicuramente più raffinato e vicino alla realtà.

1.2.5 Sensore di luminosità

Il sensore di luminosità permette di misurare il livello di luce ambientale. L'unità di misura utilizzata è il lux e, come nel caso del sensore di campo magnetico, la classe *SensorManager* definisce alcuni valori standard da associare ai rilevamenti. Ovviamente tutto ciò va considerato in un'ottica di approssimazione, visto che la sensibilità del sensore di luminosità è manufacturer-dependent.

I valori costanti associati sono:

- *SensorManager.LIGHT_CLOUDY (100 lux)*: valore di luminosità sotto un cielo nuvoloso;
- *SensorManager.LIGHT_FULLMOON (0,25 lux)*: valore di luminosità di notte con la luna piena;
- *SensorManager.LIGHT_NO_MOON (0,001 lux)*: valore di luminosità di notte in assenza della luna;
- *SensorManager.LIGHT_OVERCAST (10000 lux)*: valore di luminosità sotto un cielo coperto;
- *SensorManager.LIGHT_SHADE (20000 lux)*: valore di luminosità in ombra;
- *SensorManager.LIGHT_SUNLIGHT (110000 lux)*: valore di luminosità alla luce del sole;
- *SensorManager.LIGHT_SUNLIGHT_MAX (120000 lux)*: massimo valore di luminosità alla luce del sole;
- *SensorManager.LIGHT_SUNRISE (400 lux)*: valore di luminosità alla luce dell'alba.

1.2.6 Sensore di prossimità

Il sensore di prossimità permette di misurare la distanza (in centimetri) tra il dispositivo e un oggetto posto davanti come ostacolo. Il range di valori che può assumere va da 0 (corpo a contatto con il sensore) fino a un range massimo stabilito dalla tecnologia e tipo di sensore. Le API di Android prevedono anche l'utilizzo di hardware che permette solo di identificare valori binari: 0 se l'oggetto è all'interno del range di visibilità del sensore, altrimenti il valore massimo di distanza che il sensore riesce a ricavare, ritornato dalla funzione *Sensor.getMaximumRange()*.

1.2.7 Sensore vettore di rotazione

Il vettore di rotazione rappresenta l'orientamento del dispositivo come combinazione di un asse e un angolo: è la misura della rotazione del dispositivo di un angolo θ attorno a un asse $\langle x, y, z \rangle$. I tre elementi del vettore di rotazione sono $\langle x \cdot \sin(\theta/2), y \cdot \sin(\theta/2), z \cdot \sin(\theta/2) \rangle$, siccome la magnitudine del vettore di rotazione è pari a $\sin(\theta/2)$ e la direzione del vettore è uguale alla direzione dell'asse di rotazione. Gli elementi del vettore di rotazione non hanno unità di misura. Gli assi X, Y e Z sono definiti nella stessa maniera nella quale sono stati mostrati per il sensore di accelerazione. Il sistema di riferimento per le coordinate è definito nel seguente modo:

- X è definito come il prodotto vettoriale $Y \times Z$ (è tangente con il terreno rispetto alla posizione attuale del dispositivo e approssimativamente punta verso Est);
- Y è tangente al terreno rispetto alla posizione corrente del dispositivo e punta verso il Polo Nord magnetico;

- Z punta verso il cielo ed è perpendicolare al terreno.

Graficamente, è possibile rappresentare il sistema di riferimento come segue:

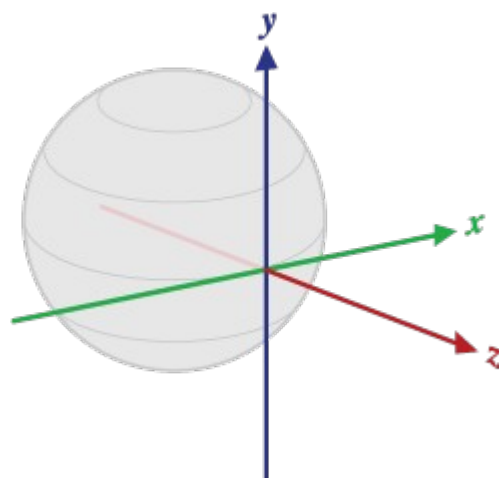


Illustrazione 5: Sistema di riferimento per il sensore vettore di rotazione

Il vettore di valori ritornati dall'acquisizione è il seguente:

- values[0]: $x \cdot \sin(\theta/2)$
- values[1]: $y \cdot \sin(\theta/2)$
- values[2]: $z \cdot \sin(\theta/2)$
- values[3]: $\cos(\theta/2)$ (opzionale e solo se value.length = 4)

1.2.8 Sensore di orientamento

Il sensore di orientamento riporta i valori, espressi in gradi, di angoli e di rotazioni. In particolare, i valori ritornati dal sensore sono:

- values[0]: **Azimuth**, angolo compreso tra la direzione del Nord magnetico e l'asse Y, attorno all'asse Z (valori che spaziano da 0 a 359 gradi). In particolare:
 - 0° = Nord
 - 90° = Est
 - 180° = Sud
 - 270° = Ovest
- values[1]: **Pitch**, rotazione attorno all'asse X (valori compresi tra -180 e 180 gradi), con valori positivi quando l'asse Z si muove verso l'asse Y;
- values[2]: **Roll**, rotazione attorno all'asse Y (valori compresi tra -90 e 90 gradi), con valori positivi quando l'asse X si muove verso l'asse Z.

Per questo tipo di misurazione, per ragioni storiche, l'angolo definito dal Roll assume valori positivi per rotazioni in senso orario, anche se in una visione strettamente matematica, tale valore dovrebbe assumere valori positivi solo per rotazioni che avvengono in senso antiorario.

2. Progetto su piattaforma Android

L'idea alla base del progetto è quella di poter creare, tramite collaborazione distribuita da parte degli utenti muniti di un dispositivo Android, una mappa esaustiva che comprende informazioni geolocalizzate sul segnale telefonico mobile (suddiviso per operatore) e sulle condizioni ambientali di acquisizione. I dati ricavati possono così essere elaborati per far fronte ad alcune problematiche tecniche reali che, al momento, non vengono affrontate in maniera soddisfacente con le soluzioni adottate fino ad ora. La principale tra di esse è la determinazione della qualità del segnale di un operatore telefonico in una zona più o meno circoscritta, utile ad esempio nella fase di decisione di stipulazione di un contratto con un operatore mobile: tramite il servizio in questo modo il cliente può consultare un database nato dalla collaborazione distribuita degli utenti, che può mostrare con una precisione accettabile il livello di potenza di segnale nella propria zona, in modo tale da mettere l'utente nella condizione di potersi regolare nelle decisioni da prendere. Dall'altra parte, chiunque in possesso dell'applicazione può effettuare un'acquisizione e quindi essere un utente attivo: nella propria zona abitativa, in vacanza o dove preferisce. Le informazioni comprendono valori propri della rete telefonica (come l'intensità del segnale, il nome dell'operatore, la presenza di roaming o meno, ecc.), valori riguardanti la luminosità, in modo da stabilire approssimativamente le condizioni atmosferiche presenti durante l'acquisizione o se ci si trova all'interno o all'esterno di un edificio, ed informazioni sul posizionamento effettuato tramite hardware aggiuntiva (GPS, in modo da avere una ottima accuratezza e precisione) o tramite celle telefoniche (che permettono di avere una posizione meno accurata, ma utile se ci si trova in ambienti interni o poco esposti). Tali valori poi vengono inviati, tramite internet, ad un server che avrà il compito di processare i dati e renderli disponibili al pubblico per una più facile consultazione. Se il dispositivo che compie l'acquisizione non è in possesso di un modulo per la connessione ad internet o se in un determinato momento la rete non fosse disponibile, i dati acquisiti vengono memorizzati all'interno del dispositivo, in modo che possano essere inviati in un secondo momento. Se il dispositivo di acquisizione non è dotato di un modulo hardware per la connessione ad internet ma fosse dotato di hardware abilitante a connessioni ad-hoc (ad esempio Bluetooth), i dati acquisiti possono essere inviati ad un dispositivo vicino (magari dotato di maggiori risorse hardware) che si farà carico dell'invio dei vari report accumulati.

2.1 Casi d'uso

Di seguito vengono illustrati alcuni possibili casi d'uso che l'utente può incontrare durante l'utilizzo dell'applicazione. Le varie possibilità si differenziano dipendentemente dal fatto che il dispositivo abbia o meno hardware sufficiente per soddisfare tutti i requisiti e quindi ricavare il maggior numero di informazioni disponibili e inviarle. Ovviamente sono possibili anche scenari intermedi in cui è presente un mix delle funzionalità (ad esempio, è presente il modulo per la connessione ad internet tramite la rete mobile ma non il modulo Wi-Fi, ecc.).

2.1.1 Primo caso d'uso: dispositivo con sufficienti risorse hardware

In questo primo caso, l'utente che compie l'acquisizione possiede un dispositivo abilitato alla connessione ad internet (su rete mobile cellulare o su rete Wi-Fi) e fornito di tutti i sensori necessari per compiere un'acquisizione dettagliata dello stato. In particolare, i passi da seguire dopo l'avvio dell'applicazione sono i seguenti:

1. Abilitazione dell'acquisizione del segnale telefonico. In questo modo l'utente, sfruttando il sensore di orientamento, ruota il telefono e lo muove nello spazio fino al completamento;

2. Abilitazione dell'acquisizione delle informazioni di luminosità. In questo modo l'utente abilita il sensore di luminosità e permette al sistema di ricavare il valore (in lux) della luce ambientale della locazione corrente;
3. Abilitazione dell'acquisizione delle informazioni di posizionamento. In questo modo l'utente si geo-localizza (con animazione a video che indica il luogo sulla mappa) ricavando le coordinate di latitudine e longitudine del luogo di acquisizione.
 - Se l'utente si trova all'esterno, può attivare l'hardware addizionale presente nel suo dispositivo (in questo caso GPS) per avere una localizzazione più accurata e con poco margine di errore (consigliata dove possibile);
 - Nel caso in cui l'utente fosse all'interno di un edificio o fosse ostacolato (e quindi senza la visione diretta del cielo) può utilizzare la rete mobile per la localizzazione. In questo modo si rinuncia all'accuratezza tipica del GPS e si accetta una posizione approssimata, ma comunque ragionevole.
4. Salvataggio e invio del report. In questo modo l'utente può decidere se salvare il report in locale per un invio successivo oppure se inviarlo direttamente al server. Eventualmente esiste anche la possibilità di inviare il report ad un altro dispositivo, in modo da minimizzare il numero di connessioni contemporanee al server, raggiunto tramite internet. Quindi è possibile effettuare le seguenti operazioni:
 - Salvataggio del report in locale per un invio successivo tramite comunicazione ad-hoc verso un altro dispositivo, o tramite internet direttamente al server di elaborazione dati;
 - Invio, senza salvataggio in locale, del report attuale al server remoto;
 - Invio, senza salvataggio in locale, del report attuale verso un altro dispositivo tramite comunicazione Bluetooth;
 - Invio, senza salvataggio in locale, del report attuale e dei report precedentemente salvati (o ricevuti da altri dispositivi) tramite comunicazione Bluetooth.

L'ultima fase prevede la chiusura dell'applicazione oppure l'inizio di una nuova acquisizione.

2.1.2 Secondo caso d'uso: dispositivo con limitate risorse hardware

Nel secondo caso, l'utente che compie l'acquisizione possiede un dispositivo non abilitato alla connessione ad internet (sia su rete mobile cellulare che su rete Wi-Fi) e senza tutti i sensori necessari per compiere un'acquisizione dettagliata dello stato. In particolare, i passi da seguire dopo l'avvio dell'applicazione sono i seguenti:

1. Abilitazione dell'acquisizione del segnale telefonico. In questo caso l'utente ricava i dati necessari senza l'ausilio del sensore di orientamento. Ne consegue una minor accuratezza dovuta ad una mancanza di rotazione nello spazio del dispositivo (per acquisire in modo più vario il segnale in una determinata area spaziale);
2. Abilitazione dell'acquisizione della luminosità. In questo caso il valore di luminosità è facoltativo.
 - Se presente, l'utente arricchisce il report abilitando il sensore di luminosità e quindi permettendo al sistema di ricavare il valore (in lux) della luce ambientale della

locazione corrente.

- Se non presente, tale dato non viene riportato e quindi impostato come “non disponibile”;
3. Abilitazione dell'acquisizione delle informazioni di posizionamento. In questo modo l'utente si geo-localizza (con animazione a video che indica il luogo sulla mappa) ricavando le coordinate di latitudine e longitudine del luogo di acquisizione. In questo caso si presume che il dispositivo non abbia un sistema di localizzazione accurato, quindi si esclude l'uso di GPS. In questo caso l'acquisizione è equivalente a quella effettuata all'interno di un edificio (o in caso di ostacoli) ed è possibile utilizzare la rete mobile per la localizzazione. In questo modo si rinuncia all'accuratezza tipica del GPS e si accetta una posizione approssimata, ma comunque ragionevole.
 4. Salvataggio e invio del report. In questo caso l'utente può solo inviarlo (tramite connessione ad-hoc) ad un altro dispositivo compatibile, in modo che quest'ultimo possa successivamente inviare i dati al server raggiunto tramite internet e minimizzare il numero di connessioni remote contemporanee. Quindi è possibile effettuare le seguenti operazioni:
 - Salvataggio del report in locale per un invio successivo tramite comunicazione ad-hoc verso un altro dispositivo;
 - Invio, senza salvataggio in locale, del report attuale verso un altro dispositivo tramite comunicazione Bluetooth;
 - Invio, senza salvataggio in locale, del report attuale e dei report precedentemente salvati (o ricevuti da altri dispositivi) tramite comunicazione Bluetooth.

L'ultima fase prevede la chiusura dell'applicazione oppure l'inizio di una nuova acquisizione.

2.2 Implementazione

La fase implementativa è stata effettuata con l'ausilio di alcuni strumenti preparati da Google e messi a disposizione degli sviluppatori. In particolare è stato utilizzato:

- Sistema operativo desktop Ubuntu Linux 11.10 x86_64;
- Eclipse IDE 3.7.0 “Indigo” 64-bit;
- SDK Android versione 2.2 (Froyo) e 2.3.3 (Gingerbread) per sistemi Linux;
- Plug-in ADT per Eclipse versione 15.0.0;

Gli ultimi due strumenti citati hanno offerto, oltre alle funzionalità di sviluppo (quali librerie, documentazione, ecc.) anche funzionalità di debug avanzate, sia a livello di codice che a livello di sistema (ad esempio, la visualizzazione dei processi in esecuzione con le relative statistiche di carico CPU/RAM ed una shell per l'esecuzione di comandi di sistema). Il debug è stato effettuato in remoto direttamente all'interno del dispositivo, senza ricorrere all'emulatore fornito all'interno dell'SDK. In particolare quest'ultimo è stato scartato per due motivazioni principali: le pessime prestazioni durante l'esecuzione (anche in presenza di un processore AMD quad-core l'ambiente di debug diventa terribilmente lento e quindi inutilizzabile per un uso intensivo) e la possibilità di avere più dispositivi fisici esterni abilitati per il debugging delle applicazioni (in questo modo vengono caricate direttamente all'interno del dispositivo e avviato il debug tramite comunicazione su porta USB).

Di seguito, viene mostrata un'immagine della GUI che presenta la console di debug (Dalvik Debug Monitor):

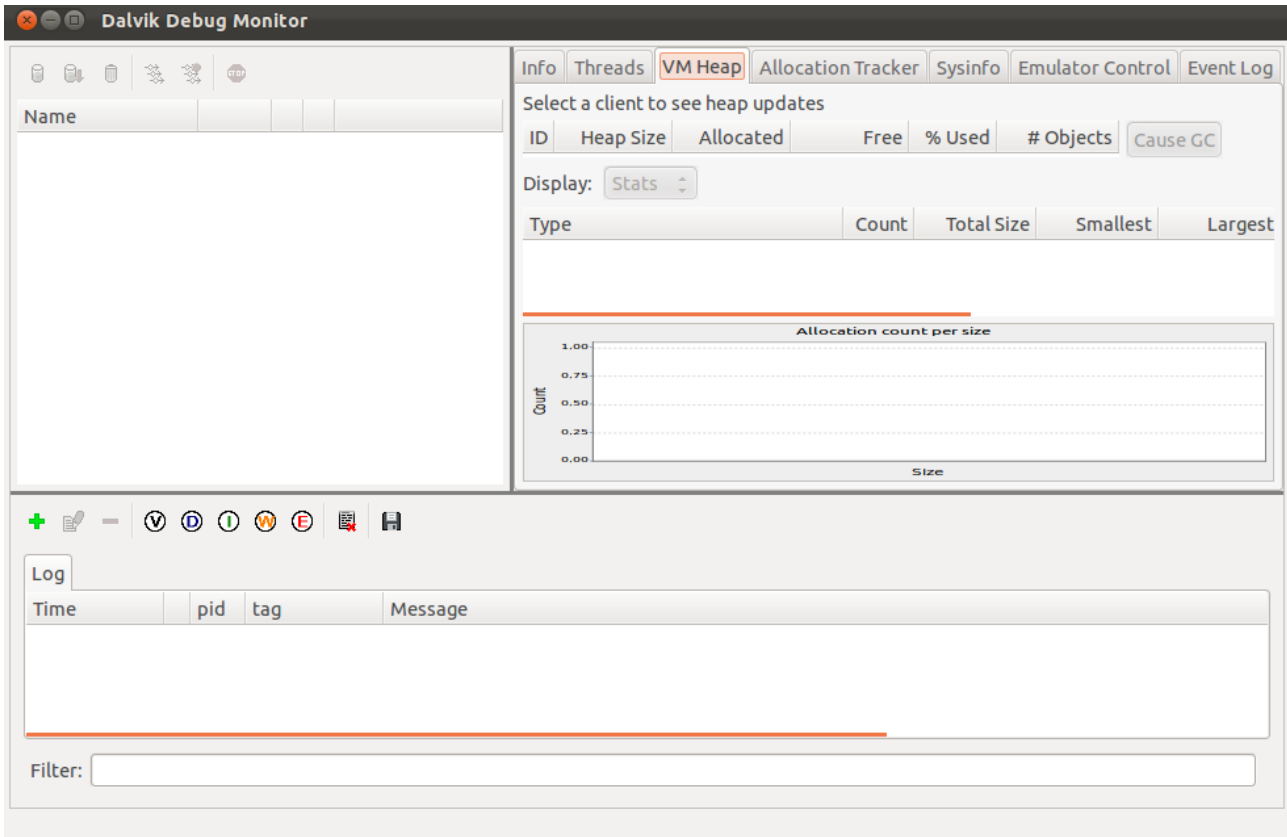


Illustrazione 6: Ambiente di debug per Android - Dalvik Debug Monitor

Da come si può notare dalla figura, sono molti i parametri che possono essere monitorati nella fase di debug di una determinata applicazione: gli eventi (verbosi, informativi, di errore, di attenzione e personalizzati), i thread in esecuzione, l'heap della Virtual Machine, ecc. In aggiunta sono disponibili funzioni avanzate, come l'invocazione forzata del garbage collector, la chiusura forzata della Virtual Machine, il dump dello stato dell'applicazione, del dispositivo e delle interfacce radio. Di seguito verrà mostrato il dettaglio del codice utilizzato per implementare le varie funzionalità.

2.2.1 Acquisizione delle informazioni sul segnale telefonico

La parte di acquisizione del segnale è uno dei punti fondamentali dell'intera applicazione. La logica è stata integrata all'interno della classe *AndroidSignalActivity* che implementa al suo interno l'Activity Android dedicata all'interazione con l'utente, insieme alle funzioni per il campionamento dei valori dai vari sensori presenti sul dispositivo. L'acquisizione prevede anche l'uso del sensore di orientamento per una duplice scelta progettuale:

- Il campionamento del segnale non deve essere istantaneo, ma si deve prevedere una finestra temporale in modo tale da diminuire l'errore di misura dovuto a picchi anomali di segnale (sia positivi che negativi), per poi restituire in uscita una media aritmetica del valore misurato nell'arco di tempo stabilito;
- Il dispositivo deve essere spostato all'interno di un'area che circonda l'utente, in modo da

non rilevare la misura di segnale in un punto ma in un determinato spazio, anche in questo caso per avere un dato più genuino e veritiero da riportare.

Il codice utilizzato per l'implementazione è il seguente:

```
// phone signal informations
signalValue = -1;
mySensorManager.unregisterListener(this);
myOrientationSensor =
mySensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION);

myTelephonyManager = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);
myProgressSignal = new ProgressDialog(AndroidSignalActivity.this);
myProgressSignal.setMax(360);
myProgressSignal.setProgressStyle(ProgressDialog.STYLE_HORIZONTAL);
myProgressSignal.setTitle("Acquiring signal...");
```

In questa parte è illustrata l'inizializzazione delle variabili necessarie per il corretto campionamento del segnale e per la corretta visualizzazione a video del progresso compiuto. In particolare, si ricava dal sistema l'oggetto per manipolare il sensore di orientamento e l'oggetto per gestire il servizio di telefonia. Si noti che in questo modo è possibile ricavare informazioni derivanti da livelli bassi del sistema operativo, direttamente tramite alcune chiamate alle API di alto livello disponibili al programmatore.

```
// behavior for the phone signal button
((ImageButton) findViewById(R.id.imgBtnSignal)).setOnClickListener(new
    OnClickListener() {
        @Override
        public void onClick(View v) {
            if (myTelephonyManager.getSimState() == TelephonyManager.SIM_STATE_READY){
                if (imgBtnSignal) {
                    sampleNetworkType = myTelephonyManager.getNetworkType();
                    mySensorManager.registerListener(AndroidSignalActivity.this,
                    myOrientationSensor, SensorManager.SENSOR_ORIENTATION);
                    myPhoneStateListener = new MyPhoneStateListener();
                    countOrientation = 0;
                    signalSample = 0;
                    myProgressSignal.setProgress(0);
                    Toast.makeText(getApplicationContext(), "Orientation
                    informations enabled!", Toast.LENGTH_SHORT).show();
                    myTelephonyManager.listen(myPhoneStateListener,
                    PhoneStateListener.LISTEN_SIGNAL_STRENGTHS);
                    myProgressSignal.show();
                    Toast.makeText(getApplicationContext(), "Signal
                    strength informations enabled!", Toast.LENGTH_SHORT).show();
                    imgBtnSignal = false;
                }
                else {
                    timestamp = System.currentTimeMillis();
                    mySensorManager.unregisterListener(AndroidSignalActivity.this,
                    myOrientationSensor);
                    myTelephonyManager.listen(myPhoneStateListener,
                    PhoneStateListener.LISTEN_NONE);
                    myPhoneStateListener = null;
                    Toast.makeText(getApplicationContext(), "Signal strength
                    informations disabled!", Toast.LENGTH_SHORT).show();
                }
            }
        }
    });
```

```

        imgBtnSignal = true;
        Toast.makeText(getApplicationContext(), "Orientation
        informations disabled!", Toast.LENGTH_SHORT).show();
    }
});

```

Il codice sopra riportato mostra il comportamento del bottone che l'utente può premere per abilitare o disabilitare l'acquisizione del segnale. Durante l'abilitazione si ricava l'informazione istantanea del tipo di rete rilevato, con relativa indicazione sullo stato della SIM per conoscere lo stato di collegamento alla rete all'inizio del campionamento (il device può essere anche in modalità aereo). Successivamente si registra un listener per l'interazione con il sensore di orientamento. Poi viene registrato un secondo listener per la parte telefonica, incaricato a gestire l'evento di cambiamento di potenza del segnale. In questo modo ad ogni variazione viene scatenato un evento e provoca l'aggiornamento del valore corrente misurato. Durante la disabilitazione (effettuata premendo nuovamente il bottone sul display), si ricava il valore temporale di disattivazione e si de-registrano tutti i listener registrati in precedenza (per il sensore di orientamento e le informazioni sulla variazione della potenza del segnale telefonico), in modo da liberare le risorse non più utilizzate.

```

private class MyPhoneStateListener extends PhoneStateListener
{
    // get the signal strength from the provider, each time there is an
    // update
    private int signal;
    @Override
    public void onSignalStrengthsChanged(SignalStrength signalStrength)
    {
        super.onSignalStrengthsChanged(signalStrength);
        if (myTelephonyManager.getSimState() ==
            TelephonyManager.SIM_STATE_READY) {
            mySignalStrength = signalStrength;
            signal = signalStrength.getGsmSignalStrength();
            Toast.makeText(getApplicationContext(), "Network info:\n" + "Operator
            name: " + myTelephonyManager.getNetworkOperatorName() + "\nNetwork type: "
            + networkType[myTelephonyManager.getNetworkType()] + "\nGSM Signal
            strength: " + String.valueOf(signal + " asu" + " (" + String.valueOf(-
            113+2*signal) + " dBm)") + "\nRoaming: " +
            String.valueOf(myTelephonyManager.isNetworkRoaming()) + "\nCell ID: " +
            String.valueOf(((GsmCellLocation)
                myTelephonyManager.getCellLocation()).getCid() & 0xffff),
                Toast.LENGTH_SHORT).show(); // 0xffff mask used when connected to 3G
            cells
        }
    }
}

```

La classe sopra riportata implementa l'handler che viene invocato ogniqualvolta il livello di potenza di segnale cambia. Viene salvato quest'ultimo dato, previo controllo dello stato della SIM, all'interno di una variabile che verrà poi utilizzata per determinare la media finale del segnale acquisito. Inoltre mostra a video, tramite un *Toast*, il resoconto in tempo reale del campionamento, indicando:

- Nome dell'operatore telefonico;

- Tipo di rete (GSM, GPRS, EDGE, UMTS, HSDPA, ecc.);
- Potenza del segnale, sia in ASU che in dBm (calcolato come: $-113+2*asu_signal$);
- Informazione che indica se si è in presenza di roaming;
- ID dell'ultima cella coinvolta nella procedura di acquisizione.

```

public void onSensorChanged(SensorEvent event) {

    // action for orientation value changed
    if (event.sensor.getType() == Sensor.TYPE_ORIENTATION) {
        // if the type of the network is changed meanwhile, stop the
        acquisition and do the average
        if (sampleNetworkType == myTelephonyManager.getNetworkType()){
            if (prevValue == 0) {
                countOrientation++;
                prevValue = event.values[0];
            }
            else if (countOrientation <= 360 && event.values[0]!
=prevValue) {
                countOrientation++;
                signalSample +=
mySignalStrength.getGsmSignalStrength();
                prevValue = event.values[0];
myProgressSignal.setProgress(countOrientation);
            }
            if (countOrientation == 360) {
                signalValue = signalSample/countOrientation;
                Toast.makeText(getApplicationContext(),String.valueOf(signalValue),
                Toast.LENGTH_SHORT).show();
                prevValue = 0;
                myProgressSignal.dismiss();
mySensorManager.unregisterListener(AndroidSignalActivity.this,
myOrientationSensor);
            }
        }
        else {
            if (countOrientation <= 360 && countOrientation>0) {
                signalValue=signalSample/countOrientation;
                Toast.makeText(getApplicationContext(),String.valueOf(signalValue),
                Toast.LENGTH_SHORT).show();
                prevValue = 0;
                myProgressSignal.dismiss();
mySensorManager.unregisterListener(AndroidSignalActivity.this,
myOrientationSensor);
            }
        }
    }
}

```

Questo metodo viene invocato all'atto del cambiamento del valore del sensore di orientamento. Infatti qui viene implementato l'handler che gestisce l'acquisizione tramite lo spostamento del dispositivo nello spazio. Un vincolo di progetto ha imposto che l'acquisizione sia valida solo se fatta per un tipo uniforme di rete. Per garantire ciò si controlla, ad ogni chiamata dell'handler, che il tipo di rete non sia cambiato: operativamente si controlla se il tipo di rete attuale è uguale al tipo di rete

2. Progetto su piattaforma Android

campionato all'inizio dell'acquisizione. Finché i valori sono identici la procedura prosegue senza problemi fino al termine, altrimenti si ferma e viene calcolata la media del segnale campionato in base al numero di misurazioni fatte fino al momento del cambio di rete. Per ottenere questo comportamento è utilizzata una variabile *countOrientation* che viene incrementata solo se il valore riportato in *values[0]* (la variazione dell'orientamento sull'asse X, secondo il sistema di riferimento riportato in 1.2.1) varia rispetto al campionamento precedente. In questo modo se l'acquisizione dovesse finire prima di raggiungere il limite massimo di acquisizioni (nello specifico, 360), si avrebbe già il denominatore pronto per calcolare la media della potenza del segnale per le *countOrientation* misure. Nel caso in cui la procedura vada a buon fine, ad ogni iterazione viene calcolata la somma del segnale accumulato fino ad arrivare al valore massimo di campionamenti (nello specifico, 360). Infine si esegue la media, dividendo il valore totale di segnale ottenuto per il numero massimo di rilevazioni effettuate. Per rendere visibile la procedura all'utilizzatore, durante l'esecuzione viene mostrata a video una barra per indicare il progresso, che aumenta il proprio valore in base alla rotazione data al dispositivo: se il dispositivo è fermo, il valore rimane costante e invariante, mentre se è in movimento sul proprio asse X, esso aumenta, indipendentemente dal verso del movimento, in modo da evitare rotazioni negative (grazie alla rilevazione del movimento assoluto del sensore tramite i parametri *countOrientation* e *prevValue* – quest'ultimo identifica il penultimo valore che ha riportato il sensore). Alla fine della procedura viene reimpostato *prevValue* a zero, in modo che possa essere pronto per una successiva acquisizione, viene eliminata dallo schermo la barra di avanzamento in modo da avere il focus sulla Activity e de-registrato il listener per il sensore di orientamento, in modo da ottimizzare il consumo di risorse e disattivare i sensori non utilizzati. Infatti, la mancata disattivazione dei listener relativi ai sensori, provoca un eccessivo consumo di risorse computazionali all'interno del dispositivo con conseguente abbattimento dell'autonomia della batteria, parametro da ottimizzare e massimizzare nel caso di dispositivi mobili. La figura sottostante mostra la finestra di interazione con l'utente:

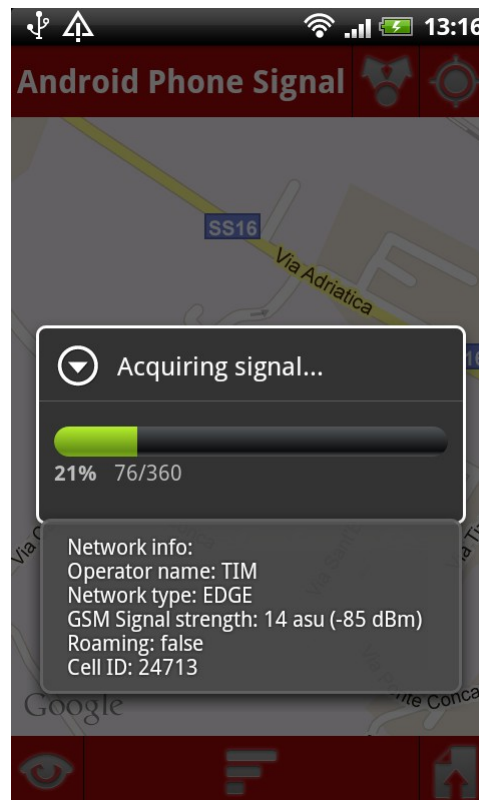


Illustrazione 7: Acquisizione delle informazioni sul segnale telefonico

2.2.2 Acquisizione delle informazioni di locazione

Le informazioni di locazione corrente estraibili dal dispositivo, che permettono di avere un dato più o meno approssimato, sono fondamentali per l'abilitazione dell'applicazione al servizio di geolocalizzazione distribuito. I dati ricavati sono due: latitudine e longitudine. Queste coordinate sono fisiche e assolute e permettono di localizzare il dispositivo in un punto della mappa mondiale. A tal proposito sono state utilizzate le API di Google Maps, tramite l'utilizzo degli overlay, per mostrare a video il punto esatto ricavato. In questo modo è possibile un confronto e verificare se la rilevazione è sufficientemente accettabile e veritiera. In particolare, i dati di posizionamento sono ricavati su richiesta. Il provider di locazione viene deciso on-demand in base a dei criteri specifici indicati all'interno del codice:

- Se all'interno del dispositivo è presente un modulo GPS, ed esso è abilitato, l'acquisizione verrà data in carico a tale modulo che, ad ogni intervallo prestabilito, fornirà i giusti valori di latitudine e longitudine.
- Se invece all'interno del dispositivo non è presente un modulo GPS, la localizzazione avviene sfruttando le informazioni derivate dalla rete mobile a cui si è collegati (tramite triangolazione delle celle telefoniche), che aggiornano i dati di latitudine e longitudine ad intervalli prestabiliti.

Ovviamente la prima possibilità è sempre preferibile, visto che fornisce valori più accurati. In ogni modo non tutti i dispositivi hanno moduli esterni dedicati alla localizzazione, quindi una seconda alternativa, anche se meno accurata, va fornita. L'implementazione di questo servizio è data dal seguente codice:

```
// location informations
imgBtnLocEnabled = true;
imgBtnLoc = ((ImageButton) findViewById(R.id.btnGetPos));
lm = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);
map = (MapView) findViewById(R.id.mapview);
mapController = map.getController();
mapOverlays = map.getOverlays();
drawable = this.getResources().getDrawable(R.drawable.androidmarker);
itemizedOverlay = new MyItemizedOverlay(drawable);
```

In particolare bisogna ricavare un oggetto di tipo *LocationManager* che abbia accesso al servizio di sistema denotato da *Context.LOCATION_SERVICE*, che permette di accedere alle informazioni di locazione a basso livello. Inoltre, per arricchire l'esperienza utente, è necessario creare una mappa, un controllore per la mappa e inizializzare l'oggetto di overlay da apporre sopra ad essa in modo da mostrare sempre la posizione aggiornata ad ogni acquisizione. Il comportamento che il cliente innesca tramite la pressione del bottone relativo è il seguente:

```
// behavior for the location button
imgBtnLoc.setOnClickListener(new OnClickListener() {
    public void onClick(View arg0) {
        if (!imgBtnLocEnabled) {
            lm.removeUpdates(myLocLis);
            myLocLis = null;
            imgBtnLocEnabled = true;
            Toast.makeText(getApplicationContext(), "Position
retrieval disabled!", Toast.LENGTH_SHORT).show();
        }
    }
});
```

```

        else {
            myLocLis = new MyLocationListener();
            criteria = new Criteria();
            criteria.setAccuracy(Criteria.ACCURACY_FINE); //
ACCURACY_LOW for Android 2.3.3 API
            criteria.setCostAllowed(true);
            criteria.setSpeedRequired(false);
            criteria.setPowerRequirement(Criteria.POWER_MEDIUM);
            bestLocationProvider = lm.getBestProvider(criteria,
true);
            lm.requestLocationUpdates(bestLocationProvider,
5000, 5, myLocLis);
            Toast.makeText(getApplicationContext(), "Position
provided by: " + bestLocationProvider, Toast.LENGTH_SHORT).show();
            imgBtnLocEnabled = false;
        }
    }
});

```

Il cliente, premendo il bottone, permette al sistema di impostare i parametri di scelta del provider di locazione tramite l'oggetto *Criteria*. In questo modo è possibile decidere di abilitare il giusto provider in base allo stato attuale dei moduli collegati. In particolare, il criterio impostato prevede di porre al primo posto il modulo che permette di avere il valore di locazione più accurato, eventualmente con costi extra accettati (anche in termini monetari), senza necessità di alta velocità nell'acquisizione e con richiesta media di consumo di energia. Con questo criterio il sistema gestisce una lista, dal modulo più rispettoso dei criteri a quello più lontano. In questo modo, se presente ed abilitato, il GPS viene utilizzato come prima opzione, altrimenti si utilizza la rete mobile e via via con soluzioni sempre meno congruenti con i parametri preimpostati. Una volta trovato il provider più adatto, si registra un listener che ascolta gli aggiornamenti di locazione apportati da tale provider. In particolare, tramite il metodo *LocationManager.requestLocationUpdates()*, si notifica all'Activity corrente il cambiamento periodico, sia a livello temporale (ogni 5 secondi), sia a livello spaziale (ogni variazione di 5 metri rispetto all'ultimo valore campionato). All'atto della disabilitazione dell'acquisizione delle informazioni di locazione, tutti i listener vengono disattivati e vengono rilasciate le risorse precedentemente allocate, ad esclusione delle informazioni riguardanti la mappa.

Il comportamento dell'applicazione all'atto del cambiamento di posizione è il seguente:

```

class MyLocationListener implements LocationListener {
    @Override
    public void onLocationChanged(Location location) {
        String Text = "My current NEW location is: " + "\nLatitude = "
+ location.getLatitude() + "\nLongitude = " + location.getLongitude();
        Toast.makeText(getApplicationContext(), Text,
Toast.LENGTH_LONG).show();
        mapController.setZoom(17);
        GeoPoint point = new GeoPoint((int) (location.getLatitude() *
1E6), (int) (location.getLongitude() * 1E6));
        mapController.animateTo(point);
        OverlayItem overlayitem = new OverlayItem(point, "", "");
        itemizedOverlay.clear();
        itemizedOverlay.addOverlay(overlayitem);
        mapOverlays.add(itemizedOverlay);
        myLocation = location;
    }
}

```

2. Progetto su piattaforma Android

La classe sopra riportata implementa l'interfaccia *LocationListener* che permette di catturare l'evento *onLocationChanged()*, scatenato quando una nuova informazione di locazione (che rispetta i criteri scelti), è disponibile. In particolare vengono mostrati a video i nuovi valori acquisiti tramite un *Toast*, viene impostato uno zoom predefinito sulla mappa e viene calcolato il punto tramite le coordinate di latitudine e longitudine, ricavate dall'oggetto di tipo *Location*, parametro del metodo inserito dalla chiamata di sistema per un nuovo aggiornamento. Successivamente si imposta l'animazione per mostrare a video la nuova posizione, si cancella la locazione visualizzata precedentemente e si aggiunge l'immagine di overlay all'interno della mappa, esattamente nella posizione denotata dalle coordinate acquisite. L'immagine sottostante mostra la finestra che dettaglia all'utente la modifica di posizione. Si noti l'elemento di overlay (il robot verde) posizionato esattamente sulle nuove coordinate:

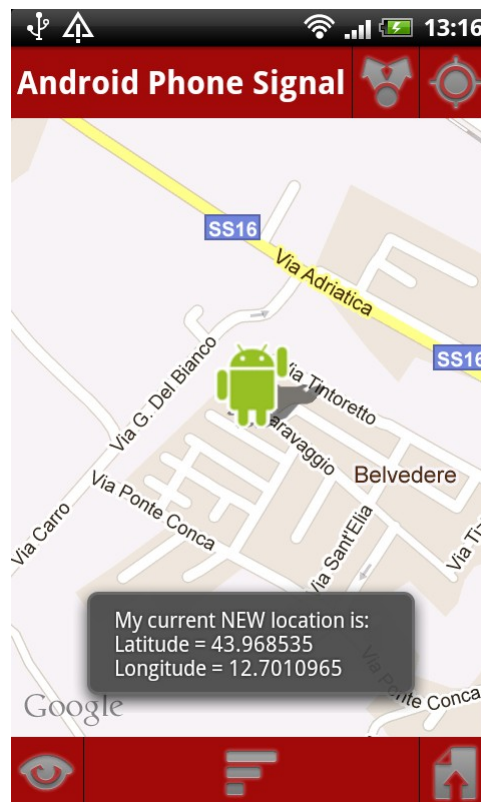


Illustrazione 8: Acquisizione delle informazioni di locazione

2.2.3 Acquisizione delle informazioni di luminosità

Le informazioni di luminosità vengono campionate per avere un'idea approssimata delle condizioni ambientali esterne all'atto dell'acquisizione. Infatti, in base ai valori ritornati dal sensore, è possibile determinare se si è in un ambiente chiuso, se si è esposti alla luce del sole, e così via. Tali soglie sono cablate all'interno di costanti presenti nella classe *SensorManager*. Ovviamente i valori ricavati possono essere diversi dipendentemente dal tipo di dispositivo ma soprattutto dalla qualità del sensore. In presenza di ostacoli ravvicinati come può essere una pellicola di protezione oppure dello sporco depositato sopra, il valore può subire variazioni, tendenzialmente sottostimando il valore reale. Purtroppo questo tipo di rumore nell'acquisizione è difficile da rimuovere visto che è relativo

al buon senso dell'utente che effettua il campionamento dei valori, il quale deve accertarsi che il sensore abbia piena visibilità dell'ambiente e non sia coperto da corpi estranei. Il codice adibito a ricavare tale dato è il seguente:

```
// light informations
myLightValue = -1;
mySensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
myLightSensor = mySensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
mySensorManager.unregisterListener(this);
imgBtnLightEnabled = true;
```

In questo caso, come nel caso del sensore di orientamento, è necessario ottenere l'oggetto che rappresenta il servizio di gestione dei sensori e da esso estrarre le informazioni riguardanti il sensore di luminosità. Per questione di efficienza energetica e computazionale ci si assicura che, all'atto dell'inizializzazione, il listener associato ai sensori sia disabilitato. Il codice del bottone utilizzabile dall'utente per compiere l'acquisizione è il seguente:

```
// behavior for the light button
((ImageButton) findViewById(R.id.imgBtnLight)).setOnClickListener(new
OnClickListener() {
    @Override
    public void onClick(View v) {
        if (imgBtnLightEnabled) {
            mySensorManager.registerListener(AndroidSignalActivity.this,
myLightSensor, SensorManager.SENSOR_DELAY_NORMAL);
            Toast.makeText(getApplicationContext(), "Light
informations enabled!", Toast.LENGTH_SHORT).show();
            imgBtnLightEnabled = false;
        }
        else {
            mySensorManager.unregisterListener(AndroidSignalActivity.this,
myLightSensor);
            Toast.makeText(getApplicationContext(), "Light informations
disabled!", Toast.LENGTH_SHORT).show();
            imgBtnLightEnabled = true;
        }
    }
});
```

All'atto della pressione del tasto, viene registrato il sensore di luminosità per la gestione dell'evento da parte dell'Activity. Insieme a questa operazione, viene impostato anche il grado di aggiornamento dell'acquisizione, con la costante *SensorManager.SENSOR_DELAY_NORMAL*. In questo modo si impone un livello medio di refresh. Quando si disabilita l'acquisizione, il listener viene de-registrato per questioni di efficienza energetica.

Il comportamento dell'handler è il seguente:

```
// action for light sensor value changed
if (event.sensor.getType() == Sensor.TYPE_LIGHT){
    Toast.makeText(getApplicationContext(), "Light value: " +
String.valueOf(event.values[0]), Toast.LENGTH_SHORT).show();
    myLightValue = event.values[0];
}
```

Allo scatenamento dell'evento di modifica del valore di luminosità, viene campionato

semplicemente il valore attuale e mostrato all'utente tramite un pop-up informativo. Se il valore di luminosità non è disponibile (sia per mancanza del sensore, sia per il rifiuto volontario del campionamento del dato), si riporta il valore “-1.0” per denotare l'evento. La seguente immagine mostra esattamente cosa viene mostrato all'utente durante la fase di campionamento:

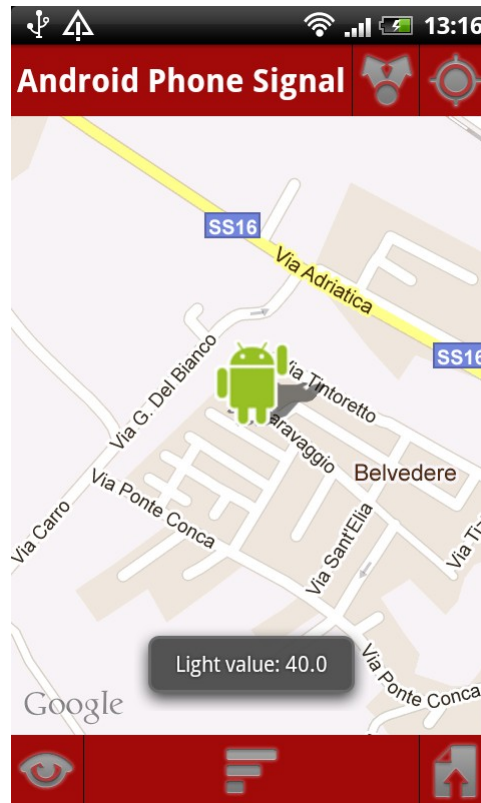


Illustrazione 9: Acquisizione delle informazioni di luminosità

2.2.4 Generazione del report

Il report di acquisizione unisce tutti i dati campionati dai vari sensori e moduli in un unico prospetto di facile lettura e adatto alla trasmissione. A tal proposito è stato creato un oggetto *SignalReportTO* che ingloba tutti i valori necessari insieme ai metodi per scriverli ed estrarli. Essi sono:

```
private String operatorName;  
private int avgSignalStrength;  
private String networkType;  
private boolean roaming;  
private String cellID;  
private float lightInfo;  
private double latitude;  
private double longitude;  
private String locationProvider;  
private long timestamp;
```

Oltre alla visualizzazione di un prospetto compatto, all'interno dell'Activity riservata al report è possibile effettuare tre diverse operazioni:

- Il salvataggio in locale del report appena acquisito

- L'invio del report ad un dispositivo vicino tramite comunicazione Bluetooth
- L'invio del report ad un server remoto dedicato all'elaborazione e presentazione dei dati

Per effettuare il salvataggio in locale dei dati è stata usata una libreria fornita da Google, chiamata *Gson*, che permette di fare una trasformazione di un oggetto Java in una stringa formato JSON e viceversa. In questo modo viene reso più facile il salvataggio dello stato dell'oggetto su un file (si salva direttamente la stringa corrispondente), permettendo di avere una maggiore interoperabilità tra piattaforme (essendo salvato in un formato standard) e facilitando eventualmente la rigenerazione a lato server, che a questo punto diventa automatica, dell'oggetto iniziale a partire dalla stringa JSON. Il salvataggio del file avviene all'interno della memoria SD esterna al dispositivo. Quindi è un requisito progettuale la presenza di un'area di memoria di massa esterna accessibile. Infatti, all'avvio dell'applicazione, vengono create le cartelle di interesse:

```
// create folder for reports in the external storage
    if (!new File(Environment.getExternalStorageDirectory().toString()
+ "/AndroidPhoneSignal").exists())
        new File(Environment.getExternalStorageDirectory().toString()
+ "/AndroidPhoneSignal").mkdir();
    if (!new File(Environment.getExternalStorageDirectory().toString()
+ "/AndroidPhoneSignal/myReport").exists())
        new File(Environment.getExternalStorageDirectory().toString()
+ "/AndroidPhoneSignal/myReport").mkdir();
```

In particolare vengono create le cartelle *AndroidPhoneSignal* e *AndroidPhoneSignal/myReport* a partire dalla directory root della memoria esterna. La prima cartella contiene tutti i report ricevuti da dispositivi esterni tramite comunicazione ad-hoc, mentre la seconda contiene esclusivamente i report effettuati dal dispositivo stesso (in caso non potesse trasferirlo ad un altro dispositivo o al server).

Il seguente codice presenta le modalità di conversione da oggetto Java a stringa JSON:

```
mySignalReport.setAvgSignalStrength(extras.getInt("SignalStrength"));
mySignalReport.setCellID(extras.getString("CellID"));
mySignalReport.setLatitude(extras.getDouble("Latitude"));
mySignalReport.setLightInfo(extras.getFloat("LightInfo"));
mySignalReport.setLongitude(extras.getDouble("Longitude"));
mySignalReport.setNetworkType(extras.getString("NetworkType"));
mySignalReport.setOperatorName(extras.getString("OperatorName"));
mySignalReport.setRoaming(extras.getBoolean("Roaming"));
mySignalReport.setLocationProvider(extras.getString("LocationProvider"));
mySignalReport.setTimestamp(extras.getLong("Timestamp"));

// convert SignalReportTo object to a Json string
jsonSignalReport = signalReportGson.toJson(mySignalReport);
```

Inizialmente si ricavano dal *Bundle*, trasferito dall'Activity di acquisizione, tutti i parametri e quindi si assegnano ad un oggetto Java di tipo *SignalReportTO* tramite i metodi setter. A questo punto viene convertito l'intero oggetto in stringa formato JSON tramite l'invocazione del metodo *toJson()* su un oggetto di tipo *Gson*. Il metodo prende in ingresso l'oggetto Java e restituisce in uscita la stringa corrispondente. Tale stringa poi viene salvata all'interno della memoria di massa in un file con nome pari al timestamp ricavato a fine acquisizione, nella cartella *AndroidPhoneSignal/myReport*. Per questione di comodità e facile riconoscimento di duplicati, ogni file di report ha estensione “.aps” e nome pari al timestamp ricavato. Come si può notare, le acquisizioni sono del tutto anonime, visto che non vengono ricavati dati sensibili o specifici, sia

dell'utente che del dispositivo mobile.

Un esempio di stringa codificata è il seguente:

```
{
  "operatorName": "vodafone IT",
  "cellID": "6701",
  "networkType": "UMTS",
  "lightInfo": 40.0,
  "longitude": 12.68985005,
  "latitude": 43.97991415,
  "avgSignalStrength": 11,
  "roaming": false
  "timestamp": 1326986383509
}
```

Tale stringa così formata può essere riconvertita in un oggetto Java tramite la funzione contraria *Gson.fromJson()* che accetta come parametro di ingresso la stringa ben formata. Di seguito viene mostrato un esempio di report grafico mostrato all'utente, unito al pop-up informativo scatenato all'atto del salvataggio in locale:

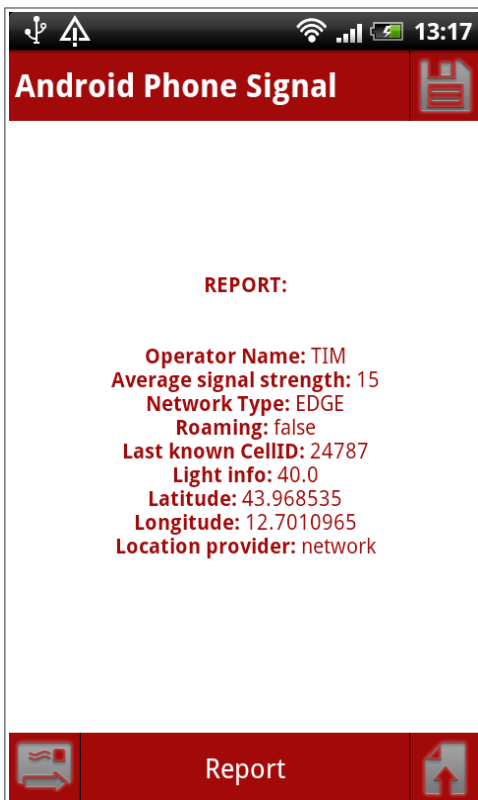


Illustrazione 10: Esempio di report finale

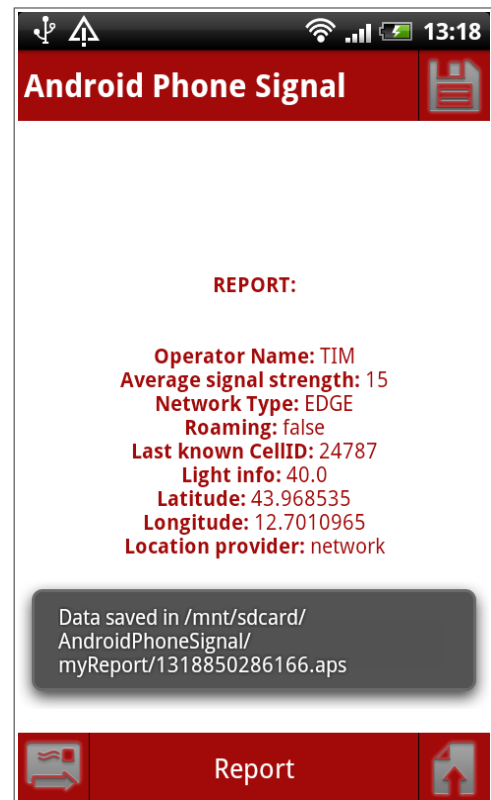


Illustrazione 11: Esempio di report finale con salvataggio in locale

Per quanto riguarda l'upload dei report all'interno di un server remoto, si è scelto di compiere tale operazione utilizzando le librerie *httpclient*, *httpcore* e *httpmime* fornite da Apache Software Foundation. In questo modo è possibile fare l'upload di report (singolo o multiplo) utilizzando il protocollo HTTP (semplificando quindi la configurazione di rete per quanto riguarda firewall e altri dispositivi intermedi) e fornendo al server una richiesta ben formata (utilizzando il metodo POST) che fa uso di oggetti *MutipartEntity*. In questo modo è necessario popolare la richiesta con le opportune informazioni (tra cui il nome del file, il contenuto e l'indirizzo del server) ed effettuare la

chiamata HTTP. Dall'altra parte il server è configurato per acquisire la richiesta, estrapolare tutte le informazioni necessarie, salvare i report in locale e quindi inviare una risposta al client in modo da riportare l'esito dell'operazione. Il seguente codice mostra la procedura utilizzata:

```

HttpResponse resp;
File f = new File(reportFileName);
HttpClient httpClient = new DefaultHttpClient();

httpClient.getParams().setParameter(HttpConnectionParams.CONNECTION_TIMEOUT, 10000);

httpClient.getParams().setParameter(HttpConnectionParams.SO_TIMEOUT, 15000);

httpClient.getParams().setParameter(CoreProtocolPNames.PROTOCOL_VERSION,
HttpVersion.HTTP_1_1);

HttpPost httpPost = new HttpPost("http://myService.net/SrvReport/");

MultipartEntity entity = new
MultipartEntity(HttpMultipartMode.BROWSER_COMPATIBLE);
entity.addPart("reportFile", new FileBody(f));
httpPost.setEntity(entity);
Toast.makeText(getApplicationContext(), "Sending...",
Toast.LENGTH_SHORT).show();
try {
    resp = httpClient.execute(httpPost);
    if (resp.getStatusLine().getStatusCode() == OK) {
        f.delete();
        reportSaved = false;
    }

    Toast.makeText(getApplicationContext(),
        resp.getStatusLine().getStatusCode()==OK?"File sent
        successfully!":"Transfer error. Retry!", Toast.LENGTH_LONG).show();
} catch (Exception e) {
    Toast.makeText(getApplicationContext(), e.getMessage() ,
    Toast.LENGTH_LONG).show();
    e.printStackTrace();
}

```

In particolare il codice mostra come impostare alcuni parametri, come il timeout per la connessione (per impostare il tempo massimo di attesa necessario sia per stabilire la connessione, sia per ricevere i dati) e la versione del protocollo. Successivamente si crea una richiesta *HttpPost* verso il server prestabilito. A questo punto si usa l'oggetto *MultipartEntity* e si imposta al proprio interno il contenuto del file da inviare. Il riconoscimento del payload viene effettuato tramite l'utilizzo di un tag (*reportFile*) utile per l'elaborazione lato server. Una volta impostata la *MultipartEntity* all'interno della richiesta, viene effettuata l'invocazione. A questo punto l'applicazione rimane in attesa del risultato determinato lato server in modo da compiere le successive operazioni in base al valore di quest'ultimo. Se il valore è "OK" (costante intera "200"), il trasferimento è andato a buon fine e il file del report viene cancellato dalla memoria di massa locale, altrimenti se è "FAIL" (costante intera "404") viene conservato per un invio successivo. Il metodo di invio multiplo è del tutto simile al metodo di invio singolo: nel primo caso, i vari report vengono aggiunti ad un archivio "zip" (il codice è mostrato nella parte successiva) in modo che il trasferimento finale sia sempre di un singolo file, in modo da ottimizzare le risorse usate per la connessione e l'elaborazione.

2.2.5 Trasferimento del report tramite Bluetooth

Il trasferimento del report tramite connessione ad-hoc viene effettuata nel caso in cui il dispositivo che ha compiuto l'acquisizione non abbia disponibilità di una connessione ad internet, sia per mancanza di un piano tariffario adeguato, sia per la mancata disponibilità di un Access Point (ad es. Wi-Fi) nelle vicinanze, sia per la mancanza del modulo fisico (sia 3G che Wi-Fi). Appena avviata l'Activity dedicata al trasferimento, se non attivo, viene indicato all'utente di attivare il modulo Bluetooth. Il codice per eseguire la procedura è il seguente:

```
// ad-hoc connection provided by Bluetooth
myBluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

if (!myBluetoothAdapter.isEnabled()) {
    Intent enableBtIntent = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
```

In particolare, si ottiene il riferimento all'adattatore Bluetooth presente nel dispositivo e lo si salva all'interno di *myBluetoothAdapter*. Successivamente si controlla se è abilitato e, in caso contrario, si utilizza un *Intent* per avviare l'Activity predefinita per l'abilitazione del modulo.



Illustrazione 12: Finestra di dialogo per l'abilitazione del modulo Bluetooth

L'immagine riporta esattamente la finestra di dialogo per l'attivazione del modulo Bluetooth. In particolare è presente, solo a fini di debug, un *Toast* che visualizza il contenuto del report in formato JSON. In caso di attivazione con successo, l'Activity (in questo caso, la *ListActivity*) presenta all'utente i dispositivi precedentemente accoppiati. Nel caso specifico, si ottengono i dispositivi già accoppiati iterando una lista di device ottenuta tramite la funzione *getBondedDevices()* e

aggiungendo i risultati ad una *ArrayAdapter* che poi la *ListView* utilizzerà come sorgente dati:

```
Set<BluetoothDevice> pairedDevices =
myBluetoothAdapter.getBondedDevices();
    myArrayAdapter = new
ArrayAdapter<String>(this,R.layout.list_item);
// Loop through paired devices
for (BluetoothDevice device : pairedDevices) {
// Add the name and address to an array adapter to show in a ListView
    myArrayAdapter.add(device.getName() + "\n" + device.getAddress());
}
l.setAdapter(myArrayAdapter);
```

A questo punto l'utente può scegliere diverse operazioni da compiere. Può rendere visibile il dispositivo ed effettuare un accoppiamento Bluetooth, può effettuare il discovery di dispositivi nelle vicinanze, può rendere il dispositivo un server per la ricezione di report da altri dispositivi e può inviare, dopo il collegamento con un altro device, il report appena creato oppure l'insieme dei report ricevuti da altri dispositivi.

Il discovery di dispositivi è implementato come segue:

```
// Create a BroadcastReceiver for ACTION_FOUND: a new Bluetooth
device has been discovered
mReceiver = new BroadcastReceiver() {
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        // When discovery finds a device
        if (BluetoothDevice.ACTION_FOUND.equals(action)) {
            // Get the BluetoothDevice object from the Intent
            BluetoothDevice device =
intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
            // Add the name and address to an array adapter to show
            // in a ListView and check if already present
            Toast.makeText(getApplicationContext(),
String.valueOf(myArrayAdapter.getPosition(device.getName() + "\n" +
device.getAddress())), Toast.LENGTH_LONG).show();
            if (myArrayAdapter.getPosition(device.getName() + "\n" +
device.getAddress())<0 && device.getName()!=null)
                myArrayAdapter.add(device.getName() + "\n" +
device.getAddress());
        }
    }
};
// behavior for the "discovery devices" button
((Button)findViewById(R.id.btnDiscovery)).setOnClickListener(new
View.OnClickListener() {
    @Override
    public void onClick(View v) {
        myBluetoothAdapter.startDiscovery();
    }
});
```

Per prima cosa si è registrato un *BroadcastReceiver* per l'azione *ACTION_FOUND*, invocata quando un nuovo dispositivo Bluetooth viene rilevato. Le azioni intraprese prevedono l'estrapolazione dell'oggetto rappresentante il dispositivo trovato tramite l'Intent e successivamente le informazioni relative, in particolare il nome simbolico e l'indirizzo MAC (quest'ultimo utile per

effettuare la connessione). Per avviare la procedura di discovery (di norma dura circa 12 secondi), viene invocata la funzione `startDiscovery()` sotto richiesta dell'utente. Un'altra opzione disponibile all'utilizzatore permette di rendere visibile il dispositivo per un numero limitato di secondi, in modo tale da essere riconosciuto in fase di discovery. L'implementazione è stata effettuata nel seguente modo:

```
// behavior for the "set discoverable" button
((Button)findViewById(R.id.btnSetDiscoverable)).setOnClickListener(new
View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Intent discoverableIntent = new
        Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
        // set the device discoverable for 300 seconds
discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION,
300);
        startActivity(discoverableIntent);
    }
});
```

La procedura avviene creando un *Intent* apposito per lanciare l'Activity di sistema che permette di impostare il device come rilevabile. L'azione da intraprendere è *BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE* con un valore extra da passare (*BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION*) che indica il numero massimo di secondi in cui il device deve rendersi rilevabile. Trascorsi questi secondi, il dispositivo risulta nascosto e quindi non più visibile per azioni di discovery da parte di altri terminali. Un'altra possibile azione che l'utente può compiere è quella di avviare un server per accettare richieste di connessione Bluetooth da parte di altri dispositivi. L'attivazione del server prevede l'avvio di un thread separato per la gestione delle richieste, in modo da non bloccare l'esecuzione della Activity:

```
while (true) {
    try {
        socket = mmServerSocket.accept();
        mySocket = socket;
        Log.d(TAG, "Connection accepted by server");
    } catch (IOException e) {
        break;
    }
    // If a connection was accepted
    if (socket != null) {
        // Do work to manage the connection (in a separate thread)
        ConnectedThread ct = new ConnectedThread(socket);
        ct.run();
        try {
            mmServerSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Il thread crea una *BluetoothServerSocket* che resta in ascolto di connessioni tramite il metodo *accept()*. Tale metodo è sincrono bloccante e attende fino all'arrivo di una richiesta di connessione. Una volta stabilita la connessione, viene ritornata la Socket utilizzabile per la comunicazione, che viene passata ad un altro thread incaricato al trasferimento dei dati tra i due dispositivi. Il thread di

accettazione delle richieste rimane attivo per sempre e rimane in attesa di nuove connessioni da parte di altri dispositivi. A questo punto il dispositivo Bluetooth cliente deve effettuare la connessione al server. Dopo aver lanciato la fase di discovery, l'utente può selezionare dalla lista il dispositivo server verso cui iniziare la connessione. Questo viene fatto premendo la voce relativa all'interno della ListView che contiene la lista completa dei device Bluetooth.

```
// Get a BluetoothSocket to connect with the given BluetoothDevice
try {
// UUID string for the application, also used by the server code
    tmp = device.createRfcommSocketToServiceRecord(uuid);
} catch (IOException e) { }
mmSocket = tmp;

// Connect the device through the socket. This will block
// until it succeeds or throws an exception
mmSocket.connect();
Toast.makeText(getApplicationContext(), "Connected to " +
mmDevice.getName(), Toast.LENGTH_LONG).show();
```

Il codice riportato, implementato all'interno di un thread, mostra come effettuare una connessione ad un server utilizzando un client. La prima operazione è quella di ricavare la Socket di comunicazione del dispositivo server, tramite la funzione *createRfcommSocketToServiceRecord()*, chiamata attraverso l'oggetto di tipo *BluetoothDevice* (che incapsula al suo interno l'indirizzo hardware del dispositivo server) e che accetta come parametro l'UUID dell'applicazione (che rimane costante per ogni istanza e installazione dell'applicazione). Una volta ricavata la Socket ci si connette attraverso la chiamata bloccante *connect()*. Alla prima connessione, in caso di dispositivi non ancora accoppiati, viene richiesto all'utente di confermare una passkey. A questo punto, a connessione ultimata, è possibile iniziare il trasferimento del singolo report o dei vari report. Nel caso del report singolo viene trasferito il singolo file corrispondente con successiva eliminazione dal dispositivo client a trasferimento avvenuto. A lato server il trasferimento viene gestito nella seguente maniera:

```
// case for single report
if (new String(command).compareTo("SNG") == 0)
    try {
        fos = new FileOutputStream(f);
        bytes = mmInStream.read(buffer);
        fos.write(buffer, 0, bytes);
        mmOutputStream.write("END".getBytes());
    }
}
```

Se il comando inviato dal client è quello di un singolo report ("SNG"), viene creato un nuovo file e al suo interno viene scritto il contenuto del buffer di ricezione. Al termine della procedura viene segnalato al client che i dati sono stati ricevuti tramite il messaggio di "END", messaggio che segnala al client la possibilità di chiudere la Socket di connessione. A lato client le azioni compiute sono le seguenti:

```
// send SiNGle report, the one just acquired
ct.write("SNG".getBytes());
ct.write(jsonSignalReport.getBytes());
try {
    mySocket.getInputStream().read(buffer, 0, buffer.length);
} catch (IOException e) { }
```

```

        e.printStackTrace();
    }
    if (new String(buffer).compareTo("END") == 0)
    {
        ct.cancel();

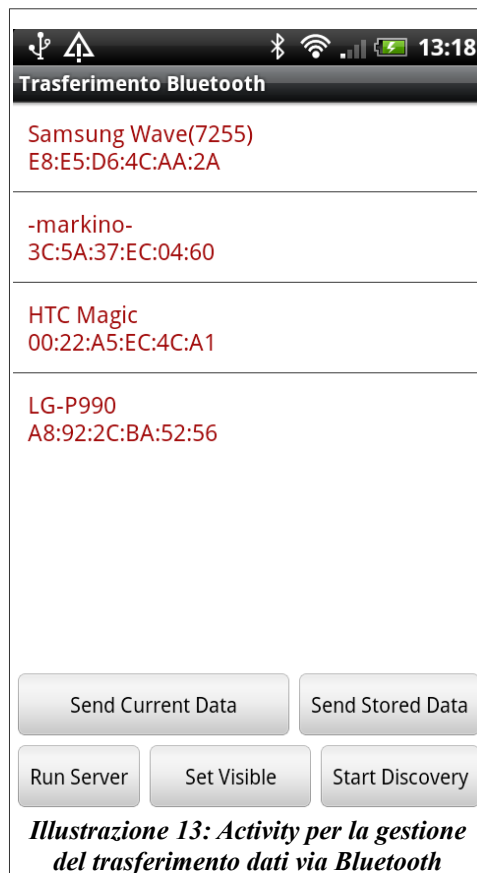
        try {
            mySocket.close();
            mySocket = null;
        }
    }
}

```

In questo caso il client invia il dato di controllo che informa il server che deve ricevere solo un report e successivamente invia la stringa JSON convertita in byte attraverso lo stream creato dalla Socket. Successivamente rimane in attesa del messaggio di avvenuta ricezione (“END”) con conseguente chiusura della connessione in corso.

Il caso di invio multiplo è analogo all'invio singolo: si è scelto di inviare comunque un solo file per ottimizzare la comunicazione. Il file comprende tutti i report, sia quelli ricavati dal dispositivo in oggetto, sia quelli ricevuti da altri dispositivi. Per effettuare questa operazione, si è scelto di utilizzare un archivio di tipo “ZIP” che potesse contenere tutti i file. Quindi si è compiuta una fase di creazione ed invio dell'archivio lato client e conseguente ricezione ed espansione dell'archivio lato server.

Di seguito viene mostrato uno screenshot dell'Activity relativa al trasferimento via Bluetooth:



3. Valutazione delle performance

Per determinare il grado di qualità dell'applicazione e per valutarne l'impatto che produce su un terminale mobile è necessario definire alcuni indicatori di performance significativi. La decisione si è focalizzata principalmente su due parametri fondamentali: carico di CPU e occupazione di memoria centrale (RAM).

3.1 Configurazione e modalità di test

La configurazione utilizzata per i benchmark è la seguente:

- Smartphone HTC Desire (CPU Qualcomm QSD 8250 Snapdragon da 1 GHz, 576 MB RAM, Android 2.3.3) fornito con i seguenti moduli e sensori: scheda microSD da 4 GB, GPS, modulo 3G, modulo Bluetooth, sensore di luminosità ambientale, sensore di prossimità, sensore di accelerazione, sensore di campo magnetico. Utilizzato come terminale mobile adibito ad eseguire l'applicazione.
- Notebook Acer Aspire 5553G equipaggiato con processore AMD Phenom II X4 N930 (quad-core, 2 GHz), 4 GB di RAM DDR3 e sistema operativo Ubuntu Linux 11.10 64-bit. Utilizzato per l'esecuzione dei test e la raccolta dei dati sfruttando Android SDK.
- Android SDK r15 per sistemi Linux che ha fornito gli strumenti di debug e di accesso remoto al terminale.
- Gnuplot per Linux. Utilizzato per mostrare su grafico i dati ricavati.

La raccolta dei dati è avvenuta compiendo 3 campionamenti consecutivi per ogni singola funzionalità fornita dall'applicazione: acquisizione delle informazioni sul segnale telefonico, acquisizione delle informazioni di posizionamento, acquisizione delle informazioni di luminosità, trasferimento di un singolo report (parte client e parte server) e trasferimento multiplo di report (sia client che server). Per ricavare le informazioni riguardanti il carico di CPU e occupazione di RAM è stato utilizzato, durante l'esecuzione di una determinato test, il seguente comando su bash:

```
~$ ./adb shell top -d 0.1 -n X | grep com.androidphonesignal
```

In questo modo è stato possibile avere un report filtrato invocando, tramite terminale esterno, il comando “top” seguito da “grep” direttamente all'interno dello smartphone, con redirectione dell'output nel terminale di invocazione (nel caso particolare, quello presente all'interno del sistema operativo del notebook). Il parametro “-d” ha permesso di impostare un tempo di refresh dei dati ad un valore predefinito, in questo caso 0.1 secondi. Invece il parametro “-n” ha permesso di impostare il numero di acquisizioni successive intervallate di 0.1 secondi. In questo caso “X” indica una variabile (di tipo numerico), dipendente dal tipo di test. Ad esempio l'acquisizione della potenza del segnale ha impiegato più tempo rispetto all'acquisizione dell'informazione sulla luce ambientale e quindi la taratura di tale parametro è stata importante per definire una finestra temporale sensata per ogni funzionalità. Successivamente i dati ricavati sono stati ulteriormente filtrati per avere solo i valori di interesse, per poi essere elaborati da Gnuplot e ricavare un grafico significativo con i valori per ciascuna delle tre prove, insieme alla media calcolata come:

$$\text{Average} = (\Sigma(\text{ist_value_1} + \text{ist_value_2} + \text{ist_value_3}) / 3) / X$$

definita sull'intervallo [0..X-1] con passo 0.1.

Dove:

- **ist_value_{1,2,3}**: valori istantanei al tempo t (con passo di 0.1 secondi) per ogni prova;
- **X**: numero di acquisizioni totali;

Di seguito verranno illustrati i risultati sperimentali ottenuti nelle varie prove, effettuate come descritto precedentemente.

3.2 Acquisizione della potenza del segnale telefonico

Il rilevamento della potenza del segnale telefonico ha previsto 170 dump dello stato dell'applicazione per avere informazioni rilevanti sul consumo di CPU e RAM. Le tre prove sono state effettuate facendo partire l'applicazione e, successivamente, compiendo 3 acquisizioni consecutive.

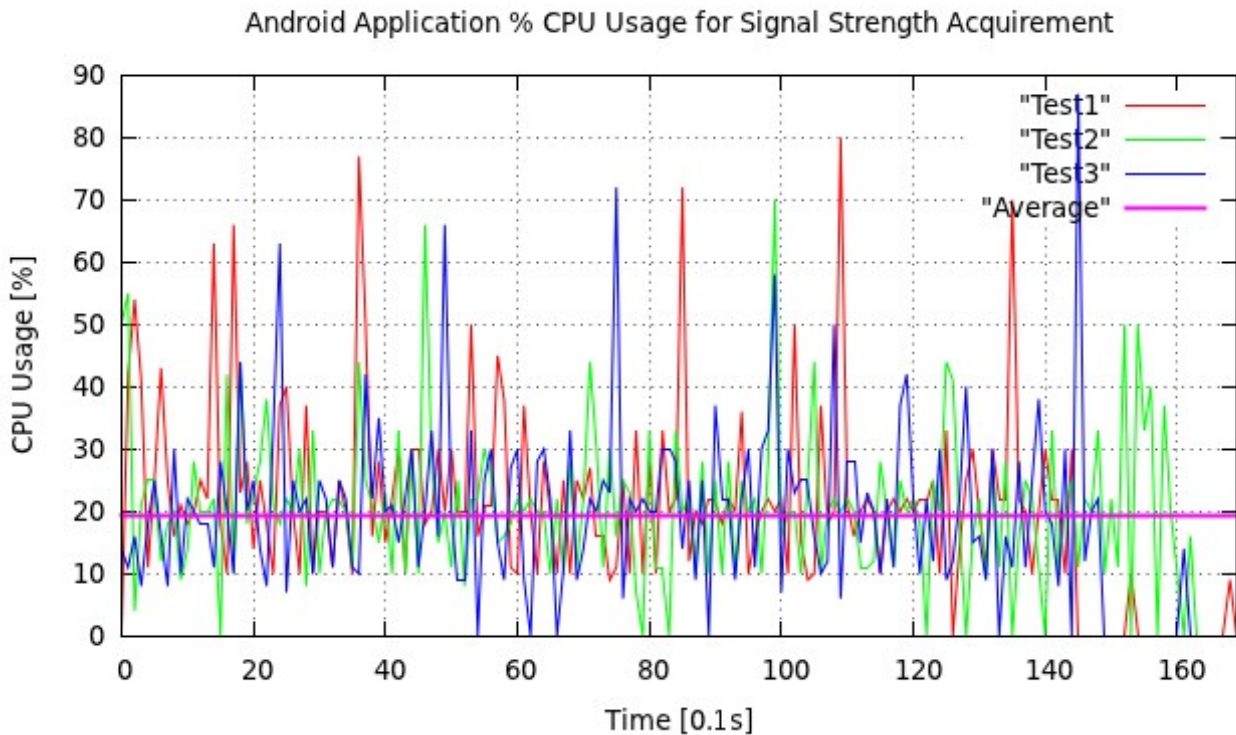


Illustrazione 14: Utilizzo della CPU nell'acquisizione del segnale telefonico

Il grafico sopra riportato mostra l'andamento dell'utilizzo di CPU (in percentuale) nelle varie prove, includendo anche il valore medio. In particolare si può notare che all'incirca tutte le curve relative ai test hanno un andamento simile e la media di carico si attesta circa sul 19,4%. Tale valore può essere giustificato grazie alla presenza dell'animazione a video che indica lo stato dell'acquisizione e per l'utilizzo del sensore di orientamento in maniera intensiva. Infatti ad ogni variazione di valore necessariamente viene effettuata un'operazione CPU-bound. Si noti inoltre che il trend delle curve non varia, indipendentemente dalla prova: la curva rossa denota la prima prova che, eseguita ad applicazione inizializzata, potrebbe far ipotizzare andamenti particolari con picchi anomali dato che il carico computazionale in questa fase può essere dedicato all'inizializzazione di strutture dati ausiliarie, strutture di conseguenza non più inizializzate nelle prove successive.

Il seguente grafico invece mostra l'andamento del consumo di memoria RAM:

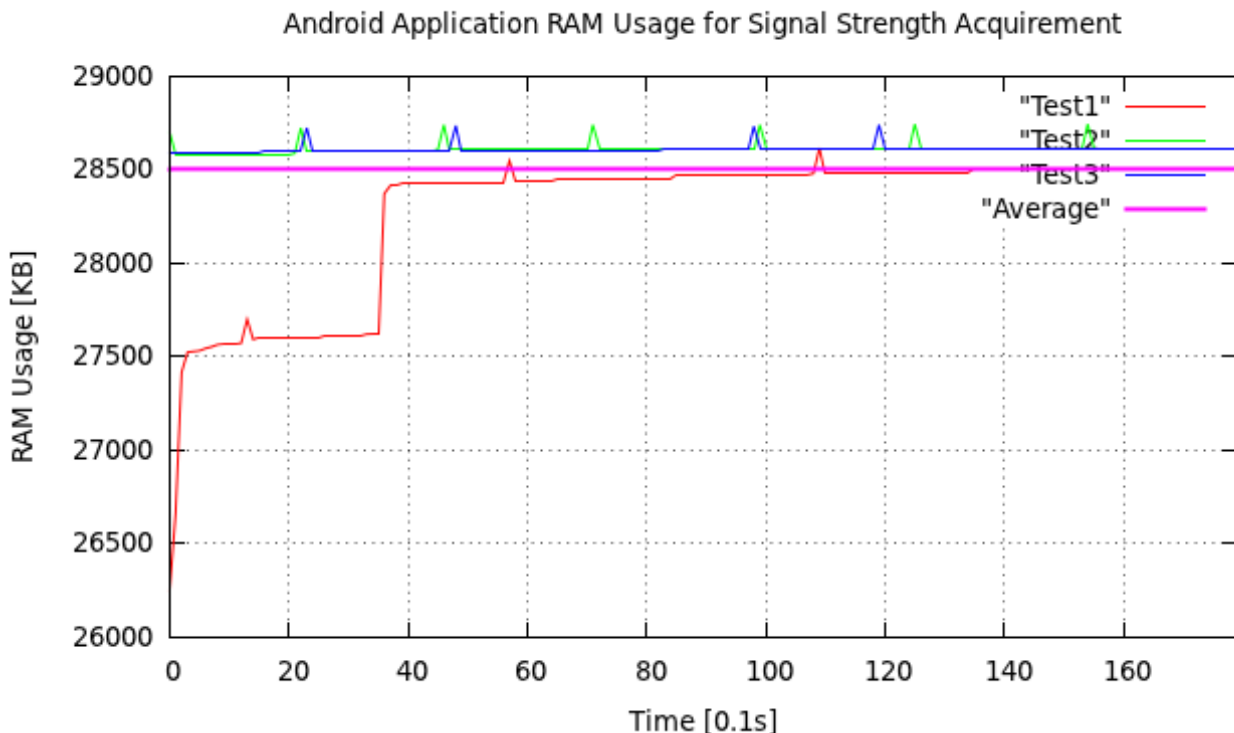


Illustrazione 15: Utilizzo della RAM nell'acquisizione del segnale telefonico

Per quanto riguarda invece il consumo di memoria RAM, si può notare che la prima prova effettuata ad applicazione appena avviata differisce rispetto alle altre. Infatti, se i valori di CPU sono in linea con quelli degli ultimi due test, in questo caso il primo test prevede una fase iniziale di caricamento delle strutture dati in memoria, per poi allinearsi all'incirca con i valori misurati nelle due prove seguenti. L'occupazione media di memoria è di circa 28.500 KB. Dal grafico si denota inoltre che in alcuni intervalli sono presenti dei picchi. Questo vale per tutte le prove effettuate e in particolare si verificano anche nella fase stabile di occupazione, presente nel primo test. Tale andamento può essere causato dalla variazione della potenza segnale telefonico. Infatti, appena avviene tale evento, viene richiamato un handler dal sistema che esegue alcune operazioni, tra le quali la presentazione a video dei dati aggiornati. La gestione di tali strutture dati può creare dei picchi (seppur piccoli) di carico e scarico della memoria RAM, fino ad arrivare alla fase finale dell'acquisizione dove si de-registra il listener e si pone fine al campionamento della variazione del segnale.

3.3 Acquisizione delle informazioni di luminosità ambientale

Per riuscire ad avere un numero sufficiente di rilevazioni in modo da determinare valori ragionevoli per studiare il comportamento di questo tipo di funzionalità è stato necessario effettuare 40 dump dello stato dell'applicazione. In particolare, come nel caso precedente, il primo test è stato effettuato ad applicazione appena avviata, mentre i due successivi sono stati consecutivi alla prima prova.

Il grafico che rappresenta l'andamento di carico della CPU è il seguente:

3. Valutazione delle performance

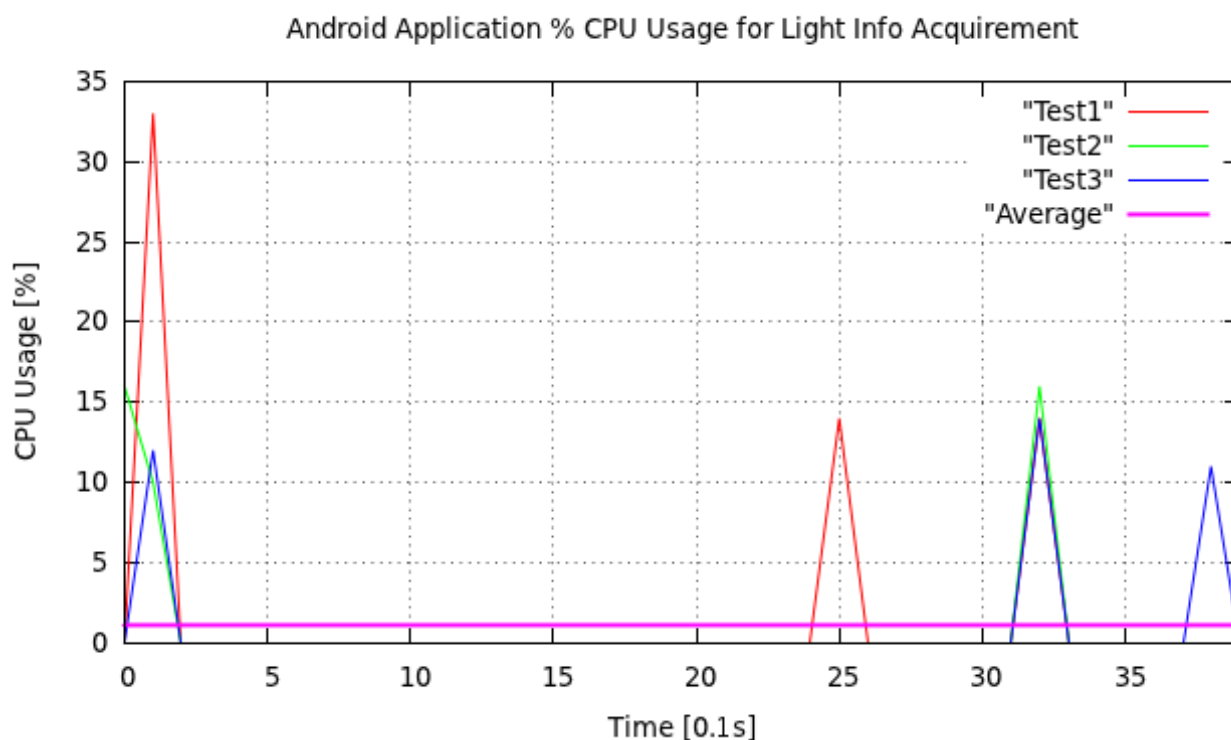


Illustrazione 16: Utilizzo della CPU nell'acquisizione dell'informazione sulla luminosità ambientale

Anche in questo caso l'andamento del carico di CPU è simile per tutte le prove effettuate. Come sempre è normale aspettarsi dei picchi di ampiezza maggiore nella prima parte del primo test, visto che è comunque presente una fase di preparazione e inizializzazione delle strutture dati che prendono parte all'elaborazione. L'andamento del carico della memoria centrale è il seguente:

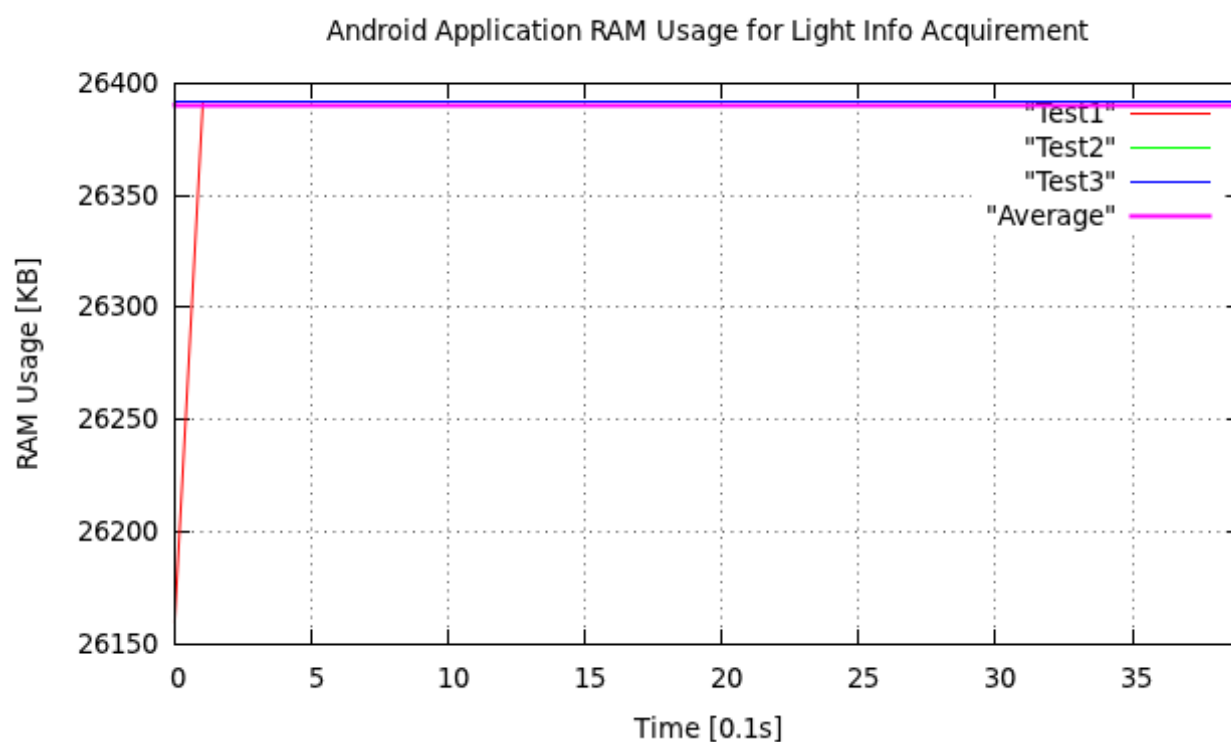


Illustrazione 17: Utilizzo della RAM nell'acquisizione dell'informazione di luminosità ambientale

3. Valutazione delle performance

Anche in questo caso è presente un transitorio iniziale nella prima parte del primo test. Alla partenza dell'acquisizione, la memoria viene occupata con le strutture dati necessarie fino ad arrivare a regime con un valore che si mantiene costante per tutta la durata della prova. In generale questa procedura occupa poche risorse, sia in termini di consumo medio di CPU, che di consumo di memoria. Infatti il consumo medio di CPU è stato del 1,1% e il consumo medio di memoria dell'applicazione è stato di 26.390 KB, con una variazione di 250 KB verificatasi all'avvio della prima acquisizione.

3.4 Acquisizione delle informazioni di locazione

I test per stabilire le performance della funzionalità di acquisizione della locazione sono stati divisi in due categorie. Le modalità di esecuzione sono state le medesime, ma la prima fase ha previsto l'utilizzo della rete telefonica come vettore di informazioni di posizionamento, mentre la seconda fase ha visto il modulo GPS per ottenere i medesimi dati in forma più accurata.

3.4.1 Uso della rete telefonica cellulare

Per riuscire ad avere dei dati significativi e in modo tale che si riuscisse a coprire l'intero intervallo di esecuzione della funzionalità, sono stati effettuati 110 dump dello stato dell'applicazione. Il grafico che mostra il consumo di CPU è il seguente:

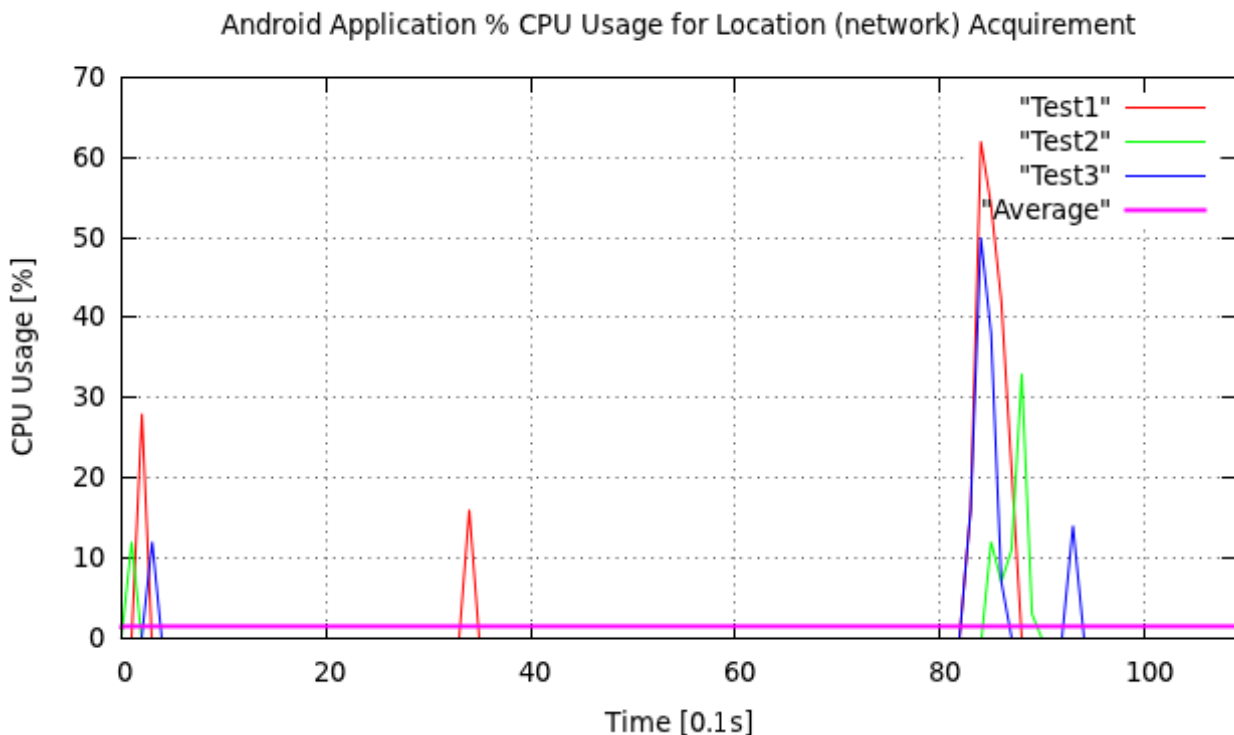


Illustrazione 18: Utilizzo della CPU nell'acquisizione della posizione tramite rete cellulare

Il carico di CPU anche in questo caso è basso per tutte e tre le prove effettuate. In particolare si possono trovare dei picchi più accentuati nella prima prova, accettabili vista l'inizializzazione delle strutture dati che l'applicazione deve compiere. Il valore medio di utilizzo del processore a partire dall'avvio dell'acquisizione fino al raggiungimento del risultato è di circa 1,34%. Il valore può dimostrare che la procedura prevede un impegno minimo delle risorse computazionali fornite.

Di seguito invece è riportato il grafico che mostra il consumo di memoria centrale:

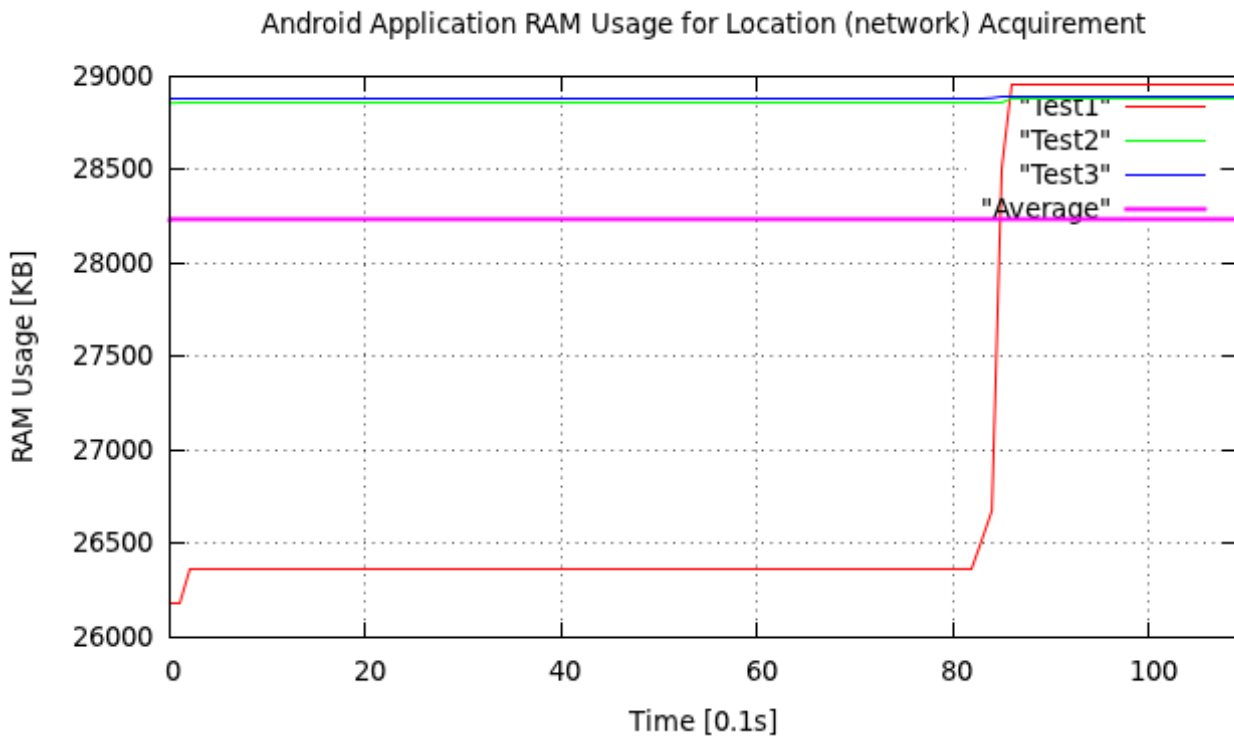


Illustrazione 19: Utilizzo della RAM nell'acquisizione della posizione tramite rete cellulare

L'andamento del consumo di memoria centrale è ben distinto per quanto riguarda la prima prova e le restanti due. Infatti, nel primo test, il quantitativo di memoria centrale utilizzata si mantiene costante a circa 26.400 KB fino al raggiungimento del risultato e la visualizzazione a video della locazione sulla mappa, dove il valore incrementa in maniera quasi istantanea fino ad allinearsi ai dati ricavati per le due acquisizioni successive. In questo caso il valore medio di RAM utilizzata dall'applicazione è di circa 28.230 KB.

3.4.2 Uso del GPS

In questo caso la localizzazione è stata gestita tramite l'uso del modulo GPS integrato all'interno del dispositivo. L'acquisizione è avvenuta, per motivi legati alla tecnologia e al funzionamento del GPS, all'aperto e senza ostacoli interposti tra il dispositivo e il cielo. Siccome il dispositivo è fornito di modulo A-GPS, è stato possibile agganciare facilmente e in tempi piuttosto rapidi il segnale proveniente dai satelliti grazie all'interazione iniziale con le celle telefoniche visibili. In questo modo lo smartphone ha ottenuto un accesso veloce alla lista dei satelliti visibili e disponibili per il collegamento, in modo da abbattere i costi di ricerca, sia in termini di tempo che in termini di consumo di energia. L'uso di tale procedura è stata necessaria per avere delle misure coerenti in uno spazio di acquisizione comune a tutte le prove effettuate, in modo da prelevare dei dati comparabili e ragionevoli. Quindi è da ricordare che il primo test eseguito ad applicazione appena avviata comprende anche questa fase di "fixing" dei satelliti, azione non più necessaria per i due test successivi.

Il grafico che rappresenta il carico di CPU è il seguente:

3. Valutazione delle performance

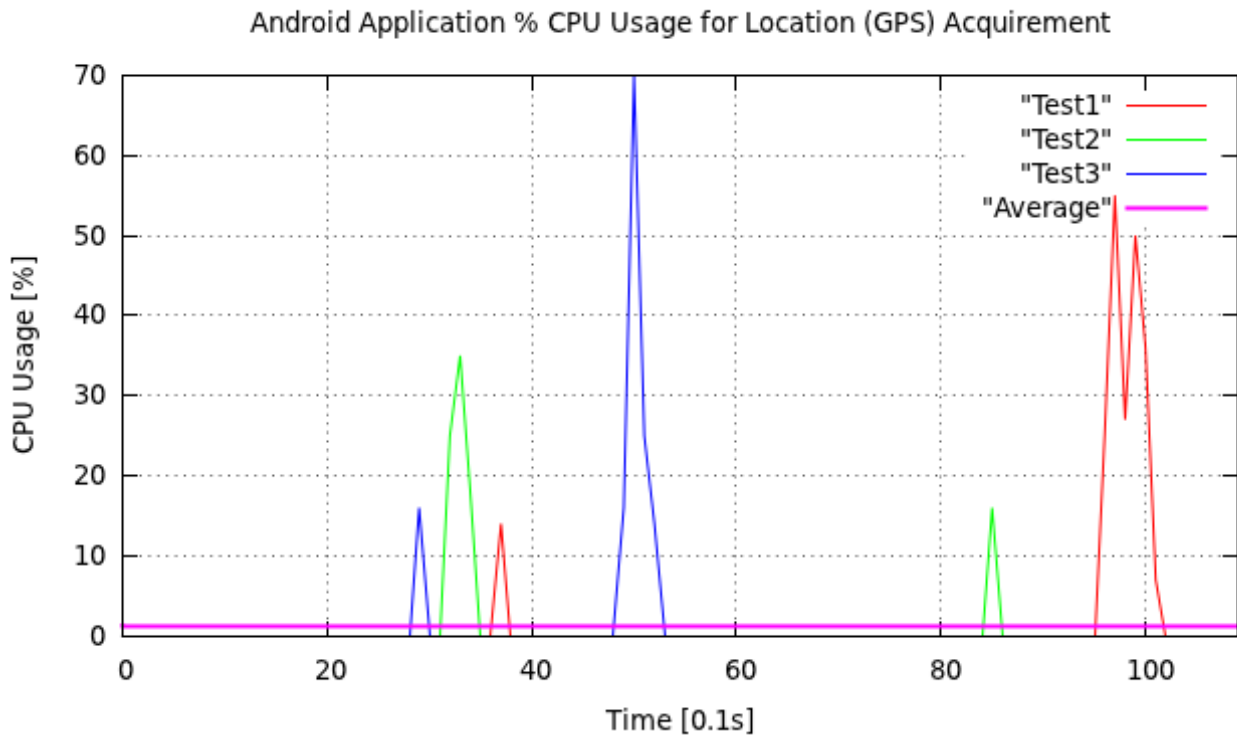


Illustrazione 20: Utilizzo della CPU nell'acquisizione della posizione tramite GPS

Così come l'acquisizione della posizione tramite rete cellulare, l'occupazione di CPU per questa funzionalità è molto bassa rispetto, ad esempio, al campionamento del segnale telefonico. A meno di comportamenti anomali (come mostra il Test3 alla 50esima acquisizione), il grafico rispecchia all'incirca il comportamento ricavato nei test con la rete telefonica mobile.

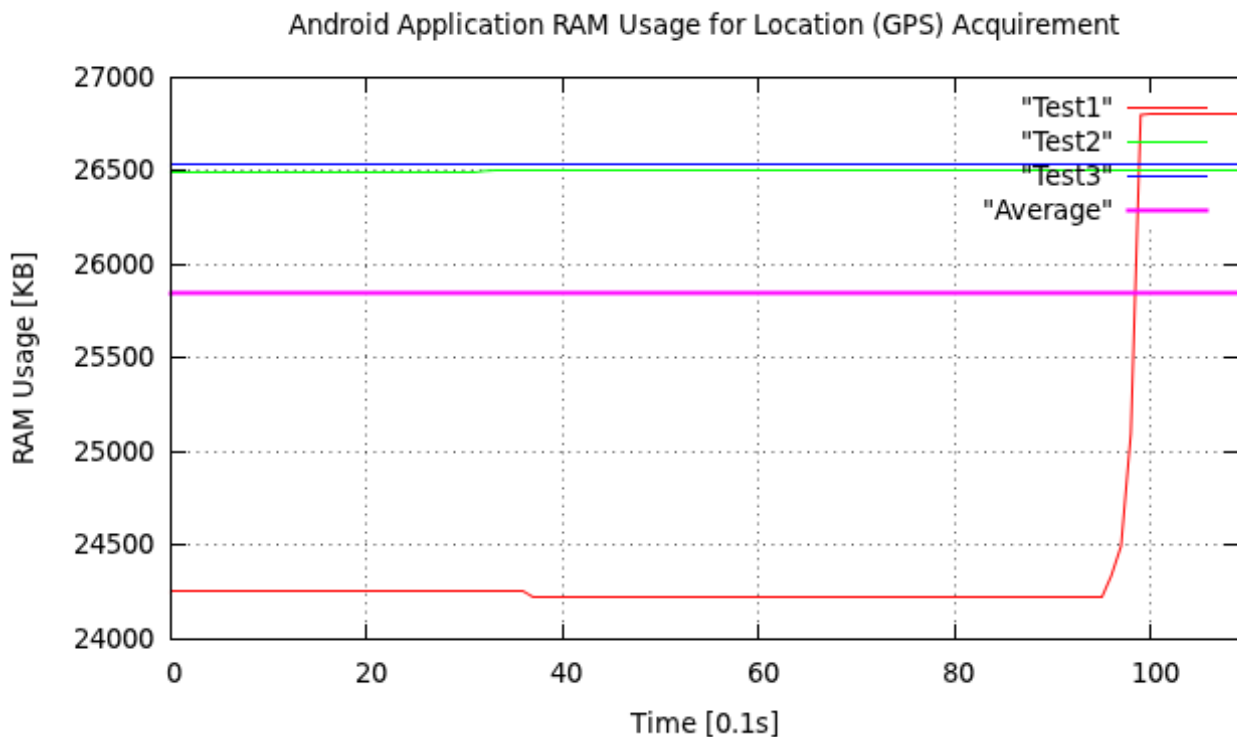


Illustrazione 21: Utilizzo della RAM nell'acquisizione della posizione tramite GPS

Il grafico mostra l'utilizzo della memoria di sistema durante l'acquisizione tramite modulo aggiuntivo di posizionamento. Anche in questo caso l'andamento è paragonabile a quello visto utilizzando la rete cellulare: nella prima prova il consumo rimane costante fino al raggiungimento del risultato, punto dove successivamente subisce un incremento esponenziale. Nelle prove seguenti il valore finale ricavato nella prima prova rimane costante per tutta la durata dell'acquisizione (presumibilmente per i medesimi motivi illustrati nel caso precedente). Anche questa modalità ha un impatto basso a livello di risorse di computazione. Infatti il carico medio di CPU rilevato è stato del 1,31%, mentre il consumo medio di RAM si è attestato sui 25.845 KB.

3.5 Trasferimento di report singolo tramite connessione Bluetooth

I test per verificare le prestazioni dell'applicazione per quanto riguarda il trasferimento di un singolo report tramite la connessione ad-hoc sono stati divisi in due categorie, vista la duplice azione che il dispositivo può compiere: client nel caso in cui debba inviare i propri report e server nel caso in cui debba fare da raccogliitore di dati per altri dispositivi. Di seguito verranno illustrati i dati ricavati nelle modalità indicate.

3.5.1 Uso del client Bluetooth

Nel primo caso è stato utilizzato il dispositivo come client. Quindi è stata effettuata un'acquisizione completa di prova e successivamente, dopo aver attivato il modulo Bluetooth, si è effettuata la connessione al dispositivo server con conseguente invio del report generato in memoria. Le prove considerano il tempo trascorso dalla fase di connessione al server all'invio completo dei dati. I valori di carico di CPU ricavati sono i seguenti:

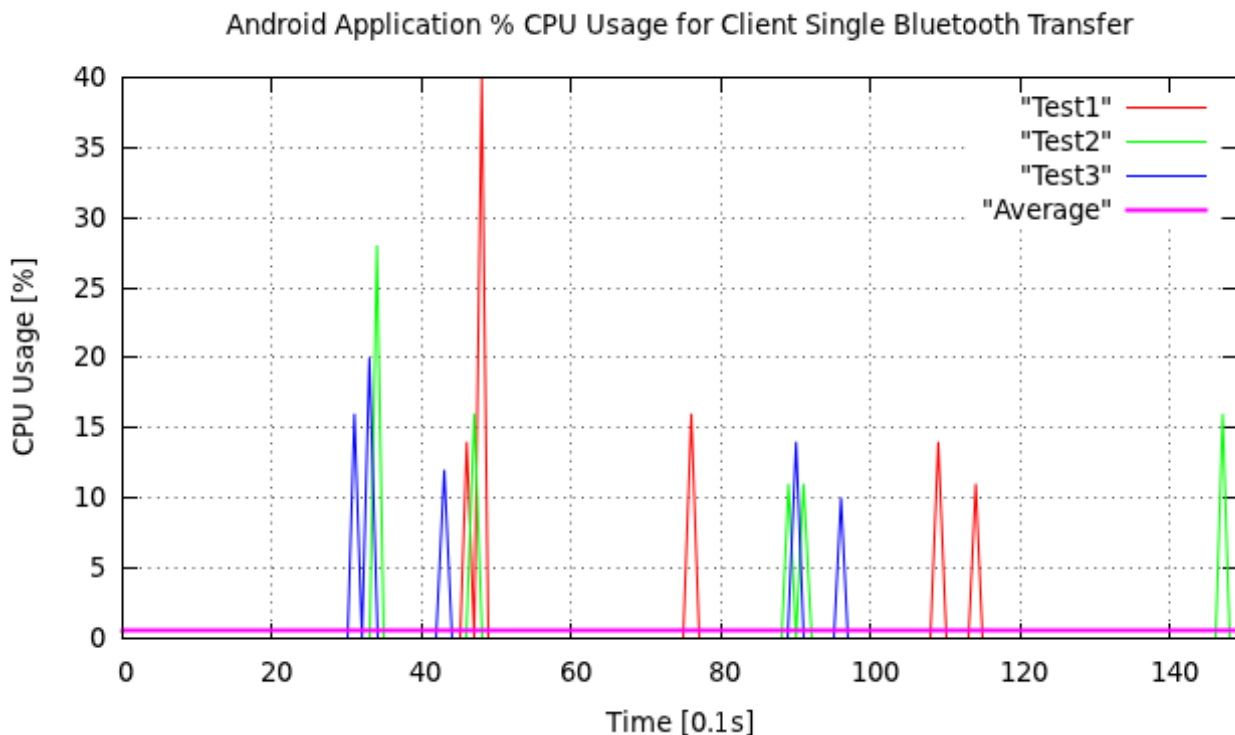


Illustrazione 22: Utilizzo della CPU nel trasferimento di singolo report via Bluetooth: parte client

I valori riportati nel grafico denotano che, mediamente, il carico è simile per tutte e tre le prove

3. Valutazione delle performance

effettuate nonostante i picchi, presenti nel primo test, superiori rispetto alle altre rilevazioni eseguite in modo consecutivo e senza riavviare l'applicazione. La ragione può essere l'inizializzazione delle strutture dati dedicate al trasferimento. Il carico medio di CPU è di circa 0,50%, valore che dimostra un basso impatto della funzionalità all'interno del sistema. Per quanto riguarda il quantitativo di memoria centrale utilizzata, il seguente grafico mostra i risultati sperimentali ottenuti nel corso delle tre prove citate precedentemente:

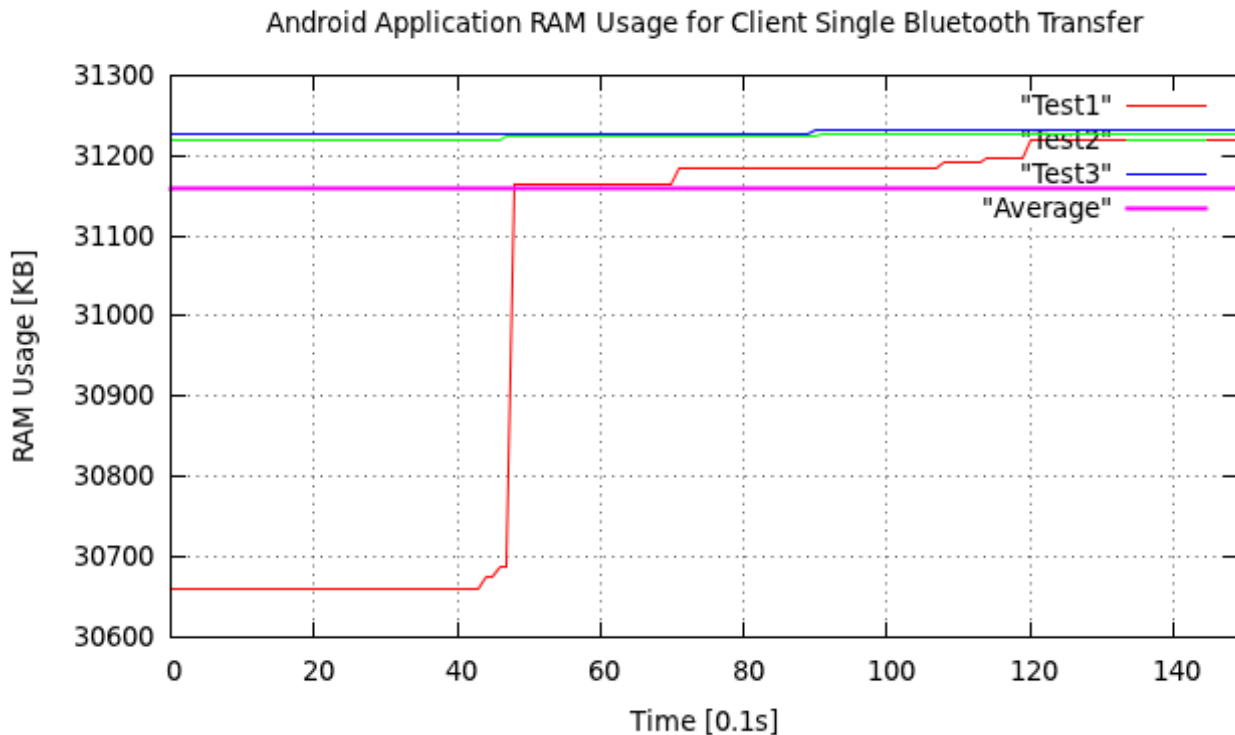


Illustrazione 23: Utilizzo della RAM nel trasferimento di singolo report via Bluetooth: parte client

L'occupazione di memoria centrale subisce un transitorio nella parte iniziale del primo test effettuato. Infatti, come indicato anche nel grafico precedente, in corrispondenza del picco di allocazione di memoria è presente anche il picco di occupazione di CPU, ad ulteriore conferma del fatto che sia dovuto all'aggiunta in RAM delle strutture dati necessarie alla connessione e al trasferimento dei dati tramite connessione Bluetooth. La variazione di occupazione di RAM si attesta all'incirca sui 500 KB. Dopo il picco, l'occupazione si stabilizza con più armonia verso il valore massimo stabilito, di circa 31.220 KB. Le due prove effettuate successivamente registrano un'occupazione costante di memoria, che si attesta al valore massimo raggiunto dal primo test. Infatti in questi ultimi due casi tutto l'occorrente è già caricato in memoria, senza la necessità di impegnare risorse computazionali aggiuntive per assolvere questo compito. Il consumo medio di memoria centrale si attesta a circa 31.160 KB, valore accettabile dato che, dall'apertura dell'applicazione, sono state effettuate tutte le acquisizioni necessarie per generare un report valido per il trasferimento remoto.

3.5.2 Uso del server Bluetooth

In questo caso è stato utilizzato il dispositivo mobile come server Bluetooth e quindi in grado di ricevere connessioni da altri dispositivi e immagazzinare il report da trasferire. La rilevazione delle performance parte dal momento in cui si attiva il server al momento in cui si riceve correttamente il flusso di dati che contiene il report, con relativo salvataggio del file all'interno della memoria di massa esterna. I valori di carico di CPU rilevati sono presentati nel seguente grafico:

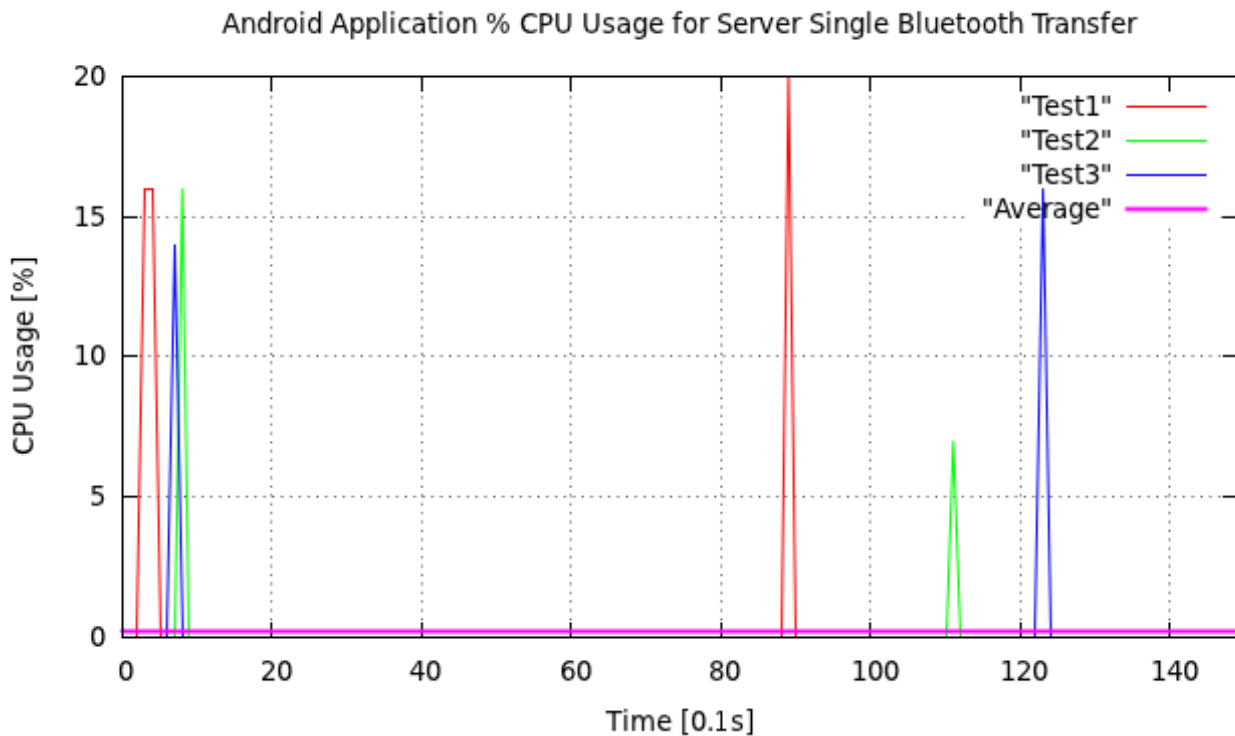


Illustrazione 24: Utilizzo della CPU nel trasferimento di singolo report via Bluetooth: parte server

Il grafico mostra un andamento simile per tutte e tre le prove effettuate. Nella prima parte infatti si verifica un picco dovuto all'abilitazione del thread adibito all'ascolto sulla Socket di accettazione di connessioni esterne. La seconda parte invece evidenzia la fase di connessione vera e propria e di trasferimento del singolo report. Quest'ultimo è lo stesso medesimo per tutte le prove (circa 200 byte), in modo da rimuovere una variabile in gioco, che può verificarsi nel caso di report di dimensione non fissa. Anche in questo caso il picco rilevato nella prima prova quindi può considerarsi un'elaborazione aggiuntiva, a differenza delle restanti due prove, per inizializzare e allocare in memoria le strutture dati necessarie per gestire il trasferimento e il salvataggio dei dati. Nonostante la connessione Bluetooth stabilita e il trasferimento del quantitativo di byte sopra indicato, il carico medio di CPU si è attestato circa allo 0,22%, con un valore massimo registrato nel primo test che ha riportato una percentuale di impegno pari al 20%. Questo dimostra che, anche se il dispositivo agisce da server, il consumo medio di risorse computazionali è irrisorio, a totale beneficio dell'autonomia e della reattività visibile dall'utilizzatore.

I valori di occupazione di memoria centrale registrati sono i seguenti:

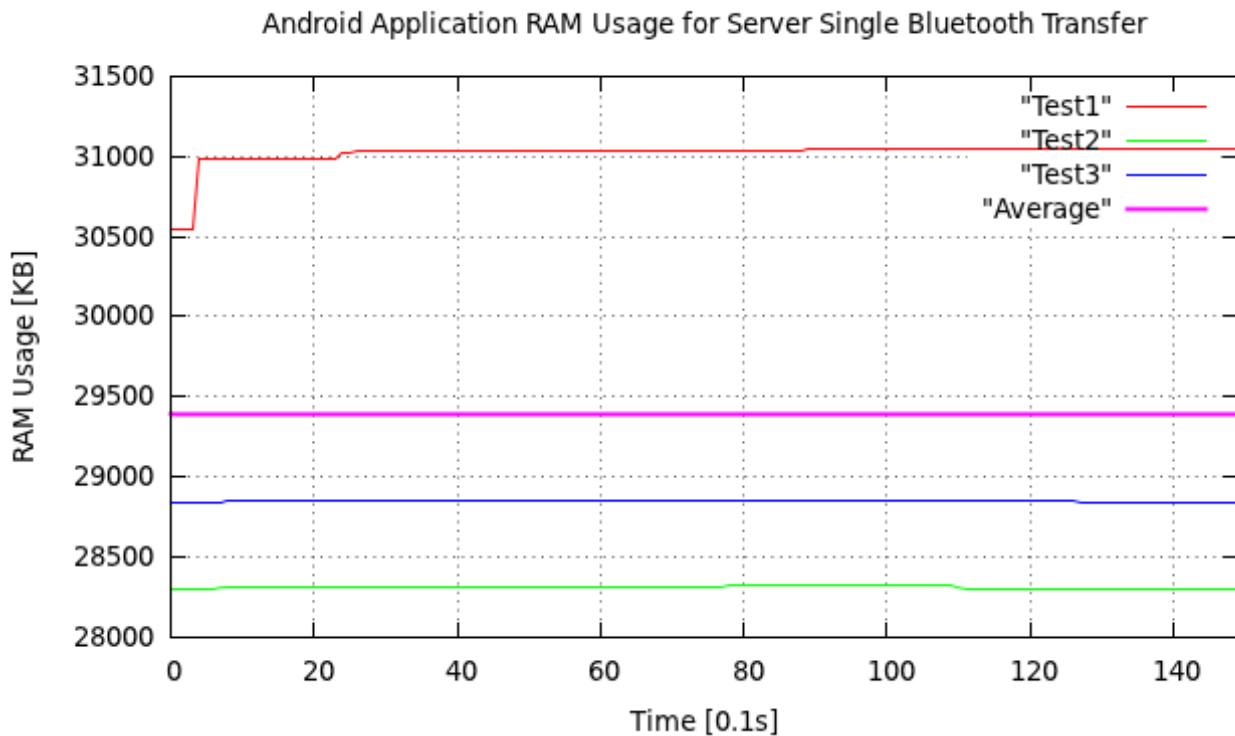


Illustrazione 25: Utilizzo della RAM nel trasferimento di singolo report via Bluetooth: parte server

Il grafico sopra riportato mostra un'anomalia nella prima prova effettuata: l'occupazione di memoria è maggiore rispetto alle due prove consecutive. Il motivo è molto probabilmente riconducibile all'attivazione del Garbage Collector che potrebbe aver ridimensionato il valore di memoria occupata de-allocando risorse non più utilizzate nel sistema. Purtroppo questo fenomeno non è direttamente controllabile in fase di esecuzione in modalità "release", ma comunque il trend delle curve offre un panorama significativo del comportamento dell'applicazione nel caso operi come server Bluetooth. All'inizio del primo test infatti è evidente un aumento della memoria occupata, dovuto all'attivazione del thread di ascolto per nuove connessioni e alla creazione delle strutture dati atte ad acquisire il dato. Successivamente si presenta una situazione costante di occupazione, situazione che si verifica fin dall'inizio per le successive due prove effettuate. Il valore medio di occupazione di memoria si attesta sui 23.400 KB, valore fortemente influenzato dalla de-allocazione, da parte del sistema, di risorse non più utilizzate che ha alterato i valori acquisiti per la seconda e la terza prova.

3.6 Trasferimento di report multiplo tramite connessione Bluetooth

Anche in questo caso il trasferimento multiplo di report è stato suddiviso in due parti distinte: la parte client con funzionalità di connessione al server, compressione dei report disponibili nella memoria di massa esterna e invio dell'archivio generato e la parte server, con funzione di accettazione delle connessioni, ricezione dell'archivio contenente i report e decompressione di quest'ultimo con conseguente salvataggio nella memoria di massa esterna locale al dispositivo.

3.6.1 Uso del client Bluetooth

La modalità di test effettuata è molto simile a quella vista in precedenza con la trasmissione di un singolo report tramite connessione ad-hoc. In questo caso il client, oltre ad effettuare la connessione al server ed inviare i dati, effettua una compressione dei file e archivia in un unico file tutti i report presenti all'interno della memoria di massa esterna. In pratica effettua una lettura di tutte le acquisizioni presenti (sia effettuate dal dispositivo che ricevute da altri dispositivi) e crea al volo l'archivio che le contiene. Successivamente invia tale file e, a trasferimento completato, cancella i dati inviati dalla memoria di massa locale. Il seguente grafico mostra l'andamento del carico di CPU:

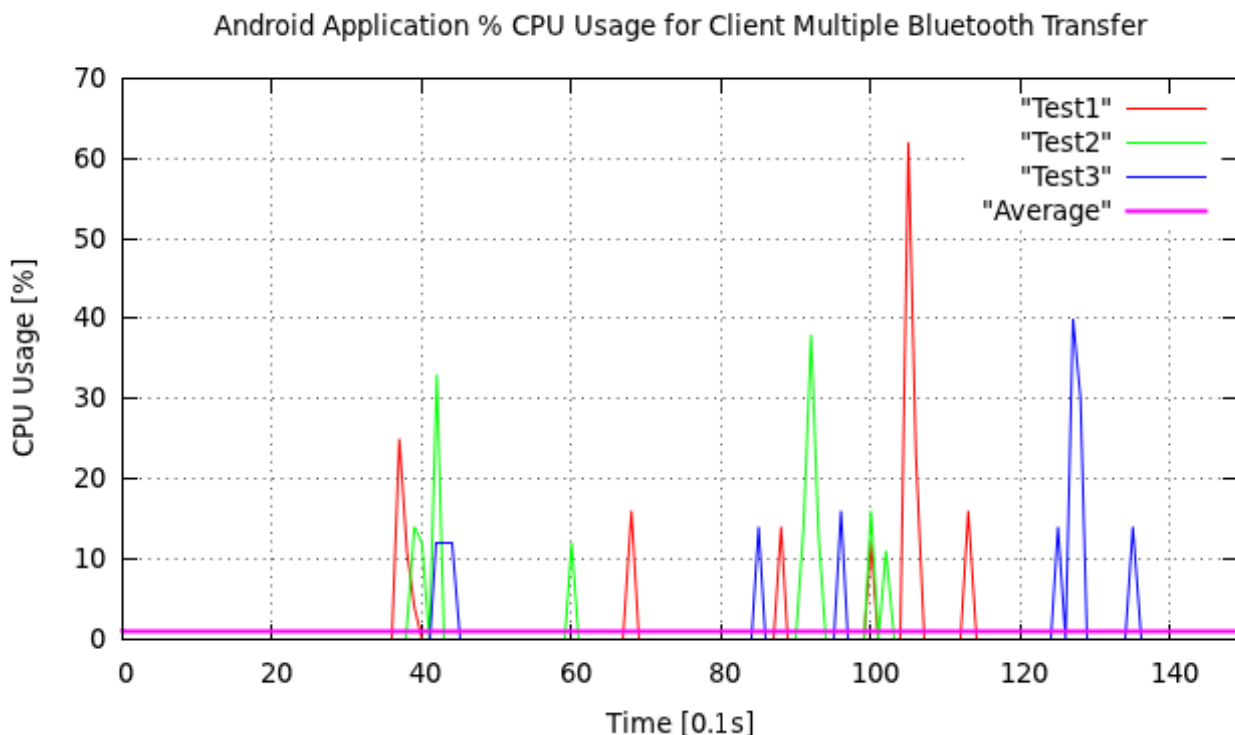


Illustrazione 26: Utilizzo della CPU nel trasferimento di report multiplo via Bluetooth: parte client

Il grafico mostra l'andamento del carico di processore nelle diverse fasi. La prima parte, nell'intorno del 40esimo campionamento, definisce la fase di connessione al server Bluetooth. I valori sono pressoché identici per tutte le prove effettuate. Successivamente i picchi più alti si verificano prima nella fase di archiviazione dei report e poi nell'invio di tale file tramite la comunicazione ad-hoc instaurata nella prima fase. In questo caso non si verifica il comportamento dell'”avvio a freddo” che ha caratterizzato la maggior parte dei test presentati in precedenza. Infatti il primo test, nella fase iniziale, mostra un andamento molto simile alle due prove seguenti, senza mostrare gli effetti dell'inizializzazione delle strutture dati necessarie al completamento delle azioni. La stessa cosa si presenta anche nelle fasi successive, dove tutti e tre i test si comportano circa nella medesima maniera. Il carico computazionale medio è circa del 1,07%, un valore molto basso, ottimo per quanto riguarda il consumo energetico del terminale.

Di seguito viene mostrato il grafico dell'occupazione di memoria RAM:

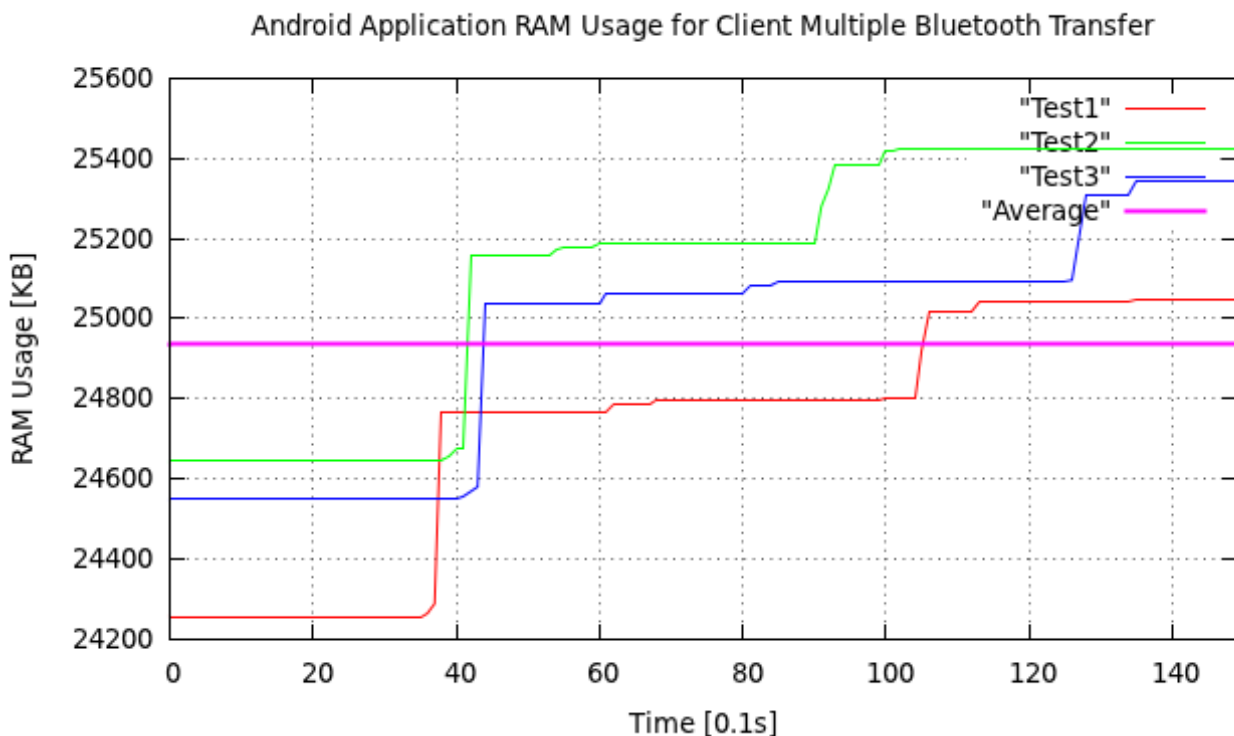


Illustrazione 27: Utilizzo della RAM nel trasferimento di report multiplo via Bluetooth: parte client

L'andamento del consumo di memoria centrale è molto simile per tutte e tre le prove effettuate. In particolare nella prima si ha, in generale, una minore occupazione di memoria dovuta all'esclusiva presenza di alcune strutture dati di base, che poi verranno riutilizzate anche nelle prove successive. Il test ha visto l'invio dello stesso tipo e numero di report (10) per tutte le prove, quindi i risultati ottenuti sono assolutamente comparabili in termini di carico computazionale e di occupazione di memoria. In corrispondenza dei picchi presenti nel grafico del carico di CPU si verifica simultaneamente anche un aumento dell'occupazione di memoria centrale. Da ciò si può dedurre che, in quelle fasi, il processore è stato dedicato all'allocazione di nuove risorse in RAM, risorse poi utilizzate per creare l'archivio dei report e per l'invio dei dati tramite Bluetooth. Il carico medio di RAM utilizzato dall'applicazione, in questo caso, si attesta sui 24.900 KB.

3.6.2 Uso del server Bluetooth

In questa ultima fase di test è stato utilizzato il terminale come server Bluetooth per la ricezione dei report da dispositivi situati nelle vicinanze. In questo caso il server è predisposto per accettare un volume di report indefinito. Ipoteticamente, è possibile inviare anche un solo report in questa modalità. In questo caso però la procedura viene integrata con la generazione di un archivio “zip”, sicuramente non efficiente dal punto di vista computazionale rispetto al caso di invio di report singolo senza compressione dati. La modalità di test è simile a quella attuata per il trasferimento di un singolo report. La procedura prevede l'avvio del server, l'accettazione della connessione, la ricezione dei dati e la decompressione dell'archivio nella cartella apposita situata nella memoria di massa esterna.

Il grafico del carico sul processore è il seguente:

3. Valutazione delle performance

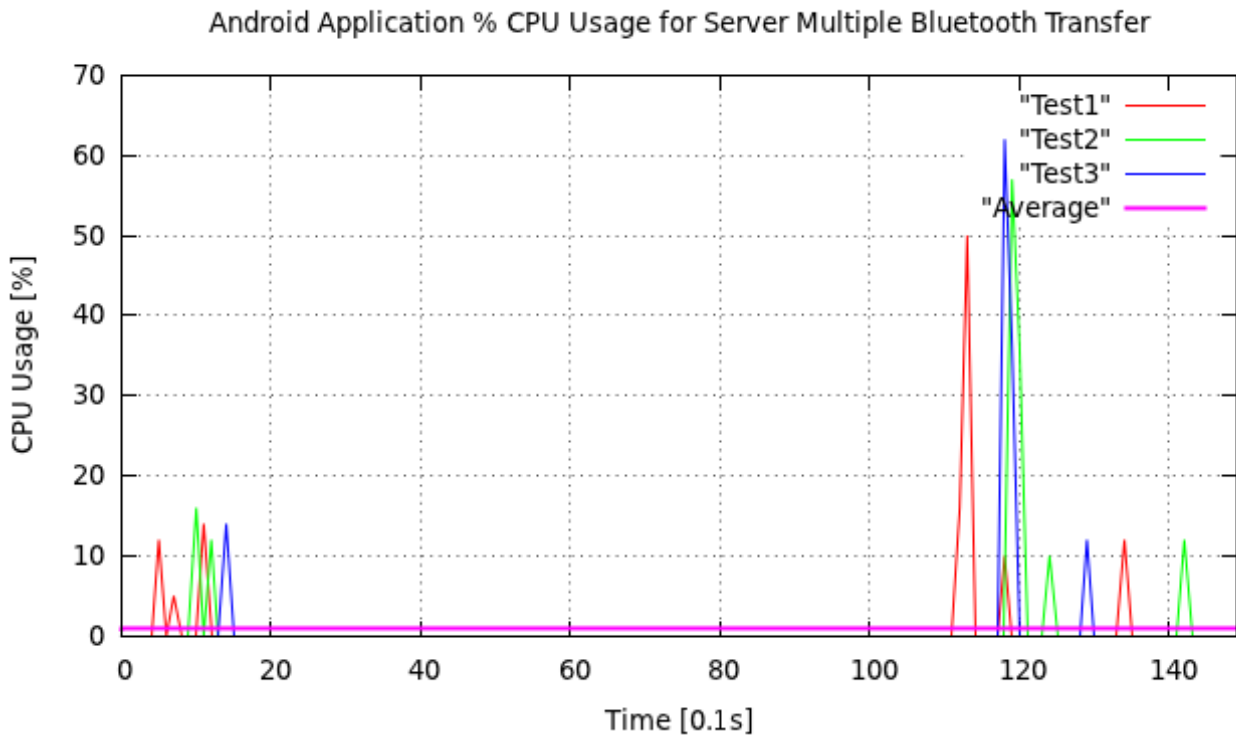


Illustrazione 28: Utilizzo della CPU nel trasferimento di report multiplo via Bluetooth: parte server

Anche in questo caso il grafico delinea le varie fasi illustrate precedentemente. La prima parte mostra l'elaborazione effettuata per stabilire la connessione, mentre la seconda parte del grafico mostra la ricezione e decompressione dell'archivio "zip" inviato. Il consumo di memoria RAM è invece riportato nel seguente grafico:

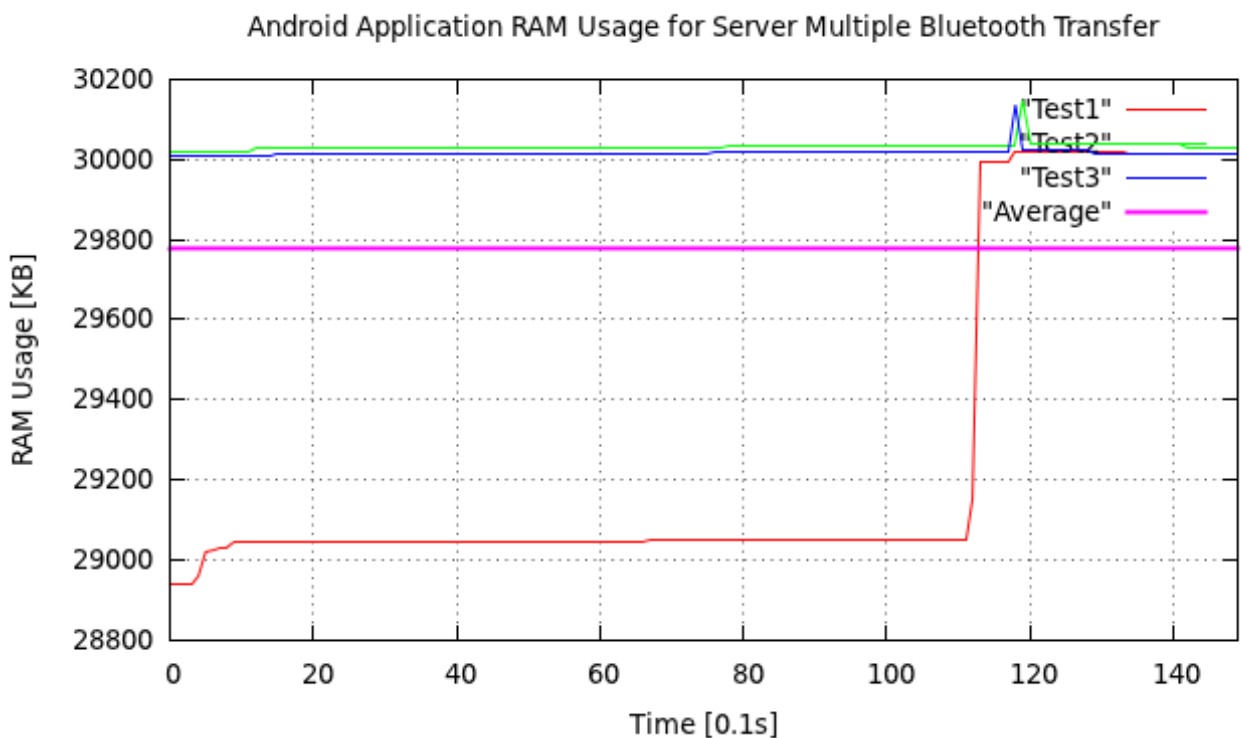


Illustrazione 29: Utilizzo della RAM nel trasferimento di report multiplo via Bluetooth: parte server

3. Valutazione delle performance

In questo grafico si denota, nel primo test, l'allocazione iniziale delle strutture dati per gestire il trasferimento dell'archivio e, successivamente, per eseguire la decompressione dello stesso su memoria di massa. Il valore finale di allocazione raggiunto rimane circa costante per le due prove successive effettuate. Rispetto al caso del singolo report, l'occupazione di CPU media è aumentata fino a circa lo 0,83%, valore comunque basso e accettabile per l'utilizzo in un dispositivo mobile con limitate risorse energetiche. Per quanto riguarda invece il consumo medio di memoria, esso si attesta sui 29.770 KB, un valore di nuovo accettabile paragonandolo con il caso di invio singolo di report, considerando il fatto che la differenza media di occupazione è minima (circa 300 KB).

4. Gestione ed elaborazione dei report lato server

Questo capitolo introduce la seconda parte del progetto e illustra le modalità di realizzazione dell'applicazione complementare a quella mobile per Android. In particolare verrà trattata la soluzione lato server per il trasferimento, la gestione e l'elaborazione dei vari report facendo uso di tecnologie a livello Web ed enterprise per la presentazione, l'elaborazione, il caching e la persistenza dei dati. La scelta della tecnologia e degli strumenti è fondamentale per la buona riuscita e riutilizzo di un progetto di tipo enterprise e per tale ragione si è deciso di utilizzare strumenti totalmente open-source, in modo da poter avere una totale trasparenza e controllo sia sull'applicazione che sul contenitore nel quale l'applicazione andrà ad eseguire, nel rispetto delle specifiche. Di seguito verrà data una breve panoramica delle tecnologie enterprise utilizzate con una particolare attenzione a RESTful Web Services, tecnologia molto usata in ambito Internet per l'interazione con risorse lato server.

4.1 J2EE Container: JBoss

L'orientamento progettuale si è diretto principalmente verso tecnologie open-source con a capo una community molto forte e collaborativa come quella Java e per questo motivo ci si è orientati verso le specifiche J2EE (quindi, Java-based) per modellare l'applicazione. Tra i tanti vendor di container che implementano le specifiche Java 2 Enterprise Edition ci si è focalizzati verso JBoss Application Server, che ha la caratteristica di essere totalmente open-source e supportato ottimamente da una comunità attiva di sviluppatori e professionisti. JBoss AS è correntemente sviluppato da Red Hat ed essendo totalmente basato su Java, è possibile farne un'installazione all'interno di un qualsiasi sistema operativo che dispone di una Java Virtual Machine, definendo un ambiente cross-platform. Di seguito sono elencate alcune delle caratteristiche del container:

- Clustering
- Supporto ad Aspect-Oriented Programming (AOP)
- Deployment API
- Distributed caching (usando JBoss Cache, un prodotto standalone)
- Distributed deployment (farming)
- Enterprise JavaBeans versione 3 e 2.1
- Failover (incluse le sessioni)
- Hibernate (per gestire la persistenza; Java Persistence API o JPA)
- Java Authentication and Authorization Service (JAAS)
- Java EE Connector Architecture (JCA)
- Java Management Extensions
- Java Server Pages (JSP) / Java Servlet 2.1/2.5 (Tomcat)
- JBossWS (JBoss Web Services) per Java EE Web Services come JAX-WS
- JDBC
- Load balancing
- Management API
- OSGi framework
- RMI-IIOP (JacORB, acronimo di “Java and CORBA”)
- SOAP con Attachments API for Java (SAAJ)
- Teiid data virtualization system

Di tutti questi moduli solo alcuni sono stati realmente utilizzati all'interno del progetto, in modo tale che possano interagire tra loro per fornire globalmente il servizio richiesto. In particolare è stato utilizzato: Enterprise JavaBean 3.x, Hibernate (JPA), JCA, JSP e Java Servlet 2.x (Tomcat).

4.2 Enterprise JavaBeans

Per Enterprise JavaBeans (EJB) si intende un componente managed server-side utilizzato per la costruzione modulare di architetture per applicazioni enterprise. La specifica EJB è una delle tante Java API all'interno della specifica Java EE. In pratica EJB è inteso come un modello server-side che incapsula la logica di business di una applicazione. La specifica EJB è stata originariamente sviluppata da IBM e successivamente è stata adottata da Sun Microsystems (EJB 1.0 e 1.1) nel 1999 e migliorata tramite il Java Community Process con JSR 19 (EJB 2.0), JSR 153 (EJB 2.1), JSR 220 (EJB 3.0) e JSR 318 (EJB 3.1). Le specifiche EJB intendono fornire un modo standard per implementare il codice business del back-end che tipicamente risiede nelle applicazioni enterprise. Questo tipo di codice risolve, nella maggior parte dei casi, sempre gli stessi tipi di problemi, e le soluzioni ad essi sono spesso ripetutamente re-implementate dai programmatori (con possibilità di introdurre errori o di scrivere codice non ottimizzato). In questo modo gli Enterprise JavaBeans sono stati ideati per gestire tipi di problemi comuni come persistenza, integrità transazionale e sicurezza in un modo standard, lasciando liberi i programmatori di concentrarsi sul particolare problema da risolvere.

Le specifiche EJB quindi dettagliano come un application server fornisce:

- Transaction processing
- Integrazione con i servizi di Persistence offerti da Java Persistence API (JPA)
- Concurrency control
- Eventi utilizzando Java Message Service
- Naming e directory services (JNDI)
- Sicurezza (Java Cryptography Extension (JCE) e JAAS)
- Deployment di componenti software in un application server
- Remote procedure call usando RMI-IIOP
- Esposizione di metodi business come Web Services
- Asynchronous Method Invocation
- Timer service

In aggiunta a quanto scritto sopra, la specifica Enterprise JavaBean definisce il ruolo ricoperto dall'EJB container e gli EJB stessi, così come la procedura per effettuare il deployment di un EJB all'interno di un container. Da precisare è il fatto che la specifica EJB 3.1 (corrente) non descrive come un application server fornisce servizi di persistenza (in questo caso, è delegato alle specifiche JPA), ma invece delinea come la business logic può essere facilmente integrata con i servizi di persistenza offerti dall'application server stesso. Per effettuare il deploy e l'esecuzione di EJB può essere utilizzato un Java EE Application server (come JBoss) che include al proprio interno un container EJB a default. Alternativamente sono comunque disponibili container standalone (come OpenEJB) che offrono funzionalità di gestione degli EJB senza l'utilizzo di un application server vero e proprio.

4.2.1 Tipi di Enterprise JavaBean

Un EJB container può gestire due tipi principali di bean:

- **Session Bean** che può essere *Stateful*, *Stateless* o *Singleton* e può essere acceduto sia tramite una interfaccia *Local* (stessa JVM) o *Remote* (diversa JVM) o direttamente senza l'utilizzo di interfacce (in questo caso si utilizza una semantica locale). Tutti i Session Bean supportano l'esecuzione asincrona per ogni tipologia di semantica;
- **Message Driven Bean** (MDB o Message Bean). Anche gli MDB supportano un modello di esecuzione asincrona, ma tramite un paradigma a messaging.

4.2.1.1 Session Bean

4.2.1.1.1 Stateful Session Bean

Gli *Stateful Session Bean* sono componenti business che hanno stato: tengono traccia del cliente con cui stanno dialogando attraverso una sessione e l'accesso all'istanza del bean è strettamente limitato ad un client alla volta. In caso che l'accesso ad un singolo bean sia effettuato in modo concorrente, il container serializza le richieste dei clienti a meno che non venga utilizzata l'annotazione `@AccessTimeout` che permette al container di lanciare un'eccezione ai client dopo un timeout predefinito. Lo stato degli *Stateful Session Bean* può essere reso persistente (passivazione) automaticamente dal container per liberare memoria di sistema nel caso in cui il cliente non abbia fatto l'accesso al bean per un determinato lasso di tempo. La *JPA extended persistence context* è esplicitamente supportata dagli *Stateful Session Bean*.

4.2.1.1.2 Stateless Session Bean

Gli *Stateless Session Bean* sono componenti business che non hanno uno stato associato. Tuttavia, l'accesso ad un singolo bean è ancora limitato ad un client alla volta e l'accesso concorrente al bean non è consentito. In caso di accesso concorrente ad un singolo bean, il container semplicemente instrada ogni singola richiesta ad un'istanza diversa del bean. Questo approccio rende automaticamente thread-safe uno *Stateless Session Bean*. Le variabili di istanza possono essere usate all'interno di un *SSB* durante una chiamata ad un singolo metodo da parte di un client, ma non è garantito che il contenuto di tali istanze sia preservato attraverso diverse chiamate a metodi del bean provenienti dallo stesso client. Le istanze di *Stateless Session Bean* sono tipicamente pooled. Tuttavia se un secondo client effettua l'accesso ad un bean specifico subito dopo la fine dell'accesso di un primo client, è possibile che il secondo client possa ottenere subito la stessa istanza che ha servito il primo client. La mancanza di overhead ottenuto dall'inutilizzo di una sessione rende questi componenti meno resource-intensive rispetto agli *Stateful Session Bean*.

4.2.1.1.3 Singleton Session Bean

I *Singleton Session Bean* sono componenti business che hanno uno stato globale condiviso all'interno della JVM. L'accesso concorrente all'unica e sola istanza di bean presente può essere controllata dal container (Container-managed concurrency, CMC) o dal bean stesso (Bean-managed concurrency, BMC). CMC può essere attivata usando l'annotazione `@Lock`, che indica se un lock in lettura o in scrittura dovrà essere usato per una chiamata ad un determinato metodo. In aggiunta, i *Singleton Session Bean* possono essere esplicitamente istanziati all'avvio dell'EJB container, utilizzando l'annotazione `@Startup`.

4.2.1.2 Message Driven Bean

I *Message Driven Bean* sono componenti business le quali esecuzioni sono innescate da messaggi invece che da chiamate a metodi. Il Message Driven Bean è usato insieme agli altri per fornire un'astrazione di alto livello e di facile utilizzo per la specifica di livello più basso Java Message Service (JMS). Esso può iscriversi alle notifiche di code o topic JMS tramite l'attributo *activationConfig* dell'annotazione *@MessageDriven*. Questa tipologia è stata aggiunta nella specifica EJB per avere un meccanismo di processamento event-driven. Al contrario del Session Bean, un MDB non ha una vista per il cliente (Local/Remote/Senza Interfaccia) e quindi i clienti non possono effettuare il lookup per un'istanza di MDB. L'unico compito è ascoltare i messaggi su una coda o topic JMS e processarli automaticamente. La specifica Java EE richiede solo il supporto a JMS, ma i Message Driven Bean possono supportare altri protocolli di messaging, i quali possono essere sia asincroni che sincroni. Siccome i Session Bean possono anche essere sincroni o asincroni, la differenza sostanziale tra un SB e un MDB non è l'asincronicità, ma la differenza tra la chiamata a metodo (object oriented) e il messaging.

4.2.1.3 Entity Bean

Versioni precedenti di EJB usavano anche un tipo di bean chiamato Entity Bean. Questi erano oggetti distribuiti aventi uno stato persistente. I bean nei quali il container era il gestore della persistenza erano detti utilizzare *Container-Managed Persistence* (CMP), mentre i bean che autogestivano il loro stato venivano descritti come utilizzatori di *Bean-Managed Persistence* (BMP). In EJB 3.0 gli Entity Bean sono stati sostituiti con Java Persistence API, che è completamente separata dalla specifica, per permettere alla specifica EJB di concentrarsi solo sui Session Bean, Message Driven Bean e sulle API mostrate al cliente. Gli Entity Bean sono comunque ancora disponibili in EJB 3.1 per retrocompatibilità, anche se è stata inoltrata ufficialmente una proposta di rimozione della specifica, tramite un processo chiamato "pruning".

Altri tipi di Enterprise Bean sono stati proposti, per esempio gli *Enterprise Media Bean* (JSR 86) sono dedicati all'integrazione di oggetti multimediali all'interno delle applicazioni Java EE.

4.3 RESTful Web Services

REpresentational State Transfer (REST) è uno stile architetturale di software per sistemi ipermediali distribuiti come il World Wide Web. Il termine "*REpresentational State Transfer*" è stato introdotto e definito nell'anno 2000 da Roy Fielding nel suo documento di tesi per il dottorato. Le architetture REST-style sono costituite da client e server. I client inoltrano richieste verso i server; i server processano la richiesta e ritornano le risposte opportunamente elaborate. Le richieste e le risposte sono costruite attorno al trasferimento della rappresentazione delle risorse. Una risorsa può essere essenzialmente un qualsiasi concetto dotato di coerenza e significato che può essere trasmesso. Una rappresentazione di una risorsa invece è di solito un documento che cattura lo stato corrente di una risorsa all'atto dell'elaborazione della richiesta. In questo modo il client inizia l'invio della richiesta quando è pronto ad effettuare la transizione verso un nuovo stato. Mentre una o più richieste sono in sospeso, il cliente viene considerato in transizione. La rappresentazione di ogni stato dell'applicazione contiene dei link che possono essere usati le volte successive, quando il client sceglie di iniziare una nuova transizione di stato.

4. Gestione ed elaborazione dei report lato server

Fielding descrive così il concetto di Representational State Transfer:

“The name 'Representational State Transfer' is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.”

REST è stato pensato inizialmente all'interno di un contesto basato su HTTP, ma in realtà non è limitato solamente a questo protocollo. Le architetture RESTful possono essere basate su altri protocolli di livello applicativo se essi supportano un ricco e uniforme vocabolario per trasferire un significativo stato di rappresentazione di una risorsa. Le applicazioni RESTful massimizzano l'uso di interfacce pre-esistenti e ben definite fornite dal protocollo utilizzato insieme ad altre particolari caratteristiche integrate e minimizzano l'aggiunta di nuove caratteristiche specifiche negli strati superiori del protocollo.

Lo stile architetturale REST descrive i seguenti sei vincoli da applicare all'architettura, mentre lascia libera l'implementazione dei singoli componenti:

- **Client-server:** un'interfaccia uniforme separa i client dai server. Questa separazione delle responsabilità significa che, per esempio, ai client non viene data la possibilità di gestire lo storage dei dati, che invece rimane interno ad ogni server in modo tale (ad esempio) da migliorare la portabilità del codice del client. I server invece non devono prendersi carico di gestire l'interfaccia utente o lo stato del client, in modo tale da rendere l'implementazione dei server più semplice e fornire un servizio scalabile. I server e i client possono anche essere sostituiti e sviluppati in maniera indipendente, lasciando però l'interfaccia inalterata;
- **Stateless:** la comunicazione client-server è ulteriormente vincolata dal fatto che il server non renda persistente il contesto del client durante le varie richieste. Ogni richiesta da ogni singolo cliente contiene tutte le informazioni necessarie per servire la richiesta, e ogni stato della sessione è tenuta dal cliente. Il server però può essere reso stateful: questo vincolo richiede che lo stato lato server sia indirizzabile attraverso un URL che identifica una particolare risorsa. Questo non solo rende il server più visibile per le operazioni di monitoraggio, ma lo rende più sicuro a fronte di failure parziali della rete;
- **Cacheable:** siccome si lavora all'interno del World Wide Web, i client possono fare caching delle risposte. Tuttavia le risposte devono, implicitamente o esplicitamente, definire se stesse come cacheable o no per evitare che i client riusino cache non rinfrescate o dati di risposta non appropriati per le richieste successive. Una gestione buona delle cache può eliminare parzialmente o totalmente alcune interazioni client-server, in modo da migliorare la scalabilità e le performance;
- **Layered system:** un cliente non può a priori dire se è collegato direttamente a un server o a un intermediario. I server intermedi possono migliorare la scalabilità di sistema abilitando un sistema di load-balancing e fornendo un sistema di cache condivise, oltre alla possibilità di applicare politiche di sicurezza;
- **Code on demand (opzionale):** i server hanno la temporanea possibilità di estendere o personalizzare le funzionalità di un client trasferendo ad esso del codice eseguibile. Alcuni esempi possono includere componenti compilati come le Java Applet e script client-side come JavaScript;
- **Uniform interface:** l'interfaccia comune e uniforme tra i client e i server semplifica e

4. Gestione ed elaborazione dei report lato server

disaccoppia l'architettura, in modo tale che ogni parte possa essere sviluppata in modo indipendente.

L'unico vincolo opzionale dell'architettura REST è il *Code on Demand*. Se un servizio viola qualsiasi altro vincolo, non può essere definito come REST-compliant.

Aderendo a questi vincoli, e quindi essendo conformi allo stile architetturale REST, permette ad ogni tipo di sistema di distribuzione di hypermedia di avere le proprietà desiderate, quali performance, scalabilità, semplicità, modificabilità, visibilità, portabilità ed affidabilità.

Un RESTful Web Service (o RESTful Web API) è un semplice Web Service implementato usando HTTP e i principi REST. È una collezione di risorse, con quattro aspetti definiti:

- La base dell'URI per il Web Service, come *http://example.com/resources/*;
- L'Internet Media Type dei dati supportati dal Web Service. Spesso viene usato JSON, XML o YAML ma possono essere supportati altri tipi validi di Internet Media Type;
- Il set di operazioni supportate dal Web Service, utilizzando i metodi HTTP (per esempio, GET, PUT, POST o DELETE);
- L'API deve essere hypertext-driven.

I metodi PUT e DELETE sono idempotenti. Il metodo GET è un *safe method*, quindi la chiamata ad esso non produce side-effects (e questo include l'idempotenza). Al contrario dei SOAP-based Web Services, non c'è uno standard ufficiale per RESTful Web Services. Questo perché REST è un'architettura al contrario di SOAP che è invece un protocollo. Anche se REST non è uno standard, una implementazione RESTful (come il Web) può usare protocolli e rappresentazioni standard come HTTP, URI, XML, ecc.

4.3.1 RESTful design vs SOAP-based design

Una delle necessità degli sviluppatori e architetti software è quella di decidere quando un particolare stile diventa una scelta appropriata per una particolare applicazione. Un design RESTful può essere appropriato quando:

- I Web Services sono completamente stateless. Una buona prova è quella di considerare se l'interazione può o meno sopravvivere al riavvio del server;
- Una infrastruttura di caching può essere resa disponibile per questioni di performance. Se i dati che ritorna il Web Service non sono generati dinamicamente e possono essere cacheable, allora l'infrastruttura di caching che i Web server o altri intermediari forniscono può essere abilitata per migliorare le performance. Tuttavia, lo sviluppatore deve fare attenzione perché tali cache sono limitate al solo metodo HTTP GET per la maggior parte dei server;
- Il produttore e il consumatore del servizio hanno entrambi una conoscenza del contesto e del contenuto che viene trasferito da uno all'altro. Siccome non esiste un modo formale per descrivere l'interfaccia del Web Service, entrambe le parti devono concordarsi *out-of-band* circa lo schema che descrive i dati da scambiare e sui modi per processarli in maniera coerente. Nel mondo reale, molte applicazioni commerciali che espongono servizi come implementazioni RESTful forniscono anche dei *value-added toolkit* che descrivono le interfacce agli sviluppatori nei più popolari linguaggi di programmazione;
- La larghezza di banda è particolarmente importante ed è necessario limitarla. REST è particolarmente utile per dispositivi *limited-profile* come PDA e telefoni cellulari, per i quali

4. Gestione ed elaborazione dei report lato server

l'overhead degli header e i layer aggiuntivi di elementi SOAP sul payload XML deve essere il minore possibile;

- L'aggregazione di Web Services all'interno di siti Web può essere facilmente applicabile con uno stile RESTful. Gli sviluppatori possono usare tecnologie come Asynchronous JavaScript with XML (AJAX) e toolkit come Direct Web Remoting (DWR) per consumare i servizi nelle loro applicazioni Web. Invece di partire da zero, i servizi possono quindi essere esposti tramite XML e consumati da pagine HTML senza un significativo refactoring dell'architettura esistente del sito Web. Gli sviluppatori in questo modo saranno più produttivi visto che aggiungeranno qualcosa a tecnologie già familiari invece di partire da zero con una nuova tecnologia.

Un design SOAP-based può essere appropriato quando:

- Un contratto formale deve essere stabilito per descrivere l'interfaccia che offre il Web Service. Il Web Services Description Language (WSDL) descrive i dettagli come i messaggi, le operazioni, i binding e la locazione dei Web Services;
- L'architettura deve far fronte a complessi requisiti non funzionali. Molte specifiche per Web Services gestiscono questi requisiti e stabiliscono un vocabolario comune per questi. Ad esempio si può parlare di transazioni, sicurezza, indirizzamento, fiducia, coordinazione e così via. Molte applicazioni reali vanno oltre alle singole operazioni CRUD (Create, Read, Update, Delete) e richiedono informazioni di contesto e uno stato della conversazione da mantenere. Con un approccio RESTful gli sviluppatori devono costruire questo schema da soli e renderlo disponibile nel layer applicativo;
- L'architettura ha bisogno di gestire invocazioni e processing asincrono. In questi casi, l'infrastruttura fornita dagli standard come WSRM e dalle API come JAX-WS con il rispettivo supporto client-side può essere fornito direttamente out-of-the-box.

5. Progetto enterprise su JBoss AS

Il progetto si fonda sull'idea di poter ricevere, ospitare, elaborare e presentare i dati provenienti dai singoli report che gli utenti Android inviano ad una applicazione Web tramite una connessione remota per l'accesso ad internet. In questo modo è possibile consultare la base di dati collettiva creata tramite l'applicazione Android presentata nei capitoli precedenti, utile nella scelta di un operatore di rete da adottare in una determinata zona di interesse. Così l'utente che possiede i dati ricavati tramite l'applicazione Android può collegarsi in modo semplice al server e inviare sia il singolo report ricavato in loco oppure la totalità di report acquisiti in zone con sola copertura di segnale GSM (e quindi con l'impossibilità di stabilire una connessione dati internet) o trasferiti da altri dispositivi mobili con scarse capacità computazionali. Tali dati vengono quindi salvati sul server in un supporto di persistenza dedicato in modo che sia possibile poi creare una vista utente personalizzata in base alla particolare esigenza. Le richieste vengono effettuate tramite un'interfaccia Web in cui è possibile impostare l'indirizzo per il quale si chiede la situazione del segnale telefonico, selezionare l'operatore di rete e il tipo di tecnologia radio. Il risultato è quindi la visualizzazione dei risultati all'interno di una mappa personalizzata creata con Google Maps API e il meccanismo visuale fornito dagli overlay item. Il sistema ha l'obiettivo di massimizzare l'ottimizzazione per gestire un alto carico di utenze contemporanee e per mostrare un risultato coerente e aggregato in base, ad esempio, ai report generati appartenenti ad una stessa zona (quindi con generazione di medie di potenza di segnale ricevuto, ecc.).

5.1 Caso d'uso

La modalità di utilizzo dell'applicazione non dipende da un particolare tipo di dispositivo o hardware. L'utente deve essere in possesso di un browser Web abilitato all'esecuzione di codice JavaScript e di una connessione ad internet. Il resto dell'infrastruttura di back-end lato server ha poi l'incarico di ricevere le richieste del cliente ed elaborarle al fine di creare un risultato esaustivo ai fini della ricerca. Quindi, i passi da effettuare affinché la procedura venga svolta correttamente sono i seguenti:

1. Raggiungere il sito Internet del servizio tramite un URL dedicato a fornire una pagina Web di presentazione dell'applicazione, composto di vari campi selezionabili e modificabili;
2. Immettere, all'interno del campo dedicato, l'indirizzo simbolico della località di cui si vogliono ricavare i dati di distribuzione di segnale telefonico. Tale indirizzo può essere completo, composto solo da alcuni campi (ad esempio solo la città o la nazione) o indicante un punto di interesse (ad esempio: "Due Torri");
3. Selezionare, tramite checkbox nell'opportuna sezione, l'operatore di interesse. In questo caso si ha la scelta tra i maggiori operatori nazionali italiani: TIM, Vodafone, Wind e H3G;
4. Selezionare, sempre tramite checkbox, il tipo di tecnologia di rete del quale si vuole un dettaglio da mostrare sulla mappa: in particolare la scelta può essere effettuata selezionando EDGE, UMTS e HSDPA;
5. Inviare, tramite apposito bottone presente all'interno del form, la richiesta al server e attendere il refresh della pagina con conseguente aggiornamento della mappa;
6. Consultare sulla mappa generata con Google Maps API, tramite overlay item correttamente posizionati in base ai dati ricavati, il valore medio di segnale per ogni zona in cui è presente un report all'interno del database. I risultati sono limitati ad una certa area determinata a partire dalla posizione immessa dall'utente in modo da avere un panorama realistico della situazione, utile per prendere decisioni in modo più coerente.

5.2 Implementazione

La fase implementativa è caratterizzata da diversi strumenti interagenti, in modo da progettare e tenere sotto controllo ogni fase dello sviluppo, dai componenti di business all'interfaccia Web fino al supporto per la persistenza. La configurazione utilizzata è la seguente:

- Sistema operativo desktop Ubuntu Linux 11.10 x86_64;
- Eclipse IDE for Java EE and Web Developers 3.7.1 "Indigo" 64-bit;
- JBoss AS 6.1 con JDK 1.6;
- Database MySQL Server 5.1;
- MySQL Administrator e MySQL Browser per Linux 64-bit;

L'IDE di sviluppo ha fornito, oltre ad un ambiente assistito per la programmazione, anche funzionalità avanzate per la gestione dei descrittori per la persistenza, i componenti e le servlet, del mapping database-Entity Bean e per il debug remoto all'interno di JBoss AS 6.1. Quest'ultima soluzione si è rivelata particolarmente utile dato che in caso di errori in fase di post-deployment (soprattutto a runtime), l'intercettazione delle cause è molto difficile da effettuare in ambiti di progetto complessi. In questo modo è stato possibile fare debug all'interno dei componenti

enterprise nello stile del classico debugging Java (quindi tramite l'utilizzo di breakpoint), rendendo le operazioni di correzione degli errori molto facili e veloci. Di seguito viene mostrata parte della GUI di Eclipse J2EE. È possibile notare alcune sezioni dedicate per l'integrazione e lo sviluppo di applicazioni enterprise:

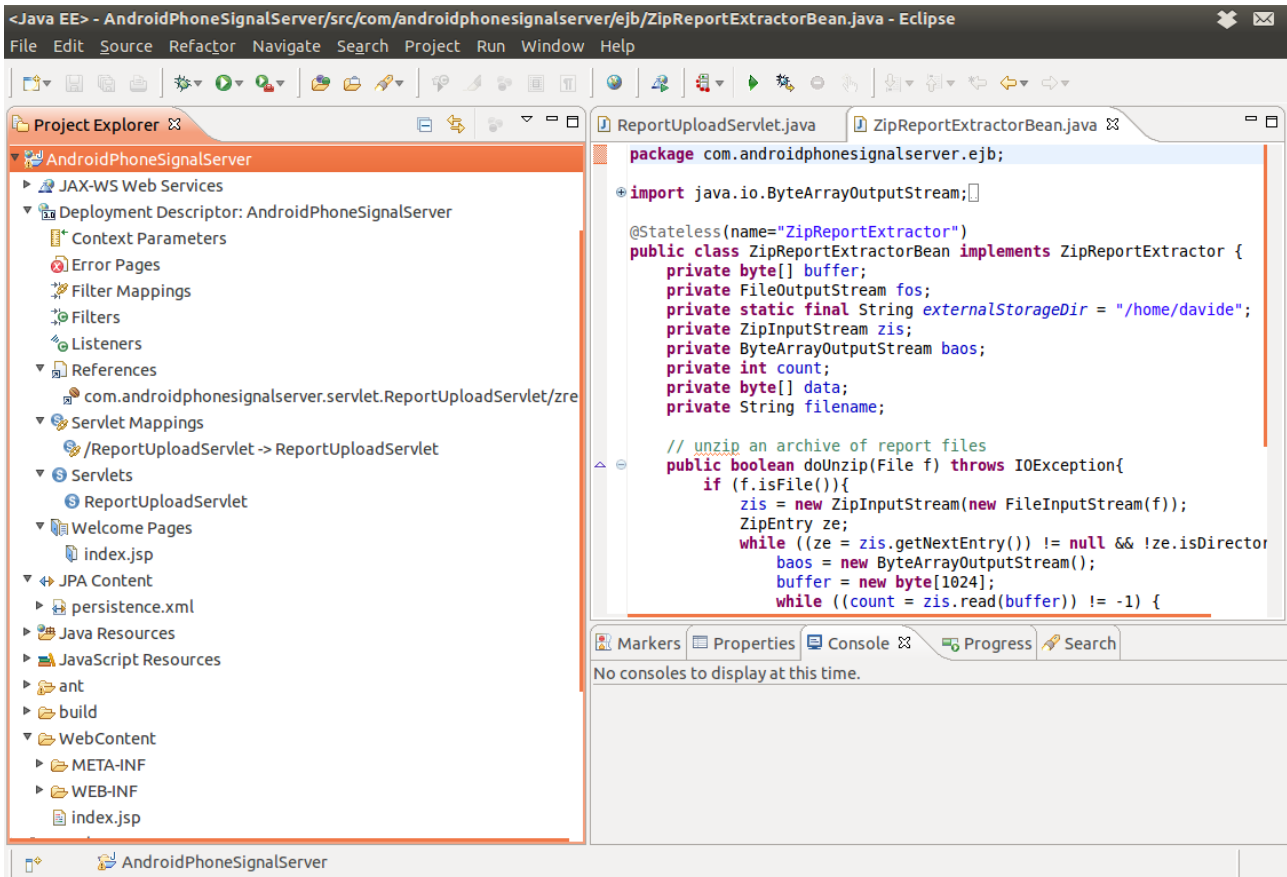


Illustrazione 30: Ambiente di sviluppo Eclipse per J2EE Developers

La figura sopra mostra come Eclipse J2EE fornisca delle viste personalizzate per gestire diversi aspetti dello sviluppo, in modo da avere un ambiente integrato e centralizzato per controllare ogni minimo aspetto: il reference-mapping di componenti EJB all'interno di altri, la gestione del contesto di persistenza tramite JPA (l'implementazione è pluggable), l'assegnazione delle pagine Web di benvenuto, ecc.. Tutto questo viene ricavato da vari descrittori XML dedicati e presenti in alcune cartelle del progetto (web.xml, persistence.xml) e annotazioni Java presenti all'interno delle singole classi.

Eclipse J2EE, come già detto, permette anche di fare debug dell'applicazione all'interno di un container. Essenzialmente questa procedura è possibile eseguirla in due modi diversi. La prima modalità consiste nel creare una configurazione per un server J2EE in modo tale da avviare un'istanza di tale server all'avvio del debug dell'applicazione. In questo modo viene avviato un nuovo server J2EE (tra i supportati è presente JBoss in diverse versioni) che viene messo in esecuzione insieme ad altri server (eventualmente già in fase di run). Quindi è necessario impostare i parametri per il binding delle porte in modo da non creare conflitti all'interno della stessa rete. La seconda modalità consiste nel configurare una socket che possa connettere l'IDE di sviluppo con il server (l'unico) in esecuzione. Per questo motivo è necessario avviare JBoss AS 6.x con alcuni parametri di avvio, in modo che mantenga una socket in ascolto per eventuali debugger esterni. I

parametri da inserire all'interno dello script *run.sh* utilizzato per l'avvio del server sono i seguenti:

```
JAVA_OPTS="$JAVA_OPTS -Xdebug -Xrunjdwp:transport=dt_socket,address=8787,server=y, suspend=n"
```

I parametri specificano che il server si debba avviare in modalità di debug (*-Xdebug*), utilizzando una connessione tramite socket (*-Xrunjdwp:transport=dt_socket*), restando in ascolto di connessioni sulla porta 8787 (*address=8787*) e facendo partire la Java Virtual Machine in uno stato diverso da “suspended” (*suspend=n*). Dal lato Eclipse, la seguente immagine mostra i parametri da impostare per collegarsi all'Application Server configurato per la modalità di debug:

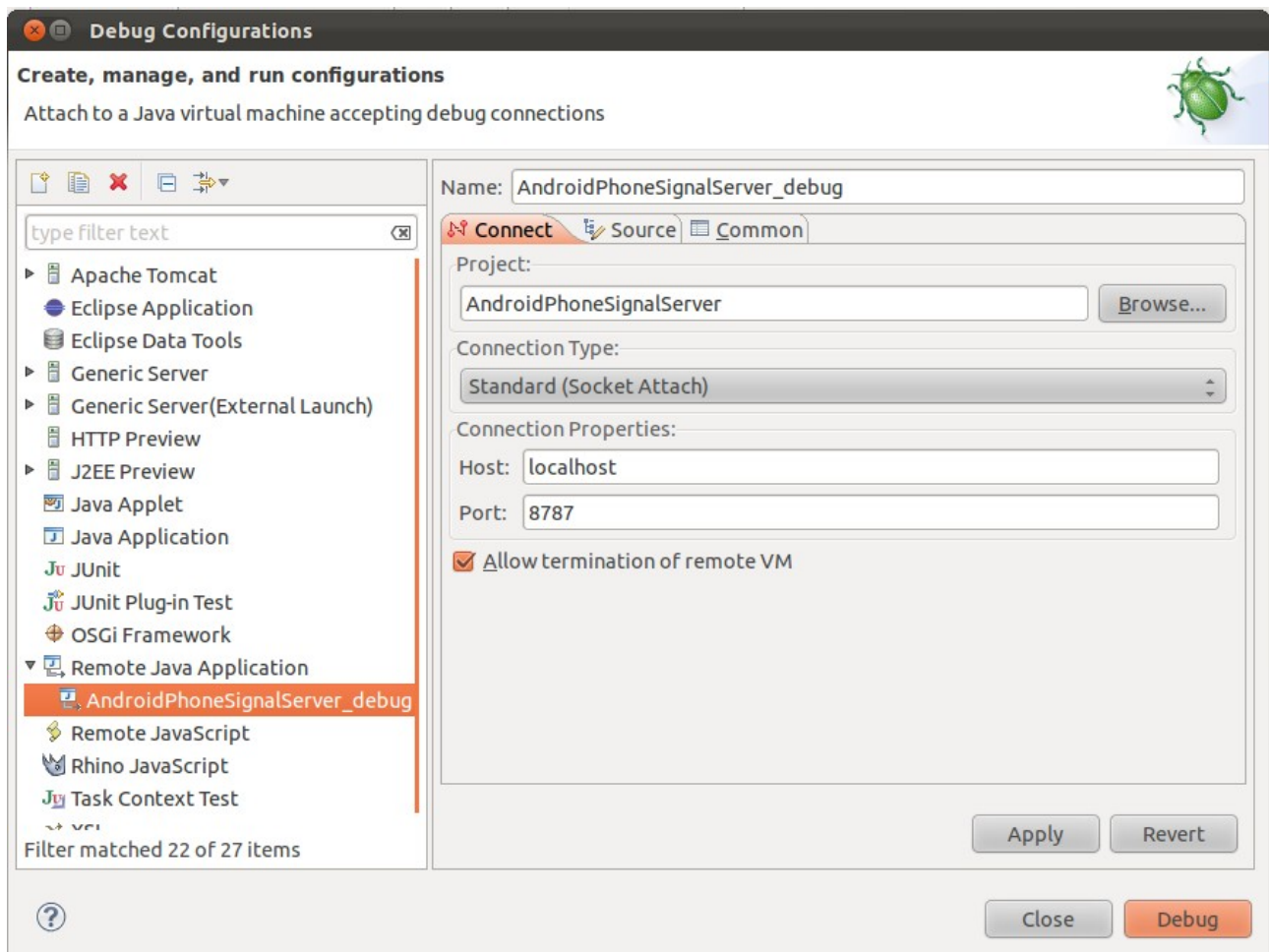


Illustrazione 31: Configurazione del debug remoto in Eclipse J2EE

L'immagine sopra mostra la finestra di configurazione per l'esecuzione dell'applicazione all'interno di una Java Virtual Machine remota. I parametri principali da immettere sono l'indirizzo dell'host e la porta. È possibile anche selezionare l'opzione che consente, sempre da remoto, di terminare la Virtual Machine in cui è in esecuzione il container e quindi l'applicazione enterprise.

Un altro tool molto importante per lo sviluppo, in questo caso relativo alla parte di persistenza, è MySQL Administrator. Tramite questo applicativo è possibile gestire in maniera veloce moltissimi aspetti che riguardano la gestione del database MySQL: amministrazione utenti, status del server, stato delle connessioni, backup, ripristino, replicazione, gestione degli schemi, logging e gestione

dei parametri di startup.
La seguente immagine mostra la schermata di diagnostica di MySQL Administrator:

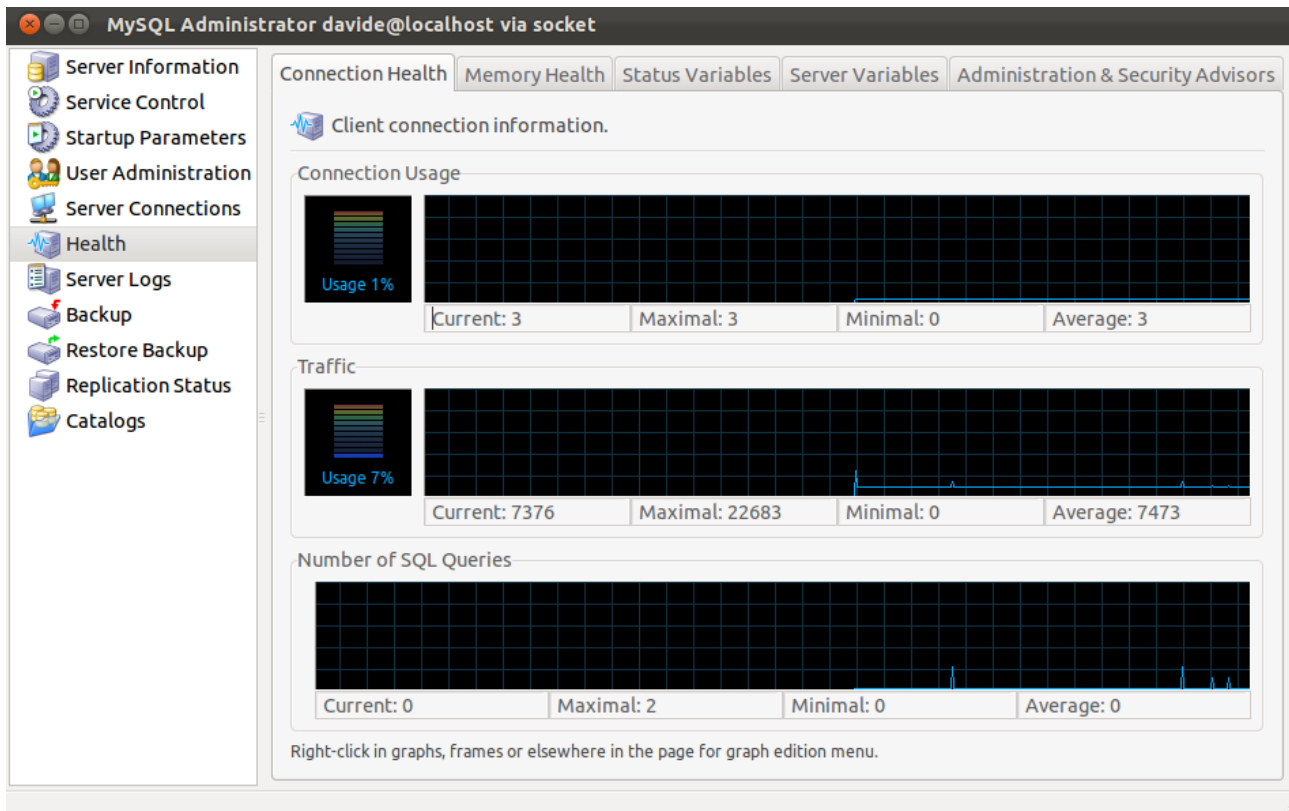


Illustrazione 32: Schermata di diagnostica di MySQL Administrator

Una delle più utili funzionalità fornite dal tool è la possibilità di monitorare il carico di richieste che vengono servite dal server MySQL. In questo modo è possibile verificare in maniera pratica quando entra in azione l'infrastruttura di caching della persistenza gestita da Hibernate (illustrata nei prossimi paragrafi); praticamente la query viene eseguita correttamente con dati presenti in memoria RAM senza rilevare alcun picco di elaborazione (sia di rete che computazionale) da parte del server MySQL.

I paragrafi successivi mostrano a livello pratico le decisioni prese per lo sviluppo di ogni singolo componente, illustrando con esempi di codice le tecnologie e le strategie utilizzate in ogni singola fase progettuale.

5.2.1 Upload del report lato server

La prima necessità da soddisfare per collegare il mondo mobile (e quindi l'applicazione Android) e il mondo server è quella di creare un componente apposito che permetta di ricevere i vari report creati con l'ausilio dei dispositivi mobili. Per abilitare questo tipo di funzione è stato deciso di creare un componente a livello Web che potesse ricevere e rispondere tramite protocollo HTTP. Tale protocollo è largamente utilizzato in ambito Internet e, lavorando sulla porta 80 e quindi in generale sempre aperta e abilitata all'interno dei firewall di sistema, permette una configurazione più facile e veloce per stabilire comunicazioni tra client e server senza inventare un nuovo protocollo adatto allo scopo. L'invio dei report può essere di due tipi: singolo o cumulativo. Nel primo caso viene inviato un singolo file di estensione “.aps” mentre nel secondo caso viene inviato un file compresso nel

formato *ZIP* che contiene al proprio interno uno o più file di report. In entrambi i casi è stato utilizzata una libreria apposita chiamata “Apache Commons FileUpload” che permette, tramite il metodo HTTP POST, di ricevere file di dimensione non necessariamente nota a priori tramite un mimetype specifico del tipo “multipart/form-data”. La descrizione della libreria è la seguente:

“The Apache Commons FileUpload package makes it easy to add robust, high-performance, file upload capability to your servlets and web applications.

FileUpload parses HTTP requests which conform to RFC 1867, 'Form-based File Upload in HTML'. That is, if an HTTP request is submitted using the POST method, and with a content type of 'multipart/form-data', then FileUpload can parse that request, and make the results available in a manner easily used by the caller.“

Il codice relativo alla Servlet è il seguente:

```
// receive a HTTP POST request to retrieve a report or an archive of
reports from the Android app
protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    try {
        DiskFileItemFactory factory = new DiskFileItemFactory();

        // Set factory constraints
        factory.setSizeThreshold(1000000);
        factory.setRepository(new File("/home/davide/"));

        // Create a new file upload handler
        ServletFileUpload upload = new ServletFileUpload(factory);

        // Set overall request size constraint
        upload.setSizeMax(100000);
        // Parse the request and save the file
        List<FileItem> items = upload.parseRequest(request);
        for (FileItem item : items) {
            if (item.getFieldName().equals("reportFile")) {
                String fileName = item.getName();
                String fileContentType = item.getContentType();
                System.out.println(fileName + " " + fileContentType + "
" + fileName.split("\\p{Punct}").length);
                File f = new
File("/home/davide/AndroidPhoneSignalServer/" + fileName);
                item.write(f);
                // check if it is a zip file, then expand the archive
                if (fileName.split("\\p{Punct}")
[fileName.split("\\p{Punct}").length-1].compareTo("zip") == 0) {
                    System.out.println(zre.doUnzip(f));
                    f.delete();
                }
            }
        }
        response.setStatus(200);
        response.getWriter().print("OK");
    } catch (FileUploadException e) {
        response.setStatus(404);
        response.getWriter().print("NO");
        throw new ServletException("Cannot parse multipart request.", e);
    }
}
```


All'interno del metodo *doPost()* della Servlet è presente il codice che gestisce l'intera funzionalità di upload. Il metodo *doGet()* non viene supportato per ragioni di implementazione della libreria FileUpload. In particolare, si utilizza un'istanza della classe *DiskFileItemFactory* per impostare alcuni parametri come la dimensione massima del file accettabile oltre la quale è imposta una scrittura diretta sul disco (altrimenti viene lasciato in memoria) e la cartella di default utilizzata come repository. Successivamente viene istanziato un oggetto della classe *ServletFileUpload* che prende, all'interno del proprio costruttore, l'oggetto del tipo *DiskFileItemFactory* creato precedentemente. In questo modo si ha a disposizione un handler per accettare le richieste di tipo HTTP POST Multipart. Quindi, una volta impostato un limite massimo di dimensione per la richiesta da accettare, l'oggetto di tipo *ServletFileUpload* fa parsing della richiesta e crea una struttura a lista che elenca i file contenuti all'interno della richiesta stessa. Tale richiesta contiene al proprio interno una sezione chiamata *reportFile*, in modo da fornire un elemento discriminante se la stessa richiesta contiene più file destinati ad applicazioni diverse. Quindi i file vengono estratti dalla richiesta e resi persistenti in una cartella di default. Se il file non è un report singolo ma è un archivio di report (l'elemento discriminante è l'estensione del file ricevuto), viene invocato un componente EJB incaricato a fornire la funzionalità di decompressione e di salvataggio dei singoli file all'interno della cartella predefinita. Il client che effettua la richiesta quindi riceve una risposta in base all'esito dell'operazione, in modo che poi possa effettuare operazioni di rollback in caso di operazione di upload fallita.

5.2.2 Decompressione dell'archivio per report multipli

La possibilità di ricevere (tramite la Servlet descritta precedentemente) archivi contenenti vari report ha richiesto l'implementazione di un EJB che potesse essere richiamato dai vari componenti appartenenti al sistema in modo da gestire la funzione di decompressione del file ricevuto all'interno di una cartella predefinita. La scelta di progettuale adottata ha portato alla creazione di uno Stateless Session Bean adatto a tale scopo. In questo modo è possibile registrare tale componente all'interno del container in modo che possa essere poi ritrovato da altri oggetti o componenti tramite una JNDI lookup (o tramite annotazioni) per facilitarne il riutilizzo.

L'interfaccia utilizzata dal componente EJB è la seguente:

```
@Local
public interface ZipReportExtractor {
    // unzip an archive of report files
    public boolean doUnzip(File f) throws IOException;
}
```

Mentre di seguito viene mostrato il codice dell'implementazione:

```
@Stateless
public class ZipReportExtractorBean implements ZipReportExtractor {

    // unzip an archive of report files
    public boolean doUnzip(File f) throws IOException{
        if (f.isFile()){
            zis = new ZipInputStream(new FileInputStream(f));
            ZipEntry ze;
            while ((ze = zis.getNextEntry()) != null && !ze.isDirectory()) {
                baos = new ByteArrayOutputStream();
                buffer = new byte[1024];
                while ((count = zis.read(buffer)) != -1) {
```

```

        baos.write(buffer, 0, count);
    }
    filename = ze.getName();
    data = baos.toByteArray();
    fos = new FileOutputStream(new File(externalStorageDir
+   "/AndroidPhoneSignalServer/" + filename.split("/")
    [filename.split("/").length-1]));
    fos.write(data);
    fos.close();
    }
    return true;
    }
    return false;
    }
}

```

Il codice sopra riportato mostra l'interfaccia utilizzata con la relativa implementazione. Visto che nel caso in oggetto i componenti eseguono all'interno della stessa Java Virtual Machine, per una questione di ottimizzazione si è utilizzata l'annotazione `@Local` per definire l'interfaccia. In questo modo le chiamate al componente bypassano l'infrastruttura remota creata dal container per la comunicazione e vengono gestite direttamente dalla JVM come chiamate locali, nettamente più prestazionali rispetto ad una chiamata RMI/IIOP. L'implementazione mostra che il componente viene registrato all'interno del sistema come Stateless Session Bean e nominato `ZipReportExtractorBean`. Quindi il bean apre il file che viene passato come parametro, ne crea un `ZipInputStream` e controlla che tale file non sia in realtà una directory. Successivamente, per ogni file contenuto nell'archivio, viene salvato il payload all'interno della cartella di default destinata al contenimento dei report.

5.2.3 Salvataggio del report su database e gestione della persistenza

Tutti i report, una volta salvati all'interno della memoria di massa come file, devono essere processati e caricati all'interno del database dedicato. Per gestire questa caratteristica si è fatto uso di un database MySQL 5.1, dell'infrastruttura offerta da Hibernate, di JPA e dei servizi presenti all'interno del container JBoss AS 6.1. Per quanto riguarda la configurazione di JBoss, è stato necessario inserire all'interno della cartella di deploy un file XML che descrivesse il database utilizzato e i parametri di connessione da utilizzare. All'interno delle cartelle di configurazione esiste un file dedicato da modificare a piacere in base al tipo di database utilizzato. In questo caso il file si chiama `mysql-ds.xml` ed è strutturato come segue:

```

<?xml version="1.0" encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>AndroidPhoneSignalDB</jndi-name>
    <connection-
url>jdbc:mysql://localhost:3306/AndroidPhoneSignal</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>davide</user-name>
    <password>xyz</password>
    <exception-sorter-class-
name>org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter</exceptio
n-sorter-class-name>
    <metadata>
      <type-mapping>mySQL</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>

```

```

</local-tx-datasource>
</datasources>

```

I tag indicano come mappare la risorsa all'interno di JNDI, l'URL per la connessione, il driver da utilizzare per l'interazione con il database da ambiente Java (posto all'interno della cartella *\$JBoss/server/default/lib*), il nome utente e la password per effettuare il login e il riferimento alla classe delle eccezioni per gestire in modo corretto errori fatali e quindi per stabilire quando una connessione deve essere eventualmente distrutta. Per gestire la persistenza all'interno dell'applicazione si è fatto utilizzo delle specifiche JPA, in particolare di una implementazione diffusa e conosciuta: Hibernate. Il framework Hibernate è già presente all'interno di JBoss AS 6.1 e fornisce tutti gli strumenti per il mapping O/R, insieme a funzionalità di caching e di interrogazione del database tramite query ottimizzate. Essendo conforme alle specifiche JPA, Hibernate consente di lavorare con Entity Bean POJO inseriti all'interno di un contesto di persistenza, in modo che possano essere gestiti secondo un ben determinato ciclo di vita, non ulteriormente approfondito in questo documento. Hibernate necessita di un file di configurazione, secondo specifica XML, per assegnare alcuni parametri fondamentali per l'esecuzione corretta del framework di persistenza. Tali parametri vengono immessi all'interno di un file chiamato *persistence.xml*:

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="AndroidPhoneSignalServer">
    <jta-data-source>java:AndroidPhoneSignalDB</jta-data-source>
    <class>com.androidphonesignalserver.entity.SignalReport</class>
    <properties>
      <property name="hibernate.cache.use_query_cache"
value="true"/>
    </properties>
  </persistence-unit>
</persistence>

```

Da come si può notare dalla definizione presente all'interno del file, viene specificata una determinata unità di persistenza chiamata *AndroidPhoneSignalServer* (creando un contesto per l'applicazione) che fa uso di Java Transaction Architecture (JTA) per l'accesso al servizio in maniera transazionale e che fa utilizzo della classe *com.androidphonesignalserver.entity.SignalReport* per il mapping O/R. Inoltre si specifica che debba essere abilitata la cache di Hibernate per l'esecuzione delle query, in modo da ottimizzare l'accesso al database. L'Entity Bean *SignalReport* è definito come segue:

```

@Entity
@Cache (usage=CacheConcurrencyStrategy.TRANSACTIONAL ,
region="SignalReport")
@NamedQuery(name = "SignalReport.findFilteredAvgReports", query = "SELECT
x.id, x.operatorName, x.cellID, x.networkType, x.locationProvider,
x.lightInfo, x.longitude, x.latitude, AVG(x.avgSignalStrength), x.roaming,
x.timestamp FROM SignalReport x WHERE x.operatorName= :operatorName AND
x.networkType= :networkType AND x.latitude BETWEEN :latitude -0.01 AND
:latitude +0.01 AND x.longitude BETWEEN :longitude -0.01 AND :longitude
+0.01 GROUP BY x.operatorName, x.latitude, x.longitude",
hints={@QueryHint(name="org.hibernate.cacheable",value="true")},

```

```

@QueryHint(name="org.hibernate.cacheRegion", value="SignalReport"))}

public class SignalReport implements Serializable{

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private String operatorName;
    private double avgSignalStrength;
    private String networkType;
    private boolean roaming;
    private String cellID;
    private float lightInfo;
    private double latitude;
    private double longitude;
    private String locationProvider;
    private long timestamp;

    [getters and setters]
}

```

La classe *SignalReport* è decorata con l'annotazione *@Entity* che permette al container J2EE di riconoscerla come Entity Bean. Al proprio interno contiene esattamente la definizione dei campi presenti all'interno del database, in modo da poter fornire un mapping 1:1. L'attributo *id* è decorato dall'annotazione *@Id*, per segnalare ad Hibernate che si tratta dell'attributo di chiave primaria, insieme a *@GeneratedValue*, annotazione che fornisce informazioni sulla modalità di generazione dell'attributo. Nel caso specifico con *GenerationType.AUTO* si segnala che, dato che è generato automaticamente dal database, il gestore della persistenza deve adattarsi e quindi scegliere la strategia migliore. Le altre annotazioni sono sempre relative al gestore di persistenza. L'annotazione *@Cache* permette di abilitare la cache di secondo livello per l'Entity Bean, specificando una strategia per l'accesso concorrente e una “region” in modo da organizzare la cache all'interno dell'infrastruttura in uno spazio ben definito. Inoltre viene specificata una Named Query chiamata *SignalReport.findFilteredAvgReports* tramite l'annotazione *@NamedQuery* che contiene al proprio interno la query utilizzata per determinare la potenza media di segnale di un singolo operatore per una singola tecnologia in una determinata area di spazio. In particolare viene mostrata la situazione all'interno di uno spazio compreso tra il valore ricavato di longitudine $\pm 0.01^\circ$ e latitudine $\pm 0.01^\circ$. Tramite l'annotazione *@QueryHint* è stato possibile inoltre introdurre proprietà personalizzate dipendenti dal contesto del framework di persistenza, in questo caso Hibernate. In questo modo è stato possibile specificare che la query in oggetto possa essere resa cacheable e che possa appartenere alla region “SignalReport”.

Per fare in modo che i report quindi vengano letti all'interno dei file presenti su disco, immessi all'interno del database e successivamente cancellati dalla memoria di massa, è stato scelto di utilizzare un componente introdotto nelle specifiche EJB 3.1: un *EJBTimer*. Esso non è altro che uno *Stateless Session Bean* corredato di alcune annotazioni che permettono al container di associarlo ad una procedura da chiamare periodicamente, in base ai parametri con cui una particolare annotazione viene configurata. Questo componente quindi fa utilizzo dell'Entity Bean precedentemente descritto per riempire i relativi campi tramite i valori derivati dai file contenenti i report, per poi successivamente far entrare tali bean all'interno del contesto di persistenza e quindi con la possibilità di gestire il ciclo di vita di ogni singola riga del database acceduta e caricata in memoria tramite i corrispettivi Entity Bean istanziati.

L'interfaccia utilizzata dal componente è la seguente:

```
@Local
public interface ReportUploadDB {
    // check the default dir from the storage and save the reports to DB
    public void scheduledTimeout(final Timer t) throws IOException;
}
```

Mentre di seguito viene mostrata l'implementazione:

```
@Stateless
public class ReportUploadDBBean implements ReportUploadDB{

    @PersistenceContext(name="AndroidPhoneSignalServer")
    private EntityManager em;

    // check periodically the default storage dir, save reports to DB
    @Schedule(second="*/10", minute="*",
        hour="*",info="ReportUploadDBBean", persistent=false)
    public void scheduledTimeout(final Timer t) throws IOException {
        f = new File(externalStorageDir +
            "/AndroidPhoneSignalServer");
        gson = new Gson();
        if (f.isDirectory()) {
            for(int i=0; i<f.listFiles().length;i++) {
                reportFile = f.listFiles()[i];
                fis = new FileInputStream(reportFile);
                buff = new byte[(int) reportFile.length()];
                fis.read(buff, 0, (int) reportFile.length());
                json = new String(buff, "UTF-8");
                // create a report Entity Bean (POJO) from the
                // Json report string
                report = new SignalReport();
                report = gson.fromJson(json, SignalReport.class);
                // attach the Entity to the Persistence Context
                em.persist(report);
                reportFile.delete();
            }
        }
    }
}
```

L'EJB Timer *ReportUploadDBBean* è definito esattamente come uno Stateless Session Bean locale e quindi, oltre al fatto di eseguire all'interno della stessa Java Virtual Machine locale al container, può essere registrato all'interno di un contesto JNDI per essere poi istanziato od invocato da diversi componenti dell'applicazione. In questo caso non è necessario e si è scelto che tale componente venga controllato direttamente dal container, visto che esegue operazioni generiche indipendenti da una specifica funzionalità. Eventualmente è possibile decorarlo con l'annotazione *@Singleton* per fare in modo che solo un'unica istanza possa essere presente a runtime. All'interno dell'implementazione viene creato un riferimento al contesto di persistenza utilizzando l'annotazione *@PersistenceContext*, indicando al proprio interno il nome indicato all'interno del file *persistence.xml* e che fa appunto riferimento al contesto creato per l'Entity Bean definito e quindi alla particolare tabella all'interno del database MySQL utilizzato per l'archiviazione dei report. Il metodo *ScheduledTimeout (final Timer t)* è decorato con l'annotazione *@Schedule*, che permette di specificare al proprio interno l'intervallo di invocazione del suddetto metodo. La configurazione

impostata permette al container di gestire automaticamente la chiamata periodicamente ogni 10 secondi, in qualsiasi minuto e ora del giorno. In caso di caduta dell'Application Server, EJB Timer prevede un servizio di persistenza del timer stesso, in modo che il tempo di schedulazione possa essere recuperato in un successivo avvio del server. Nel caso specifico si è scelto di non abilitare la persistenza del timer per due principali motivi:

- il progetto non richiede un timer persistente vista l'alta frequenza di schedulazione dell'azione, caratteristica necessaria invece se lo scheduling prevedesse tempi precisi di invocazione del metodo, magari molto lontani tra loro;
- JBoss AS 6.1, grazie a un bug noto alla comunità, non elimina dal supporto persistente i componenti EJB Timer, anche dopo una rimozione dal progetto e un riavvio dell'Application Server. In questo modo è necessario cancellare manualmente la cartella destinata alla passivazione e persistenza dei componenti. Tale problema dovrebbe essere risolto con le release successive.

Le operazioni di business del componente prevedono la lettura dei report precedentemente salvati all'interno della cartella predefinita da parte del componente adibito a tale funzionalità; tali report poi vengono salvati assegnando i relativi campi ad un'istanza di Entity Bean utilizzando la libreria Gson: in questo modo è possibile prelevare la stringa in formato Json all'interno del file del report e convertirlo in modo molto semplice in un oggetto Java qualsiasi (ma conforme alla signature), specificato all'interno del metodo *fromJson()*. Infine viene inserita l'istanza di Entity Bean all'interno del contesto di persistenza tramite il metodo *EntityManager.persist()* e successivamente il file viene cancellato dal File System.

5.2.4 Estrazione dei dati dal supporto di persistenza

Una delle funzionalità fondamentali dell'applicazione è quella del ritrovamento corretto dei dati in base all'interrogazione specifica da parte dell'utente. Tale modulo viene implementato attraverso un componente EJB incaricato di ottenere il contesto di persistenza fornito da Hibernate, invocare la query specifica e quindi ritornare i risultati come output. Siccome tale componente non necessita di salvare uno stato persistente, si è utilizzato uno Stateless Session Bean. Di seguito viene illustrata l'interfaccia:

```
@Local
public interface DataDBExtractor {

    // get all the location reports from the DB
    public List<SignalReport> getLocationReports();
    // get all the location reports filtered by latitude and longitude
    public List<SignalReport> getFilteredLocationReports(double
latitude, double longitude);
    // get all the location reports filtered by latitude, longitude,
carrier and network type
    public List<SignalReport> getFilteredAvgLocationReports(String
operatorName, String networkType, double latitude, double
longitude);
}
```


Mentre l'implementazione è stata eseguita come segue:

```

@Stateless
public class DataDBExtractorBean implements DataDBExtractor{
    @PersistenceContext(name="AndroidPhoneSignalServer")
    private EntityManager em;

    // get all the location reports from the DB
    public List<SignalReport> getLocationReports() {
        Query q = em.createQuery("SELECT x FROM SignalReport x");
        @SuppressWarnings("unchecked")
        List<SignalReport> resultsQuery = (List<SignalReport>)
q.getResultList();
        return resultsQuery;
    }
    // get all the location reports filtered by latitude and longitude
    public List<SignalReport> getFilteredLocationReports(double
latitude, double longitude) {
        Query q = em.createQuery("SELECT x FROM SignalReport x WHERE
x.latitude BETWEEN " + String.valueOf(latitude-0.05) + " AND " +
String.valueOf(latitude+0.05) + " AND x.longitude BETWEEN " +
String.valueOf(longitude-0.05) + " AND " +
String.valueOf(longitude+0.05));
        @SuppressWarnings("unchecked")
        List<SignalReport> resultsQuery = (List<SignalReport>)
q.getResultList();
        return resultsQuery;
    }
    // get all the location reports filtered by latitude, longitude,
carrier and network type
    public List<SignalReport> getFilteredAvgLocationReports(String
operatorName, String networkType, double latitude, double longitude)
{
        @SuppressWarnings({ "rawtypes", "unchecked" })
        List<SignalReport> results = new LinkedList();
        if (operatorName == null)
            return results;
        Query q =
em.createNamedQuery("SignalReport.findFilteredAvgReports");
q.setParameter("operatorName", operatorName);
q.setParameter("networkType", networkType);
q.setParameter("latitude", latitude);
q.setParameter("longitude", longitude);
        @SuppressWarnings({ "rawtypes" })
        List resultsQuery = q.getResultList();
        SignalReport temp = null;
        @SuppressWarnings("rawtypes")
        Iterator t = resultsQuery.iterator();
        while(t.hasNext()) {
            temp = new SignalReport();
            Object[] result = (Object[]) t.next();
            temp.setId(((Integer) result[0]).intValue());
            temp.setOperatorName((String) result[1]);
            temp.setCellID((String) result[2]);
            temp.setNetworkType((String) result[3]);
            temp.setLocationProvider((String) result[4]);
            temp.setLightInfo(((Float) result[5]).floatValue());
        }
    }
}

```

```

        temp.setLongitude(((Double) result[6]).doubleValue());
        temp.setLatitude(((Double) result[7]).doubleValue());
        temp.setAvgSignalStrength(((Double)
            result[8]).doubleValue());
        temp.setRoaming(((Boolean) result[9]).booleanValue());
        temp.setTimestamp(((Long) result[10]).longValue());
        results.add(temp);
    }
    return results;
}
}
}

```

Il componente *DataDBExtractor* presenta 3 funzionalità diverse, specificate nell'interfaccia: la prima ricava indistintamente tutti i record dal database, la seconda ritorna solo i record appartenenti ad un certo intervallo di latitudine e longitudine mentre la terza ricava tutti i report di un determinato operatore, all'interno di un determinato range spaziale e di una determinata tecnologia di rete. Inoltre, se sono presenti più report alla stessa coordinata, ne effettua la media e la riporta come valore finale.

Per tutte e tre le funzioni viene invocato il contesto di persistenza tramite l'Entity Manager. Mentre le prime due sono state implementate per eventuali estensioni future dell'applicazione, la terza è di particolare importanza perché è la sola che viene effettivamente invocata e quindi utilizzata. Nel metodo *getFilteredAvgLocationReports()* viene passato in ingresso il nome dell'operatore di rete, il tipo di rete da ricercare e i valori di latitudine e longitudine. Successivamente si crea una query utilizzando la NamedQuery creata all'interno della classe dell'Entity Bean. Per crearla è necessario invocare il metodo *EntityManager.createNamedQuery()* che accetta come parametro la stringa che riferenzia la NamedQuery all'interno del contesto di persistenza. In questo modo è possibile attivare i meccanismi di caching dell'interrogazione con i conseguenti vantaggi sulle performance. Dopo aver impostato i parametri per la ricerca, si invoca il metodo *Query.getResultList()* per ottenere la lista dei risultati. Siccome all'interno della query si fa utilizzo di una funzione di aggregazione che va a sostituire il valore di un campo (quello del valore medio del segnale), non è possibile ritornare i risultati direttamente all'interno di una lista di tipo SignalReport, visto che il mapping 1:1 non viene effettuato dal gestore di persistenza. Quindi è necessario ricavare ogni singolo valore e creare manualmente la lista di risultati da ritornare ad operazione eseguita. L'utilizzo di una cache per la query risolve il problema del caching dei singoli Entity Bean: in questo modo è possibile avere in memoria centrale i risultati più richiesti in base alla posizione di ricerca, senza effettuare l'accesso continuativo al database.

5.2.5 Localizzazione tramite Google Maps Web Services RESTful API

Un altro punto fondamentale da considerare è il ritrovamento delle coordinate (latitudine e longitudine) del luogo che l'utente immette sotto forma di indirizzo simbolico all'interno della pagina Web. Tale azione è necessaria poi per mettere in moto l'intero meccanismo integrato di elaborazione e visualizzazione corretta dei risultati. Quindi, quando l'utente digita in un apposito form il luogo per il quale è interessato avere una mappa del segnale telefonico e quindi avvia la ricerca, entra in azione un particolare componente EJB che, utilizzando Google Maps Web Services RESTful API, permette di ricavare le coordinate del luogo analizzando la risposta ritornata dal server. Tale risposta poi viene inoltrata ai componenti di interesse descritti precedentemente, che estrarranno i dati dal database e forniranno dei risultati corretti che verranno poi mostrati all'interno di una appositamente apposta mappa.

L'interfaccia del componente è la seguente:

```
@Local
public interface GMapsLocation {

    // retrieve the latitude and longitude from a symbolic geo-address
    public String getLatlngFromAddress(String address) throws
        IOException, InvalidSyntaxException;
}
```

Mentre la classe del bean che la implementa è la seguente:

```
@Stateless
public class GMapsLocationBean implements GMapsLocation{
    private final static String defaultLocation =
        "43.9797480,12.6957660";

    private JsonNode parsedJson;

    // retrieve the latitude and longitude from a symbolic geo-address
    public String getLatlngFromAddress(String address) throws IOException,
        InvalidSyntaxException {
        if (address == null) {
            return defaultLocation;
        }
        else {
            // REST Web Service request to use Google Maps Web Services API
            - JSON encoding
            URL mapsCoord = new
            URL("http://maps.googleapis.com/maps/api/geocode/json?address=" +
            address + "&sensor=false");
            URLConnection mc = mapsCoord.openConnection();
            BufferedReader in = new BufferedReader(new
            InputStreamReader(mc.getInputStream()));
            String inputLine, output = "";
            while ((inputLine = in.readLine()) != null)
                output += inputLine;
            in.close();
            // parse the request with Argo JSON parser
            parsedJson = new JdomParser().parse(output);
            if (parsedJson.getStringValue("status").compareTo("OK") ==
                0) {
                parsedJson = new JdomParser().parse(output)
                    .getArrayNode("results")
                    .get(0).getNode("geometry")
                    .getNode("location");
                return parsedJson.getNumberValue("lat") + "," +
                    parsedJson.getNumberValue("lng");
            }
            else {
                return defaultLocation;
            }
        }
    }
}
```

Anche in questo caso è stato implementato uno Stateless Session Bean locale, visto che non è

necessario mantenere uno stato relativo ai clienti che effettuano la richiesta. Il componente accetta in ingresso una stringa che contiene la locazione simbolica da convertire in locazione fisica. Il vincolo sintattico da rispettare è l'assenza di spazi: la stringa fornita dal cliente quindi deve essere ristrutturata convertendo il carattere spazio con il carattere "+". Solo in questo modo è possibile creare una richiesta corretta da inviare al server. Quindi avrà la seguente forma:

```
http://maps.googleapis.com/maps/api/geocode/json?address=Viale+del+Risorgimento,+2,+Bologna&sensor=true_or_false
```

E la risposta del server, in formato Json, è ritornata come segue:

```
{
  "results" : [
    {
      "address_components" : [
        {
          "long_name" : "2",
          "short_name" : "2",
          "types" : [ "street_number" ]
        },
        {
          "long_name" : "Viale del Risorgimento",
          "short_name" : "Viale del Risorgimento",
          "types" : [ "route" ]
        },
        {
          "long_name" : "Bologna",
          "short_name" : "Bologna",
          "types" : [ "locality", "political" ]
        },
        {
          "long_name" : "Bologna",
          "short_name" : "BO",
          "types" : [ "administrative_area_level_2", "political" ]
        },
        {
          "long_name" : "Emilia Romagna",
          "short_name" : "Emilia Romagna",
          "types" : [ "administrative_area_level_1", "political" ]
        },
        {
          "long_name" : "Italia",
          "short_name" : "IT",
          "types" : [ "country", "political" ]
        },
        {
          "long_name" : "40135",
          "short_name" : "40135",
          "types" : [ "postal_code" ]
        }
      ],
      "formatted_address" : "Viale del Risorgimento, 2, 40135 Bologna, Italia",
      "geometry" : {
```

```

    "bounds" : {
      "northeast" : {
        "lat" : 44.48801110,
        "lng" : 11.32810140
      },
      "southwest" : {
        "lat" : 44.4879980,
        "lng" : 11.32809790
      }
    },
    "location" : {
      "lat" : 44.4879980,
      "lng" : 11.32809790
    },
    "location_type" : "RANGE_INTERPOLATED",
    "viewport" : {
      "northeast" : {
        "lat" : 44.48938568029150,
        "lng" : 11.32948073029150
      },
      "southwest" : {
        "lat" : 44.48668771970850,
        "lng" : 11.32678276970850
      }
    }
  },
  "types" : [ "street_address" ]
},
"status" : "OK"
}

```

In questo caso il componente si aspetta che l'indirizzo ricevuto sia ben formato. La richiesta prevede l'invio dei parametri in modalità HTTP GET e, in particolare, l'indirizzo ben formato e un flag che indica se la richiesta è inviata da un dispositivo che possiede un sensore di geolocalizzazione. Una volta inviata la richiesta, si riceve la risposta leggendo dall'InputStream e salvando i dati all'interno di una stringa. A questo punto è necessario fare il parsing della risposta Json: a tal proposito è stata utilizzata una libreria esterna, Argo, che permette di navigare l'albero con semplici chiamate. Quindi si è controllato lo stato della richiesta e successivamente si sono ricavati i due valori di latitudine e longitudine dal campo *location* della risposta. In caso di invocazione del componente con una stringa malformata, o con una stringa non istanziata oppure se, in generale, la richiesta non va a buon fine, viene ritornata una locazione di default specificata all'interno della costante *defaultLocation*.

5.2.6 Interfaccia utente via Web

L'ultima parte da trattare è l'interfaccia Web presentata all'utente utilizzatore. Quest'ultima è stata scritta utilizzando la tecnologia JSP. In questo modo è stato possibile progettare una pagina Web dinamica, utilizzando lo stesso paradigma delle servlet (di fatto, ogni pagina JSP viene successivamente convertita in una servlet dal container). Questo ha permesso di utilizzare il protocollo HTTP e le richieste di tipo GET per passare i relativi parametri che l'utente ha impostato all'interno del relativo form. All'interno del codice della pagina vengono istanziati anche alcuni componenti EJB, utilizzando esplicitamente il lookup fornito dal servizio JNDI, dato che la

dependency injection non è consentita all'interno di componenti Web quali sono le servlet.

L'istanziamento dei componenti EJB è stata eseguita come segue:

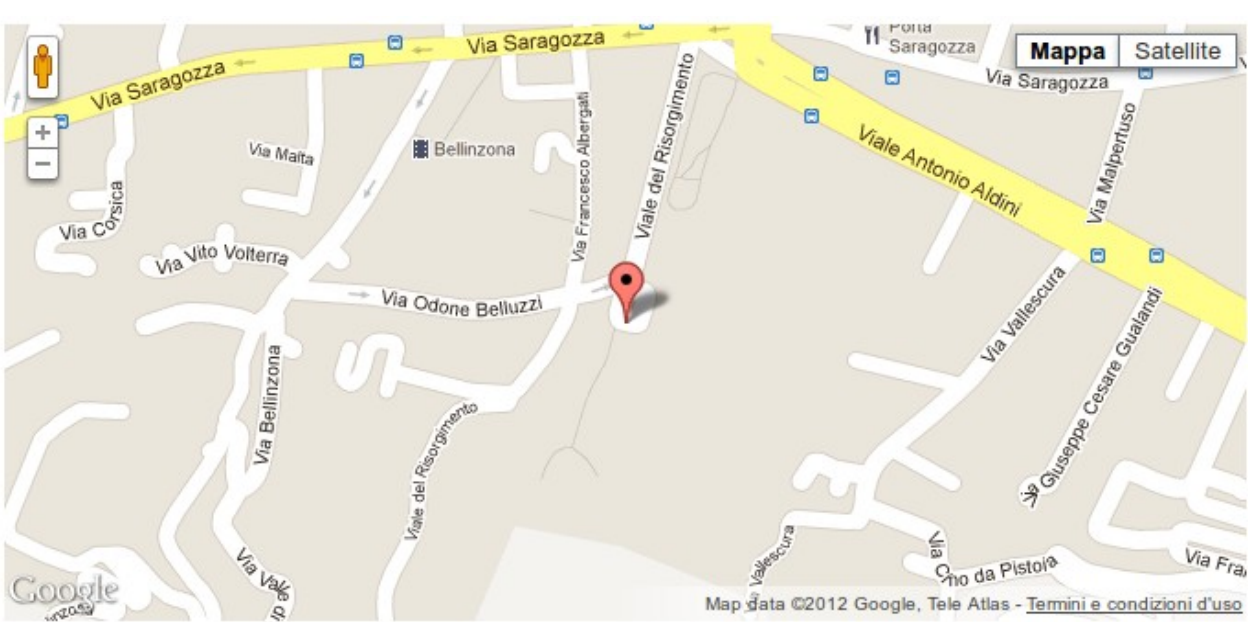
```
private GMapsLocation locationbean = null;
private DataDBExtractor datadbextbean = null;
public void jspInit () {
    Context jndiContext;
    try {
        jndiContext = new javax.naming.InitialContext();
        locationbean = (GMapsLocation)
PortableRemoteObject.narrow(jndiContext.lookup("GMapsLocationBean/local"),
GMapsLocation.class);
        datadbextbean = (DataDBExtractor)
PortableRemoteObject.narrow(jndiContext.lookup("DataDBExtractorBean/local"
), DataDBExtractor.class);
    } catch (NamingException e) { }
}
```

All'inizializzazione della JSP viene estratto il riferimento al contesto JNDI tramite il metodo *InitialContext()* e successivamente si estrae il riferimento ai due Stateless Bean incaricati al ritrovamento delle coordinate data la locazione simbolica (*GMapsLocation*) e all'estrazione dei dati dal database (*DataDBExtractor*). Il procedimento quindi prevede l'utilizzo della classe *PortableRemoteObject* per effettuare il narrowing e quindi controllare che un determinato oggetto con interfaccia remota o astratta possa essere correttamente convertito in un tipo specificato. Questo passaggio può essere evitato in questo caso specifico visto che i componenti sono locali e si può utilizzare una semantica "call by reference" con il solo uso dell'operazione di casting. Quindi il metodo *lookup()* effettua la ricerca del bean specificato dalla stringa tra quelli registrati all'interno del contesto. Se l'operazione di narrowing e di casting avviene con successo, i riferimenti agli EJB diventano validi con conseguente possibilità di invocare operazioni attraverso essi. Per quanto riguarda la visualizzazione della mappa con i relativi risultati come overlay item, è stato utilizzato Google Maps Javascript API per effettuare il corretto rendering. La fase di popolamento della mappa è stato effettuato come segue:

```
<%
// populate the map with the retrieved position
String loc =
locationbean.getLatlngFromAddress(request.getParameter("location")!=null?
request.getParameter("location").replace(" ", "+"):null);
    List<SignalReport> results =
datadbextbean.getFilteredAvgLocationReports(request.getParameter("operator"
), request.getParameter("networkType"), Double.parseDouble(loc.split(",")
[0]), Double.parseDouble(loc.split(",")[1]));
    for (int i=0; i<results.size() && results.get(i)!
=null;i++) {
        out.println("marker = new google.maps.Marker({\n\t\tt
position: new google.maps.LatLng(" + results.get(i).getLatitude() + "," +
results.get(i).getLongitude() + " ),\n\t\t\t title:\"\" +
results.get(i).getOperatorName() + " \" +
results.get(i).getAvgSignalStrength() + " \" +
results.get(i).getNetworkType() + \"\");\n marker.setMap(map);");
    }
%>
```

Inizialmente vengono ricavati i due valori di latitudine e longitudine invocando il metodo

`getLanlngFromAddress()` contenuto all'interno dello `SSB GmapsLocation`. In particolare viene passata una stringa ben formata, sostituendo gli spazi con il carattere "+". Successivamente viene memorizzato il risultato in una stringa con il seguente formato: "lat,lng". Se nella richiesta il parametro non viene inserito, si invia un valore `null` in modo da mostrare la località di default sulla mappa. Una volta ricavati i valori di latitudine e longitudine, viene invocato il metodo `getFilteredAvgLocationReports()` del componente EJB `DataDBExtractor`. Processando i parametri derivati dall'elaborazione dell'EJB `GmapsLocation` e dalla richiesta HTTP GET (`operator` e `networkType`), viene generata una lista di locazioni, prossime al luogo di ricerca, che sono utilizzate per posizionare il relativo marker all'interno della mappa. Il marker, oltre ad indicare il punto preciso della rilevazione, indica anche il livello medio della potenza di segnale per la tecnologia di rete mobile impostata all'atto della ricerca. Infine i risultati vengono visualizzati attraverso del codice JavaScript, utilizzando le API specifiche fornite da Google per la manipolazione delle mappa. L'interfaccia mostrata all'utente è la seguente:



Insert location here (example: Misano Adriatico)

TIM
 Vodafone
 Wind
 H3G

EDGE
 UMTS
 HSDPA

Illustrazione 33: Interfaccia Web presentata all'utente

6. Valutazione delle performance lato server

Per valutare le performance e quindi la qualità dell'applicazione a runtime si è deciso di utilizzare una piattaforma di testing che permetta di analizzare il comportamento dell'applicazione in caso di carico variabile di richieste da parte del cliente. In questo modo è così possibile estrarre informazioni circa l'utilizzo sia in un caso reale, sia in un caso ipotetico di alto carico di richieste contemporanee.

6.1 Configurazione e modalità di test

La configurazione usata per il benchmark è la seguente:

- Notebook Acer Aspire 5553G equipaggiato con processore AMD Phenom II X4 N930 (quad-core, 2 GHz), 12 GB di RAM DDR3 e sistema operativo Ubuntu Linux 11.10 64-bit;
- JBoss Application Server 6.1 come container per l'applicazione Web/Enterprise;
- PC Desktop equipaggiato con processore AMD Athlon 64 X2 4200+ (dual-core, 2.2 GHz), 2 GB di RAM DDR e sistema operativo Ubuntu Linux 12.04 beta 1 32-bit;
- Router Ethernet/Wi-Fi Netgear DGN2200 IEEE 802.11 b/g/n (max 300 Mbps);
- Apache JMeter 2.6 come software di simulazione di carico per l'applicazione.

La fase di testing è stata suddivisa in base all'intensità di carico e alla locazione (remota o meno) dei client che effettuano le richieste. Per quanto riguarda il carico di richieste si è scelto di impostare JMeter con due diversi profili, in modo da poter confrontare i possibili casi d'utilizzo:

- In uno scenario di **carico reale**, si è deciso di impostare alcuni parametri di test in modo tale che al server possano arrivare richieste da un numero variabile di client all'interno di un certo intervallo di tempo che permetta al container di gestirle correttamente senza andare in sovraccarico e cercando di garantire uno throughput il più costante e accettabile possibile. I valori impostati prevedono un "Ramp-up Period" di 30 secondi, che indica l'intervallo di tempo entro il quale tutti i thread che simulano i clienti devono essere sicuramente attivati. Quindi, per esempio, se il numero di thread da attivare è 30 e il "Ramp-up Period" è di 30 secondi, il simulatore avvia 1 thread cliente al secondo. In aggiunta è stato inserito un "Gaussian Timer" che simula un ritardo di tempo tra una richiesta client ed un'altra, in modo da ritrovare un comportamento piuttosto vicino a quello reale, in cui tra una richiesta ed un'altra ci sia un particolare intervallo di tempo. In questo caso è stato impostato a 3 secondi con una deviazione standard (rispetto alla media) di 1 secondo. Fissando questi parametri quindi si è deciso di variare il numero di client da connettere e così confrontare i tempi di risposta dell'applicazione e di elaborazione delle richieste.
- In uno scenario di **carico di picco**, si è deciso di impostare i parametri di test in modo tale da indurre un sovraccarico dell'Application Server. In questo caso si è scelto un "Ramp-up Period" di 0 secondi, in modo tale che JMeter possa avviare tutti i thread nel tempo più breve possibile, simulando un possibile attacco di Denial of Service. Il "Gaussian Timer" è stato disattivato in modo tale che non ci sia un intervallo temporale tra una richiesta client ed un'altra. Anche in questo caso si sono fatti diversi test variando il numero di client attivati in contemporanea, fino a raggiungere il limite massimo computabile dal server per quanto riguarda il numero di connessioni parallele gestibili.

6. Valutazione delle performance lato server

Per quanto riguarda la locazione, i test di carico sono stati ripetuti in ugual modo sia raggiungendo l'applicazione in uno scenario remoto dove i client risiedono all'interno di un host remoto in rete locale sia all'interno dell'host stesso che ospita il server (quindi nell'host locale). In questo modo è stato possibile identificare eventuali latenze dovute al carico di rete o dovute all'alto carico computazionale della macchina che ospita l'Application Server, in particolare nel caso in cui la stessa debba gestire l'applicazione server e i client in modo simultaneo. In tutti i casi si è scelto di configurare il test in modo che richiedesse la pagina "index.jsp" tramite una HTTP Request con i seguenti parametri passati attraverso il metodo GET:

- location = Viale+del+Risorgimento,+2,+Bologna
- operator = TIM
- networkType = EDGE

6.2 Carico reale in localhost

Per verificare il comportamento dell'applicazione in localhost con un carico di client realistico si è deciso di utilizzare il notebook descritto nella configurazione, visto il quantitativo di processori e memoria RAM disponibile. In questo modo è stato avviato il server e quindi Apache JMeter con la configurazione adatta ad un test per carico reale. Successivamente sono state fatte tre acquisizioni, variando solamente il numero di client presenti nella simulazione in modo da poter effettuare una comparazione sensata dipendente solamente da tale parametro. Il test effettuato con 500 client ha prodotto il seguente risultato:

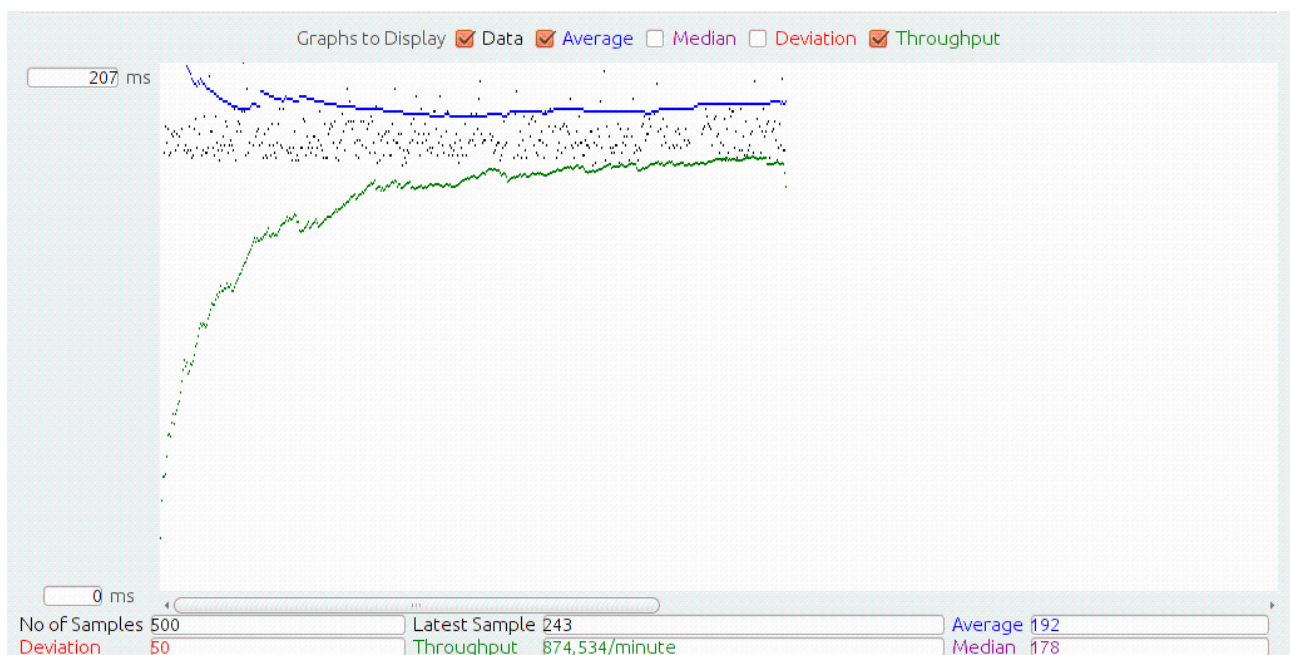


Illustrazione 34: Test applicazione server con carico reale in localhost: 500 client

Il grafico sopra riportato mostra la latenza per ogni richiesta (punti neri), la latenza media considerando tutte le richieste (linea blu) e lo throughput (richieste/min) del server. Dal grafico quindi si può notare la generale bassa latenza nel campionamento delle richieste (il valore massimo è di 243 ms) insieme ad uno throughput costante di circa 900 richieste/minuto una volta arrivato a regime, indice del fatto che l'applicazione è riuscita ad elaborare tutte le richieste senza perderne

6. Valutazione delle performance lato server

alcuna e senza avere comportamenti dovuti al sovraccarico (come ad esempio un abbassamento evidente dello throughput). A questo punto è interessante mostrare qual è il comportamento in caso di raddoppio dei client adibiti alle richieste, in modo da comparare i risultati sia in termini di tempi di latenza che in throughput generale del sistema.

Il test effettuato con 1000 client ha prodotto il seguente risultato:

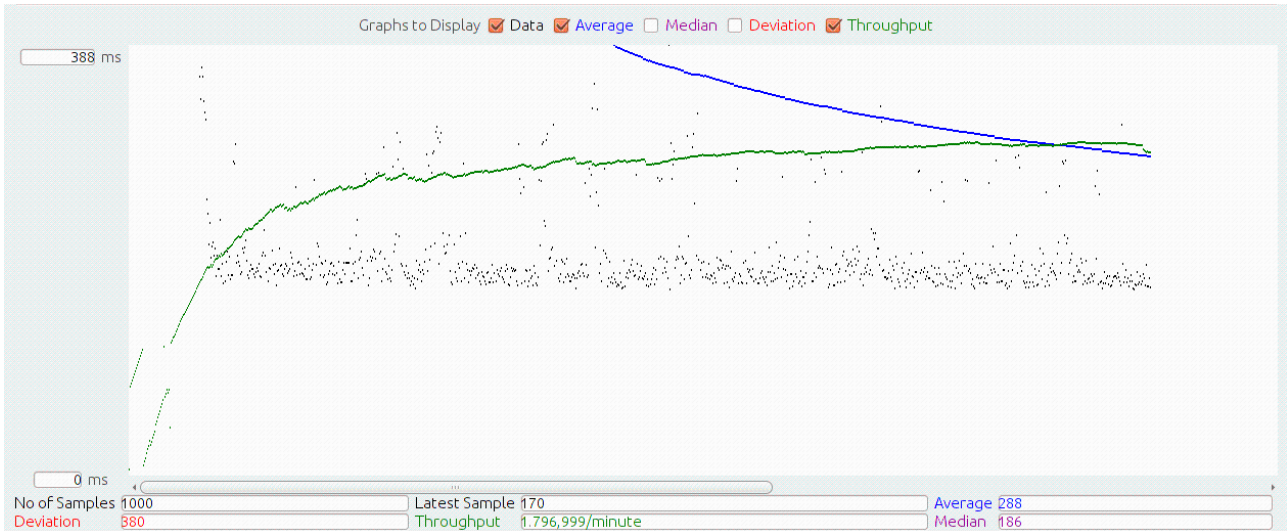


Illustrazione 35: Test applicazione server con carico reale in localhost: 1000 client

Anche in questo caso il tempo di campionamento di ogni singola richiesta è stato relativamente piccolo, con un ritardo massimo che si attesta sui 388 ms. Allo stesso modo lo throughput si è mantenuto all'incirca costante per tutto il tempo della prova, escludendo la fase iniziale in cui il server va a regime. Ciò è dovuto al fatto che l'Application Server riesce ancora a gestire le richieste in modo opportuno, senza generare situazioni di sovraccarico. Di seguito invece viene mostrato l'andamento prestazionale dell'applicazione con 2500 client:

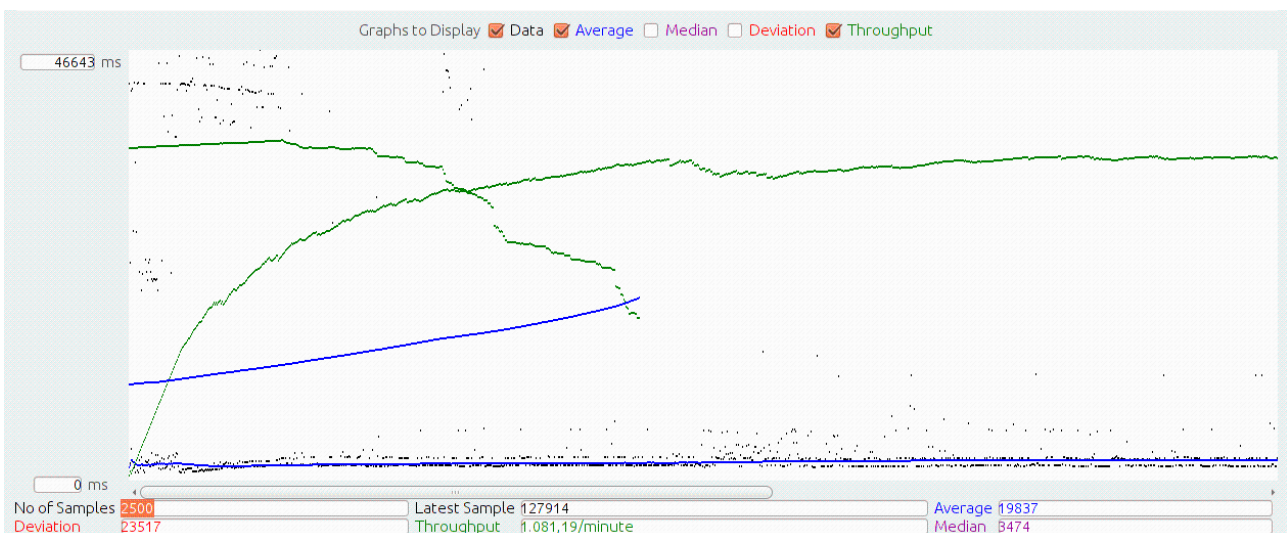


Illustrazione 36: Test applicazione server con carico reale in localhost: 2500 client

Questo caso interessante mostra il comportamento che si delinea con un carico che si avvicina al limite reale di dispatching delle richieste da parte del server. Infatti si può delineare un

6. Valutazione delle performance lato server

comportamento iniziale del tutto analogo alle precedenti prove effettuate, caratterizzato da un basso throughput e bassa latenza nella gestione delle richieste. Con l'aumentare e l'accumularsi di richieste pendenti, come si nota nelle curve sovrapposte e nei punti di maggior valore nella prima parte del grafico, l'andamento prestazionale del server tende a deteriorarsi, mostrando picchi di latenza delle richieste che si aggirano sui 128000 ms con conseguente aumento della latenza media e diminuzione dello throughput che, nella fase finale, cessa la sua andatura costante per diminuire inevitabilmente. Infatti, nel caso di 1000 client, l'ultimo valore di throughput rilevato è stato di circa 1800 richieste/minuto (valore circa costante per tutta la fase di test), mentre in questo caso l'ultimo dato rilevato è di circa 1080 richieste/minuto.

6.3 Carico di picco in localhost

Per valutare le performance in caso di carico elevato ed istantaneo in localhost è stato configurato JMeter nella modalità di simulazione di un carico di picco. In questo modo sono stati effettuati quattro test differenti: i primi 3 utilizzando un numero variabile di client, mentre l'ultimo utilizzando il numero massimo di client con l'aggiunta di un "Gaussian Timer" per evidenziare le differenze prestazionali nel caso di richieste inoltrate in maniera istantanea e in caso di richieste inviate con un intervallo prestabilito. Il seguente grafico mostra l'andamento nel caso di 500 client:

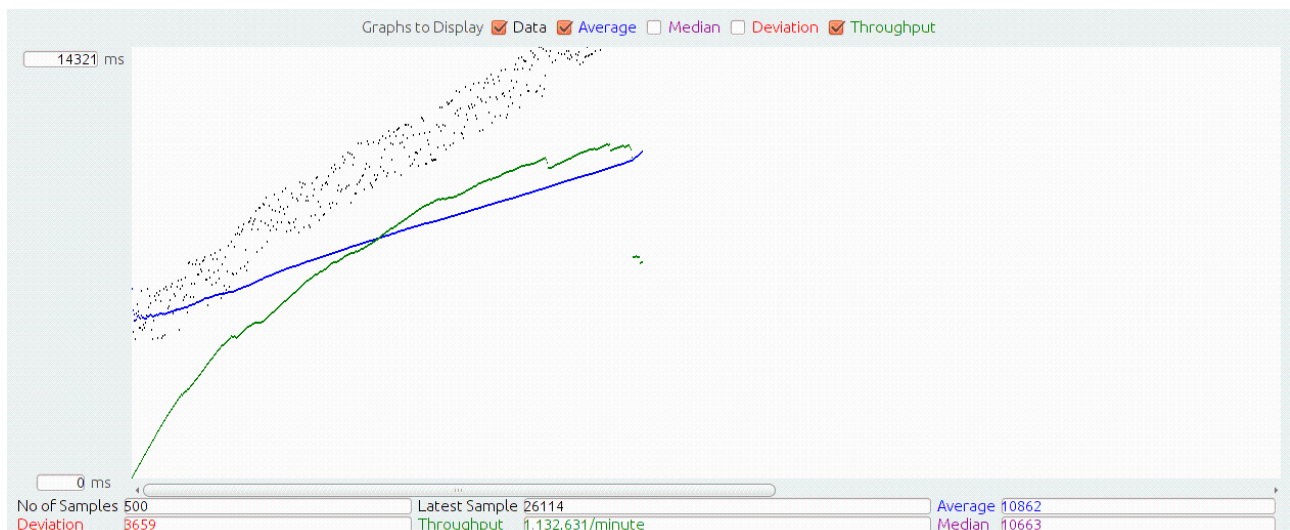


Illustrazione 37: Test applicazione server con carico di picco in localhost: 500 client

Con l'utilizzo di un "Ramp-up Period" pari a zero ed eliminando il "Gaussian Timer", le prestazioni dell'applicazione sono degradate per quanto riguarda il tempo di campionamento della richiesta rispetto all'analogo test condotto nel caso di carico reale. Al contrario lo throughput è aumentato, dato l'aumento repentino di richieste che il server ha dovuto elaborare in un breve lasso di tempo, costringendo quindi il server ad andare in un regime di computazione superiore rispetto al caso reale. In questo modo si è raggiunto il valore di circa 1600 richieste/minuto (valore poi crollato nella gestione delle ultime richieste pendenti) e un tempo di latenza massimo determinato dall'ultima richiesta elaborata di 26114 ms.

6. Valutazione delle performance lato server

Il seguente grafico mostra lo stesso test ripetuto con 1000 client in esecuzione:

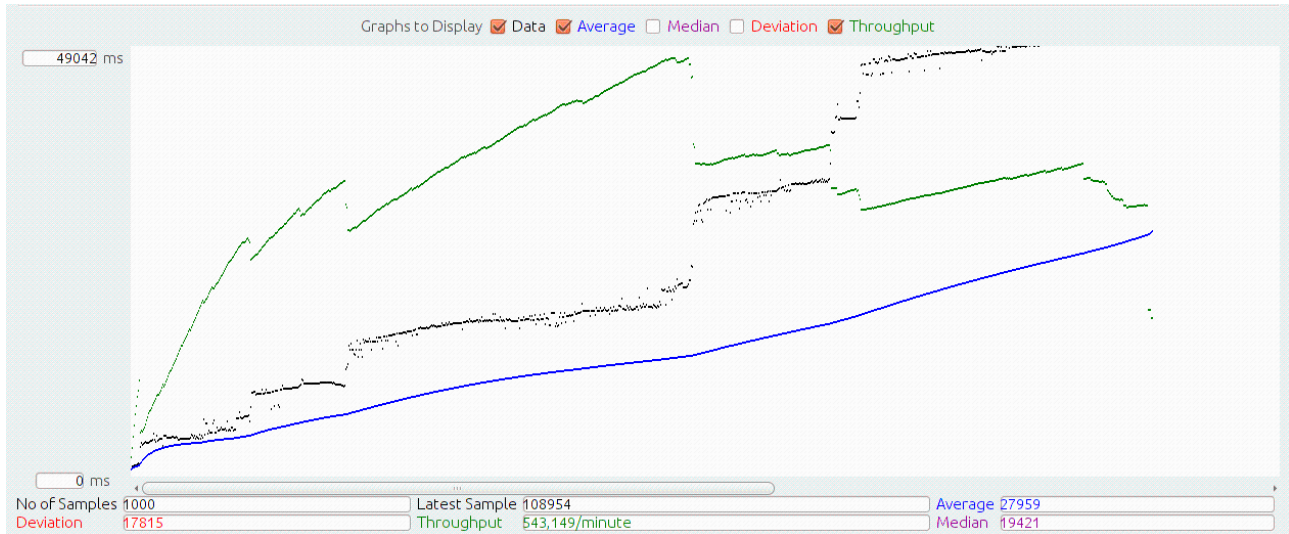


Illustrazione 38: Test applicazione server con carico di picco in localhost: 1000 client

In questo caso si può notare un degrado continuo delle prestazioni dopo una certa quantità di richieste elaborate. Il tempo di latenza delle richieste è in costante aumento, fino a raggiungere il picco di circa 109000 ms, mentre per quanto riguarda lo throughput si può notare un radicale degrado della capacità computazionale dopo 600 richieste elaborate. Infatti l'ultima rilevazione mostra un valore di 543 richieste/minuto, indice del fatto che il server sta raggiungendo il limite di elaborazione di richieste concorrenti. Il limite oltre il quale l'Application Server respinge le richieste è stato rilevato sperimentalmente ed è di 1800 client attivi in parallelo. Il seguente grafico mostra i risultati ottenuti:

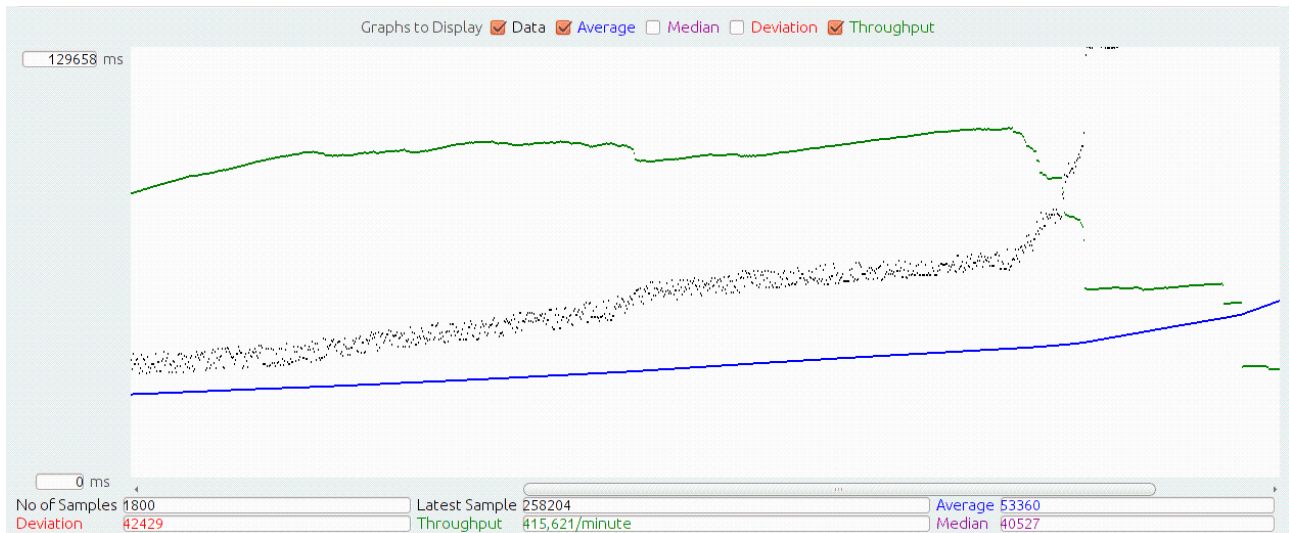


Illustrazione 39: Test applicazione server con carico di picco in localhost: 1800 client

L'utilizzo di 1800 client nella configurazione impostata per rilevare il carico di picco massimo ha permesso di individuare il limite massimo dell'applicazione per quanto riguarda il numero di connessioni gestibili contemporanee. Oltre tale numero, il server JBoss impone una politica di rifiuto di nuove connessioni, denotata da un'eccezione socket Java con il messaggio "Too many

6. Valutazione delle performance lato server

connections”. In questo caso il valore massimo di campionamento delle richieste è stato di 258204 ms, valore ricavato dall'ultima richiesta elaborata. Lo throughput totale è rimasto costante fino al 75% del test, fino a crollare ad un valore minimo di 415 richieste/minuto. Di seguito vengono mostrati i valori ricavati aggiungendo, alla configurazione di test usata in questo caso, un “Gaussian Timer” impostato a 3 secondi di ritardo tra una richiesta ed un'altra con una deviazione standard di 1 secondo:

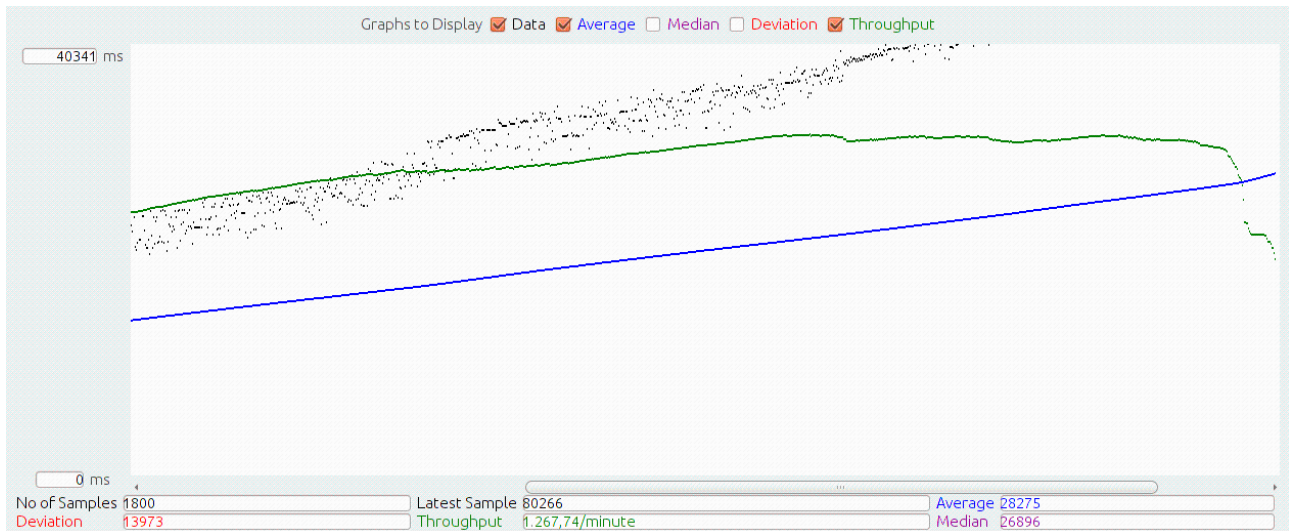


Illustrazione 40: Test applicazione server con carico di picco in localhost e Gaussian Timer: 1800 client

Il grafico mostra il netto aumento dello throughput e l'abbassamento conseguente dei tempi di latenza delle richieste. Infatti si è passati, per il primo parametro, a circa 1267 richieste/minuto come valore minimo (a fronte di 415 del test senza il timer) e a 80266 ms di latenza massima per il secondo parametro (a fronte di 258204 ms). La curva blu delinea anche in questo caso un progressivo aumento della latenza media, a maggior supporto dei dati ricavati sulle singole richieste indicate nel grafico tramite punti neri.

6.4 Carico reale in rete locale

Un'altra fase di testing molto importante per verificare le performance in caso di utilizzo dell'applicazione in un contesto remoto è stata svolta all'interno di uno scenario in rete locale. In questo modo è possibile apprezzare la variazione dei tempi di latenza e dello throughput per ogni test, condizionati dall'infrastruttura di rete intermedia, in modo poi da comparare tali dati con quelli ricavati negli scenari impostati in localhost. La prima fase si è svolta configurando Apache JMeter nella macchina desktop in modo che possa ricavare dati utili simulando uno scenario di uso reale. Quindi è stato impostato un numero variabile di client per verificare l'andamento del carico computazionale che il server può gestire. Anche in questo caso si è scelto di utilizzare il notebook come server, dato il superiore quantitativo di memoria RAM e di capacità computazionale disponibile. Per questo tipo particolare di test è stato necessario (sia per JBoss AS che per Apache JMeter) aumentare il quantitativo di memoria dedicato all'Heap in modo tale da non incorrere a problemi dovuti all'esaurimento della stessa a causa delle troppe risorse allocate destinate alla gestione delle connessioni.

6. Valutazione delle performance lato server

Il seguente grafico mostra l'andamento utilizzando 500 client:

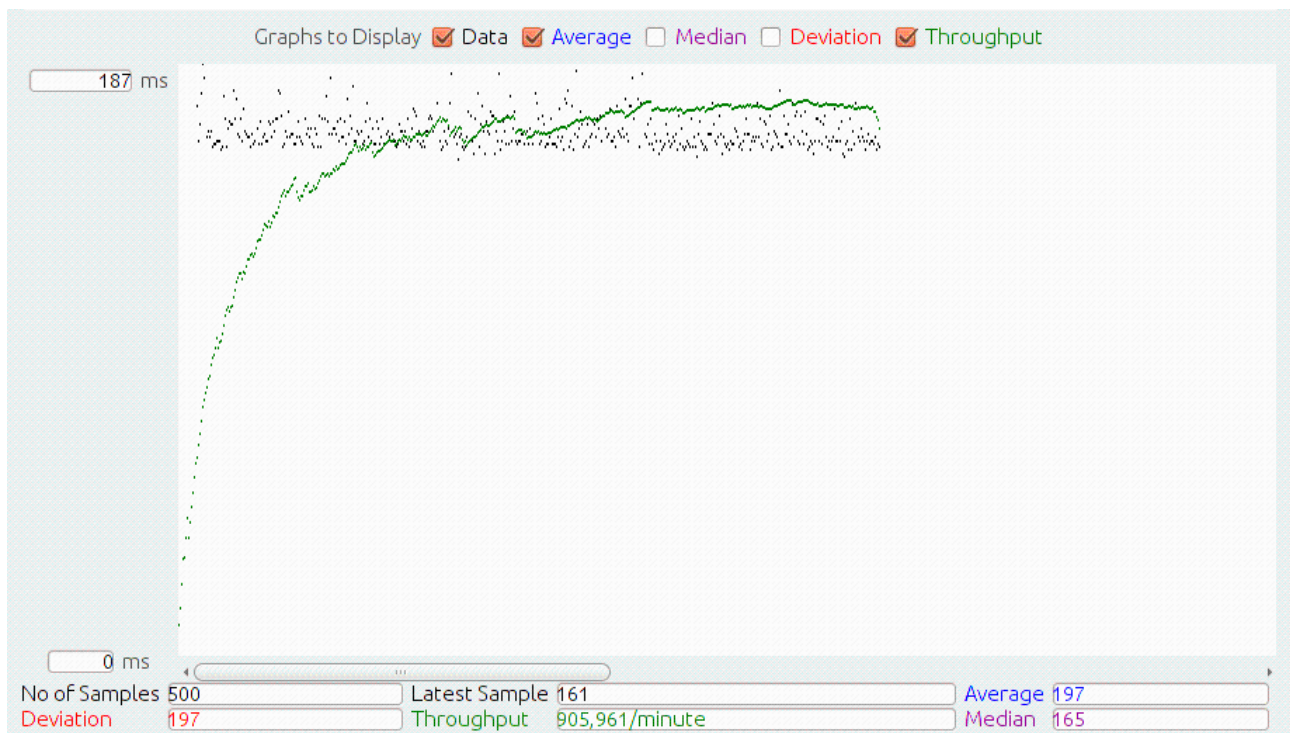


Illustrazione 41: Test applicazione server con carico reale in rete locale: 500 client

I dati ricavati dalla simulazione con 500 client mostra dei valori migliori rispetto allo stesso test effettuato in localhost: la latenza massima per il campionamento delle richieste è stata di 187 ms (a fronte di 243 ms) e lo throughput minimo rilevato a regime è di 905,961 richieste/minuto (a fronte di 874,534 richieste/minuto in localhost). Questo dato può essere influenzato dal fatto che, in questo caso, le Java Virtual Machine istanziate per l'esecuzione dei test (una per la parte cliente, l'altra per la parte server) risiedono su due macchine differenti e quindi, al contrario del caso in localhost, i costi di context switch e di gestione delle risorse per il mantenimento delle connessioni vengono ripartiti su due macchine differenti invece che in una sola. Infatti nel caso localhost, la macchina notebook ha l'incarico di mantenere almeno due JVM distinte all'interno dello stesso sistema, gestire le risorse di JBoss AS e di Apache JMeter (in modalità simulazione) e quindi di effettuare un numero maggiore di cambi di contesto che, nel caso di un numero di richieste relativamente basso, possono incidere in modo tangibile sulle performance dell'applicazione. Quindi l'infrastruttura di rete intermedia non contribuisce in modo visibile nel degradare le performance, rendendo del tutto trascurabile la latenza dei pacchetti scambiati. Lo stesso tipo di comportamento si è verificato anche con l'utilizzo di 1000 client: i valori di latenza di campionamento delle richieste e dello throughput hanno fatto registrare un andamento migliore rispetto allo stesso caso testato in localhost. In particolare si è registrato una latenza massima di 252 ms e uno throughput minimo a regime di 1720,183 richieste/minuto. Si suppone che tali risultati siano dovuti al motivo indicato sopra: la divisione dei carichi tra la macchina desktop (che esegue i client) e la macchina notebook (che esegue il server) aumenta in modo evidente le prestazioni dell'applicazione, tralasciando anche in questo caso il contributo dato dalla latenza dell'infrastruttura di rete.

6. Valutazione delle performance lato server

I dati ricavati sono rappresentati in questo grafico:

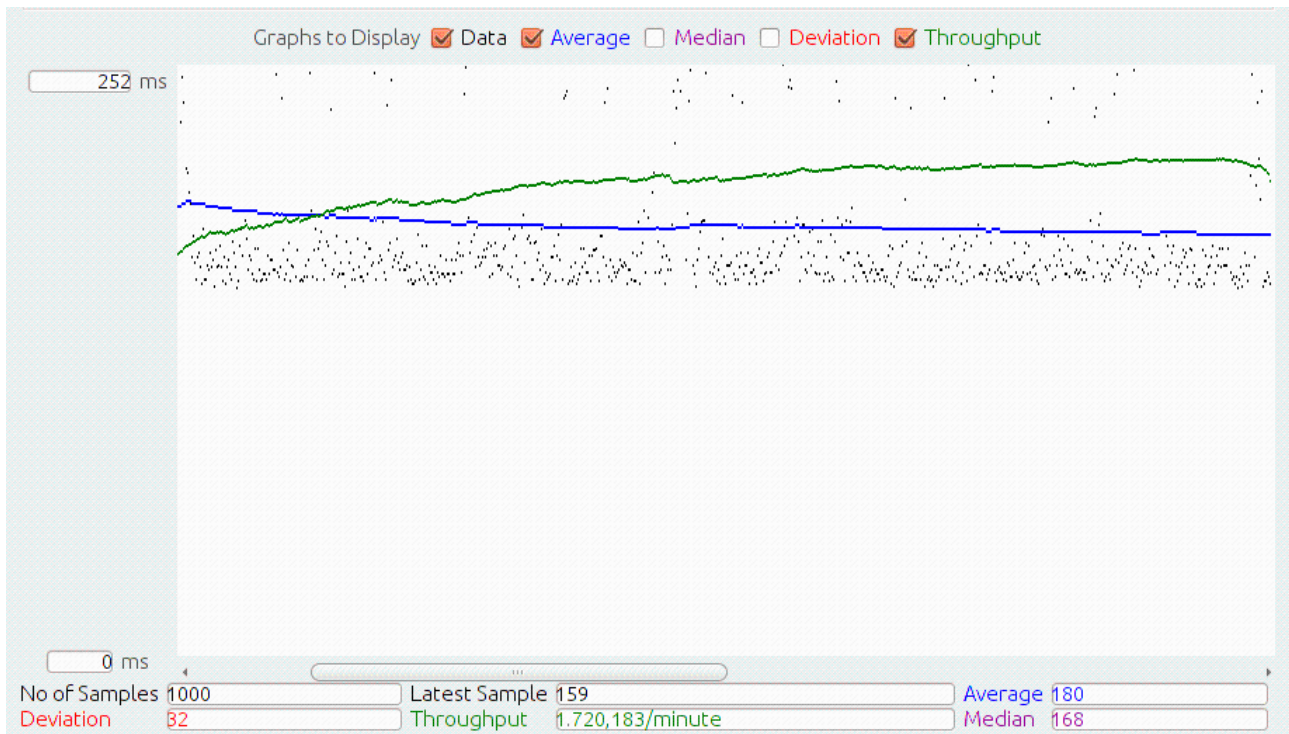


Illustrazione 42: Test applicazione server con carico reale in rete locale: 1000 client

Così come nel caso sopra, lo stesso comportamento si è verificato anche con l'utilizzo di 2500 client:

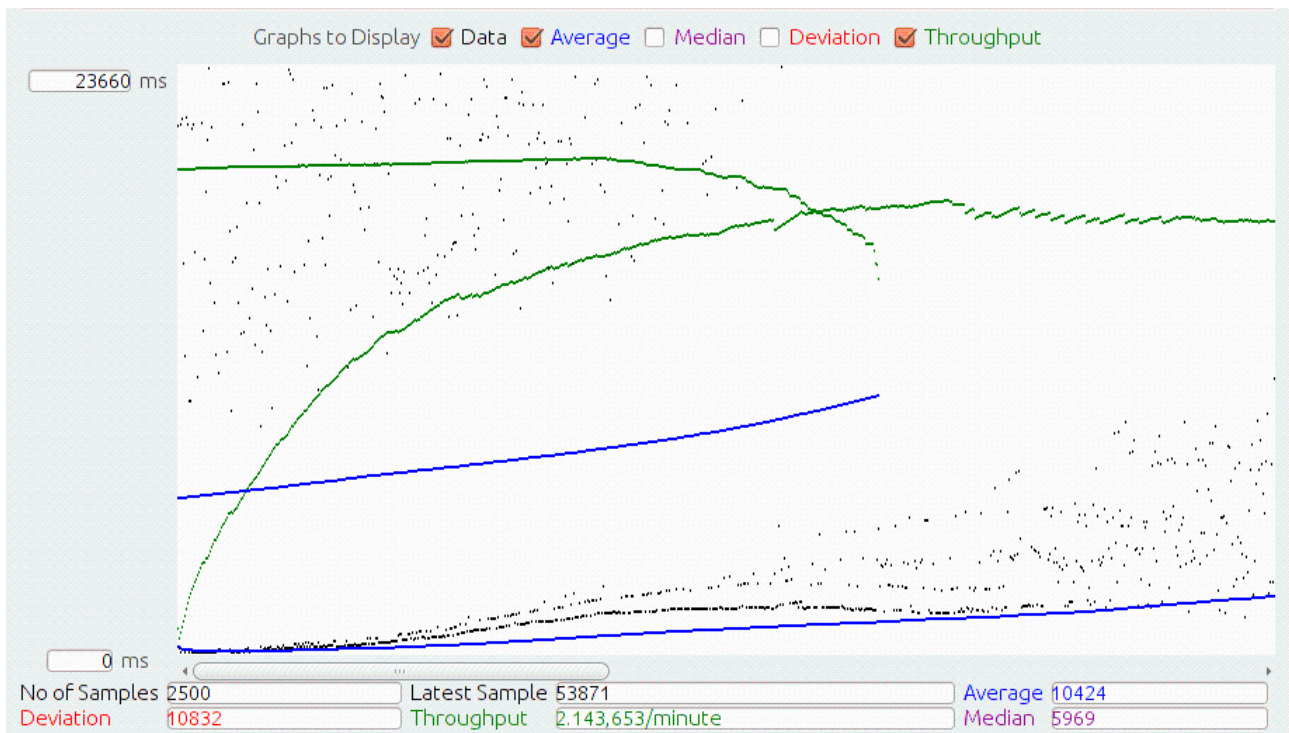


Illustrazione 43: Test applicazione server con carico reale in rete locale: 2500 client

6. Valutazione delle performance lato server

Il valore massimo di latenza ricavato è di 53871 ms (a fronte di 127914 ms ricavato in localhost) mentre lo throughput minimo ricavato a regime è di 2143,653 richieste/minuto (a fronte di 1081,19 richieste/minuto in localhost). Anche in questo caso è ipotizzabile un miglioramento delle performance a causa della divisione del carico tra le due macchine coinvolte.

6.5 Carico di picco in rete locale

In uno scenario di rete è importante valutare il carico massimo che l'infrastruttura server riesce a gestire, determinando quindi i tempi di risposta e la quantità di richieste processabili nell'unità di tempo. Il tutto influenzato dall'infrastruttura di rete che, con un numero alto di richieste attivate in un intervallo di tempo minimo, può introdurre ritardi o perdite di pacchetti. Come nei test effettuati in localhost, è stato configurato JMeter nella modalità di testing per la rilevazione del carico massimo gestibile da JBoss AS, variando analogamente il numero di client da attivare. Il primo test è stato eseguito con 500 client attivi e i risultati sono riportati all'interno del seguente grafico:

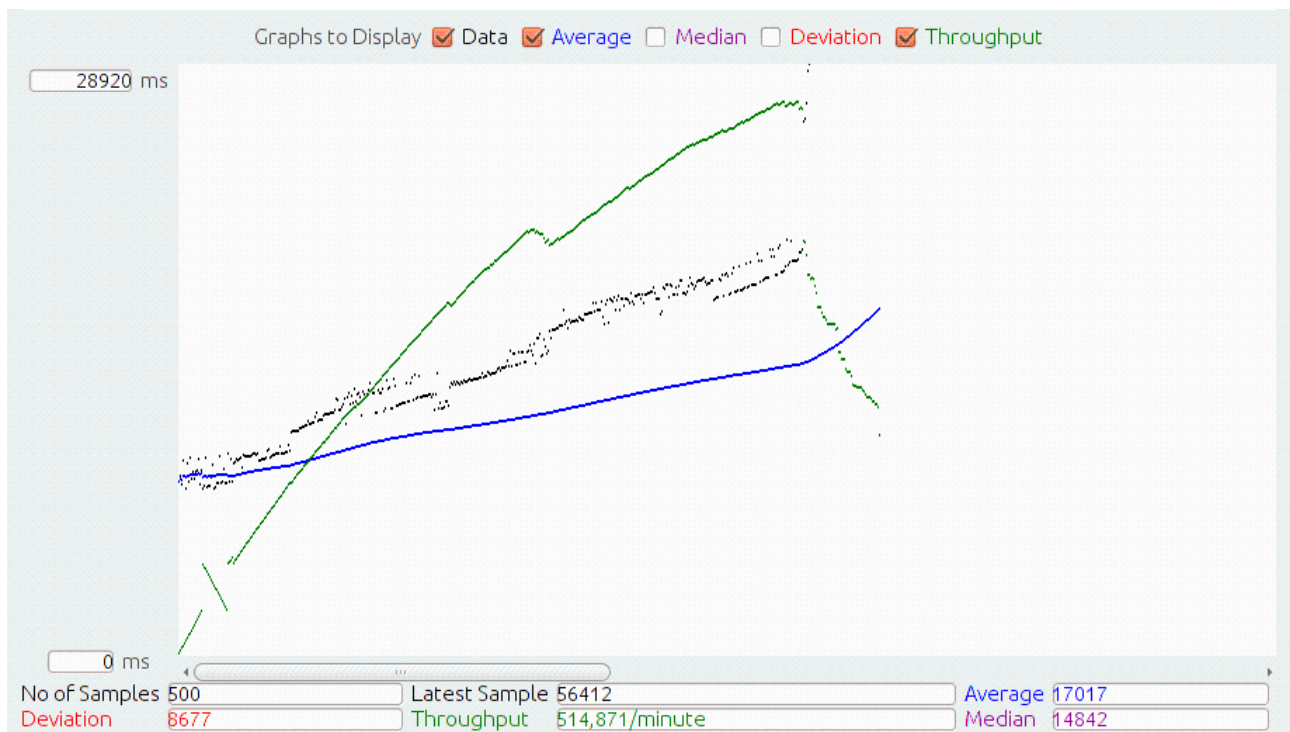


Illustrazione 44: Test applicazione server con carico di picco in rete locale: 500 client

Contrariamente a quanto si è verificato nei test di carico reale, la simulazione restituisce dei dati che mostrano il degrado di performance rispetto al test effettuato in localhost. Il valore massimo di latenza ricavato (quello dell'ultimo campionamento di richiesta) è di 56412 ms mentre il valore di throughput minimo è di 514,871 richieste/minuto. La causa di questo degrado può essere determinata da diversi fattori, tra cui un eccesso di pacchetti in coda all'interno del router e la latenza del trasferimento di questi all'interno della rete. Infatti in questo caso le connessioni vengono avviate in parallelo interamente all'avvio della simulazione. Questo, oltre che a presentare una possibile causa di sovraccarico per l'applicazione, può essere altrettanto causa di sovraccarico dell'infrastruttura di rete che può introdurre ritardi sempre più significativi all'avanzare della simulazione.

6. Valutazione delle performance lato server

La simulazione effettuata utilizzando 1000 client ha prodotto i seguenti risultati:

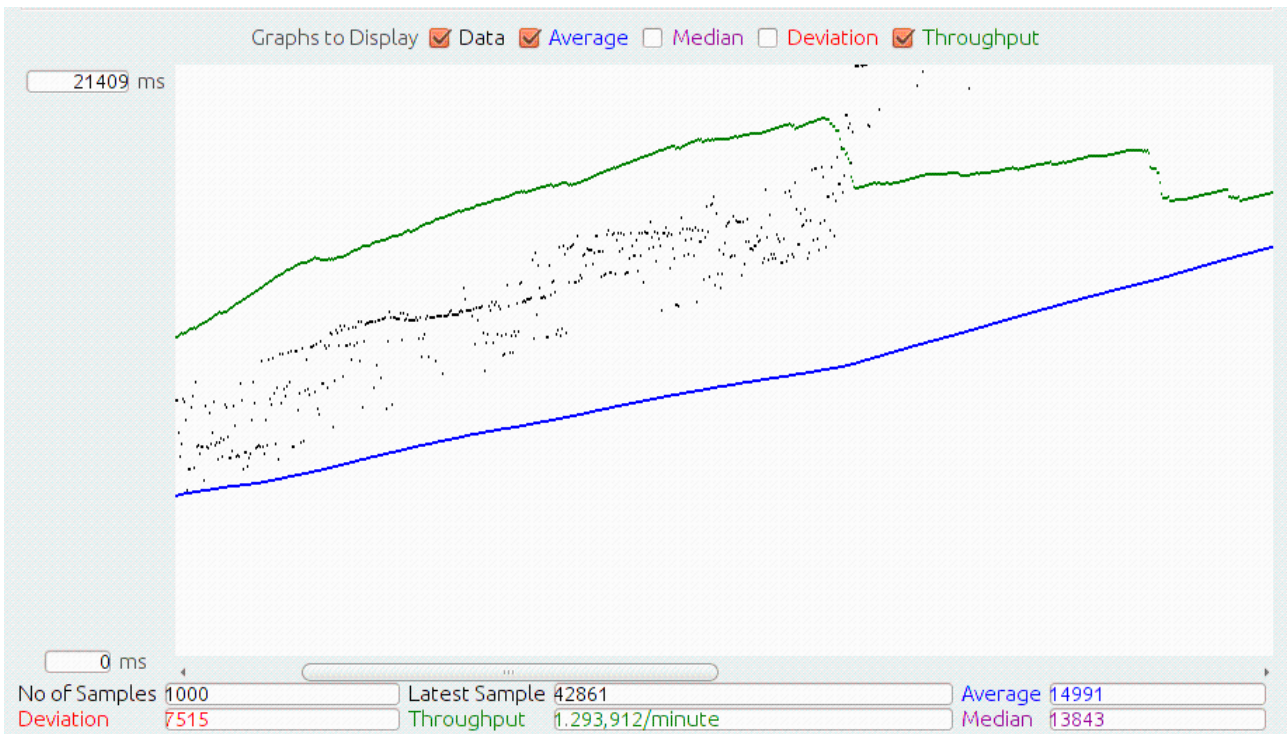


Illustrazione 45: Test applicazione server con carico di picco in rete locale: 1000 client

In questo caso si può notare un comportamento opposto rispetto a quello avuto nel test precedente. I valori di latenza e di throughput sono prestazionalmente migliori rispetto al benchmark effettuato in localhost, ponendo il valore di latenza massima pari a 42861 ms e lo throughput minimo a circa 1293,912 richieste/minuto. Il test con 1800 client ha prodotto il seguente grafico:

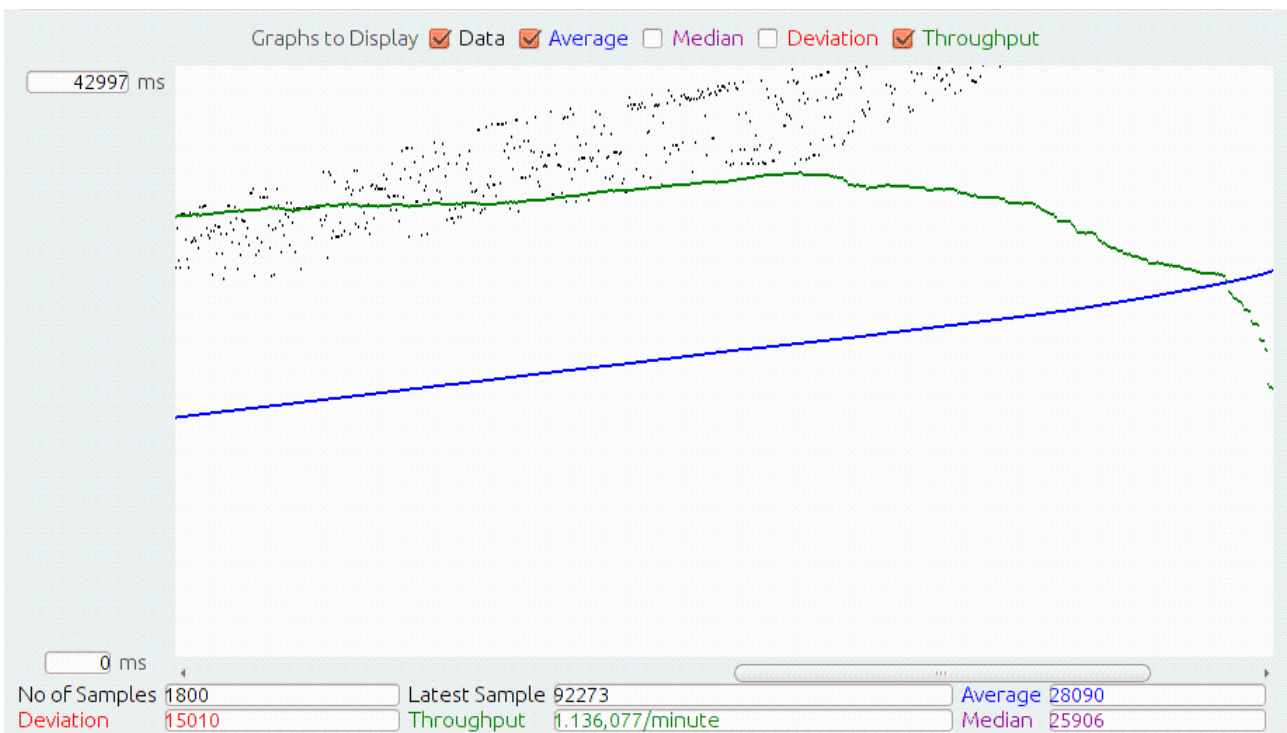


Illustrazione 46: Test applicazione server con carico di picco in rete locale: 1800 client

6. Valutazione delle performance lato server

Anche con l'avvio di 1800 client in modalità di carico di picco si può notare che i valori misurati mostrano una condizione più performante rispetto alla controparte in localhost. Il valore massimo di latenza ricavato è di 92273 ms mentre lo throughput si è attestato sui 1136,077 richieste/minuto come valore di minimo a regime. Un motivo può essere ricondotto allo stato dell'applicazione durante la fase di test: nel caso di 500 client, l'Application Server si è preso in carico di inizializzare il contesto di persistenza e quindi lo start-up dell'applicazione. Nei test successivi non è stato necessario ricreare molte strutture dati e ciò, oltre al fatto che client e server non risiedono sulla stessa macchina, potrebbe essere un motivo che giustifica l'aumento delle performance. Infine è stato fatto un test con 1800 client ma con l'aggiunta di un "Gaussian Timer" impostato come da test di carico reale:

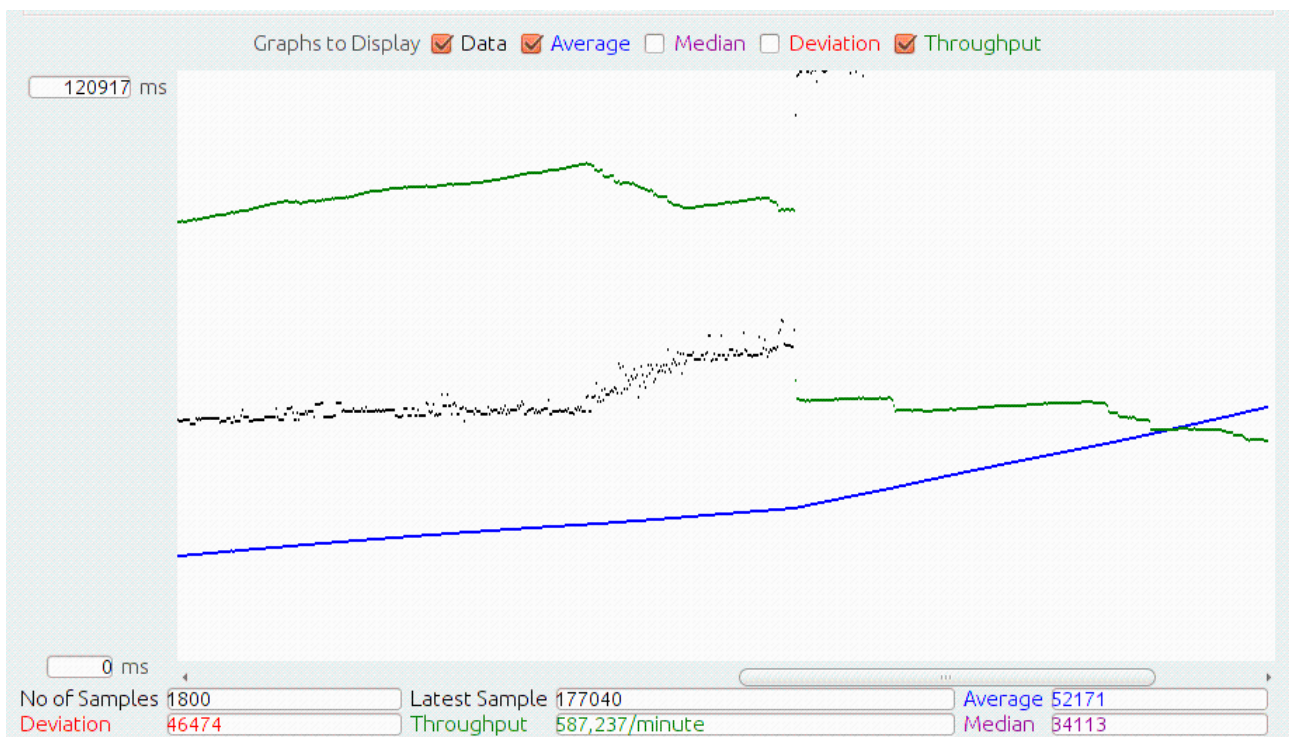


Illustrazione 47: Test applicazione server con carico di picco in rete locale e Gaussian Timer: 1800 client

Diversamente da quanto previsto, il test effettuato con un timer che impone un certo intervallo di tempo tra una richiesta e un'altra presenta valori di performance inferiori rispetto allo stesso caso privo di timer. In particolare si è ottenuta una latenza massima di campionamento della richiesta pari a 177040 ms e un valore di throughput minimo a regime pari a 587,237 richieste/minuto. Dal grafico si può notare che dopo il 60% del test le prestazioni cominciano a degradare fino ad avere un crollo evidente, probabilmente a causa di una improvvisa congestione di rete che ha reso l'infrastruttura incapace di recuperare i tempi di risposta avuti nei campioni precedenti.

Indice delle illustrazioni

Illustrazione 1: Stack architetturale della piattaforma Android.....	3
Illustrazione 2: Un esempio di dispositivo Android - “Nexus S” progettato da Google e Samsung....	4
Illustrazione 3: Ciclo di vita di un'Activity su piattaforma Android.....	6
Illustrazione 4: Sistema di coordinate utilizzato da SensorEvent API.....	10
Illustrazione 5: Sistema di riferimento per il sensore vettore di rotazione.....	13
Illustrazione 6: Ambiente di debug per Android - Dalvik Debug Monitor.....	17
Illustrazione 7: Acquisizione delle informazioni sul segnale telefonico.....	21
Illustrazione 8: Acquisizione delle informazioni di locazione.....	24
Illustrazione 9: Acquisizione delle informazioni di luminosità.....	26
Illustrazione 10: Esempio di report finale.....	28
Illustrazione 11: Esempio di report finale con salvataggio in locale.....	28
Illustrazione 12: Finestra di dialogo per l'abilitazione del modulo Bluetooth.....	30
Illustrazione 13: Activity per la gestione del trasferimento dati via Bluetooth.....	34
Illustrazione 14: Utilizzo della CPU nell'acquisizione del segnale telefonico.....	36
Illustrazione 15: Utilizzo della RAM nell'acquisizione del segnale telefonico.....	37
Illustrazione 16: Utilizzo della CPU nell'acquisizione dell'informazione sulla luminosità ambientale	38
Illustrazione 17: Utilizzo della RAM nell'acquisizione dell'informazione di luminosità ambientale	38
Illustrazione 18: Utilizzo della CPU nell'acquisizione della posizione tramite rete cellulare.....	39
Illustrazione 19: Utilizzo della RAM nell'acquisizione della posizione tramite rete cellulare.....	40
Illustrazione 20: Utilizzo della CPU nell'acquisizione della posizione tramite GPS.....	41
Illustrazione 21: Utilizzo della RAM nell'acquisizione della posizione tramite GPS.....	41
Illustrazione 22: Utilizzo della CPU nel trasferimento di singolo report via Bluetooth: parte client	42
Illustrazione 23: Utilizzo della RAM nel trasferimento di singolo report via Bluetooth: parte client	43
Illustrazione 24: Utilizzo della CPU nel trasferimento di singolo report via Bluetooth: parte server	44
Illustrazione 25: Utilizzo della RAM nel trasferimento di singolo report via Bluetooth: parte server	45
Illustrazione 26: Utilizzo della CPU nel trasferimento di report multiplo via Bluetooth: parte client	46
Illustrazione 27: Utilizzo della RAM nel trasferimento di report multiplo via Bluetooth: parte client	47
Illustrazione 28: Utilizzo della CPU nel trasferimento di report multiplo via Bluetooth: parte server	48
Illustrazione 29: Utilizzo della RAM nel trasferimento di report multiplo via Bluetooth: parte server	48
Illustrazione 30: Ambiente di sviluppo Eclipse per J2EE Developers.....	58
Illustrazione 31: Configurazione del debug remoto in Eclipse J2EE.....	59
Illustrazione 32: Schermata di diagnostica di MySQL Administrator.....	60
Illustrazione 33: Interfaccia Web presentata all'utente.....	74
Illustrazione 34: Test applicazione server con carico reale in localhost: 500 client.....	76
Illustrazione 35: Test applicazione server con carico reale in localhost: 1000 client.....	77
Illustrazione 36: Test applicazione server con carico reale in localhost: 2500 client.....	77
Illustrazione 37: Test applicazione server con carico di picco in localhost: 500 client.....	78
Illustrazione 38: Test applicazione server con carico di picco in localhost: 1000 client.....	79
Illustrazione 39: Test applicazione server con carico di picco in localhost: 1800 client.....	79
Illustrazione 40: Test applicazione server con carico di picco in localhost e Gaussian Timer: 1800 client.....	80
Illustrazione 41: Test applicazione server con carico reale in rete locale: 500 client.....	81

Illustrazione 42: Test applicazione server con carico reale in rete locale: 1000 client.....	82
Illustrazione 43: Test applicazione server con carico reale in rete locale: 2500 client.....	82
Illustrazione 44: Test applicazione server con carico di picco in rete locale: 500 client.....	83
Illustrazione 45: Test applicazione server con carico di picco in rete locale: 1000 client.....	84
Illustrazione 46: Test applicazione server con carico di picco in rete locale: 1800 client.....	84
Illustrazione 47: Test applicazione server con carico di picco in rete locale e Gaussian Timer: 1800 client.....	85

Fonti:

[1] **Android Developers:** <http://developer.android.com/>

[2] **Apache HttpComponents:** <http://hc.apache.org/index.html>

[3] **Google Gson:** <http://code.google.com/p/google-gson/>

[4] **JBoss Community:** <https://community.jboss.org/>

[5] **RESTful Web Services:** <http://www.oracle.com/technetwork/articles/javase/index-137171.html>

[6] **Fielding Dissertation - CHAPTER 6 - Experience and Evaluation:**
http://www.ics.uci.edu/~fielding/pubs/dissertation/evaluation.htm#sec_6_1

[7] **The Google Geocoding API - Google Maps API Web Services:**
<https://developers.google.com/maps/documentation/geocoding/>

[8] **Argo - A simple JSON parser and generator for Java:** <http://argo.sourceforge.net/>

[9] **Apache JMeter - User's Manual:** <http://jmeter.apache.org/usermanual/>