ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI INGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

Attività progettuale in Sistemi Distribuiti M e Sistemi Mobili M

Applicazione per il tracciamento di escursioni alpine e condivisione dei dati raccolti tra utenti

Autore

Stefano Poli

Docente

Prof. Ing. Paolo Bellavista

Anno accademico 2011/2012

Indice generale

Capitolo 1: Introduzione

La storia degli smartphone per il mercato di massa comincia con il successo dell'iPhone di Apple, ma in realtà il primo modello di telefono con funzioni "allargate" è datato 1993, costruito dalla IBM. In meno di 20 anni si è passati da rudimentali apparecchi dall'aria preistorica ai moderni gioielli della tecnologia odierna, forti di processori con potenza di calcolo paragonabile ai migliori supercomputer dei primi anni 90. Grazie all'evoluzione della microelettronica, gli attuali smartphone sono un agglomerato di moduli e funzionalità che li rendono la principale fonte di buisiness del mondo della telefonia. I produttori tendono ad integrare sempre più moduli all'interno dei loro smartphone, ponendo ricche funzionalità ed opportunità a disposizione dell'utente e degli sviluppatori di applicazioni. Moduli WiFi e GPS, sensori di prossimità, luminosità, rumore, ecc. estendono le potenzialità dei vecchi telefonini, trasformandoli in gioielli della tecnologia in grado di interpretare i gesti degli utenti e l'ambiente in cui sono immersi. Questa conoscenza del contesto in cui gli smartphone si trovano ad operare sta diventando sempre più il principale ramo di ricerca degli sviluppatori e delle università, che si trovano a realizzare strumenti user friendly in grado di aiutare l'utente in moltissime situazioni di vita. Per garantire un utilizzo uniforme e "pulito" di tutte le funzionalità offerte dagli attuali smartphone sono nati stack software avanzati che, mettendo a disposizione API e meccanismi automatici per la gestione delle risorse, semplificano notevolmente la vita degli sviluppatori. Esempi di questi evoluti stack software sono iPhone OS ed Android.

Se la piattaforma Java ha rivoluzionato il modo di pensare allo sviluppo software, la specifica Enterprise JavaBeans (EJB) ha rivoluzionato il modo di pensare allo sviluppo di software per l'impresa. EJB riunisce componenti lato server con tecnologie a oggetti distribuiti e servizi web per semplificare lo sviluppo di applicazioni; tiene conto automaticamente di molti dei requisiti dei sistemi aziendali, come la sicurezza, la gestione delle risorse, la persistenza, la concorrenza e l'integrità transazionale.

Il lavoro spiegato in questo documento si basa principalmente su due tecnologie: Android per la realizzazione di un'applicativo che fungerà da client, ed EJB per la realizzazione dell'applicativo che funzionerà come server.

Nel capitolo 2 è presente una panoramica delle tecnologie utilizzate per la realizzazione del nostro progetto. In particolare, saranno descritte la piattaforma Android con i relativi supporti all'utilizzo dei sensori, la tecnologia WiFi Direct che consente l'instaurazione di reti WiFi ad-hoc, la specifica Java Enterprise Edition comprendente la tecnologia Enterprise JavaBeans, la tecnologia Java Persistence API che assiste la persistenza dei dati, l'approccio REST alla realizzazione di Web Services e l'application server JBoss.

Il capitolo 3 descrive l'applicativo realizzato per la piattaforma Android, entrando nei dettagli implementativi ed illustrando le tecniche utilizzate per l'utilizzo delle funzionalità avanzate di un moderno smartphone.

Il capitolo 4 illustra la parte server realizzata con tecnologia Enterprise JavaBeans, sviluppata secondo l'architettura 4-tier e composta quindi da una parte dedicata alla persistenza dei dati, una dedicata alla logica di business, una dedicata alla parte di presentazione ed una dedicata alla visualizzazione dei risultati da parte del cliente.

Il capitolo 5 mostra e spiega i risultati ottenuti dai test sperimentali eseguiti sulla parte server del nostro progetto, focalizzando l'attenzione sui tempi di risposta del server a richieste mirate a sollecitare le sue funzionalità chiave.

1.1 Obiettivi

Questo progetto ha lo scopo di approfondire le tecnologie Android ed EJB, sfruttando opportunamente sensori e moduli particolari di cui sono dotati i moderni smartphone, ed utilizzando tecnologie enterprise per la realizzazione di un'applicazione in grado di memorizzare, elaborare ed aggregare informazioni raccolte dall'applicazione client.

Lo scenario che ci poniamo per consentirci lo sviluppo di un sistema sensato ed utilizzabile nella realtà, è uno scenario alpino. Il sistema ha il compito di realizzare una sorta di “diario dell'escursionista alpino” condiviso tra più utenti. Il sistema si dividerà quindi in due parti che ci consentono di approfondire separatamente le tecnologie sopra citate: una parte client realizzata su dispositivo Android ed una parte server implementata con tecnologia EJB.

Il client Android avrà il compito di tracciare il percorso seguito dall'escursionista e di raccogliere altri dati utili all'elaborazione di un report contenente valutazioni statistiche sul percorso svolto.

Oltre a questo, dovrà essere possibile scattare fotografie geotaggate, scambiare dati con altri dispositivi incrociati dall'escursionista durante il suo percorso ed inviare dati ad un server centrale sfruttando un approccio di tipo REST.

Il server EJB dovrà consentire una condivisone dei dati raccolti a tutti i suoi utenti. In particolare dovrà implementare un sistema di ricezione dati dai dispositivi client seguendo un approccio REST, memorizzare tali dati e renderli disponibili per la visualizzazione testuale e grafica (i tracciati seguiti dagli escursionisti saranno visualizzati su una mappa) da parte degli utenti. Dovrà, inoltre, essere implementato un meccanismo di ottimizzazione per il ritrovamento dei dati richiesti dai vari utenti del sistema. Tale obiettivo sarà raggiunto realizzando un meccanismo di prefetch orientato verso un principio di località geografica: i dati delle escursioni verranno precaricati in una cache in base alla loro posizione geografica.

Capitolo 2: Tecnologie utilizzate

All'interno di questo progetto sono state utilizzate diverse tecnologie, prodotti e metodologie su cui vale la pena scrivere qualche riga. Iniziamo suddividendo il tutto in base alla parte di progetto in cui sono utilizzate:

- Parte client
  - Piattaforma Android
  - Sensori e GPS
  - WiFi Direct
- Parte server
  - Java Enterprise Edition ed EJB
  - Java Persistence API e Hibernate
  - RESTful Web Services
  - JBoss

Nel seguito è presente una panoramica su ognuna delle tecnologie citate.

2.1 Piattaforma Android

Android è una piattaforma software per dispositivi mobili formata da uno stack di software disposto su quattro livelli (Fig. 2.1):

- un sistema operativo
- un livello di librerie scritte in linguaggio nativo C/C++
- un application framework che fornisce servizi evoluti, sempre incapsulati in oggetti java, alle applicazioni

- contiene l'*Android Runtime*, ambiente di esecuzione delle applicazioni basato su una *Dalvik Virtual Machine*

- un livello delle applicazioni comprendente le applicazioni:
  - core: fornite dal sistema
  - scritte dagli sviluppatori

Fig. 2.1: Stack della piattaforma Android

Ogni applicazione Android viene eseguita in un proprio processo che è in esecuzione su una propria istanza di una *Dalvik Virtual Machine*, una macchina virtuale *registry-based* creata in modo da avere un'esecuzione multipla efficiente su un singolo dispositivo. Android si basa sul kernel Linux versione 2.6 (le ultime versioni di Android si basano sul kernel 3.x) per i servizi core system come sicurezza, gestione della memoria, gestione dei processi, rete e driver. Tale kernel funge anche da livello di astrazione tra l'hardware ed il resto dello stack software.

Un'applicazione Android è raggruppata in un *Android package* (file di archivio con suffisso `.apk`) e contiene file java e di altri tipi. Questo package è il veicolo di distribuzione e di installazione dell'applicazione sui dispositivi. Di default, ogni applicazione Android vive dentro un

proprio mondo, infatti:

- ogni applicazione è eseguita in un proprio processo Linux. Android lancia il processo quando qualche parte del codice dell'applicazione necessita di essere eseguita e termina il processo quando ciò non è più necessario e altre applicazioni hanno bisogno di risorse di sistema;
- ogni processo ha una propria virtual machine, così ogni applicazione è eseguita in modo isolato dalle altre;
- ad ogni applicazione è assegnato un unico ID Linux. I permessi sono impostati in modo che i file di un'applicazione siano visibili solo all'utente e all'applicazione stessa, ma volendo possono essere anche condivisi.

Queste sono tutte proprietà modificabili.

Uno degli aspetti centrali di Android è che un'applicazione può far uso di elementi di altre applicazioni (se queste ultime lo permettono). Android è stato creato per permettere ciò, infatti a differenza di molti altri sistemi, le applicazioni Android non hanno un entry point unico nell'applicazione (ad es. `main()`), ma sono composte da componenti essenziali che il sistema può istanziare e avviare secondo necessità (non tramite importazione di codice o creazione di link). Questi componenti sono di quattro tipi:

- *activity*: componente che consente l'interazione diretta tra utente e applicazione. Rappresenta l'interfaccia grafica dell'applicazione;
- *service*: esegue operazioni in background, non ha interfaccia utente ed ha tipicamente vita lunga. Non ha un processo/thread dedicato;
- *broadcast receiver* : risponde agli *Intent* (richiesta di compiere un'operazione) compatibili eseguendo le operazioni previste; in genere svolge azioni di notifica. Il suo ciclo di vita è limitato alla risposta all'*Intent*.
- *content provider*: consente l'accesso a risorse come file e dati. Per la memorizzazione persistente dei dati, la piattaforma Android si avvale del RDBMS *SQLite*.

Prima che Android possa avviare un componente di un'applicazione deve sapere dell'esistenza di tale componente, quindi le applicazioni devono dichiarare i propri componenti in un file *manifest* dal nome `AndroidManifest.xml`.

L’idea base del sistema Android è che le applicazioni saranno eseguite su dispositivi con limitata capacità di memoria. Questo comporta la possibilità che un processo (e gli elementi di un’applicazione in esso contenuti) possa essere eliminato dal sistema in caso di carenza di memoria. La decisione riguardo a quale processo rimuovere è strettamente legata allo stato dell’interazione che l’utente ha con questo. Il sistema calcola l’importanza di un processo in base all’elemento più importante in esso contenuto. Il sistema terminerà il processo meno importante prima di essere obbligato a terminare i più importanti.

2.1.1 Ciclo di vita di un'*Activity*

Ogni componente Android è una macchina a stati gestita dalla piattaforma. L’invio di un intent ad un componente può innescarne l’esecuzione, cioè provocare la creazione del componente e il passaggio dallo stato di componente-creato allo stato di componente-attivo all’interno del proprio specifico processo di esecuzione. La figura 2.2 mostra il ciclo di vita di una Activity, che si articola in diversi stati, tra cui:

- *active* (o *running*), se l’activity è visualizzata in primo piano sullo schermo. Il processo che ospita tale activity ha massima priorità in questo momento e quindi è l’ultimo candidato per essere terminato dal sistema.
- *paused*, se ha perso il focus ma è ancora visibile (se l’activity in primo piano non è a schermo intero). Se un processo ospita solo questa activity (opppure ospita anche altre activity in stato *paused* o *stopped*) ha alta priorità in questo momento e nel caso di grave mancanza di memoria sarà eliminato dal sistema dopo tutti gli altri processi con priorità inferiore ma prima dei processi con priorità superiore (ad es. un processo che ospiti un’activity in stato *active*).
- *stopped*, se l’activity è completamente coperta da un’altra activity. Se un processo ospita solo activity in questo stato allora ha bassa priorità in questo momento e nel caso di mancanza di memoria sarà eliminato dal sistema dopo tutti gli altri processi con priorità inferiore ma prima dei processi con priorità superiore (ad es. un processo che ospiti un’activity in stato *active* o *paused*).

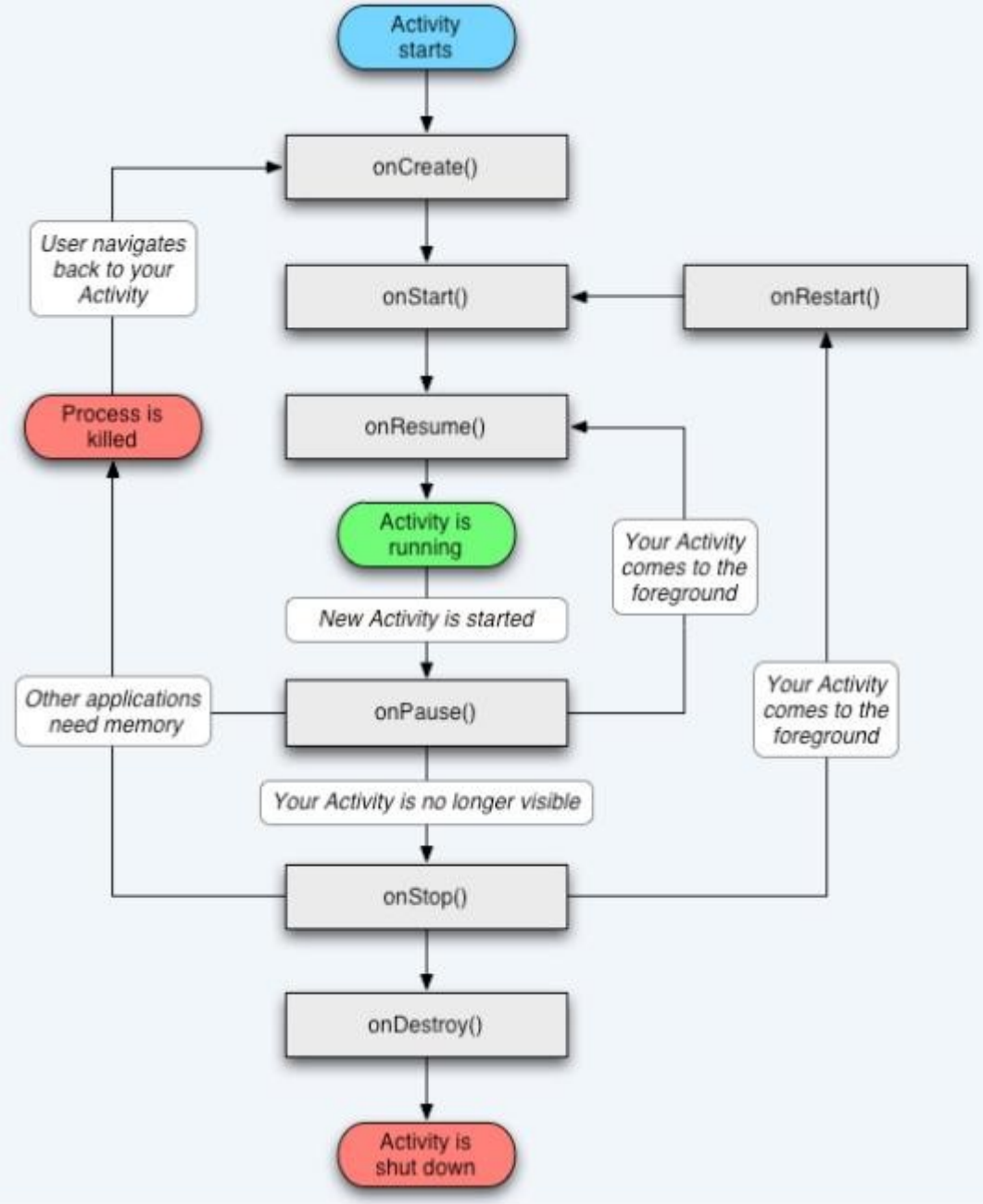

Fig. 2.2: Ciclo di vita di una activity

Se un'activity in stato *paused* o *stopped* viene rimossa dal sistema, allora, quando vi sarà necessità di essere di nuovo mostrata all'utente, il componente (e il relativo processo) verrà avviato di nuovo e il suo stato sarà ripristinato nella situazione precedente all'eliminazione. La priorità di un processo è stabilita in base al componente più importante del processo.

Le transizioni di stato sono percepibili dall'utente finale in quanto la piattaforma invoca per ciascuno di essi uno specifico metodo di callback del componente, ad es. `onCreate`, `onDestroy`, ecc..

La figura 2.3 mostra in modo schematico alcuni aspetti essenziali dell'architettura logica di supporto al funzionamento di una Activity. Si noti in particolare che:

- ogni activity può avviare un'altra activity, e Android organizza l'esecuzione delle activity in uno stack ;
- lo stack delle activity è la rappresentazione Android di un *task*, inteso come una sequenza di activity che l'utente esegue per portare a termine un compito;
- una activity è dotata di un *Main Thread* e può accedere alla coda di messaggi gestita da questo Thread. Inoltre un'activity può lanciare altri Thread per eseguire lavoro in background.

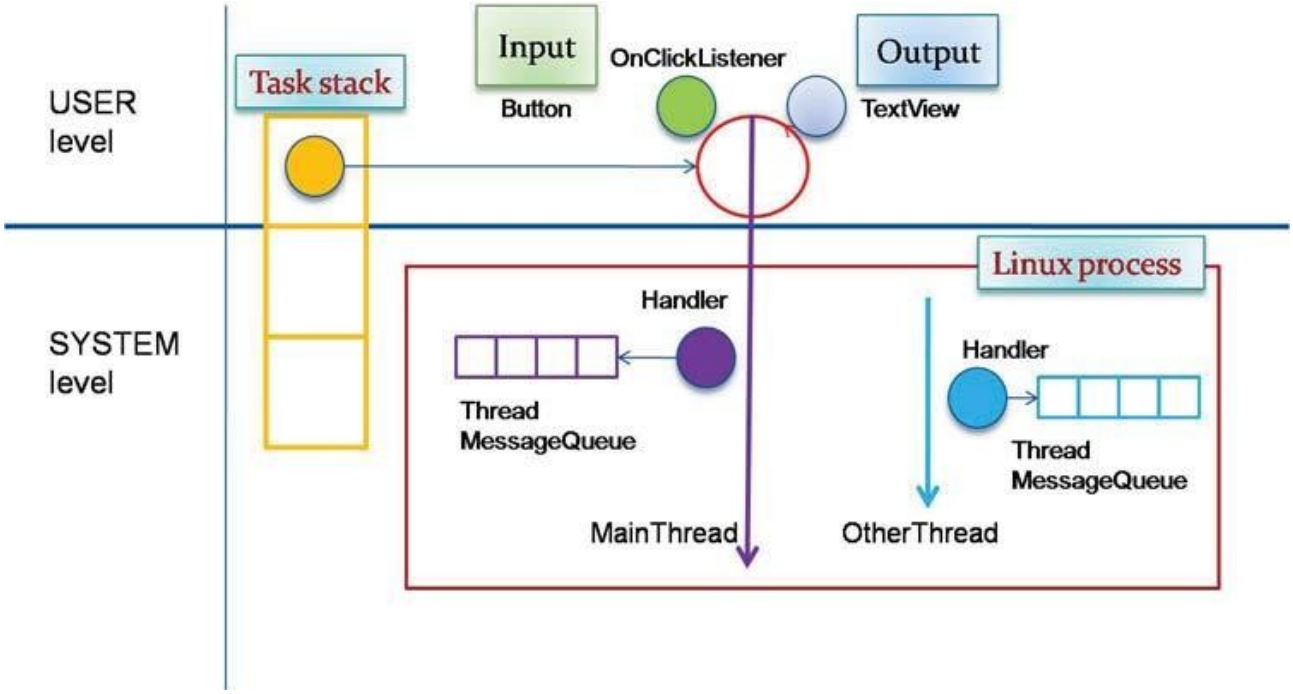

Fig. 2.3: Architettura logica per activity

2.2 Global Positioning System (GPS)

Il Sistema di Posizionamento Globale, (in inglese: *Global Positioning System*, abbreviato *GPS*) è un sistema di posizionamento e navigazione satellitare che, attraverso una rete satellitare dedicata di satelliti artificiali in orbita, fornisce ad un terminale o ricevitore GPS informazioni sulle sue coordinate geografiche ed orario, in ogni condizione meteorologica, ovunque sulla Terra o nelle sue

immediate vicinanze ove vi sia un contatto privo di ostacoli con almeno quattro satelliti del sistema, tramite la trasmissione di un segnale radio da parte di ciascun satellite e l'elaborazione dei segnali ricevuti da parte del ricevitore.

Il sistema GPS è gestito dal governo degli Stati Uniti d'America ed è liberamente accessibile da chiunque dotato di ricevitore GPS. Il suo grado attuale di accuratezza è dell'ordine dei metri, in dipendenza dalle condizioni meteorologiche, dalla disponibilità e dalla posizione dei satelliti rispetto al ricevitore, dalla qualità e dal tipo di ricevitore, dagli effetti di radiopropagazione del segnale radio in ionosfera e troposfera (es. riflessione) e degli effetti della relatività.

Attualmente sono in orbita 31 satelliti attivi nella costellazione GPS (più alcuni satelliti dismessi, alcuni dei quali riattivabili in caso di necessità). I satelliti supplementari migliorano la precisione del sistema permettendo misurazioni ridondanti. Al crescere del numero di satelliti, la costellazione è stata modificata secondo uno schema non uniforme che si è dimostrato maggiormente affidabile in caso di guasti contemporanei di più satelliti.

Il principio di funzionamento si basa su un metodo di posizionamento sferico (*trilaterazione*), che parte dalla misurazione del tempo impiegato da un segnale radio a percorrere la distanza satellite-ricevitore (*ToA: Time Of Arrival*). Poiché il ricevitore non conosce quando è stato trasmesso il segnale dal satellite, per il calcolo della differenza dei tempi il segnale inviato dal satellite è di tipo orario, grazie all'orologio atomico presente sul satellite: il ricevitore calcola l'esatta distanza di propagazione dal satellite a partire dalla differenza (dell'ordine dei microsecondi) tra l'orario pervenuto e quello del proprio orologio sincronizzato con quello a bordo del satellite, tenendo conto della velocità di propagazione del segnale. L'orologio a bordo dei ricevitori GPS è però molto meno sofisticato di quello a bordo dei satelliti e quindi, non essendo altrettanto accurato sul lungo periodo, deve dunque essere corretto frequentemente. In particolare la sincronizzazione di tale orologio avviene all'accensione del dispositivo ricevente utilizzando l'informazione che arriva dal quarto satellite, venendo così continuamente aggiornata. Se il ricevitore avesse anch'esso un orologio atomico al cesio perfettamente sincronizzato con quello dei satelliti basterebbero le informazioni fornite da 3 satelliti, ma nella realtà non è così e dunque il ricevitore deve risolvere un sistema di 4 incognite (latitudine, longitudine, altitudine e tempo) e per riuscirci necessita dunque di 4 equazioni. La figura 2.4 mostra, nel caso a 2 dimensioni, come l'errore di clock (righe continue) porti a determinare una posizione scorretta: la vera posizione del nodo mobile è nel punto A. La figura 2.5 mostra come viene corretto l'errore con l'aggiunta di un terzo satellite (sempre nel caso a

2 dimensioni): i cerchi con tratto continuo (errati) intersecano in 3 punti B, quindi è sufficiente effettuare uno shift dell'orologio fino a che i 3 cerchi non intersecano in un unico punto (tratto tratteggiato) A.

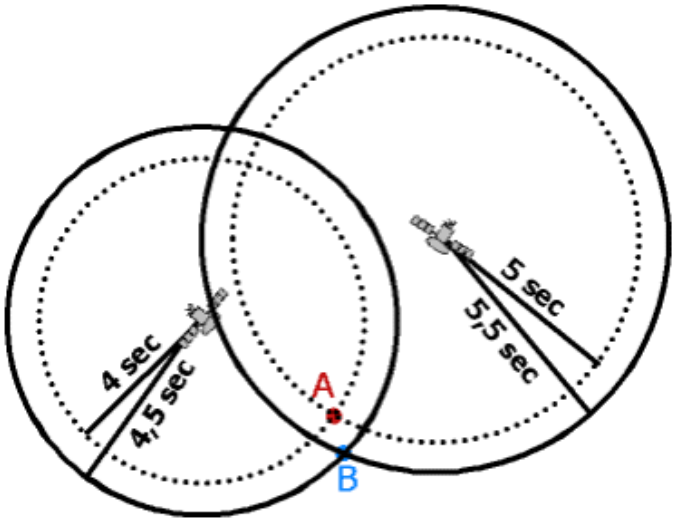

Fig. 2.4: Errore di clock

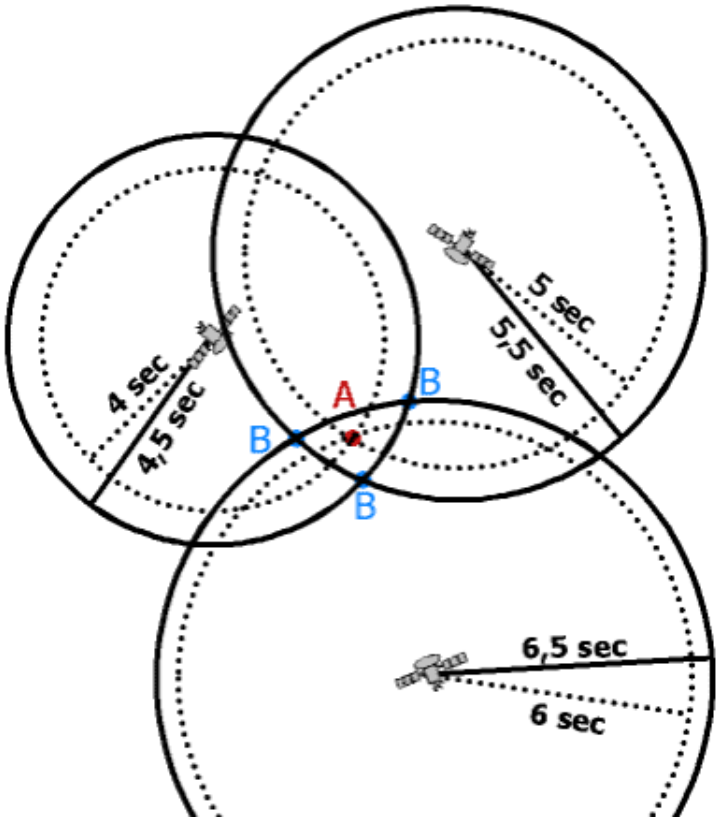

Fig. 2.5: Correzione dell'errore di clock

Un semplice protocollo di sincronizzazione è il seguente: il dispositivo calcola la propria posizione utilizzando 3 satelliti, poi misura il ToA con un quarto satellite (conoscendo già la propria posizione, il dispositivo sa quanto dista il quarto satellite) e, se il ToA è quello atteso, gli orologi sono sincronizzati, altrimenti la differenza tra ToA atteso e misurato rappresenta la correzione da effettuare sul clock del dispositivo.

Oltre al problema dato dalla sincronizzazione degli orologi, ne esistono altri che possono essere causa di errori nel sistema di posizionamento GPS:

- condizioni atmosferiche: ionosfera e troposfera rifraggono il segnale causando quindi cambiamenti nella velocità di propagazione;
- errori dovuti alle effemeridi: errori di posizionamento dei satelliti nell'orbita satellitare;
- clock drift: errori negli orologi atomici dei satelliti;
- “rumore” di misurazione;
- selective availability: errori ad-hoc introdotti dal dipartimento dio difesa americano;
- multipath: errori dovuti all'arrivo del segnale al ricevitore attraverso più percorsi (causati da edifici, montagne, ecc.).

Per correggere buona parte di questi errori è stato introdotto il GPS Differenziale (D-GPS). Il D-GPS sfrutta una stazione a terra con posizione perfettamente nota per calcolare le differenze di posizionamento esatto ed eliminare gli errori. L'errore misurato dal calcolo della posizione della stazione fissa vicina con sistema GPS non differenziale, viene inviato via radio e ricevuto dai nodi mobili, che lo potranno quindi utilizzare per correggere la loro posizione da loro stessi calcolata. Il sistema D-GPS è quello utilizzato da tutti i comuni apparecchi GPS oggi in commercio.

2.3 Sensori e GPS nei moderni smartphone

Gli attuali smartphone sono dotati di una molteplicità di sensori che possono essere impiegati per ricavare informazioni utili alle applicazioni installate sul dispositivo stesso. Non tutti gli smartphone in commercio sono dotati dello stesso insieme di sensori ed hardware addizionale ma, oggigiorno, la maggior parte di essi integrano l'hardware di interesse per il nostro progetto: accelerometro, bussola ed antenna GPS.

L'accelerometro è in grado di misurare l'accelerazione impressa al dispositivo; la bussola individua i punti cardinali in base all'orientamento del dispositivo; l'antenna GPS è in grado di interagire con il sistema di posizionamento globale GPS.

2.3.1 Sensori nella piattaforma Android

La piattaforma Android supporta nativamente diversi tipi di sensori, mettendo a disposizione del programmatore una serie di API per il loro facile utilizzo. Come già detto precedentemente, per ragioni legate alla diversità dei modelli dei vari produttori, i 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 . Esiste comunque una documento denominato *CDD* (*Compatibility Definition Document*) [CDD] che specifica quali sensori dovrebbero essere integrati nei dispositivi equipaggiati di sistema Android in ogni sua versione.

Le API per l'utilizzo dei sensori in Android comprendono due classi rilevanti denominate `Sensor` e `SensorEvent`. Le API utili per l'estrazione e la manipolazione dei dati dei sensori sono state implementate dalla versione 3 dell'SDK (Android 1.6). Alcune proprietà all'atto dell'acquisizione sono comuni all'interno della classe `SensorEvent` e comprendono:

- `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` per avere un livello alto di precisione nell'acquisizione dei dati;
  - `SensorEvent.SENSOR_STATUS_ACCURACY_MEDIUM` 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` per un livello basso di precisione.
- `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`.
- `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).

- `values` (vettore): 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.

Sistema di coordinate in SensorEvent API

Il sistema di coordinate viene definito relativamente allo schermo del telefono nel caso sia orientato nella sua posizione predefinita (Fig. 2.6). 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.

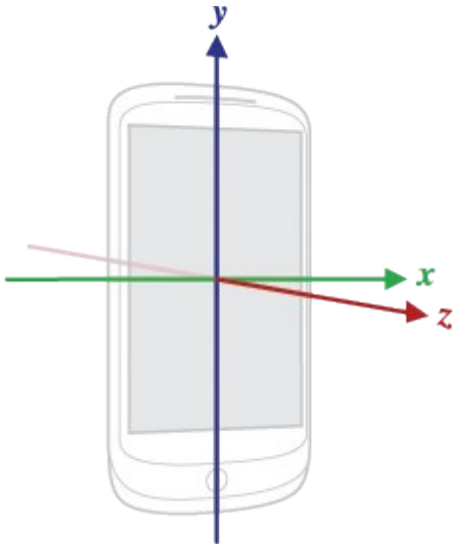

Fig. 2.6: Disposizione degli assi

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.

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

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

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 $9{,}81\, m/s^2$. In modo analogo, quando il dispositivo viene posto in caduta libera e quindi ha un'accelerazione verso terra pari a $9{,}81\, m/s^2$, l'accelerometro rileva una magnitudine di $0\, m/s^2$.

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.

Sensore di orientamento

Il sensore di orientamento riporta i valori, espressi in gradi, di angoli e di rotazioni. In

particolare, i valori restituiti 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.

Sistema di posizionamento e antenna GPS

In ambito scientifico e pratico esistono svariati sistemi per la determinazione della posizione di un dispositivo. Questi sistemi possono essere suddivisi in diverse categorie in base a diversi criteri. Basandoci sul tipo di informazioni restituite, possiamo individuare due categorie:

- *sistemi fisici*: restituiscono informazioni adatte all'elaborazione da parte di macchine (es. coordinate GPS);
- *sistemi simbolici*: restituiscono informazioni human friendly, solitamente strutturate a livelli (es.: {Italy, Bologna, EngFaculty, DEIS, Lab2}).

Un secondo criterio di categorizzazione può basarsi sul tipo di sistema di riferimento:

- *assoluto*: unico sistema di riferimento per tutti gli oggetti localizzati; può essere *fisico* o *simbolico*;
- *relativo*: relativo alla locazione di un altro oggetto; tipicamente fisico ed utili per scenari ad-hoc.

Un'altra categorizzazione può essere determinata dal tipo di processamento delle informazioni raccolte:

- *centralizzato*: un sistema centrale processa le informazioni e determina la locazione per tutti gli oggetti posizionati;
- *distribuito*: ognuno determina la propria locazione.

Si possono poi suddividere i vari sistemi di posizionamento a seconda della loro:

- *accuratezza*: range di errore (es. metri);
- *precisione*: grado di fiducia nel range di errore (in percentuale);
- *scalabilità*: capacità di elaborazione a seconda del numero di utenti, dell'area da considerare, delle risorse computazionali a disposizione, ecc.;
- *recognition*: quanto è personale il dispositivo utilizzato (ad es. la carta di credito è uno strumento personale, l'antenna GPS no);
- *costo*: in termini di
  - tempo
  - infrastruttura e deployment di infrastruttura (ad es. GPS ha bisogno di infrastruttura formata dai satelliti)
  - lato cliente
  - hardware addizionale (ad es. GPS necessita di un'antenna)
  - consumo di batteria
  - consumo di memoria
  - ecc.

Esistono molte tecniche di base che consentono di implementare meccanismi di posizionamento più o meno precisi ed efficienti a seconda dell'hardware a disposizione. Tra queste ricordiamo:

- *lateration*: basata sulla misura della differenza di distanza tra due o più stazioni aventi posizione nota;
- *angolazione*: basata sull'osservazione dell'angolo con cui arrivano i segnali provenienti da

antenne direzionali di stazioni con posizioni note;

- *scene analysis*: basata sull'osservazione passiva di fenomeni fisici, ma senza sfruttare valori fisici come distanze, angoli, iperboli, ecc.. La fase preliminare si basa sul collezionamento di di caratteristiche fisiche dell'ambiente in esame (ad es. intensità di segnali wireless in ogni punto della stanza), dopodichè si determina la posizione del punto mobile in base alle intensità dei segnali da esso rilevati;
- *prossimità*: la locazione corrente viene approssimata al punto con posizione nota più vicino (es.: RFID, carta di credito).

I sistemi di posizionamento sono, ovviamente, soggetti a diversi tipi di errore. Tra questi possiamo citare quelli dovuti a:

- *Non Line Of Sight* e *fading*: è presente un ostacolo tra il trasmettitore ed il ricevitore del segnale che causa attenuazione e riflessione;
- *sincronizzazione del clock*: spesso la distanza tra due oggetti è misurata in base al tempo impiegato dal segnale per arrivare dal trasmettitore al ricevitore, sollevando quindi il problema di sincronizzazione dei rispettivi clock.

La piattaforma Android può sfruttare più meccanismi (anche contemporaneamente) per ottenere indicazioni sulla posizione geografica attuale del dispositivo. Tra questi, in ordine di accuratezza crescente, spiccano:

- posizionamento tramite *rete cellulare* (Cell-ID);
- posizionamento tramite rete *WiFi*;
- posizionamento tramite *GPS*.

La scelta di quale sistema impiegare è una questione di trade-off tra accuratezza, velocità di determinazione della posizione e consumo di batteria. La determinazione della posizione tramite rete cellulare è un processo a basso consumo di potenza ma poco accurato; il sistema basato su rete WiFi consuma più energia del precedente ma è più accurato; il sistema GPS è il più dispendioso energeticamente ma anche il più accurato.

L'utilizzo dei sistemi di posizionamento offerti da Android è consentito da API native che garantiscono un accesso semplice ed immediato alle informazioni disponibili sul dispositivo. Il punto d'accesso alle API di posizionamento è `LocationManager`, che consente il recupero dei gestori

dei diversi sistemi di localizzazione (*provider*) e la registrazione di listener per la notifica dell'aggiornamento della posizione.

Un'applicazione Android che vuole recuperare informazioni di posizionamento può seguire il seguente tipico flusso di azioni (Fig. 2.7) [LOCAPI]:

1. Avvio dell'applicazione
2. Avvio dell'ascolto degli aggiornamenti di posizione da qualche provider di locazione
3. Mantenimento della migliore posizione stimata filtrando i nuovi ma meno accurati aggiornamenti
4. Arresto dell'ascolto degli aggiornamenti di posizione
5. Elaborazione della migliore posizione trovata

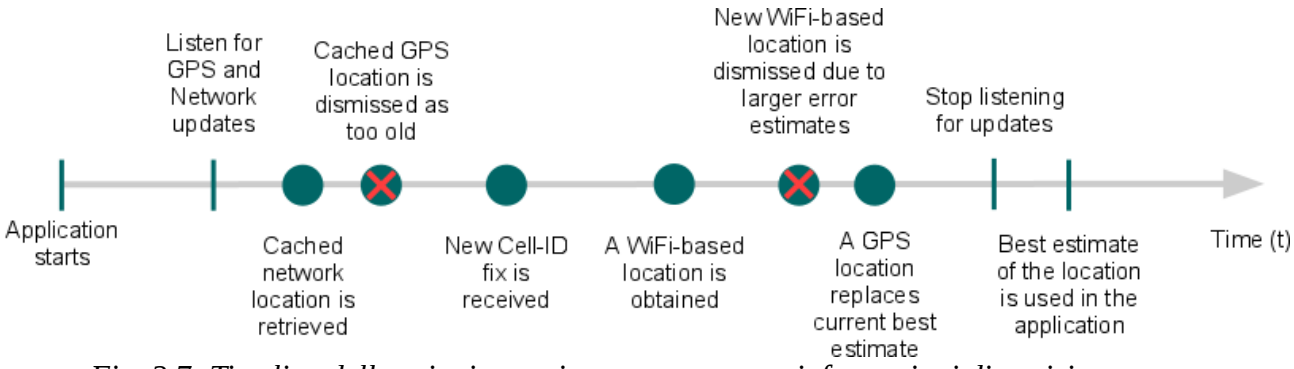

Fig. 2.7: Timeline delle azioni eseguite per recuperare informazioni di posizionamento

2.4 WiFi Direct

Molta gente associa la tecnologia WiFi alla connettività internet. I dispositivi certificati *WiFi Direct* estendono la tecnologia WiFi per consentire una semplice connessione diretta tra più utenti. Le specifiche *WiFi Peer-to-Peer* estendono le capacità di WiFi attraverso il supporto alla connessione diretta tra dispositivi. Con WiFi Direct è possibile connettersi con chiunque e dovunque in ogni momento, senza sacrificare robustezza e sicurezza della connessione, e garantendo compatibilità con dispositivi WiFi standard. Sfruttando WiFi Direct, gli sviluppatori possono realizzare applicazioni innovative e creare nuove opportunità per produttori, fornitori di contenuti ed utenti.

Alcuni dei benefici portati da WiFi Direct sono:

- *mobilità e portabilità*: WiFi Direct consente la creazione di connessioni ovunque ed in qualsiasi momento in quanto non c'è necessità di un router WiFi o di un AP;
- *utilizzo immediato*: un nuovo dispositivo WiFi Direct può creare connessioni dirette con ogni altro dispositivo WiFi standard;
- *facilità di utilizzo*: le funzionalità WiFi Direct *Devices Discovery* e *Service Discovery* consentono il rapido ed immediato ritrovamento di dispositivi e servizi ancora prima di stabilire una connessione;
- *connessioni semplici e sicure*: i dispositivi WiFi Direct utilizzano *WiFi Protected Setup* (*WTS*) per stabilire connessioni sicure: l'utente dovrà soltanto premere un pulsante o inserire un PIN.

WiFi Direct consente di stabilire vere e proprie connessioni P2P tra dispositivi, in modo da poter usufruire di specifiche applicazioni per la condivisione di contenuti multimediali, per il controllo remoto e per lo scambio di informazioni in genere.

I dispositivi WiFi Direct compatibili possono connettersi tra loro formando *gruppi* (con topologia uno a uno o uno a molti) che funzionano in modo molto simile ad una infrastruttura BSS. Un singolo dispositivo WiFi Direct si incarica della gestione del gruppo (diventando *manager* del gruppo), e decide così quando crearlo e distruggerlo e quali dispositivi ammettere al suo interno. Tale dispositivo si comporta come un Access Point agli occhi di dispositivi WiFi legacy, fornendo alcuni dei servizi tipici di un'infrastruttura ad Access Point (Fig. 2.8).

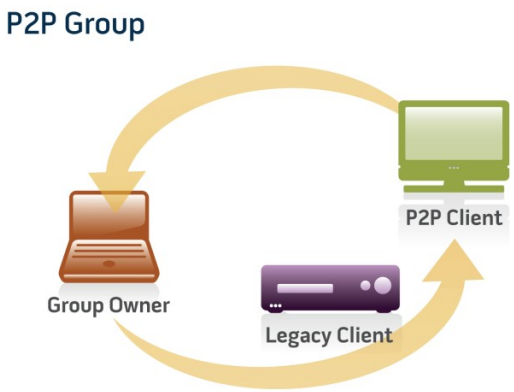

Fig. 2.8: Gruppo WiFi Direct

Siccome il manager del gruppo WiFi Direct non offre tutti i servizi di un AP, l'infrastruttura tradizionale ad AP resta la scelta migliore per la realizzazione di reti domestiche ed aziendali.

Ogni dispositivo certificato WiFi Direct deve essere in grado di occupare il ruolo di manager del gruppo a seguito di un processo di negoziazione, nel caso in cui siano presenti più dispositivi WiFi Direct.

La Tabella 2.1 riporta i meccanismi chiave definiti nelle specifiche *WiFi Peer-to-Peer*, indicando quali di questi sono obbligatoriamente supportati dai dispositivi certificati WiFi Direct, e quali opzionali.

Meccanismo	**Obbligatorio**	**Opzionale**
Device Discovery consente il ritrovamento di dispositivi WiFi Direct e lo scambio di informazioni su di essi	X	
Service Discovery facilita il ritrovamento di servizi di alto livello. Può essere eseguito prima della realizzazione della connessione WiFi Direct		X
Group Formation determina quale dispositivo WiFi Direct deve ricoprire il ruolo di manager del gruppo	X	
Invitation consente ad un dispositivo WiFi Direct di invitare altri dispositivi WiFI Direct ad unirsi al gruppo esistente		X
Client Discovery abilita un dispositivo WiFi Direct al ritrovamento di altri dispositivi WiFi Direct all'interno del gruppo esistente	X	
Power Management		
P2P-PS and P2P-WMM®-PS adattamenti di sistemi legacy di risparmio energetico e sistemi WMM-Power Save che abilitano funzioni aggiuntive di risparmio energetico per dispositivi WiFi Direct	X	
Notice of Absence nuova tecnica che consente, al dispositivo WiFi Direct che svolge il ruolo di manager del gruppo, di ridurre il consumo energetico comunicando periodi programmati di sua indisponibilità	X	
Opportunistic Power Save nuova tecnica che consente, al dispositivo WiFi Direct che svolge il ruolo di manager del gruppo, di ridurre il consumo energetico entrando in uno stato di inattività qualora lo siano anche gli altri dispositivi WiFi Direct connessi	X	

Tabella 2.1: Meccanismi chiave di WiFi Direct

2.5 Java Enterprise Edition ed EJB

Java EE (dall'inglese Java Enterprise Edition) è la versione enterprise della piattaforma Java. È costituita da specifiche che definiscono le caratteristiche e le interfacce di un insieme di tecnologie pensate per la realizzazione di applicazioni di tipo enterprise, compresi web services ed altre applicazioni che necessitano di requisiti di scalabilità, affidabilità e sicurezza. Java EE estende Java Platform Standard Edition (Java SE) fornendo un insieme di API per object relational mapping e per la realizzazione di web services e di applicazioni distribuite e multi-tier. La piattaforma suggerisce un design delle applicazioni basato largamente sul principio di modularità dei componenti che vengono eseguiti su un application server. Il software Java EE compliant è scritto principalmente in linguaggio Java e utilizza XML per la configurazione dei componenti.

Essendo un insieme di specifiche aperte, chiunque ne può realizzare un'implementazione e produrre application server compatibili con le specifiche Java EE, garantendo quindi a qualsiasi applicazione Java EE compliant di poter essere eseguita su uno qualunque di tali application server.

La specifica Java EE include molte tecnologie che estendono le funzionalità di base della piattaforma Java. La specifica descrive i seguenti componenti:

- *JavaServer Faces (JSF)*: tecnologia per la realizzazione dell'interfaccia utente dei componenti;
- *Servlet*: consente la creazione di servlet ed include le specifiche *Java Server Pages (JSP)*;
- *Enterprise JavaBeans (EJB)*: definiscono un sistema a componenti distribuito che rappresenta il cuore della specifica Java EE. Tale sistema, infatti, fornisce le tipiche caratteristiche richieste dalle applicazioni *enterprise*, come scalabilità, sicurezza, persistenza dei dati e altro;
- *JNDI*: definisce un sistema di nomi per identificare e elencare risorse generiche, come componenti software o sorgenti di dati;
- *JDBC*: è un'interfaccia per l'accesso a qualsiasi tipo di basi di dati (compresa anche nella standard edition);
- *JTA*: è un sistema per il supporto delle transazioni distribuite;
- *JAXP*: è un API per la gestione di file in formato XML;
- *Java Message Service (JMS)*: descrive un sistema per l'invio e la gestione di messaggi;

- *Java Persistence API (JPA)*: definisce le specifiche del framework per la persistenza dei dati.

Le applicazioni enterprise Java EE sono, come già detto, orientate verso un'architettura a componenti. Per consentire una gestione efficace ed efficiente dei componenti mettendo a disposizione dello sviluppatore diversi servizi per la risoluzione di problematiche comuni come controllo della concorrenza, load-balancing, sicurezza, gestione delle risorse, ecc., Java EE prevede l'implementazione di un modello a contenimento, nel quale un gestore di componenti, detto *container*, si occupa di tutti gli aspetti che non riguardano direttamente la logica di business dell'applicazione.

Il container

Gli application server Java EE compliant devono realizzare un modello a contenimento in grado di fornire servizi utili ai componenti da eseguire, tra i quali:

- gestione del ciclo di vita dei componenti
- supporto al sistema di nomi
- supporto alla qualità del servizio
- sicurezza
- dependency injection
- …

Le funzionalità sopra riportate sono offerte da un componente dell'application server detto *container*, che si interpone a tutte le operazioni richieste dai componenti che costituiscono l'applicazione (Fig. 2.9).

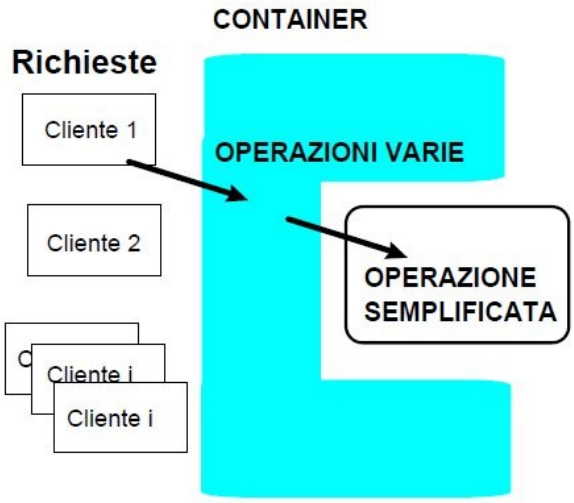

Fig. 2.9: Lavoro del container

Esistono diversi tipi di container (Web, EJB, Applet, ecc.) a seconda dei requisiti richiesti, ma tutti con la stessa filosofia di approccio basata sul principio di separazione tra logica di business e codice di sistema. Il container è il fornitore dei servizi di sistema verso i componenti di un'applicazione.

Componenti EJB

I componenti principali di un'applicazione Java enterprise sono denominati *Enterprise JavaBeans* (*EJB*). La specifica EJB consente lo sviluppo di componenti server-side portabili, garantendo il tanto caro principio Java del *write once, run everywhere* o meglio, come più appropriato in questo caso, *write onece, then deploy on any server platform that supports the Enterprise JavaBeans specification*.

L'architettura a componenti garantisce disaccoppiamento, scalabilità, transazionalità e sicurezza multiutente. Le applicazioni EJB non si occupano della gestione delle risorse (che è invece compito del container, con obiettivi di massima efficienza), consentendo quindi allo sviluppatore di dedicarsi esclusivamente alla logica di business. Questo approccio consente anche l'integrazione tra loro di componenti sviluppati da vendor differenti: l'unica cosa importante è che espongano le interfacce richieste.

La filosofia adottata da EJB rende possibile l'utilizzo dei componenti in diverse architetture N-tier e da più clienti simultaneamente (Fig. 2.10).

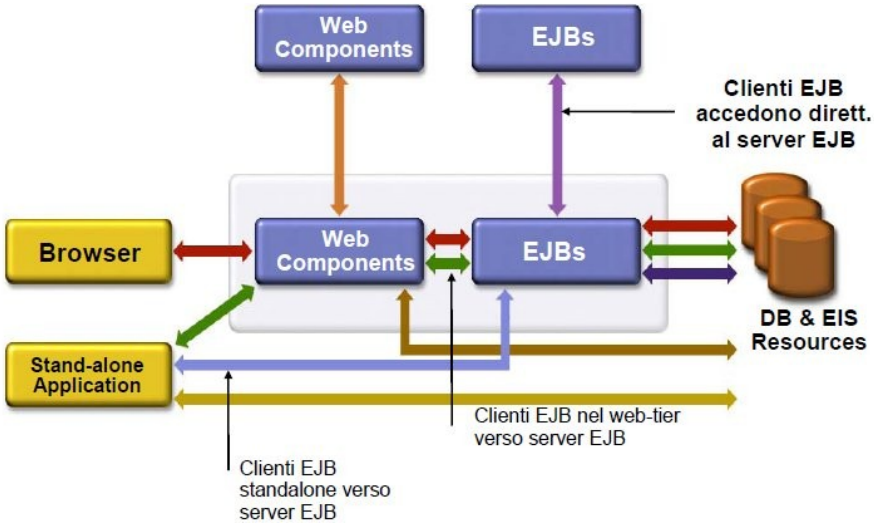

Fig. 2.10: EJB in applicazioni N-tier

Come già detto precedentemente, i componenti EJB sono gestiti da un container che offre servizi di sistema e prende decisioni per garantire un'esecuzione efficiente dell'applicazione enterprise. Queste decisioni vengono prese in base a parametri di configurazione specificabili dall'assemblatore di applicazioni oppure a default. La tecnologia EJB offre quindi anche supporto ad una facile configurazione dell'applicazione e dei componenti a deployment-time.

Esistono vari tipi di componenti EJB:

- *Session Bean*: lavorano tipicamente per un singolo cliente e consentono di modellare oggetti di processo o di controllo e per muovere la logica applicativa di business dal lato cliente a quello servitore
  - *Stateless*: esegue una richiesta e restituisce un risultato senza salvare alcuna informazione di stato relativa al cliente
  - *Stateful*: può mantenere stato specifico per un cliente
- *Entity Bean* (normali oggetti POJO a partire dalla specifica EJB3.x): forniscono una vista ad oggetti dei dati mantenuti in un database, consentendo un accesso ai dati condiviso tra più clienti
- *Message Driven Bean*: svolgono il ruolo di consumatori di messaggi asincroni e non possono essere invocati direttamente dai clienti ma soltanto tramite l'invio di messaggi

verso *code* o *topic* per i quali questi componenti sono in ascolto.

Formato di deployment

Le applicazioni Java EE (chiamate anche applicazioni EJB) sono tipicamente racchiuse in archivi di estensione *.EAR* (*Enterprise ARchive*), i quali possono contenere anche moduli web-tier racchiusi in file *.WAR* (*Web ARchive*). La parte relativa ai componenti EJB è tipicamente racchiusa all'interno di un file *jar* denominato *EJB-JAR*. All'interno di un archivio *.EAR* deve apparire anche un descrittore di deployment denominato `application.xml`, che specifica i parametri di configurazione per realizzare il corretto deployment dell'intera applicazione.

Riassumendo, un'applicazione EJB è rappresentata da un file di estensione *.EAR* che contiene:

- uno o più moduli EJB (file jar *EJB-JAR*);
- un descrittore di deployment `application.xml`;
- zero o più moduli web-tier (archivi *.WAR*).

Alcuni container consentono anche il deployment diretto di file *EJB-JAR*.

2.6 Java Persistence API e Hibernate

Una parte rilevante degli sforzi nello sviluppo di ogni applicazione distribuita di livello enterprise si concentra sul layer di persistenza. Solitamente i dati vengono mantenuti in DBMS relazionali che non consentono un immediato mapping con il mondo ad oggetti tipico dei moderni linguaggi di programmazione. A questo proposito, è nata l'esigenza di standardizzare metodi che consentono di eseguire automaticamente il mapping Object/Relational (*ORM*), cioè di trasformare automaticamente dati tabellari in oggetti e viceversa.

Java Persistence API (*JPA*) è la specifica standard Java EE del framework di supporto al mapping O/R, in grado di gestire in modo trasparente la persistenza di *POJO* (*Plain Old Java Object*). Gli oggetti persistenti possono così seguire le usuali modalità di progettazione e programmazione, disinteressandosi di potenziali mismatch tra gli oggetti stessi e la loro rappresentazione con dati relazionali. *JPA* supporta ricche funzionalità di modellazione del dominio come ereditarietà e polimorfismo, realizza un mapping O/R standard ed efficiente, supporta capacità di querying ricche e flessibili e si integra con servizi di persistenza di terze parti.

Il vantaggio più evidente derivante dall'utilizzo di *JPA* è la possibilità di evitare la gestione esplicita di query, connessioni al database e incapsulamento dei risultati in oggetti da utilizzare nell'applicazione, configurando il tutto con semplici annotazioni o file di configurazione XML.

Il concetto fondamentale di *JPA* è quello di *Entity*, cioè un oggetto “leggero” appartenente ad un dominio di persistenza. L'*Entity*, usualmente, rappresenta una tabella in un database relazionale, ed ogni sua istanza corrisponde ad una riga di questa tabella. Lo stato persistente di un'*Entity* è rappresentato da *campi persistenti* (o da *proprietà persistenti*) che usano annotazioni per specificare il mapping O/R. I campi (o le proprietà) persistenti sono costituti da tutti i campi (o proprietà) non marcate con l'annotazione `@Transient`. La flessibilità di *JPA* consente di creare intere gerarchie di classi specificando, tramite annotazioni o file di configurazione XML, quali dovranno essere *Entity* e quali no.

Secondo le specifiche di JPA, le *Entity* sono gestite da un *EntityManager* che è associato ad un *contesto di persistenza* (insieme di istanze che esistono in un particolare data store), il quale definisce lo scope al cui interno le istanze di *Entity* sono create, gestite e rimosse. L'*EntityManager* può essere gestito a *livello di container* (qualora sia all'interno di un container, ad es. EJB) o a *livello di applicazione* (e quindi gestito dallo sviluppatore). Come già detto, l'*EntityManager* gestisce le *Entity* ed il loro intero ciclo di vita, facendogli assumere diversi stati (Fig. 2.11):

- *New/Transient entity*: nuove istanze non hanno ancora identità di persistenza e non sono ancora associate ad uno specifico contesto di persistenza;
- *Managed entity*: istanze con identità di persistenza e associate ad un contesto di persistenza;
- *Detached entity*: istanze con identità di persistenza e correntemente disassociate da contesti di persistenza;
- *Removed entity*: istanze con identità di persistenza, associate ad un contesto e la cui eliminazione dal datastore è stata già schedulata.

Il cambiamento di stato di un'*Entity* è subordinato all'esecuzione di particolari operazioni offerte *dall'EntityManager*.

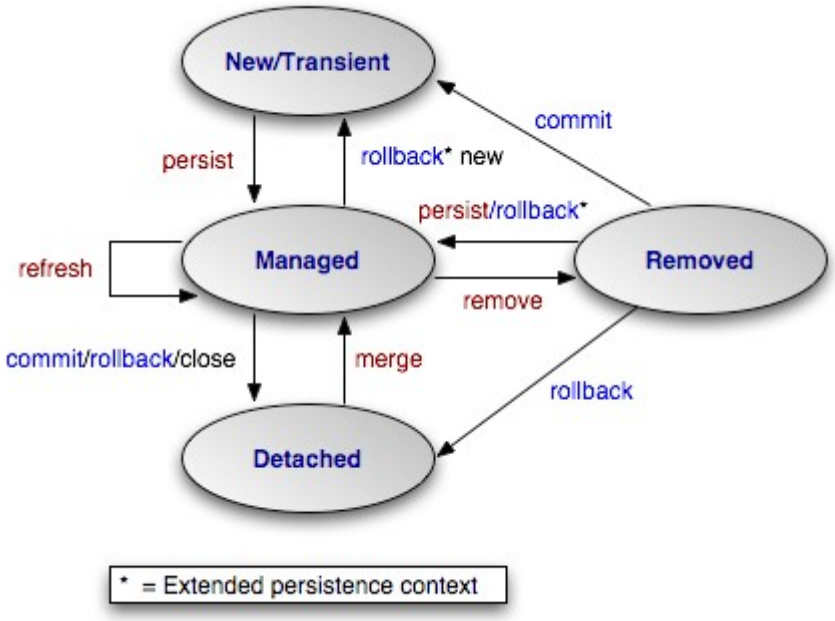

Fig. 2.11: Transizione degli stati di Entity

Le specifiche JPA affermano che la sincronizzazione tra oggetti *Entity* e database è automatica, ed è eseguita al momento del *commit* della transazione nella quale viene richiamata un'operazione dell'*EntityManager* o richiamando il metodo `commit()` dello stesso. Le transazioni all'interno di JPA possono essere di tipo *locali alle risorse* oppure *Java Transaction API (JTA)*. Se JPA è usato all'interno di un container (ad esempio un container EJB), il tipo di transazioni di default è JTA.

2.6.1 Un middleware di persistenza: Hibernate

Hibernate (talvolta abbreviato in *H8*) è una piattaforma middleware open source per lo sviluppo di applicazioni Java che fornisce un servizio di Object/relational mapping (ORM), ovvero che gestisce la rappresentazione e il mantenimento su database relazionale di un sistema di oggetti Java. Hibernate è stato originariamente sviluppato da un team internazionale di programmatori volontari coordinati da Gavin King; in seguito il progetto è stato proseguito sotto l'egida di *JBoss*, che ne ha curato la standardizzazione rispetto alle specifiche Java EE.

L'obiettivo di Hibernate è quello di esonerare lo sviluppatore dall'intero lavoro relativo alla persistenza dei dati. Hibernate si adatta al processo di sviluppo del programmatore, sia se si parte da zero sia se da un database già esistente. Hibernate genera le chiamate SQL e solleva lo sviluppatore dal lavoro di recupero manuale dei dati e dalla loro conversione, mantenendo l'applicazione

portabile in tutti i database SQL. Hibernate fornisce una persistenza trasparente per POJO; l'unica grossa richiesta per la persistenza di una classe è la presenza di un costruttore senza argomenti. In alcuni casi si richiede un'attenzione speciale per i metodi equals() e hashcode().

Hibernate è tipicamente usato sia in applicazioni Swing che Java EE facenti uso di servlet o EJB session beans.

La versione 3 di Hibernate, arricchisce la piattaforma con nuove caratteristiche come una nuova architettura *Interceptor/Callback*, filtri definiti dall'utente, e annotazione stile JDK 5.0 (Java's metadata feature). Hibernate *3* è vicino anche alle specifiche di EJB 3.0 (nonostante sia stato terminato prima di EJB 3.0 le specifiche erano già state rilasciate dalla Java Community Process) ed è usato come spina dorsale per l'implementazione EJB 3.0 di *JBoss*.

Hibernate permette di astrarre dalle API JDBC/JTA sottostanti, consentendo al livello di applicazione di disinteressarsi di questi dettagli (Fig. 2.12).

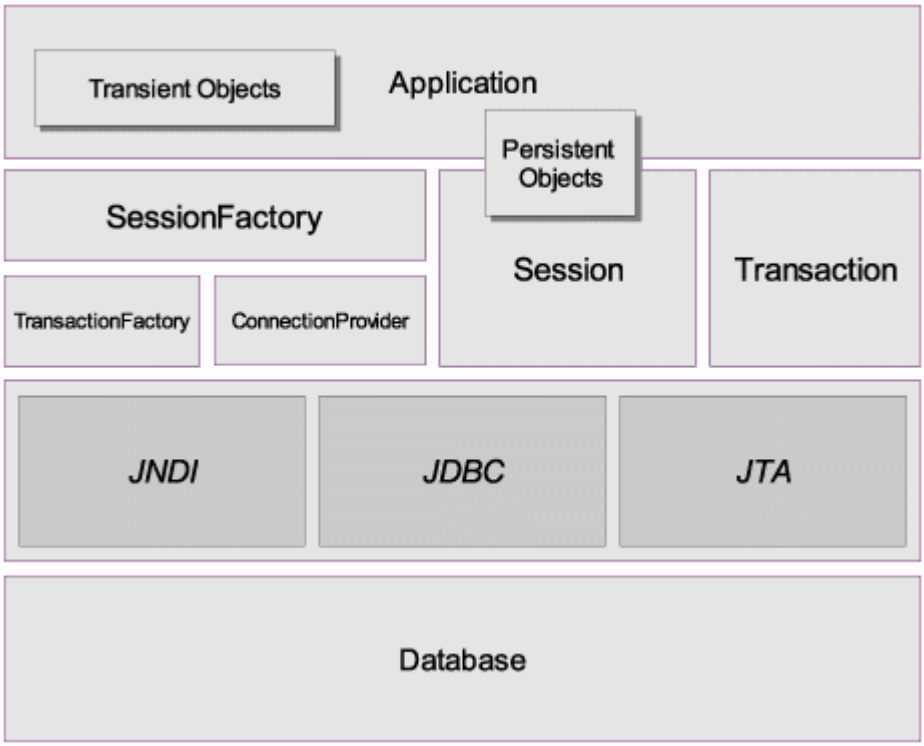

Fig. 2.12: Architettura di Hibernate

I componenti di base in Hibernate sono i seguenti:

- *SessionFactory*: rappresenta la factory per oggetti *Session* ed è cliente del componente

ConnectionProvider (factory delle connessioni JDBC) . Si occupa di mantenere una cache di primo livello associata all'oggetto *Session* e usata di default, e può mantenerne una di secondo livello (opzionale) associata all'oggetto *SessionFactory* stesso con dati riusabili in diverse transazioni;

- *Session*: rappresenta un contesto di persistenza e la sua vita è delimitata dall'inizio e dalla fine di una transazione logica. Gestisce le operazioni del ciclo di vita degli oggetti persistenti e fa da factory per oggetti *Transaction*;
- *Persistent Objects*: sono oggetti single-threaded che contengono stato persistente e logica di business. Possono essere normali JavaBeans oppure POJO; l'unico aspetto peculiare è che devono essere associati esattamente ad un oggetto *Session*. Le modifiche fatte sugli oggetti persistenti sono automaticamente riportate sulle tabelle del database;
- *Transient and Detached Objects*: sono istanze di classi persistenti non correttamente associate a nessuna *Session* quindi le modifiche fatte su questi oggetti non si riflettono su database; è possibile farli tornare *Persistent* attraverso comandi di `merge` e `persist`;
- *Transaction*: è un oggetto single-threaded usato dall'applicazione per specificare unità atomiche di lavoro, permettendo di astrarre dai dettagli dei sottostanti meccanismi transazionali (JDBC, JTA, CORBA, ecc.); creato tramite *TransactionFactory*, non direttamente accessibile dall'applicazione.

Hibernate può essere utilizzato attraverso JPA, in modo da rendere standard le chiamate al middleware di persistenza e garantirne il corretto funzionamento a qualsiasi applicazione che, appunto, utilizza JPA.

2.7 RESTful Web Services

Il Web è nato negli anni ’90 come una piattaforma per la condivisione di documenti distribuiti su diverse macchine e tra loro interconnessi. Il tutto è nato e si è evoluto grazie alla standardizzazione di alcuni semplici concetti:

- *URI*: meccanismo per individuare risorse in una rete
- *HTTP*: protocollo semplice e leggero per richiedere una risorsa ad una macchina

- *HTML*: linguaggio per la rappresentazione dei contenuti

Questa semplice idea iniziale si è evoluta nel corso degli anni non tanto nei concetti di base, quanto nel modo di intenderli e di utilizzarli.

Al testo si sono aggiunti contenuti multimediali, i documenti sono generati dinamicamente e non pubblicati come pagine statiche, e poi il Web esce dalla visione esclusivamente ipertestuale per diventare un contenitore di applicazioni software interoperabili: una piattaforma applicativa distribuita.

Questa prospettiva ha dato origine, intorno all'anno 2000, al concetto di *Web Service*: un sistema software progettato per supportare un'interazione tra applicazioni, utilizzando le tecnologie e gli standard Web. Il meccanismo dei Web Service consente di far interagire in maniera trasparente applicazioni sviluppate con linguaggi di programmazione diversi, eseguiti su sistemi operativi eterogenei.

Questo meccanismo consente di realizzare porzioni di funzionalità in maniera indipendente e su piattaforme potenzialmente incompatibili facendo interagire i vari pezzi tramite tecnologie Web e creando un'architettura facilmente componibile.

2.7.1 SOAP e REST

Allo stato attuale esistono due approcci alla creazione di Web Service:

1. basato sul protocollo standard *SOAP* (*Simple Object Access Protocol*) che consente lo scambio di messaggi per l'invocazione di servizi remoti e si prefigge di riprodurre in ambito Web un approccio a chiamate remote (*Remote Procedure Call*) tipico di protocolli di interoperabilità come CORBA, DCOM e RMI;
2. ispirato ai principi architetturali tipici del web concentrandosi sulla descrizione di risorse, sul modo di individuarle nel web e sul modo di trasferirle da una macchina all'altra. Questo è l'approccio che prende il nome di *REST* (*REpresentational State Transfer*).

Volendo essere formali, *REST* definisce un insieme di principi architetturali per la progettazione di un sistema. Rappresenta uno stile architetturale, cioè non si riferisce ad un sistema concreto e ben definito nè si tratta di uno standard stabilito da un organismo di standardizzazione.

La sua definizione è apparsa per la prima volta nel 2000 nella tesi di Roy Fielding, *Architectural Styles and the Design of Network-based Software Architectures* [RESTTH], discussa presso

l’Università della California, ad Irvine. In questa tesi si analizzavano alcuni principi alla base di diverse architetture software, tra cui appunto i principi di un’architettura software che consentisse di vedere il web come una piattaforma per l’elaborazione distribuita.

È bene precisare che i principi *REST* non sono necessariamente legati al web, nel senso che si tratta di principi astratti di cui il World Wide Web ne risulta essere un esempio concreto.

All’epoca questa visione del web passò un po' inosservata, ma negli ultimi anni l’approccio *REST* è venuto alla ribalta come metodo per la realizzazione di Web Service altamente efficienti e scalabili ed ha al suo attivo un significativo numero di applicazioni. Il motivo è semplice: dal momento che il web ha tutto quello che serve per essere considerata una piattaforma di elaborazione distribuita secondo i principi *REST*, non sono necessarie altre sovrastrutture per realizzare quello che è il web programmabile. Questa è una dichiarazione di aperto antagonismo ai Web Service basati su SOAP.

L'approccio *REST* può essere riassunto nei seguenti cinque principi:

- identificazione delle risorse;
- utilizzo esplicito dei metodi HTTP;
- risorse autodescrittive;
- collegamenti tra risorse;
- comunicazione senza stato.

Questi principi rappresentano in realtà concetti ben noti perchè ormai insiti nel web che conosciamo. Li analizzeremo tuttavia sotto una prospettiva diversa, quella della realizzazione di Web Service.

Identificazione delle risorse

Per risorsa si intende un qualsiasi elemento oggetto di elaborazione. Per fare qualche esempio concreto, una risorsa può essere un cliente, un libro, un articolo, un qualsiasi oggetto su cui è possibile effettuare operazioni.

Il principio che stiamo analizzando stabilisce che ciascuna risorsa deve essere identificata univocamente. Essendo in ambito Web, il meccanismo più naturale per individuare una risorsa è dato dal concetto di URI.

Utilizzo esplicito dei metodo HTTP

Questo principio ci indica di sfruttare i metodi (o verbi) predefiniti del protocollo HTTP, e cioè *GET*, *POST*, *PUT* e *DELETE* per indicare quali operazioni eseguire sulle risorse. In ambito *REST*, l'insieme delle operazione eseguibili è definito con il nome *CRUD* (*Create*, *Read*, *Update*, *Delete*).

Metodo HTTP	Operazione CRUD	Descrizione
POST	Create	Crea una nuova risorsa
GET	Read	Ottiene una risorsa esistente
PUT	Update	Aggiorna una risorsa o ne modifica lo stato
DELETE	Delete	Elimina una risorsa

Tabella 2.2: Operazioni CRUD di REST

È opportuno notare che questo principio è in contrasto con quella che è la tendenza generale nell'utilizzo dei metodi HTTP. Infatti, molto spesso viene utilizzato il metodo GET per eseguire qualsiasi tipo di interazione con il server. Ad esempio, spesso per l'inserimento di un nuovo cliente nell'ambito di un'applicazione web viene eseguita una richiesta di tipo GET su un URI del tipo:

```
http://www.myapp.com/addCustomer?name=Rossi
```

Questo approccio non è conforme ai principi *REST* perchè il metodo GET serve per accedere alla rappresentazione di una risorsa e non per crearne una nuova.

Risorse autodescrittive

Le risorse di per sè sono concettualmente separate dalle rappresentazioni restituite al client. Ad esempio, un web service non invia al client direttamente un record del suo database, ma una sua rappresentazione in una codifica dipendente dalla richiesta del client e/o dall'implementazione del servizio.

I principi REST non pongono nessun vincolo sulle modalità di rappresentazione di una risorsa. Virtualmente possiamo utilizzare il formato che preferiamo senza essere obbligati a seguire uno standard. Di fatto, però, è opportuno utilizzare formati il più possibile standard in modo da semplificare l'interazione con i client. Inoltre, sarebbe opportuno prevedere rappresentazioni multiple di una risorsa, per soddisfare client di tipo diverso.

Il tipo di rappresentazione inviata dal Web Service al client è indicato nella stessa risposta HTTP tramite un tipo MIME, così come avviene nella classica comunicazione tra web server e browser.

Collegamenti tra risorse

Un altro vincolo dei principi REST consiste nella necessità che le risorse siano tra loro messe in relazione tramite link ipertestuali. Questo principio è anche noto come *HATEOAS*, dall'acronimo di *Hypermedia As The Engine Of Application State*, e pone l'accento sulle modalità di gestione dello stato dell'applicazione.

In sostanza, tutto quello che un client deve sapere su una risorsa e sulle risorse ad essa correlate deve essere contenuto nella sua rappresentazione o deve essere accessibile tramite collegamenti ipertestuali.

Ad esempio, la rappresentazione di un ordine in un linguaggio XML-based deve contenere gli eventuali collegamenti agli articoli ed al cliente correlati:

```
<ordine>
	<numero>12345678</numero>
	<data>01/07/2011</data>
	<cliente rif="http://www.myapp.com/clienti/1234" />
	<articoli>
		<articolo rif="http://www.myapp.com/prodotti/98765" />
		<articolo rif="http://www.myapp.com/prodotti/43210" />
	</articoli>
</ordine>
```

In questo modo il client può accedere alle risorse correlate seguendo semplicemente i collegamenti contenuti nella rappresentazione della risorsa corrente.

Comunicazione senza stato

Il principio della comunicazione stateless è ben noto a chi lavora con il Web. Questa è infatti una delle caratteristiche principali del protocollo HTTP, cioè ciascuna richiesta non ha alcuna relazione con le richieste precedenti e successive. Lo stesso principio si applica ad un *Web Service RESTful*, cioè le interazioni tra client e server devono essere senza stato.

È importante sottolineare che sebbene *REST* preveda la comunicazione stateless, non vuol dire che un'applicazione non deve avere stato. La responsabilità della gestione dello stato dell'applicazione non deve essere conferita al server, ma rientra nei compiti del client. La principale ragione di questa scelta è la scalabilità.

2.7.2 Java e RESTful Web Services: JAX-RS

La piattaforma Java consente la facile realizzazione di web services RESTful grazie alle specifiche denominate *JAX-RS*. JAX-RS è un insieme di specifiche atte a definire API per lo sviluppo di RESTful Web Services. Implementazioni di JAX-RS devono fornire un insieme di annotazioni, interfacce e classi da utilizzare con oggetti POJO per consentirne l'esposizione come servizi web. Tali specifiche assumono HTTP come protocollo di comunicazione, e il meccanismo delle URI come metodologia per accedere alle risorse REST. I messaggi HTTP hanno un body che deve poter contenere svariati tipi di informazioni, e deve essere possibile aggiungerne di nuovi in modo standard.

JAX-RS promuove il principio di indipendenza dal container, ovvero tutti gli artefatti prodotti attraverso tali specifiche devono poter essere eseguite all'interno di una vasta varietà di container web. Oltre a questi aspetti, le specifiche JAX-RS definiscono l'ambiente in cui devono essere ospitate le risorse web all'interno di container Java EE, e come utilizzare i componenti e le funzionalità di Java EE all'interno delle classi di risorse web.

JAX-RS definiscono soltanto la parte server di servizi web, la parte client è demandata ad altre specifiche.

Come già detto, JAX-RS fornisce un insieme di annotazioni che consentono di esporre oggetti POJO come servizi web; tra queste citiamo:

- `@Path`: specifica il path relativo con cui accedere alle risorse o ai metodi;
- `@GET`, `@PUT`, `@POST`, `@DELETE` e `@HEAD`: specificano il tipo di richiesta verso la risorsa;
- `@Produces`: specifica il tipo MIME delle risposte fornite dal servizio web;
- `@Consumes`: specifica il tipo MIME delle richieste cui il servizio web deve rispondere.

Per ulteriori annotazioni, interfacce, classi e dettagli, si rimanda alla documentazione ufficiale di JAX-RS (JSR-311) [JAXRS].

Un'implementazione di JAX-RS: RESTEasy

RESTEasy [RESTEASY] è un progetto JBoss che fornisce framework per la realizzazione di RESTful Web Services e applicazioni Java RESTful implementando le specifiche JAX-RS.

RESTEasy può funzionare in ogni servlet container ed è integrato nell'application server JBoss AS. Siccome JAX-RS è un insieme di specifiche server-side, RESTEasy le porta anche dalla parte client, fornendo un framework che consente la costruzione di applicazioni che si interfacciano con RESTful Web Services. Questo framework client-side consente il mappaggio di richieste HTTP verso server remoti utilizzando annotazioni e proxy definiti dalle specifiche JAX-RS.

Le principali caratteristiche di RESTEasy sono:

- portabilità verso ogni application server o versione di Tomcat che esegue sulla JDK 5 o superione;
- implementazioni integrate per test in stile junit;
- supporto all'integrazione con EJB e Spring;
- client framework per la semplice realizzazione di client HTTP.

2.8 JBoss AS

JBoss [JBOSS] è un application server open source che implementa l'intera suite di servizi Java EE. Essendo basato su Java, JBoss è un application server multipiattaforma, utilizzabile su qualsiasi sistema operativo che supporti Java. Il gruppo di sviluppo di JBoss era originariamente assunto presso la *JBoss Inc.* fondata da Marc Fleury. Nell'aprile del 2006, Red Hat ha comprato JBoss per 420 milioni di dollari. Come progetto open source, JBoss è supportato e migliorato da una enorme rete di sviluppatori.

JBoss è stato pioniere nel mondo dell'open source professionale, dove gli sviluppatori iniziali del progetto creano un prodotto e offrono i loro servizi. Legati a JBoss esistono inoltre molti altri progetti, tra cui JBoss AS, Hibernate, Tomcat, JBoss ESB, jBPM, JBoss Rules (ex Drools), JBoss Cache, JGroups, JBoss Portal, SEAM, JBoss Transaction, JBoss Messaging tutti sotto il marchio della JBoss Enterprise Middleware Suite (JEMS).

JBoss AS è costruito su noti progetti open source (Fig. 2.13), tra i quali citiamo (quelli di maggior interesse per il nostro progetto sono scritti in grassetto):

- **Hibernate Core**: middleware per la persistenza dei dati
- Transactions

- Infinispan
- IronJacamar
- **RESTEasy**: framework per la rapida realizzazione di servizi RESTful
- Weld
- HornetQ
- JGroups
- Mojarra
- Apache CXF
- Arquillian

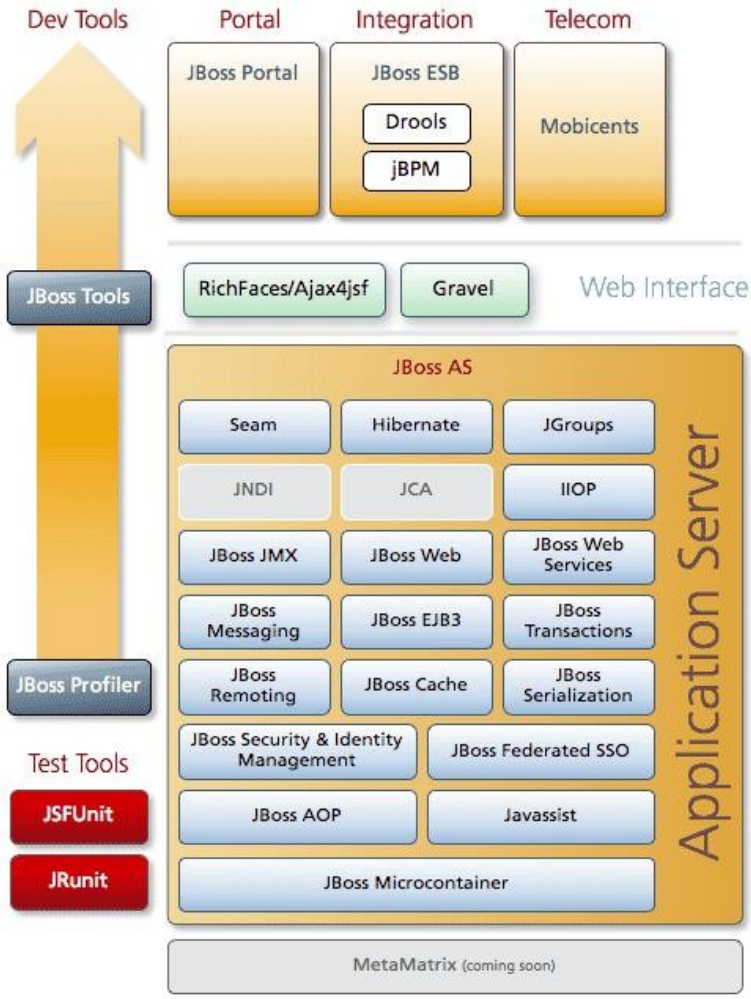

Fig. 2.13: Architettura modulare di JBoss 7

Le principali caratteristiche della versione 7 di JBoss AS sono:

- alta velocità di avvio (meno di 3 secondi);
- leggerezza: il miglioramento della gestione della memoria rispetto alle versioni precedenti hanno reso il server “leggero” ed eseguibile anche su dispositivi con risorse limitate;
- core modulare: i moduli con i quali JBoss è costruito offrono isolamento delle applicazioni, astrazione dalle classi di implementazione del server e la possibilità di caricare soltanto le classi di cui l'applicazione necessita;
- velocità di deployment delle applicazioni: il deployment delle applicazioni è eseguito in modo concorrente;
- console di amministrazione user-friendly;
- gestione domini: il server JBoss 7 può essere lanciato sia in modalità standalone che in modalità multi-server.

Capitolo 3: L'applicazione per piattaforma Android

L'applicazione *Trekker* consente il collezionamento di dati relativi ad escursioni montane, con lo scopo di realizzare una sorta di diario dell'escursionista e di tracciare percorsi e relative statistiche. Nello specifico, le funzionalità principali dell'applicazione sono le seguenti:

- collezionamento delle posizioni GPS;
- calcolo di statistiche quali distanza percorsa, tempo impiegato, ritmo di marcia istantaneo e medio, altitudine corrente, dislivello percorso, ecc.;
- fotografie geotaggate;
- inserimento di POI (Point Of Interest);
- invio dei dati ad un sistema remoto (Capitolo 4);
- scambio di dati con altri dispositivi incrociati lungo il sentiero.

Per realizzare tutte le sue funzionalità, l'applicazione fa uso di alcuni sensori e componenti hardware, che saranno quindi indispensabili per il corretto funzionamento (paragrafo 3.2).

3.1 Tecnologie utilizzate

Nella realizzazione dell'applicazione *Trekker* sono state utilizzate le seguenti tecnologie:

- piattaforma Android versione 4.1 [AND];
- *Google Maps APIs* [GMAP]: API per l'accesso alle mappe fornite da Google;
- *Gson [GSON]*: libreria di Google per la de/serializzazione di oggetti in formato; *Json*
- *Cling [CLING]*: libreria per l'utilizzo del protocollo *UpnP;*
- *WiFi Direct [WIFID]*: tecnologia per la connessione punto-punto ad hoc.

Le *Google MAPS APIs* sono state utilizzate per realizzare la funzionalità che consente di tracciare il percorso delle escursioni direttamente sulle mappe, offrendo all'utente una loro visualizzazione grafica.

Le librerie *Gson* e *Cling* sono state scelte per la realizzazione delle funzionalità di invio dati ad altri dispositivi (*Gson* + *Cling*) e al server remoto (solo *Gson*).

3.2 Componenti Hardware coinvolti

Le funzionalità offerte dall'applicazione prevedono l'utilizzo di alcuni componenti hardware di cui, normalmente, i dispositivi Android 4.1 sono dotati:

- *GPS*: per tracciare il percorso, memorizzare i POI e geotaggare le fotografie scattate;
- *Accelerometro*: per realizzare la funzionalità di contapassi e gestire l'orientamento del display;
- *Bussola*: per individuare il punto cardinale verso cui l'utente è orientato;
- *WiFi Direct*: per lo scambio di informazioni tra dispositivi.

Secondo il *CDD* (*Compatibility Definition Document*) [CDD], i dispositivi equipaggiati con piattaforma Android 4.1 DOVREBBERO essere forniti dei suddetti componenti hardware, ed offrire le API per un loro facile utilizzo.

3.3 Struttura generale

L'applicazione *Trekker* è costituita e fa uso dei componenti tipici della piattaforma Android:

- *Activity*
- *Service*
- *Content provider*
- *Broadcast receiver*

Tali componenti consentono una realizzazione modulare dell'applicazione separando l'interfaccia grafica dal vero motore dell'applicazione, costituito principalmente da servizi e oggetti POJO.

Ignorando i componenti tipici della piattaforma Android, possiamo suddividere l'applicazione *Trekker* nei seguenti moduli funzionali:

- *GUI:* modulo che gestisce l'interfaccia grafica; costituito da *activity;*
- *camera:* modulo che gestisce la fotocamera;
- *data exchange:* modulo per la condivisione dei dati tra dispositivi; fa uso dei *service;*
- *wifidirect: modulo per la creazione di una connessione ad hoc;*
- *db:* modulo che gestisce la persistenza dei dati raccolti su database;
- *GPS:* modulo che gestisce il modulo GPS; fa uso dei *service;*
- *sensors:* modulo che gestisce i sensori accelerometro e bussola; fa uso dei *service;*
- *statistics:* modulo che gestisce l'elaborazione delle statistiche quali distanza percorsa, tempo impiegato, ritmo di marcia istantaneo e medio, altitudine corrente, dislivello percorso, ecc.;
- *step:* modulo che implementa il meccanismo di contapassi;
- *upload:* modulo che consente l'upload dei dati verso il server remoto descritto nel capitolo Capitolo 4:;

3.3.1 Modulo *db*

Il modulo *db* ha lo scopo di gestire la persistenza dei dati dell'applicazione *Trekker*. A tale scopo viene sfruttato il database SQLite presente di default all'interno della piattaforma Android. L'utilizzo del database viene semplificato dalla presenza di una classe di supporto di Android chiamata `SQLiteOpenHelper`, dalla quale deriviamo una nostra classe personalizzata denominata `DatabaseHelper`. Questa classe ha il compito di creare il nostro database.

All'interno di questo modulo sono presenti anche tutte le classi che contengono i vari dati necessari alla nostra applicazione e le classi utili all'implementazione del pattern DAO.

Le principali entità dati necessarie all'applicazione *Trekker* sono le seguenti:

- *Excursion*: rappresenta un'escursione;
- *ExcursionCoordinate*: rappresenta una coordinata catturata durante un'escursione;
- *GPSCoordinate*: contiene i dati GPS di una coordinata;

- *Photo*: rappresenta una fotografia scattata;
- *Poi*: rappresenta un POI (Point Of Interest) all'interno di un'escursione.

La figura 3.7 mostra le relazioni tra le entità persistenti del modulo *db* sopra descritte.

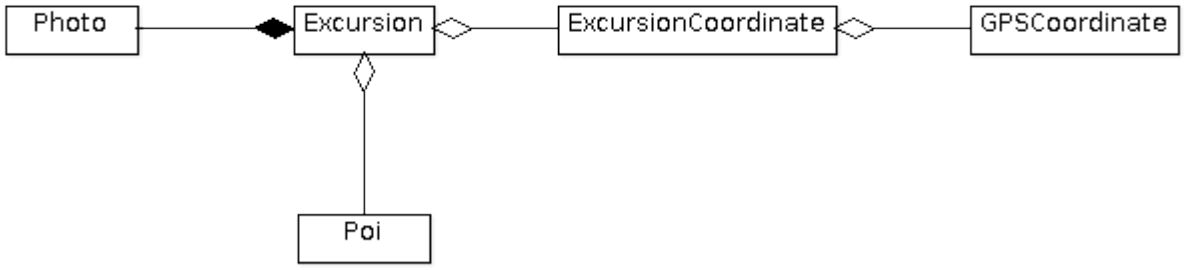

Fig. 3.1: Entità persistenti del modulo db

3.3.2 Modulo *GUI*

Il modulo dell'interfaccia grafica è costituito da una serie di *activity* che consentono di visualizzare schermate con le quali l'utente può interagire. Queste *activity* consentono di eseguire diverse azioni quali:

- avvio di una nuova escursione
- visualizzazione ed eliminazione di escursioni esistenti
- visualizzazione delle statistiche real time dell'escursione in corso
- salvataggio, visualizzazione ed eliminazione di POI
- scatto di una nuova fotografia geotaggata
- upload dei dati raccolti verso il server remoto
- modifica dei parametri di configurazione dell'applicazione

L'esecuzione di queste azioni viene demandata agli altri moduli funzionali dell'applicazione illustrati nel paragrafo 3.3, i quali effettueranno le dovute elaborazioni per poi restituire i risultati a specifiche *activity* che ne consentono la visualizzazione all'utente.

Activity di visualizzazione dell'escursione con Google Maps

L'activity più interessante da descrivere è probabilmente quella che consente la visualizzazione del percorso seguito direttamente sulle mappe di Google. La figura 3.2 mostra uno screenshot della visualizzazione del percorso di un'escursione.

Fig. 3.2: visualizzazione del percorso

L'activity preposta alla visualizzazione del percorso di un'escursione è denominata `ExcursionMapActivity`, ed utilizza le API di *Google Maps* per disegnare la mappa e gli oggetti su di essa. L'activity estende la classe `MapActivity` che implementa già alcuni aspetti basilari per la visualizzazione di una mappa di Google. Iniziamo mostrando il file XML che descrive il layout della suddetta activity.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <com.google.android.maps.MapView
        xmlns:android="http://schemas.android.com/apk/res/android"
```

```
            android:id="@+id/mapview"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:apiKey="0XR_rxO17-YWxpfJgC4YqudID2IfgQy_NMdsgPg"
            android:clickable="true" />

        <ImageButton
            android:id="@+id/add_poi"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="bottom|right"
            android:background="#444444"
            android:contentDescription="@string/add_new_poi"
            android:padding="10dip"
            android:src="@drawable/add_poi_icon"
            android:textSize="12sp" />
    </FrameLayout>
```

Notiamo l'esistenza di un tag (`com.google.android.maps.MapView`) che definisce una view di tipo `MapView` e denominata `mapview`. Tale MapView deve contenere un attributo denominato `android:apiKey` il cui valore contiene la chiave personale per l'accesso alle API di Google Maps. Questa chiave è richiedibile gratuitamente attraverso il seguente link: https://developers.google.com/android/maps-api-signup

Il codice dell'activity deve recuperare il cosiddetto *controller* della mappa stessa ed inserire degli *overlay*, cioè layer grafici che verranno disegnati sopra la mappa stessa.

```
MapView mapView = (MapView) findViewById(R.id.mapview);
MapController mapController = mapView.getController();
List<Overlay> mapOverlays = mapView.getOverlays();

PathOverlayManager pathOverlayManager = new PathOverlayManager(this, mapOverlays,
mapView.getProjection());
drawPath(pathOverlayManager);
```

La classe `PathOverlayManager` ha lo scopo di disegnare il layer relativo al percorso seguito, cioè la riga rossa che congiunge tutte le coordinate collezionate durante l'escursione. Il metodo `drawPath(PathOverlayManager)` ha il compito di recuperare tutte le coordinate dell'escursione e di aggiungerle al layer del percorso.

Nel caso in cui siano state scattate fotografie durante il percorso, queste risulteranno geotaggate e visualizzabili sulla mappa nel punto esatto in cui sono state catturate. Il seguente codice mostra come viene creato il layer relativo alle fotografie.

```
Drawable marker = this.getResources().getDrawable(R.drawable.camera_icon);
PhotoItemizedOverlay photoItemizedOverlay = new PhotoItemizedOverlay(this,
mapView , marker);
mapOverlays.add(photoItemizedOverlay);
```

```
insertPhotoPoints(photoItemizedOverlay);
```

Il metodo `insertPhotoPoints(PhotoItemizedOverlay)` recupera tutte le foto scattate durante l'escursione e le aggiunge al layer delle fotografie. La classe `PhotoItemizedOverlay` rappresenta il layer delle fotografie.

Durante la visualizzazione del percorso, toccando l'icona di una fotografia ne verrà mostrata un'anteprima (Fig. 3.3); toccando l'anteprima verrà aperto il visualizzatore di deafult delle immagini; scorrendo il dito verso sinistra o verso destra verrà visualizzata l'anteprima dell'immagine successiva o precedente.

Fig. 3.3: Anteprima di una fotografia

Per ulteriori dettagli sulla realizzazione del modulo *GUI* si rimanda al codice sorgente.

Interfaccia di gestione dell'escursione in fase di esecuzione

L'interfaccia che ci si presenta davanti durante l'esecuzione di un'escursione è composta da tre

schermate:

- schermata delle statistiche in tempo reale
- schermata della mappa
- schermata dei POI

L'activity demandata alla visualizzazione delle statistiche in tempo reale è denominata `RealTimeStatisticsActivity`. La figura 3.4 mostra l'interfaccia utente esposta dall'activity.

Fig. 3.4: Statistiche in tempo reale

L'activity visualizza le informazioni relative al ritmo istantaneo, alla bussola, ai dislivelli totale e parziali percorsi, alla distanza coperta ed al numero di passi eseguiti (l'immagine proviene da un emulatore quindi i dati mostrati non sono da considerare realistici).

La schermata relativa alla mappa mostra, in tempo reale, l'attuale posizione sulla mappa (Fig. 3.5), ed è gestita dall'activity denominata `RealTimeMapActivity`.

Fig. 3.5: Schermata della mappa

Da questa activity è possibile aggiungere punti di interesse sfiorando il pulsante in basso a destra.

La schermata dei POI (Fig. 3.6) è gestita dall'activity `DistanceToPoiActivity`, e consente la visualizzazione di informazioni relative ai POI aggiunti durante l'escursione e ricevuti da altri dispositivi (vedi paragrafo 3.3.7).

Fig. 3.6: Schermata dei POI

3.3.3 Modulo *camera*

Il modulo della fotocamera si occupa dell'accesso all'hardware adeguato per catturare fotografie, della loro persistenza e marcatura geografica.

Lo scatto della fotografia viene demandato all'applicazione fotocamera preinstallata sul dispositivo, sfruttando il meccanismo degli *intent*.

```
// create a file to save the image
fileUri = UtilityCamera.getOutputMediaFileUri(UtilityCamera.MEDIA_TYPE_IMAGE);

            if(!UtilityCamera.launchCamera(this, fileUri,
CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE)) {
                    Toast.makeText(this, "Unable to create file",
Toast.LENGTH_LONG).show();
            }
```

Il metodo `getOutputMediaFileUri(int type)` della classe `UtilityCamera` ha lo scopo di preparare il file in cui verrà salvata la fotografia. Vediamo nel dettaglio il metodo che consente di lanciare un

intent che verrà catturato dall'applicazione di gestione della fotocamera preinstallata:

```
public static boolean launchCamera(Activity parent, Uri fileUri, int requestCode) {
	// create Intent to take a picture and return control to the calling
application
	Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);

	if(fileUri != null) {
		// set the image file name
		intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri);

		// start the image capture Intent
		parent.startActivityForResult(intent, requestCode);

		return true;
	}

	return false;
}
```

Notiamo come viene richiamata un'*activity* con richiesta di restituzione di un risultato, e come viene inserito nell'*intent* l'*URI* del file in cui salvare la fotografia.

L'activity dalla quale proviene la richiesta di scattare una fotografia dovrà implementare un metodo denominato onActivityResult per catturare il risultato prodotto dall'applicazione di gestione della fotocamera.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
	if(requestCode == CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE) {
		if(resultCode == RESULT_OK) {
			if(statisticsFromStartPoint.isGpsEnabled()) {
				// Image captured and saved to fileUri specified in the
Intent
				Toast.makeText(this, "Image saved to:\n" + fileUri,
Toast.LENGTH_LONG).show();
				Location location =
statisticsFromStartPoint.getCurrentLocation();
				UtilityCamera.storeGeotagInfo(fileUri.getPath(),
location.getLatitude(), location.getLongitude());
				UtilityCamera.deleteImageFromDCIM(this,
fileUri.getPath());
				UtilityCamera.updateGallery(this);

				// Save photo into database
				UtilityCamera.savePhotoIntoDatabase(this, excursionId,
fileUri, location);
			}
			else {
				Toast.makeText(this, "GPS not available: image saved in
the default gallery", Toast.LENGTH_LONG).show();
			}
		}
```

```
                else if(resultCode == RESULT_CANCELED) {
                        Toast.makeText(this, "Operation aborted",
Toast.LENGTH_LONG).show();
                }
                else {
                        Toast.makeText(this, "Failed to take photo",
Toast.LENGTH_LONG).show();
                }
        }
        else {
                super.onActivityResult(requestCode, resultCode, data);
        }
}
```

Notiamo come, in caso di successo dello scatto fotografico, venga recuperata la posizione GPS corrente per essere poi memorizzata nel file della fotografia e all'interno del database attraverso i seguenti metodi della classe `UtilityCamera`:

```
public static void savePhotoIntoDatabase(Context ctx, int excursionId, Uri fileUri,
Location location) {
        PhotoDAO photoDAO = new PhotoDAO(ctx);
        photoDAO.open();
        Photo photo = new Photo(fileUri.getPath(), excursionId);
        photo.setLatitude(location.getLatitude());
        photo.setLongitude(location.getLongitude());
        photo.setId(photoDAO.insertPhoto(photo));
        photoDAO.close();
}


public static void storeGeotagInfo(String mediaPath, double latitude, double
longitude) {

        String[] latitudeDMS = toDMSFormat(latitude, LATITUDE);
        String[] longitudeDMS = toDMSFormat(longitude, LONGITUDE);

        try {
                ExifInterface exif = new ExifInterface(mediaPath);
                exif.setAttribute(ExifInterface.TAG_GPS_LATITUDE, latitudeDMS[0]);
                exif.setAttribute(ExifInterface.TAG_GPS_LATITUDE_REF, latitudeDMS[1]);
                exif.setAttribute(ExifInterface.TAG_GPS_LONGITUDE, longitudeDMS[0]);
                exif.setAttribute(ExifInterface.TAG_GPS_LONGITUDE_REF,
longitudeDMS[1]);
                exif.saveAttributes();
        } catch (IOException e) {
                e.printStackTrace();
        }
}
```

Il metodo `storeGeotagInfo(String mediaPath, double latitude, double longitude)` memorizza le informazioni di geotagging direttamente nel file della fotografia attraverso gli adeguati tag del formato *ExIf*.

3.3.4 Modulo *GPS*

Il modulo *GPS* ha il compito di rilevare l'avanzamento dell'utente tracciando il percorso da esso seguito. Il suo funzionamento prevede “l'ascolto” delle coordinate rilevate dall'antenna GPS del dispositivo per poterle memorizzare all'interno del database dell'applicazione. Il meccanismo di registrazione del percorso seguito fa uso di un servizio denominato `GPSLocationService` che si pone in ascolto degli aggiornamenti di localizzazione dati dall'antenna GPS del dispositivo e, a sua volta, avvisa ulteriori *listener* ad esso registrati. Il più importante *listener* nei confronti del servizio `GPSLocationService` è la classe `GPSDataCollector`, che ha il compito di registrare su database tutte le coordinate rilevate in modo da tracciare il percorso seguito dall'utente. Questo meccanismo consente all'applicazione di tracciare sempre il percorso seguito, anche quando la sua esecuzione viene spostata in background a causa dell'arrivo di una telefonata o di altre azioni volute dall'utente. Altro *listener* del servizio `GPSLocationService` è `PoiDistanceStatistics` del modulo *statistics* (paragrafo 3.3.6), che ha il compito di rilevare le distanze percorse dai vari POI aggiunti dall'utente lungo il percorso e di aggiornare così le informazioni contenute all'interno di `PoiRealTimeInfo` (del medesimo modulo *statistics)*: in questo modo è possibile scambiare tali informazioni con altri dispositivi (vedere paragrafo 3.3.7). La figura 3.7 mostra l'interazione appena descritta tra le entità. La classe `LocationManager` è una classe di supporto di Android per l'accesso alle informazioni di localizzazione del dispositivo.

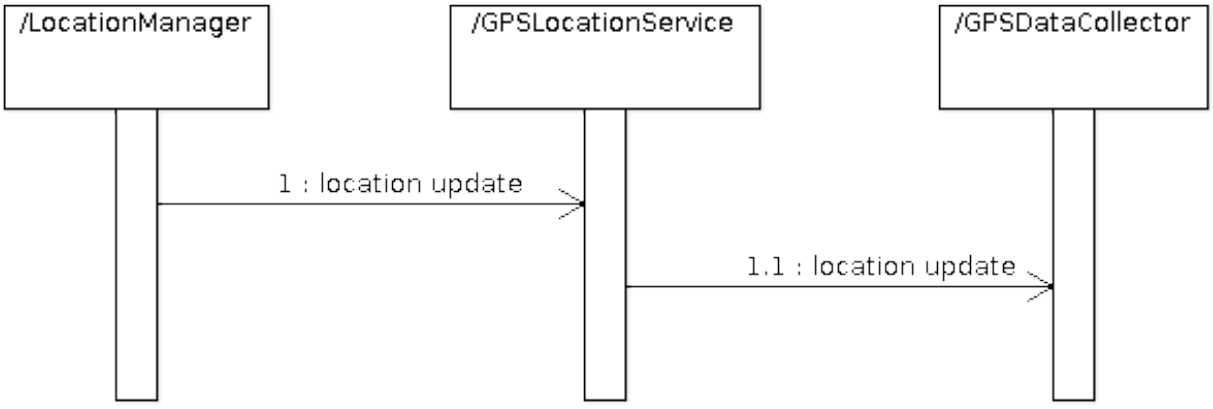

Fig. 3.7: Interazione per l'aggiornamento delle coordinate GPS

Diamo uno sguardo a qualche riga di codice per capire il funzionamento del modulo *GPS*. Iniziamo mostrando la registrazione del servizio `GPSLocationService` come *listener* verso il manager (`LocationManager`) del *location service* di Android.

```
LocationManager locationManager = (LocationManager)
getSystemService(Context.LOCATION_SERVICE);   // recuperiamo il LocationManager
locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, minTime,
minDistance, this);
locationManager.addGpsStatusListener(this);
```

Il servizio, come già accennato, implementa meccanismo di notifica verso eventuali listener che vogliono registrarsi ad esso, in modo che questi possano essere notificati al momento del cambiamento di locazione del dispositivo.

```
public void addGPSListener(IGPSListener listener) {
	gpsListeners.add(listener);
}
```

Al momento del cambiamento di posizione GPS del dispositivo, il `LocationManager` di Android richiama il seguente metodo di `GPSLocationService` che, a sua volta, notifica tutti i *listener* ad esso registrati:

```
public void onLocationChanged(Location location) {
	for(IGPSListener listener : gpsListeners) {
		listener.onLocationChanged(location);
	}
}
```

Per catturare e registrare le coordinate GPS in modo da tracciare tutto il percorso seguito dall'utente, la classe `GPSDataCollector` non deve fare altro che registrarsi come *listener* al `GPSLocationService` ed implementare il metodo `onLocationChanged(Location location)` come segue:

```
public void onLocationChanged(Location location) {
	ExcursionCoordinate coord = new ExcursionCoordinate(location);
	coord.setExcursionId(excursion.getId());
	gpsCoordinateDAO.insertGPSCoordinate(coord);
}
```

Il metodo sopra riportato non fa altro che creare una nuova coordinata di escursione (`ExcursionCoordinate`) e registrarla nel database.

Per ulteriori dettagli e spiegazioni sul modulo *GPS* si rimanda al codice sorgente ed ai commenti in esso riportati.

3.3.5 Modulo *sensors*

Il modulo *sensors* realizza le funzionalità di gestione dei sensori *accelerometro* e *bussola* del dispositivo Android.

La parte relativa al sensore *accelerometro* ha lo scopo principale di calcolare l'accelerazione misurata dall'hardware del dispositivo in modo da rendere disponibile tale valore al modulo *step* di contapassi (paragrafo 3.3.9).

La parte relativa al sensore bussola ha lo scopo di individuare il punto cartesiano verso cui il dispositivo è rivolto.

Come il modulo *GPS*, anche il modulo *sensors* prevede l'impiego di un servizio (`SensorsListenerService`) che ha il compito di porsi in ascolto verso aggiornamenti provenienti dai sensori *accelerometro* e *bussola*, e di notificare eventuali *listener* ad esso registrati. Questo meccanismo garantisce la continua elaborazione dei dati provenienti dai sensori, anche nel caso in cui l'esecuzione dell'applicazione *Trekker* venga posta in background.

I principali *listener* del servizio `SensorsListenerService` sono:

- la classe `StepDector` del modulo *step* (paragrafo 3.3.9), interessata al cambiamento di accelerazioni del dispositivo (implementa l'interfaccia `IAccelerationListener`);
- la classe `RealTimeStatistics` del modulo *statistics* (paragrafo 3.3.6), interessata al cambiamento dell'orientamento del dispositivo per realizzare la funzionalità di bussola (implementa l'interfaccia `IorientationListener`); il punto cardinale verrà mostrato a video dall'activity `RealTimeStatisticsActivity` del modulo *GUI*.

La figura 3.8 mostra come le entità dell'applicazione possono interagire con il modulo *sensors* per essere notificate di aggiornamenti di informazioni derivate dai sensori del dispositivo.

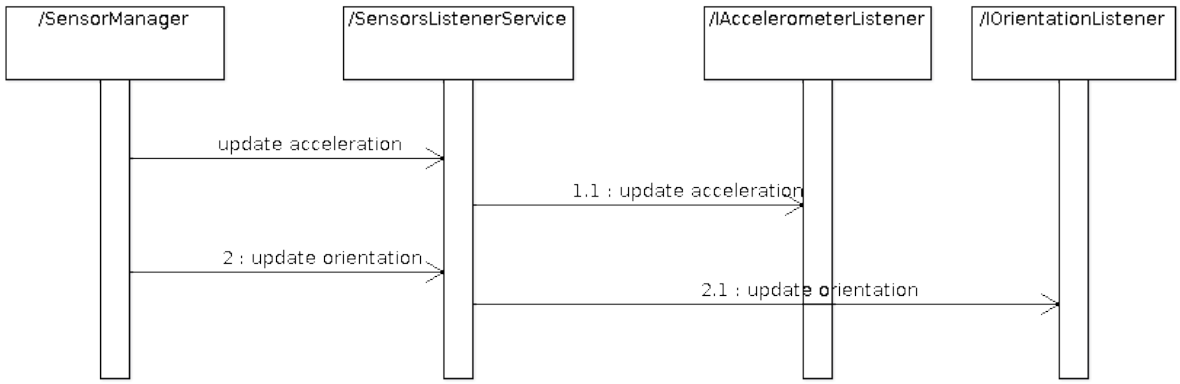

Fig. 3.8: Interazione tra le entità per la notifica dei cambiamenti derivati dai sensori

Nel seguente estratto di codice è mostrato il meccanismo con cui il servizio `SensorsListenerService` recupera il manager Android dei sensori (`SensorManager`) e i sensori stessi (`Sensor`), e come avviene la registrazione al loro servizio di notifica di aggiornamento.

```
SensorManager sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);

Sensor accelerometerSensor =
sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
sensorManager.registerListener(this, accelerometerSensor,
SensorManager.SENSOR_DELAY_NORMAL);

Sensor orientationSensor = sensorManager.getDefaultSensor(Sensor.TYPE_ORIENTATION);
sensorManager.registerListener(this, orientationSensor,
SensorManager.SENSOR_DELAY_UI);
```

Il servizio `SensorsListenerService` mette a disposizione due metodi per registrare i listener: uno per gli interessati all'aggiornamento dell'accelerazione ed uno per gli interessati all'aggiornamento dell'orientamento.

```
public void addAccelerometerListener(IAccelerometerListener listener) {
	accelerometerListeners.add(listener);
}

public void addOrientationListener(IOrientationListener listener) {
	orientationListeners.add(listener);
}
```

Per ulteriori approfondimenti si rimanda al codice sorgente del modulo *sensors*.

3.3.6 Modulo *statistics*

Il modulo *statistics* ha il compito di calcolare le statistiche di un'escursione, sia essa in corso di svolgimento o semplicemente selezionata dall'utente per visualizzarne i dettagli.

Le entità all'interno di questo modulo necessitano delle funzionalità offerte dai moduli *GPS* e *sensors*, in quanto elaborano i dati da essi pervenuti per calcolare le seguenti informazioni:

- distanza percorsa dall'inizio dell'escursione e da ogni POI aggiunto durante il percorso;
- ritmo istantaneo;
- ritmo medio;
- altitudine corrente;
- dislivello percorso;
- tempo impiegato;
- bussola.

Per realizzare il calcolo delle suddette statistiche, il modulo si avvale principalmente di tre entità:

- `GPSStatistics`: il cuore del calcolo delle statistiche; è la classe che implementa gli algoritmi che consentono di ricavare tutte le informazioni necessarie all'applicazione in tempo reale, semplicemente iniettando al suo interno l'ultima posizione GPS registrata;
- `RealTimeStatistics`: ha il compito di pilotare GPSStatistics nel calcolo delle statistiche passandole l'ultima coordinata di posizionamento registrata. Ha anche lo scopo di facilitare l'accesso alle statistiche real time alle activity che le dovranno visualizzare a video;
- `PoiDistanceStatistics`: estende le funzionalità di `RealTimeStatistics` con lo scopo di gestire le statistiche real time relative ai POI inserite dall'utente durante l'escursione. In particolare, consente di elaborare le coordinate GPS ricevute dal servizio `GPSLocationService` del modulo *GPS* (paragrafo 3.3.4) per tenere traccia della distanza percorsa dall'aggiunta di ogni POI.

Come i moduli precedenti, anche questo fa uso di meccanismi di *callback* per notificare l'aggiornamento delle informazioni inerenti le statistiche a tutti i *listener* interessati (principalmente

activity che mostrano le statistiche a video). Il principale *listener* di `RealTimeStatistics` è l'activity `RealTimeStatisticsActivity` del modulo *GUI* (paragrafo 3.3.2), la quale contiene un oggetto che implementa l'interfaccia di callback (`IRealTimeCallback`) richiesta da `RealTimeStatistics` per notificare i propri *listener*.

```
private RealTimeStatistics.IRealTimeCallback statisticsCallback = new
RealTimeStatistics.IRealTimeCallback() {

	@Override
	public void paceChanged(double value) {
		// value*1000 to preserve decimal digits
		mHandler.sendMessage(mHandler.obtainMessage(PACE_MSG, (int)
(value*1000), 0));
	}

	@Override
	public void distanceChanged(double value) {
		mHandler.sendMessage(mHandler.obtainMessage(DISTANCE_MSG, (int)
(value*1000), 0));
	}

	@Override
	public void altitudeChanged(double value) {
		mHandler.sendMessage(mHandler.obtainMessage(ALTITUDE_MSG, (int)
(value*1000), 0));
	}

	@Override
	public void altitudeDifferenceChanged(AltitudeDifferenceObject diff) {
		mHandler.sendMessage(mHandler.obtainMessage(ALTDIFF_MSG, diff));
	}

	@Override
	public void stepsChanged(long value) {
		Long steps = new Long(value);
		mHandler.sendMessage(mHandler.obtainMessage(STEPS_MSG, steps));
	}

	@Override
	public void orientationChanged(OrientationCoord orientation) {
		mHandler.sendMessage(mHandler.obtainMessage(ORIENTATION_MSG,
orientation));
	}
};
```

L'oggetto `mHandler` che viene utilizzato nel codice sopra riportato è un gestore della coda di messaggi dell'activity, meccanismo impiegato da Android per inviare messaggi all'activity stessa in modo asincrono. Tale meccanismo è spesso utilizzato per l'aggiornamento dell'interfaccia grafica.

Il principale *listener* di `PoiDistanceStatistics` è l'activity `DistanceToPoiActivity` del modulo *GUI* (paragrafo 3.3.2), la quale contiene un oggetto che implementa l'interfaccia di callback

(`IDistanceToPoiCallback`) richiesta da `PoiDistanceStatistics` per notificare i propri *listener*. Tale oggetto riceve notifiche inerenti l'aggiornamento delle statistiche relative ai POI ed alla loro aggiunta, potendo così eseguire rinfrescare le informazioni a video.

La figura 3.9 mostra le interazioni tra le entità del modulo *statistics* e le entità esterne ad esso legate.

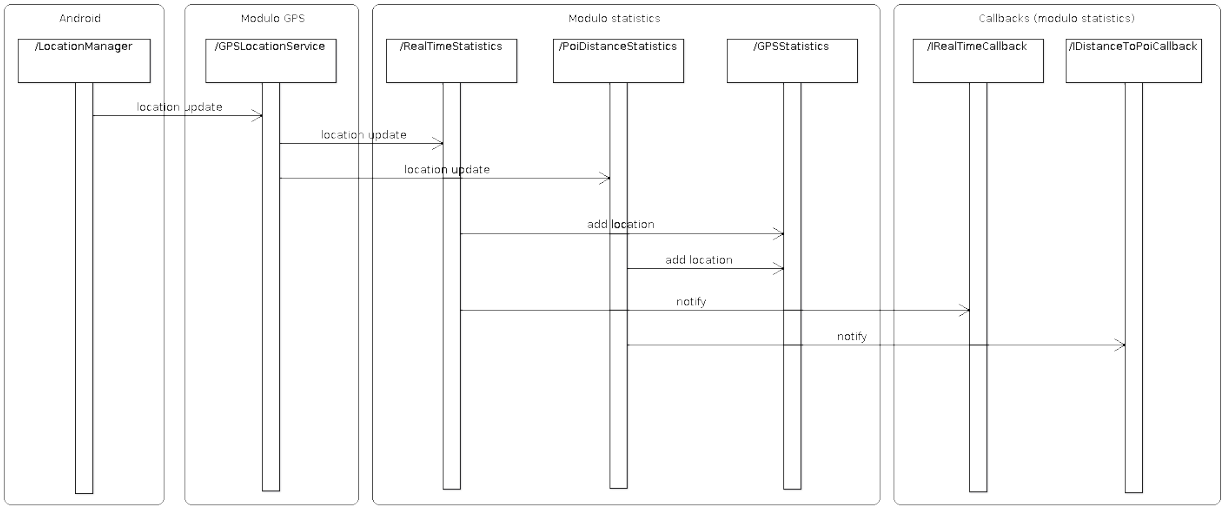

Fig. 3.9: Meccanismo di aggiornamento delle statistiche e loro notifiche

In figura 3.9 possiamo notare come il manager `LocationManager` di Android notifichi al servizio `GPSLocationService` del modulo GPS il cambiamento di posizione del dispositivo. A sua volta, `GPSLocationService` notifica a `RealTimeStatistics` ed a `PoiDistanceStatistics` l'avvenuto cambiamento di posizione, i quali eseguiranno le elaborazioni opportune per aggiornare le statistiche previste ed avvisare i *listener* registrati attraverso oggetti di *callback*.

3.3.7 Modulo *data exchange*

Il modulo *data exchange* consente lo scambio di dati tra dispositivi che vengono incrociati lungo il percorso seguito e che stanno eseguendo l'applicazione *Trekker*.

Questo modulo fa uso della libreria *Cling* [CLING] per sfruttare il protocollo *UPnP* che consente di scoprire ed usufruire di servizi presenti su altri dispositivi. Nel nostro caso specifico, la necessità che abbiamo è quella di verificare la presenza di un servizio di scambio dati offerto dall'applicazione *Trekker* presente sugli altri dispositivi che incontriamo lungo il cammino.

Nel seguito verrà definito *server* il pari che invia i dati, *client* il pari che li richiede e li riceve.

Scendiamo un po' più nel dettaglio analizzando fin da subito il codice sorgente perchè, si sa, un po' di codice vale più di mille parole.

Il seguente scorcio di codice illustra il servizio di scambio dati dell'applicazione *Trekker*.

```
@UpnpService(
	serviceId = @UpnpServiceId("TrekkerDataExchange"),
	serviceType = @UpnpServiceType(value = "DataExchange", version = 1)
)
public class DataExchange {

	@UpnpStateVariable
	private String data;

	@UpnpAction(
		name = "RetrieveData",
		out = @UpnpOutputArgument(name = "DataList", stateVariable = "data")
	)
	public String retrieveData() {

		List<PoiRealTimeInfo> pois =
PoiDistanceStatistics.getSharedInstance().getPoisList();

		if(pois.size() > 0) {
			IJsonConverter converter = new JsonConverter();
			data = converter.PoiRealTimeInfoArray2Json((PoiRealTimeInfo[])
pois.toArray());
		}
		else {
			data = "";
		}

		return data;
	}

}
```

La prima cosa che possiamo notare è la presenza di alcune annotazioni all'interno della nostra classe java. La libreria *Cling* ci consente di specificare i metadati che descrivono il servizio direttamente sottoforma di annotazioni java, rendendo così molto più rapida la sua implementazione. Come specificato nella documentazione di Cling, sono possibili altri meccanismi per la descrizione del servizio da offrire attraverso il protocollo *UPnP*:

> *The annotations are used by Cling to read the metadata that describes your service, what UPnP state variables it has, how they are accessed, and what methods should be exposed as UPnP actions. You can also provide Cling metadata in an XML file or programmatically through Java code - both options are discussed later in this manual. Source code annotations are usually the best choice. [DOCCL]*

L'annotazione `@UpnpService` consente di specificare l'identificativo del servizio ed il tipo, in

modo da poterlo riconoscere facilmente al momento del discovery.

L'annotazione `@UpnpAction` consente di specificare l'azione accessibile dall'esterno attraverso il protocollo *UPnP*, nel nostro caso è denominata *RetrieveData* e restituisce un risultato denominato *DataList* rappresentato dalla variabile *data*. Il risultato dell'azione *RetrieveData* sarà la lista dei POI collezionati dal dispositivo che offre il servizio, associati alla distanza ed al ritmo medio rilevati dal momento di aggiunta del POI, al momento dell'invio verso un altro dispositivo.

Chiariamo con un esempio pratico quali dati vengono scambiati: supponiamo che l'utente *User1* aggiunga un POI denominato *POI1* al proprio dispositivo all'istante *i1*; all'istante *i2*, *User1* incrocia sul proprio percorso un utente *User2* dotato di un dispositivo Android 4.1 che sta eseguendo la stessa applicazione *Trekker* con i servizi di *data exchange* funzionanti. I dispositivi di *User1* e *User2* stabiliscono una connessione P2P utilizzando la tecnologia *WiFi Direct* ed iniziano il discovery del servizio *data exchange*. All'istante *i3*, il servizio *data exchange* di *User2* richiede i dati al dispositivo di *User1*, che invia dunque informazioni che descrivono *POI1*, la distanza percorsa dall'istante *i1* (istante di aggiunta di *POI1*) e l'istante *i3* (istante di invio dei dati) e il ritmo medio mantenuto da *User1*. Queste informazioni sono calcolate dal modulo *statistics* di paragrafo 3.3.6, e vengono scambiate con il supporto della classe `PoiRealTimeInfo` per la serializzazione in formato *Json*.

Il server

La prima cosa da fare nella parte server è creare un *device UPnP* che offra il servizio implementato precedentemente.

```
private LocalDevice createDevice() throws ValidationException,
LocalServiceBindingException, IOException {

	DeviceIdentity identity =
			new DeviceIdentity( UDN.uniqueSystemIdentifier("Data
exchange") );

	DeviceType type =
			new UDADeviceType("DataExchange", 1);

	DeviceDetails details =
			new DeviceDetails(
					"Friendly Data Exchange",
					new ManufacturerDetails("ACME"),
					new ModelDetails(
							"TrekkerDataExchange",
							"A demo data exchange",
							"v1"
```

```
                                        )
                        );

        LocalService<DataExchange> dataExchangeService =
                        new AnnotationLocalServiceBinder().read(DataExchange.class);

        dataExchangeService.setManager(
                        new DefaultServiceManager(dataExchangeService,
DataExchange.class)
                        );

        return new LocalDevice(identity, type, details, dataExchangeService);
}
```

Come illustrato in [DOCCL] paragrafo 2.2 *Binding a UPnP device*, è necessario specificare diversi parametri per poter creare un *device* conforme alle specifiche *UPnP*. Innanzitutto dobbiamo specificare un *DeviceIdentity*, cioè un identificatore univoco per il *device*, sia esso un *root device* o un *device innestato*. Questo identificatore deve essere sempre lo stesso per questo *device*, non deve cambiare alla sua riaccensione. Altro parametro da specificare è il *DeviceType*, ovvero il tipo del *device*. Questo serve al protocollo *UPnP* per realizzare il *discovery* e consentire ai *control points* di accedere correttamente ai servizi richiesti. Stesso discorso può essere fatto per i parametri che costruiscono il *DeviceDetails*: hanno lo scopo di estendere la descrizione del *device* creato.

Specificati i parametri, è necessario creare il *service* (notare che viene creato a partire da una classe `AnnotationLocalServiceBinder` che “legge” le annotazioni) ed associarlo ad un *ServiceManager* che funge da collante tra i metadati che descrivono il servizio e la sua implementazione vera e propria. In altre parole, il *ServiceManager* è la factory che istanzia l'attuale implementazione del servizio *UPnP* descritto dai metadati.

Per avviare il server dobbiamo dapprima lanciare il servizio UPnP per Android implementato dalla libreria *Cling* ed agganciarci ad esso, poi possiamo aggiungere il *device* appena creato al registro di questo servizio.

```
public void startUPnP(boolean startServer, boolean startClient) {

        ...

        if(startServer || startClient) {
                upnpServiceConnection = createServiceConnection();
                context.bindService(
                        new Intent(context, UPnPDataExchangeService.class),
                        upnpServiceConnection,
                        Context.BIND_AUTO_CREATE
                );
        }
```

```
}


private void pullUpServer() {
	try {

		upnpService.getRegistry().addDevice( createDevice() );

	} catch (RegistrationException e) {
		...
	}
}
```

Il metodo createServiceConnection() restituisce una classica ServiceConnection di Android.

```
private ServiceConnection createServiceConnection() {
	return (new ServiceConnection() {
		@Override
		public void onServiceConnected(ComponentName className, IBinder
service) {
			upnpService = (AndroidUpnpService) service;

			if(startClient) {
				pullUpClient();
			}
			if(startServer) {
				pullUpServer();
			}
		}

		@Override
		public void onServiceDisconnected(ComponentName className) {
			upnpService = null;
		}
	});
}
```

Notiamo che il *service* passato al metodo `onServiceConnected(...)` è di tipo `AndroidUpnpService`, classe della libreria *Cling* che realizza il servizio per l'utilizzo del protocollo *UPnP*.

Il Client

Il client UPnP prevede la creazione di un *listener* che “ascolta” il registro UPnP per catturare l'aggiunta di nuovi *device* scoperti, poi recupera il control point dal servizio UPnP ed avvia la prima ricerca. Per garantire il ritrovamento di altri *device* durante il tragitto, viene schedulata ad intervalli regolari l'esecuzione di un task che esegue la ricerca.

```
private void pullUpClient() {

	registryListener = createRegistryListener(upnpService);

	// Getting ready for future device advertisements
	upnpService.getRegistry().addListener(registryListener);

	// Search asynchronously for all devices
	upnpService.getControlPoint().search();

	timerSearch = new Timer();
	timerSearch.scheduleAtFixedRate(createTimerTaskSearch(), 0,
searchingInterval);
}
```

Il metodo `createRegistryListener(...)` restituisce l'implemntazione di un `RegistryListener` (della libreria Cling) che avvia il recupero dei dati da ogni *device* che offre il servizio illustrato nel paragrafo *Il server*. Nel seguito possiamo notare come vengono filtrati i *device* ritrovati che offrono il servizio di nostro interesse, e come viene invocato il metodo per il recupero dei dati voluti.

```
RegistryListener createRegistryListener(final AndroidUpnpService upnpService) {
	return (new DefaultRegistryListener() {
	ServiceId serviceId = new UDAServiceId("TrekkerDataExchangeServiceId");

	@Override
	public void remoteDeviceAdded(Registry registry, RemoteDevice device) {

		Service dataExchange;
		if ((dataExchange = device.findService(serviceId)) != null) {
			Log.d(TAG, "Service discovered: " + dataExchange);
			executeAction(upnpService, dataExchange);
		}

	}

	@Override
	public void remoteDeviceRemoved(Registry registry, RemoteDevice device) {
		Service dataExchange;
		if ((dataExchange = device.findService(serviceId)) != null) {
			Log.d(TAG, "Service disappeared: " + dataExchange);
		}
	}

});
}


protected void executeAction(AndroidUpnpService upnpService, Service
dataExchangeService) {

	ActionInvocation getDataInvocation = new
GetDataActionInvocation(dataExchangeService);

	// Executes asynchronous in the background
	upnpService.getControlPoint().execute(
		new ActionCallback(getDataInvocation) {

			@Override
```

```
                    public void success(ActionInvocation invocation) {
                        // Save retrieved data
                        …
                    }

                    @Override
                    public void failure(ActionInvocation invocation,
                        UpnpResponse operation, String defaultMsg) {
                        …
                    }
                }
        );
    }


    protected class GetDataActionInvocation extends ActionInvocation {
        GetDataActionInvocation(Service service) {
            super(service.getAction("RetrieveData"));
        }
    }
```

La classe Cling `ActionInvocation` consente un controllo molto fine dell'invocazione del servizio richiesto, dando, ad esempio, la possibilità di settare parametri di input, di specificare come l'azione deve essere eseguita.

3.3.8 Modulo *wifidirect*

La libreria *Cling* per il protocollo UPnP non funziona senza infrastruttura; in altre parole è necessario disporre di una rete IP sulla quale poter scambiare i messaggi multicast per fare *discovery*, *advertising* ed utilizzare i servizi UPnP. A questo scopo, il nostro progetto fa uso della tecnologia *WiFi Direct* [WIFID] disponibile a partire da Android 4.1 per realizzare un'infrastruttura WiFi ad hoc tra due o più dispositivi comunicanti. La tecnologia *WiFi Direct* consente la creazione di cosiddetti *gruppi* che, una volta creati, operano come normali reti IP. Risulta inoltre possibile creare connessioni tra dispositivi non tutti *WiFi Direct* compatibili, basta che lo sia soltanto chi inizia la connessione.

Il modulo *wifidirect* ha lo scopo di creare una rete ad hoc utilizzando la tecnologia *WiFi Direct*, consentendo così la fruizione del protocollo UPnP per realizzare la funzionalità di condivisione dati tra dispositivi che si incrociano lungo il percorso. In realtà le specifiche *WiFi Direct* indicano la disponibilità di un meccanismo di *service discovery* integrato nella tecnologia ma, essendo opzionale, sembra non essere ancora (e forse non lo sarà mai) disponibile su piattaforma Android.

Le classi rilevanti di questo modulo sono le seguenti:

- `WiFiDirectBroadcastReceiver`: estende il componente `BroadcastReceiver` di Android e ha lo

scopo di “ascoltare” i cambiamenti di stato dell'interfaccia di connessione *WiFi Direct*;

- `WiFiDirectConnectionInfoListener`: implementa l'interfaccia di *WiFi Direct* `ConnectionInfoListener` e ha lo scopo di avviare il discovery del servizio di scambio dati utilizzando il modulo *dataexchange* (paragrafo Errore: sorgente del riferimento non trovata) al termine della negoziazione dei parametri di connessione;
- `WiFiDirectPeerListListener`: implementa le interfacce di *WiFi Direct* `PeerListListener` e `ChannelListener`, con ha lo scopo di “ascoltare” il ritrovamento di nuovi dispositivi, avviare nuove connessioni con essi e gestire le disconnessioni.

Nel seguito mostriamo come viene avviato il meccanismo di gestione di *WiFi Direct* e di scambio dati (paragrafo Errore: sorgente del riferimento non trovata).

Intent per ricevere i diversi eventi *WiFi Direct* che ci interessano:

```
intentFilter.addAction(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION);
intentFilter.addAction(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION);
intentFilter.addAction(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION);
intentFilter.addAction(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION);
```

Istanziazione ed avvio dei componenti sopra descritti:

```
WifiP2pManager manager = (WifiP2pManager)
getSystemService(Context.WIFI_P2P_SERVICE);
Channel channel = manager.initialize(this, getMainLooper(), null);

WiFiDirectPeerListListener peerListener = new WiFiDirectPeerListListener(this,
manager, channel);
ConnectionInfoListener infoListener = new WiFiDirectConnectionInfoListener();

BroadcastReceiver receiver = new WiFiDirectBroadcastReceiver(manager, channel,
peerListener, infoListener);
registerReceiver(receiver, intentFilter);
```

Schedulazione della ricerca di nuovi dispositivi WiFi cui connettersi per effettuare scambio dati:

```
Timer timerSearch = new Timer();
timerSearch.scheduleAtFixedRate(createTimerTaskSearch(), 0, 1000);
```

Avvio della parte UPnP di scambio dati:

```
// Start data exchange service

SharedPreferences sharedPref = PreferenceManager.getDefaultSharedPreferences(this);
boolean startClient = sharedPref.getBoolean("upnp_client_enable", true);
boolean startServer = sharedPref.getBoolean("upnp_server_enable", true);
int searchingInterval =
```

```
Integer.parseInt(sharedPref.getString("upnp_searching_interval", "2000"));
            dataExchangeService =
ClingDataExchangeService.getInstanceAndCreateIfNotExist(getApplicationContext(),
PoiDistanceStatistics.getSharedInstance());
dataExchangeService.setSearchingInterval(searchingInterval);
dataExchangeService.startUPnP(startServer, startClient);
```

Ogni volta che accade qualcosa relativo ad azioni di cambiamento stato di WiFi Direct, ritrovamento di nuovi dispositivi, cambiamenti di connessione e cambiamenti di stato dei dispositivi già trovati, viene invocato il metodo `onReceive(Context context, Intent intent)` della classe `WiFiDirectBroadcastReceiver`.

```
public void onReceive(Context context, Intent intent) {
      String action = intent.getAction();
      if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals(action)) {
            ...
      }
      else if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {
            // request available peers from the wifi p2p manager. This is an
            // asynchronous call and the calling activity is notified with a
            // callback on PeerListListener.onPeersAvailable()
            if (manager != null) {
                  manager.requestPeers(channel, peerListListener);
            }
      }
      else if (WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(action)) {

            if (manager == null) {
                  return;
            }

            NetworkInfo networkInfo = (NetworkInfo)
intent.getParcelableExtra(WifiP2pManager.EXTRA_NETWORK_INFO);

            if (networkInfo.isConnected()) {

                  // we are connected with the other device, request connection
                  // info to find group owner IP
                  manager.requestConnectionInfo(channel, connectionInfoListener);
            }
            else {
                  // It's a disconnect
                  peerListListener.resetData();
            }
      }
      else if (WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION.equals(action)) {
      }
}
```

In questo scorcio di codice notiamo che quando viene trovato un nuovo dispositivo WiFi (*`WIFI_P2P_PEERS_CHANGED_ACTION`*) si richiede al manager di WiFi Direct la lista dei dispositivi ritrovati. Al completamento della chiamata `requestConnectionInfo` viene notificato l'oggetto `peerListListener` di tipo `WiFiDirectPeerListListener`, richiamando il seguente metodo:

```
public void onPeersAvailable(WifiP2pDeviceList peerList) {
	peers.clear();
	peers.addAll(peerList.getDeviceList());
	if (peers.size() == 0) {
		Log.d(TAG, "No devices found");
		return;
	}

	for(WifiP2pDevice device : peers) {
		WifiP2pConfig config = new WifiP2pConfig();
		config.deviceAddress = device.deviceAddress;  // MAC address
		config.wps.setup = WpsInfo.PBC;  // WiFi protected Setup information
		connect(config);
	}
}
```

Notiamo che, quando viene scoperto un nuovo dispositivo, si avvia una connessione con esso attraverso il metodo `connect(...)` (per il metodo si rimanda al codice sorgente).

Quando viene intercettato l'evento *`WIFI_P2P_CONNECTION_CHANGED_ACTION`* causato da una connessione, vengono richieste le informazioni ad essa relative tramite la chiamata al metodo `requestConnectionInfo` del `manager`, il quale notificherà, al completamento, l'oggetto `connectionInfoListener` di tipo `WiFiDirectConnectionInfoListener` richiamando il seguente metodo:

```
public void onConnectionInfoAvailable(final WifiP2pInfo info) {
	// Search the service and use it if exist
	ClingDataExchangeService dataExchange =
ClingDataExchangeService.getInstance();
	if(dataExchange != null)
		dataExchange.search();
}
```

Il metodo riportato sopra ha il compito di avviare la ricerca del servizio di scambio dati della nostra applicazione e, se questo è presente, verrà utilizzato (vedere paragrafo Errore: sorgente del riferimento non trovata).

3.3.9 Modulo *step*

Il modulo *step* realizza la funzionalità di contapassi attraverso un'unica entità denominata `StepDetector`. Tale entità ha il compito di porsi in ascolto di aggiornamenti provenienti dal sensore accelerometro sfruttando il servizio `SensorsListenerService` del modulo *sensors* (paragrafo 3.3.5), e rilevare le accelerazioni tipiche dell'esecuzione di un passo. Ogni volta che viene rilevato un passo vengono notificati eventuali *listener* interessati (e preventivamente registrati presso la classe `StepDetector`). Il principale *listener* del contapassi è la classe `RealTimeStatistics` del modulo

statistics (paragrafo 3.3.6).

L'algoritmo per rilevare l'esecuzione di un passo sfruttando i dati dell'accelerometro è ispirato a quello proposto in [STEPC]. Nel seguito è riportato il codice atto a rilevare un passo.

```
long k = 0;
long i = k+1;
float max = 0;
double th = 0.0;
static double alpha = 0.8;       // constant
static final double beta = 3.1541;     // constant

private void detectStep(float accZ) {
       th = alpha / (i-k) + beta;

       double diff = Math.abs(max - accZ);
       if(diff >= th) {    // step detected
              k = i;
              max = accZ;

              for(IStepDetectorListener listener : stepDetectorListeners) {
                     listener.onStep();
              }
       }
       else {
              if(accZ > max) {
                     max = accZ;
              }
       }
       i++;

       // Reset to avoid overflow
       if(i == Long.MAX_VALUE) {
              long iMinusK = i-k;
              k = 0;
              i = iMinusK;
       }
}
```

L'algoritmo si basa sul rilevamento di picchi negativi e positivi di accelerazione che determinano rispettivamente l'inizio e la fine di un passo. Se la differenza tra questi due picchi supera una certa soglia (calcolata dinamicamente per adattarsi al ritmo sostenuto dall'utente) viene rilevato un passo.

I parametri coinvolti sono i seguenti:

- `th`: soglia da superare per rilevare un passo; calcolata dinamicamente;
- `i-k`: tentativi per rilevare il passo corrente;
- `1/(i-k)`: frequenza di tentativi per rilevare il passo corrente;
- `i`: numero del campione corrente;

- `k`: numero del campione dell'ultimo passo rilevato;
- `a`, `b`: costanti.

Per approfondimenti tecnici si rimanda alle spiegazioni contenute in [STEPC].

3.3.10 Modulo *upload*

Il modulo *upload* ha il compito di consentire l'invio dei dati relativi alle escursioni effettuate ad un server remoto. Questo modulo recupera dal database le informazioni da inviare, le trasforma in formato *json* e le inoltra ad un server che implementa un servizio *RESTful* (Capitolo 4:). Le informazioni inviate devono essere marcate dall'utente che le ha generate, cioè è necessario inserire all'interno delle informazioni un riferimento all'utente che ha realizzato le escursioni in esame.

Entriamo maggiormente nel dettaglio di questo modulo. Le entità implementate sono principalmente quattro:

- `ExcursionAdapter`: classe che consente di raggruppare le seguenti informazioni relative ad una singola escursione da inviare:
  - `Excursion`: informazioni relative all'escursione (corrisponde alla classe `Excursion` del modulo *db* illustrata nel paragrafo Errore: sorgente del riferimento non trovata)
  - lista di `ExcursionCoordinate`: informazioni relative alle coordinate collezionate durante il percorso di un'escursione (corrisponde ad una lista di oggetti di classe `ExcursionCoordinate` del modulo *db* illustrata nel paragrafo Errore: sorgente del riferimento non trovata)
  - lista di `Poi`: informazioni relative ai POI collezionati durante un'escursione (corrisponde ad una lista di oggetti di classe `Poi` del modulo *db* illustrata nel paragrafo Errore: sorgente del riferimento non trovata)
  - `User`: informazioni relative all'utente che ha generato l'escursione (corrisponde alla classe `User` del modulo *upload*)
- `User`: informazioni relative all'utente che genera un'escursione
- `UploadHelper`: classe di utilità per facilitare l'upload di informazioni verso il server remoto
- `UploadHelperREST`: classe derivata da `UploadHelper` che consente l'interazione con il server

remoto tramite approccio *REST*

I passaggi dettagliati per eseguire un upload sono i seguenti:

1. accorpamento delle informazioni di un'escursione all'interno di un oggetto di classe `ExcursionAdapter`
2. costruzione di una lista di oggetti di tipo `ExcursionAdapter` (uno per ogni escursione da inviare al server)
3. serializzazione su un file temporaneo della lista di oggetti di tipo `ExcursionAdapter` in formato json
4. recupero del file json delle escursioni e invio del suo contenuto verso il server remoto con approccio *REST*
5. eliminazione del file temporaneo costruito al punto 3

Il principale utilizzatore della classe `UploadHelperREST` è l'activity `ExcursionListViewActivity` del modulo *GUI* (Fig. 3.10). Il server risponde con un messaggio HTTP 200 OK (in caso di esito positivo) o con altri messaggi in caso di errore.

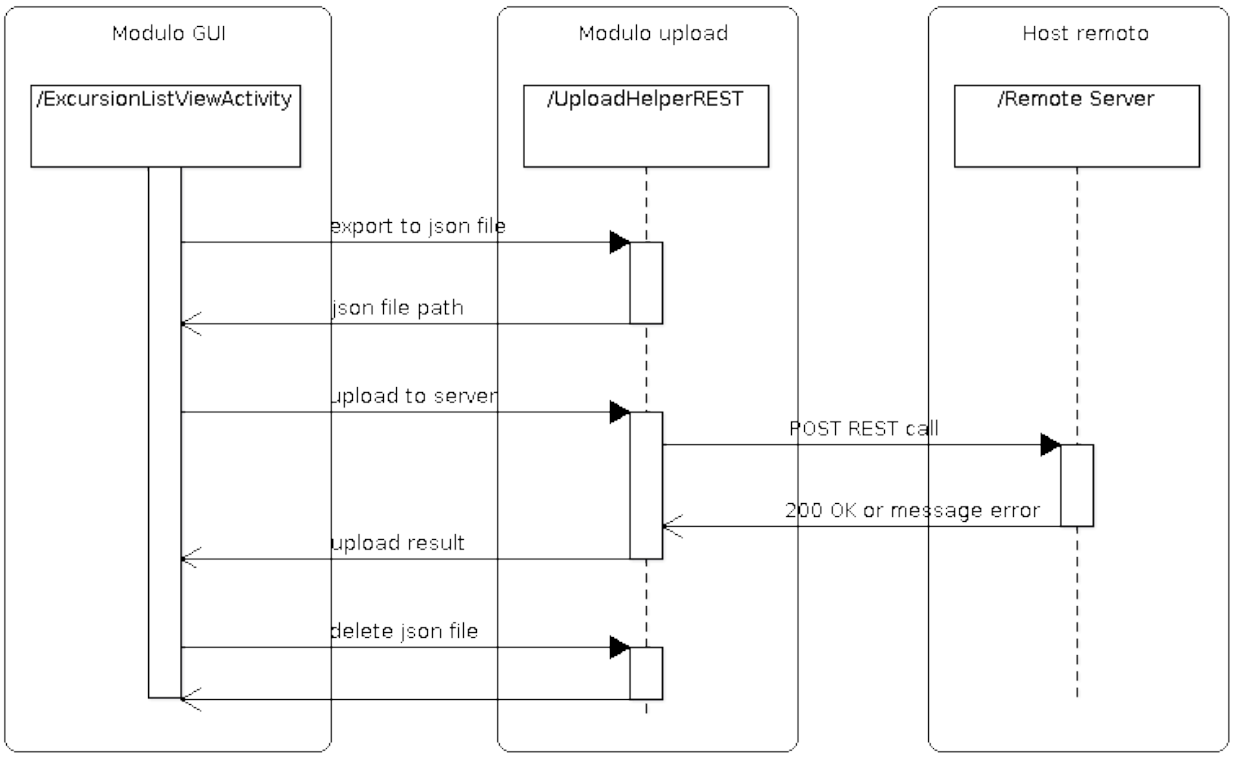

Fig. 3.10: Creazione e upload di una lista di escursioni

Vediamo ora il codice che implementa la funzionalità di upload del contenuto del file json.

```
public boolean uploadFile(String filePath, String serverUrl) {
	File f = new File(filePath);

	HttpClient httpclient = new DefaultHttpClient();
	HttpPost httppost = new HttpPost(serverUrl);

	try {
		FileEntity entity = new FileEntity(f, "application/json");
		httppost.setEntity(entity);
		HttpResponse response = httpclient.execute(httppost);

		if(response.getStatusLine().getStatusCode() == 200)
			return true;
	}
	catch(...) {
		...
	}

	return false;
}
```

Notiamo l'impiego di oggetti come `HttpClient` e `HttpPost` della libreria *HttpComponents* di *apache* [HTTPCOM] (già presente su piattaforma Android) che consentono di costruire facilmente messaggi http. Il contenuto del messaggio è di tipo *application/json* e viene caricato da un file ed inserito all'interno dell'oggetto `httppost`.

Per ulteriori dettagli sul modulo *upload* si rimanda al codice sorgente.

Capitolo 4: Server

Il server dell'applicazione *Trekker* ha lo scopo di raccogliere tutti i dati relativi alle escursioni realizzate dagli utenti, costruendo così un database globale accessibile a tutti.

Le funzionalità implementate dal server sono:

- raccolta delle informazioni inviate dagli utenti;
- ricerca di escursioni di specifici utenti o localizzate in un'area circoscritta ad un indirizzo dato;
- visualizzazione dei dettagli di un'escursione;
- prefetching dei dati in modo da velocizzare la visualizzazione di un'escursione da parte di un utente.
- autenticazione da parte dell'utente per l'eliminazione delle proprie escursioni.

Il server è sviluppato secondo un'architettura 4-tier, composta quindi da una parte dedicata alla persistenza dei dati, una dedicata alla logica di business (EJB), una dedicata alla parte di presentazione (JSP/Servlet) ed una dedicata alla visualizzazione da parte del cliente (browser).

4.1 Tecnologie utilizzate

Per la realizzazione delle funzionalità del server sono state adottate diverse tecnologie ed approcci implementativi.

Il server sfrutta la tecnologia EJB versione 3.1 per la realizzazione dei componenti che ne costituiscono il motore centrale, e impiega le *Java Server Pages (JSP)* per realizzare il tier di presentazione sottoforma di pagine web. Il server mette inoltre a disposizione un *web service RESTful* che consente l'upload dei dati da parte dei clienti, sia via interfaccia web che direttamente da smartphone tramite l'applicazione client descritta nel capitolo *3*.

Il server realizzato in questo progetto necessita di un application server che implementi l'intera

suite di servizi *Java EE* e che dia supporto alla rapida realizzazione di *web service RESTful*. Per queste ragioni la scelta è ricaduta sulla versione 7 dell'application server *JBoss* [JBOSS], ed in particolare sull'ultima release stabile denominata *Brontes* (versione *7.1.1 final*).

4.2 I componenti EJB

I componenti EJB realizzati all'interno di questo progetto hanno tre funzionalità principali:

- garantire l'accesso ai dati contenuti all'interno del database (paragrafo 4.2.1);
- offrire la funzionalità di prefetching per velocizzare il caricamento e la visualizzazione dei dettagli dell'escursione selezionata (paragrafo 4.2.2);
- garantire la memorizzazione nel database dei dati ricevuti dall'esterno (paragrafo 4.2.3).

4.2.1 Componenti DAO per la gestione dei dati nel database

La persistenza è gestita tramite le *Java Persistence API (JPA)* grazie all'utilizzo di *Hibernate*, framework integrato in *JBoss*.

Le entità da rendere persistenti sono le seguenti:

- *Excursion*: rappresenta un'escursione;
- *ExcursionCoordinate*: rappresenta una coordinata GPS rilevata lungo l'escursione;
- *Poi*: rappresenta un POI (Point Of Interest);
- *User*: rappresenta un utente del sistema.

Le suddette entità sono gestite da componenti EJB la cui implementazione segue il pattern DAO.

Mostriamo innanzitutto, attraverso un esempio, come sono realizzati gli oggetti che dovranno essere resi persistenti. La loro configurazione è realizzata tramite annotazioni JPA.

```
@Entity
public class Excursion implements Serializable {

	…

	@Id
	@GeneratedValue
	public int getId() {
		return id;
	}
```

```
        …

        @OrderBy("timestamp ASC")
        @OneToMany(
               fetch = FetchType.LAZY,
               cascade = {CascadeType.ALL},
               mappedBy = "excursion"
        )
        public Set<ExcursionCoordinate> getCoordinates() {
               return coordinates;
        }

        …

        @OneToMany(
               fetch = FetchType.LAZY,
               cascade = {CascadeType.ALL},
               mappedBy = "excursion"
        )
        public Set<Photo> getPhotos() {
               return photos;
        }

        …

        @OneToMany(
               fetch = FetchType.LAZY,
               cascade = {CascadeType.ALL},
               mappedBy = "excursion"
        )
        public Set<Poi> getPois() {
               return pois;
        }

        …

        @Transient
        public GPSCoordinate getStartPoint() {
               …
        }

        …

        @ManyToOne(
               fetch = FetchType.EAGER,
               cascade = {CascadeType.PERSIST}
        )
        public User getUser() {
               return user;
        }

        …

}
```

Notiamo l'utilizzo di diverse annotazioni che consentono di specificare il ruolo e le relazioni degli attributi cui sono associati. Evidenziamo la presenza dell'annotazione di classe `@Entity`, il cui scopo è specificare al container *J2EE* che `Excursion` è una classe le cui istanze dovranno essere rese

persistenti attraverso l'utilizzo di strumenti esterni (in *JBoss 7.1.1 final* viene utilizzato il framework *Hibernate*).

Per la creazione di questi componenti è necessario specificare alcuni parametri di configurazione che possano indicare all'application server come devono essere trattati; a questo scopo, ancora una volta, risulta molto comodo utilizzare le annotazioni java. Riportiamo un esempio di configurazione per mostrare cosa viene specificato in tutti i componenti EJB che implementano le funzionalità DAO verso gli oggetti da rendere persistenti.

```
@Stateless
@Local(IExcursionDAO.class)
public class EJB3ExcursionDAO implements IExcursionDAO {
	...
}
```

Notiamo la presenza delle seguenti annotazioni:

- `@Stateless`, indica che gli oggetti di tipo `EJB3ExcursionDAO` dovranno essere trattati dal container come *Stateless Session Bean*;
- `@Local`, indica che i *Session Bean* di classe `EJB3ExcursionDAO` saranno acceduti localmente al server e non direttamente da clienti remoti; questo consente di gestire molto più efficientemente le chiamate a metodo (evitando *RMI* su *IIOP*) ed il passaggio di parametri (che verrà realizzato quindi by reference evitando il processo di *un/marshalling*).

Per maggiori dettagli si rimanda al codice sorgente.

4.2.2 Componenti per il prefetching dei dati

La funzionalità di *prefetching* offerta dal server garantisce un miglioramento di prestazioni dato dal caricamento preventivo di informazioni riguardanti le escursioni che l'utente vuole visualizzare. La politica di caricamento dei dati si basa sul ben noto *principio di località*: se un utente ha richiesto la visualizzazione di un'escursione situata in un certo punto, è probabile che la successiva richiesta riguardi un'escursione vicina a quella visualizzata in quel momento. Il meccanismo che implementa la politica appena descritta prevede l'impiego di una cache per il mantenimento delle escursioni precaricate e l'esecuzione dei seguenti passi:

1. ricezione della richiesta di visualizzazione di una determinata escursione;
2. ricerca dell'escursione richiesta all'interno della cache;

2.1. se l'escursione è in cache viene restituita

2.2. se l'escursione non è in cache, viene cercata nel database e restituita immediatamente al cliente, dopodichè la cache viene riempita (operazione in background) con escursioni vicine a quella richiesta

Altro obiettivo da raggiungere è la realizzazione di un'interfaccia che consenta di segnalare graficamente (tramite un'icona) escursioni vicine a quella visualizzata in un determinato momento. La figura 4.1 mostra tale funzionalità.

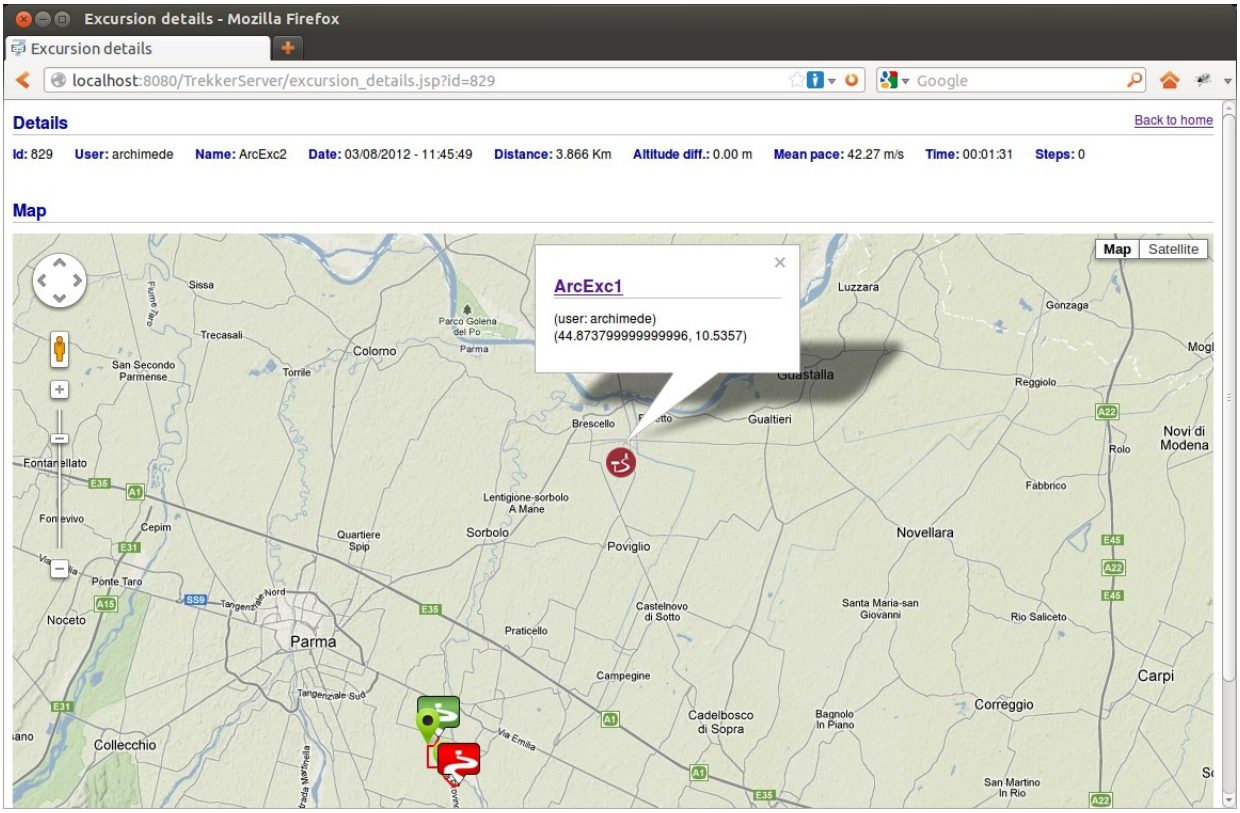

Fig. 4.1: Escursioni vicine a quella visualizzata

Per offrire quest'ultima funzionalità è necessario implementare un meccanismo che consenta al cliente di recuperare una lista di tutte le escursioni vicine a quella richiesta. Siccome la cache è già fornita dei dati relativi alle escursioni vicine a quella richiesta, risulta molto semplice recuperane una lista.

Funzionamento della cache

La cache memorizza al suo interno una lista di escursioni comprese in un'area di raggio

SEARCHING_RADIUS centrata in un dato punto geografico. Quando la cache è vuota e viene richiesta la visualizzazione di un'escursione, il centro della ricerca coincide con il punto di partenza della stessa escursione richiesta.

La figura 4.2 mostra come viene riempita la cache. Il punto blu rappresenta il *centro di riferimento della cache*, cioè il centro dell'area che contiene le escursioni da caricare (cerchio esterno arancione). Il cerchio interno verde racchiude le escursioni che saranno restituite come "lista di escursioni vicine al punto dato". In altre parole, la cache conterrà tutte le escursioni all'interno del cerchio arancione e, quando viene richiesta la lista delle escursioni vicine al punto blu, restituirà tutte le escursioni comprese nel cerchio verde. In questo modo la cache conterrà sempre un insieme di escursioni uguale o maggiore a quello da restituire quando richiesto.

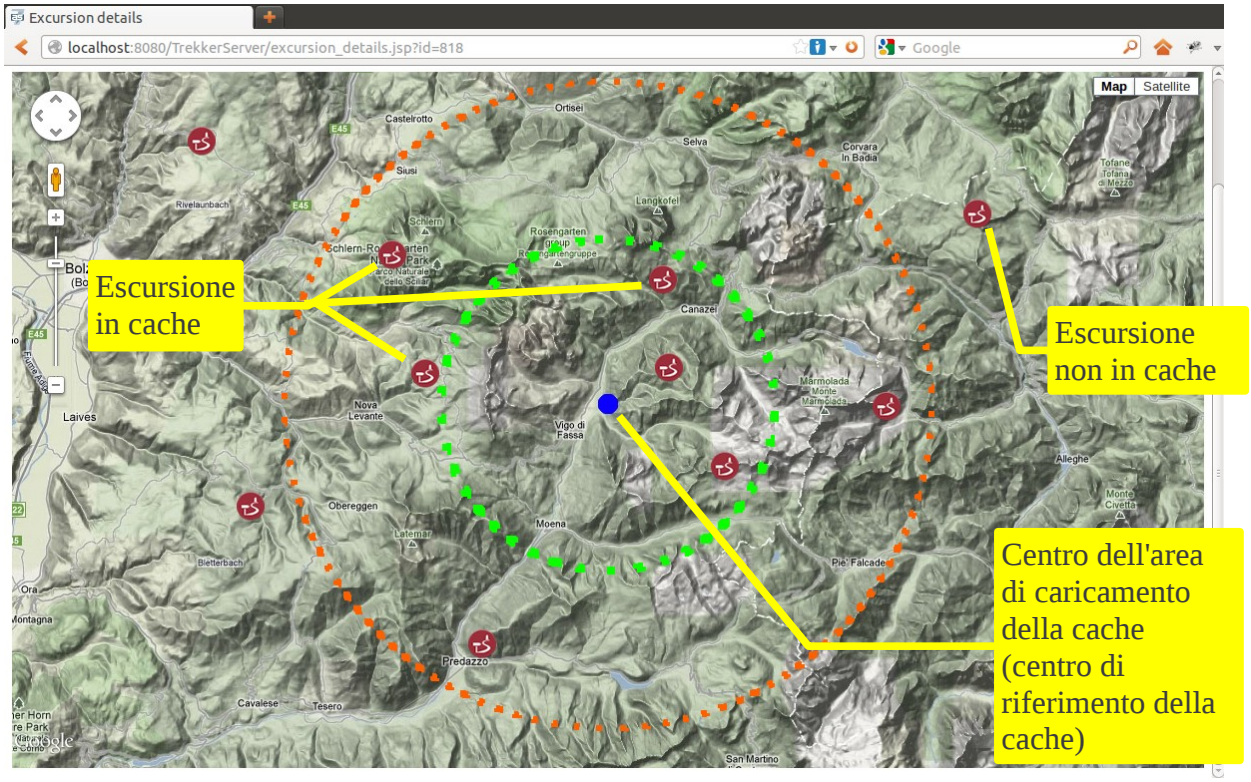

Fig. 4.2: Escursioni caricate in cache

Se il centro dell'area delle escursioni richieste cambia, la cache non deve necessariamente essere ripopolata in quanto, per il principio di località, probabilmente contiene già la lista di tutte le escursioni da restituire (Fig. 4.3).

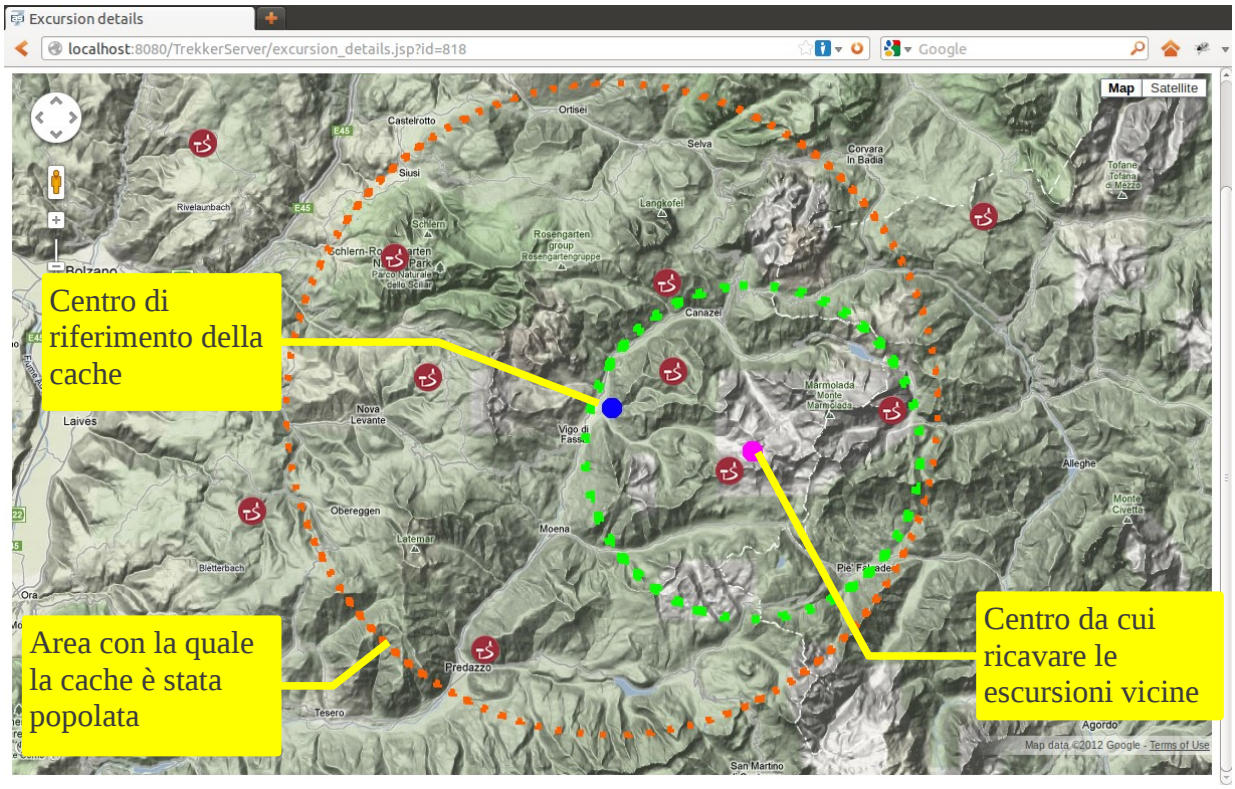

Fig. 4.3: Spostamento del centro di ricerca

Se il centro da cui ricavare le escursioni vicine viene allontanato dal *centro di riferimento della cache* di una distanza tale per cui la nuova area richiesta non è stata precaricata, è necessario ripopolare la cache (Fig. 4.4): il punto richiesto (punto viola) diventerà il nuovo *centro di riferimento della cache*, cioè il centro dell'area (cerchio arancione) con cui la cache stessa verrà ripopolata (Fig. 4.5).

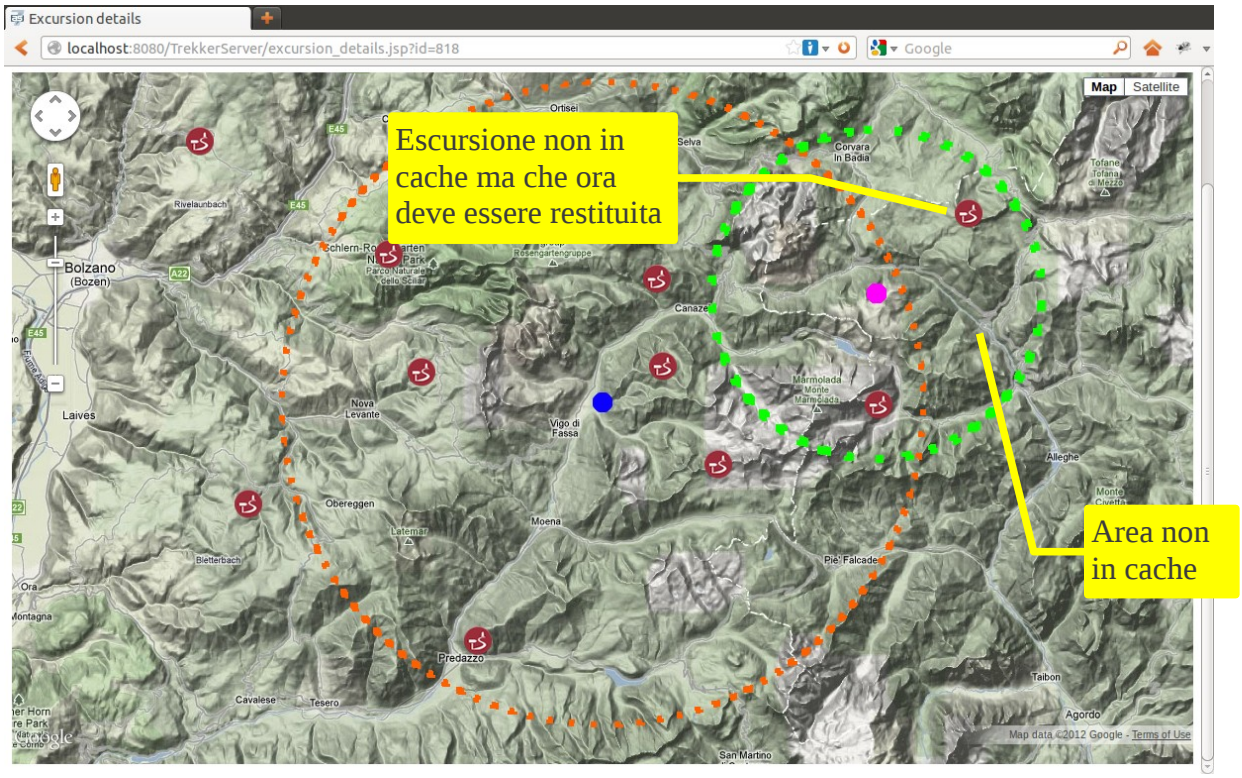

Fig. 4.4: L'area richiesta non è in cache

Fig. 4.5: Nuovo centro di riferimento della cache

All'interno del progetto, la cache è realizzata dalla classe `ExcursionsCache` ed utilizzata dai componenti EJB `ExcursionDAOWithPrefetch` e `CacheLoader` descritti in *Componenti EJB per il prefetching*.

Componenti EJB per il prefetching

I componenti che implementano le funzionalità di prefetching sono realizzati in tecnologia EJB3. Il caricamento della cache secondo le politiche descritte in *Funzionamento della cache* dovrà essere eseguito in modo asincrono per non “bloccare” il richiedente fino alla fine del prefetch.

La classe che realizza il Session Bean da utilizzare nel caso si voglia accedere ai dati con servizio di prefetching è `ExcursionDAOWithPrefetch`. Nel seguito mostriamo come viene configurato sfruttando le annotazioni java.

```
@Stateful
@Local(IExcursionDAOWithPrefetch.class)
public class ExcursionDAOWithPrefetch implements IExcursionDAOWithPrefetch {
	…
}
```

Notiamo la presenza delle annotazioni:

- `@Local`, per indicare che l'accesso al bean sarà locale con le relative conseguenti ottimizzazioni;
- `@Stateful`, per indicare che sarà uno *Stateful Session Bean*, in quanto utilizzerà una cache che verrà riempita a seconda delle richieste successive del cliente, legandolo quindi alla sua sessione.

Il corretto funzionamento di questo bean necessita di altri componenti che vengono inettati utilizzando la dependency injection offerta dal container J2EE:

```
@EJB
IExcursionDAO excursionDAO;

@EJB
ICacheLoader cacheLoader;
```

`IExcursionDAO` è l'interfaccia del Session Bean che implementa il pattern DAO per l'accesso ai dati di un'escursione, mentre `ICacheLoader` è l'interfaccia del Session Bean il cui compito è il caricamento asincrono della cache.

Nel seguito è mostrato il codice che implementa il meccanismo per la restituzione di una

particolare escursione.

```
public Excursion findExcursionById(int id, boolean loadCoordinates) {
	// Retrieve from cache
	Excursion excursion = cache.getExcursionById(id);

	if(excursion == null) {
		// Retrieve from database
		excursion = excursionDAO.findExcursionById(id, loadCoordinates);
		centerOfTheSearch = excursion.getStartPoint();
		cache.invalidateCache();

		/*
		 * With an external stateless session bean annotated with
@Asynchronous,
		 * loadCache() works asynchronously for an external client.
		 */
		cacheLoader.loadCacheWithCloseExcursions(cache, excursion,
SEARCHING_RADIUS/2);
	}
	else {
		if(GPSCoordinate.distanceBetween(centerOfTheSearch,
excursion.getStartPoint()) > SEARCHING_RADIUS/2) {
			cache.invalidateCache();
			cacheLoader.loadCacheWithCloseExcursions(cache, excursion,
SEARCHING_RADIUS/2);
		}
	}

	return excursion;
}
```

In realtà si potrebbe evitare di impiegare un Session Bean asincrono di interfaccia `ICacheLoader`: basterebbe marcare con `@Asynchronous` (disponibile dalla specifica *EJB3.1*) un metodo privato (chiamiamolo `loadCache(...)`) di `ExcursionDAOWithPrefetch` che si occupa di caricare la cache (esattamente quello che fa il Session Bean di interfaccia `ICacheLoader`), e richiamarlo nel seguente modo:

```
ctx.getBusinessObject(IExcursionDAOWithPrefetch.class).loadCache(excursion);
```

dove `ctx` è il contesto di sessione da iniettare con la *dependency injection* (`@Resource SessionContext ctx;`)

La riga di codice appena mostrata andrebbe a sostituire la seguente:

```
cacheLoader.loadCacheWithCloseExcursions(cache, excursion, SEARCHING_RADIUS/2);
```

Il metodo `loadCache(...)` risulterebbe così dichiarato:

```
@Asynchronous
public void loadCache(Excursion excursion) {
	…
}
```

Purtroppo, un bug presente in *JBoss 7.1.1 final* non consente la corretta esecuzione asincrona del metodo `loadCache(...)`, motivo per cui è necessario costruire il nuovo Session Bean di interfaccia `ICacheLoader` mostrato nel seguito.

```
@Stateless
@Local(ICacheLoader.class)
@Asynchronous
public class CacheLoader implements ICacheLoader {
	…
}
```

4.2.3 Componenti per la memorizzazione dei dati ricevuti

La ricezione dei dati inviati dall'utente (via interfaccia web o via smartphone con l'applicazione client descritta nel capitolo 3) è realizzata attraverso un *web service RESTful*. Tale servizio ha il compito di ricevere richieste di tipo HTTP POST e salvarne nel database i dati contenuti in formato json. La presenza del framework *RESTEasy* all'interno dell'application server *JBoss 7.1.1 final* consente la rapidissima realizzazione del nostro servizio.

La classe che implementa il nostro *web service RESTful* è così definita:

```
@Path("/upload")
@ManagedBean
public class UploadJsonREST {
	…
}
```

Notiamo la presenza di due annotazioni:

- `@Path(...)`: indica a quale URI dovrà rispondere il servizio implementato in questa classe;
- `@ManagedBean`: indica al container J2EE che questa classe dovrà essere gestita come un Session Bean, offrendole quindi funzionalità di dependency injection.

Il metodo che risponde alle richieste di tipo HTTP POST è il seguente:

```
@POST
@Path("/json")
public Response uploadJsonFile(String json) {
	IJsonConverter converter = new JsonConverter();
	ExcursionAdapter[] list = converter.Json2ExcursionAdapterArray(json);
```

```
        saveInDatabase(list);

        return Response.ok().build();
}
```

L'annotazione `@POST` indica il tipo di messaggio HTTP cui il metodo deve rispondere; l'annotazione `@Path(...)` indica l'URI a cui il metodo deve rispondere. L'oggetto `converter` di interfaccia `IJsonConverter` fa uso della libreria *Gson* per la trasformazione dei dati dal formato json al mondo java ad oggetti. Il metodo `saveInDatabase(...)` riceve la lista dei dati da salvare nel database ed effettua le opportune operazioni di persistenza.

4.3 Web tier

La parte di presentazione web è realizzata attraverso le Java Server Pages (JSP), in modo da facilitare l'accesso ai Session Bean necessari. Alcune parti delle JSP includono script in linguaggio javascript per consentire l'accesso alle API di Google Maps, le quali facilitano l'interazione con i web services di tipo RESTful di Google stesso.

L'accesso ai Session Bean realizzati precedentemente avviene sempre allo stesso modo in tutte le pagine JSP del tier web, sfruttando un'oggetto factory:

```
DAOFactory daoFactory =
DAOFactory.getDAOFactory(application.getInitParameter("dao"));
IExcursionDAOWithPrefetch excursionDAO = daoFactory.getExcursionDAOWithPrefetch();
IUserDAO userDAO = daoFactory.getUserDAO();
```

Il parametro di contesto `dao` è specificato all'interno del file di configurazione `web.xml`.

4.3.1 Le Google Maps Javascript API

La visualizzazione delle escursioni su una mappa fa uso delle *Google Maps Javascript API v3* che facilitano l'interazione con i web services di tipo RESTful di Google. Tali API sono disponibili per il linguaggio Javascript, quindi è necessario introdurre script di questo tipo all'interno delle pagine JSP con cui il nostro web tier è realizzato.

La mappa di Google va inizializzata nel seguente modo:

```
var startPos = new google.maps.LatLng(<%= startCoord.getLatitude() %>, <%=
startCoord.getLongitude() %>);
var myOptions = {
      zoom : 8,
```

```
        center : startPos,
        mapTypeId : google.maps.MapTypeId.TERRAIN
};

var map = new google.maps.Map(document.getElementById("map_canvas"), myOptions);
```

Questo estratto di codice ci consente di caricare la mappa di Google centrandola alle coordinate definite nell'oggetto `startPos`. La mappa verrà visualizzata in un elemento `div` della pagina denominato `map_canvas`.

```
<div id="map_canvas" style="width:100%; height:100%"></div>
```

Successivamente viene disegnata la linea del tracciato seguito durante l'escursione selezionata:

```
function setPath(map) {
        var pathCoordinates = new google.maps.MVCArray();
        <% for (ExcursionCoordinate loc : locations) { %>
                var location = new google.maps.LatLng(<%=
loc.getCoordinate().getLatitude() %>, <%= loc.getCoordinate().getLongitude() %>);
                pathCoordinates.push(location);
        <% } %>


        var excursionPath = new google.maps.Polyline({
                path : pathCoordinates,
                strokeColor : "#FF0000",
                strokeOpacity : 1.0,
                strokeWeight : 2
        });

        excursionPath.setMap(map);
}
```

Per implementare la visualizzazione delle icone dei POI con possibilità di cliccare su di esse per far apparire un popup di informazioni, è stato usato il seguente codice:

```
function setPoiMarkers(map) {
        var image = 'images/poi_marker.png';
        <%List<Poi> pois = poiDAO.findPoisByExcursion(excursion); %>
        <%for(Poi poi : pois) { %>
                var myLatLng = new google.maps.LatLng(<%=
poi.getLocation().getLatitude() %>, <%= poi.getLocation().getLongitude() %>);
                var contentInfo = "<h1><%= poi.getFriendlyName() %></h1><br />(<%=
poi.getLocation().getLatitude() %>, <%= poi.getLocation().getLongitude() %>)"
                var poiMarker = new google.maps.Marker({
                        position: myLatLng,
                        map: map,
                        clickable: true,
                        icon: image,
                        title: "<%= poi.getFriendlyName() %>",
                        content: contentInfo
                  });
                poiMarker.info = new google.maps.InfoWindow({
                   content: contentInfo
```

```
                });
                google.maps.event.addListener(poiMarker, 'click', function() {
                        poiMarker.info.setContent(this.content);
                        poiMarker.info.open(map, this);
                });

        <%} %>
}
```

Notiamo che si fa uso di oggetti di tipo `Marker` (le icone dei POI da mostrare) e `InfoWindow` (le finestre popup) esposti dalle API di Google, oltre a funzioni di callback che rispondono al “click” del mouse sulle icone (aggiunte tramite `addListener(...)`). Le icone relative alle escursioni vicine vengono disegnate in modo analogo.

Per sfruttare le Google Maps API è sufficiente inserire la seguente riga di codice all'interno della sezione header della pagina html:

```
<script type="text/javascript" src="http://maps.google.com/maps/api/js?
sensor=false"></script>
```

In questo modo le API di Google per l'accesso alle mappe sono disponibili ed utilizzabili.

Per approfondimenti si rimanda al codice sorgente ed alla documentazione delle Google Maps API [GMAPSAPI].

4.3.2 Controllo dell'accesso con JAAS

L'application server *JBoss 7.1.1 final* implementa un meccanismo di controllo degli accessi appoggiandosi al servizio Java *JAAS* [JAAS], il quali realizza una versione Java dello standard *Pluggable Authentication Module* (*PAM*).

Il suo utilizzo è molto semplice. Per proteggere l'accesso a determinate pagine web è sufficiente configurare il server in modo da indicargli quali pagine proteggere. Il seguente codice mostra come fare all'interno del file `web.xml`.

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>Protected</web-resource-name>
                <url-pattern>*.jsp</url-pattern>
        </web-resource-collection>
        <auth-constraint>
                <role-name>viewer</role-name>
        </auth-constraint>
</security-constraint>
```

Nel nostro caso vogliamo proteggere tutte le pagine JSP (`*.jsp`) rendendole accessibili soltanto agli utenti che ricoprono il ruolo di `viewer`.

La modalità che l'utente ha di autenticarsi è attraverso una form html definita nella pagina `login.jsp`, la quale verrà mostrata automaticamente ad ogni utente che chiede l'accesso a qualsiasi pagina protetta. Il seguente codice del file `web.xml` configura questo aspetto.

```
<login-config>
      <auth-method>FORM</auth-method>
      <realm-name>TrekkerRealm</realm-name>
      <form-login-config>
            <form-login-page>/login.jsp</form-login-page>
            <form-error-page>/errorpages/loginerror.html</form-error-page>
      </form-login-config>
 </login-config>
```

La form è definita in modo classico, unico requisito è che il valore dell'attributo `action` venga posto pari a `j_security_check`. La figura 4.6 mostra la pagina di login.

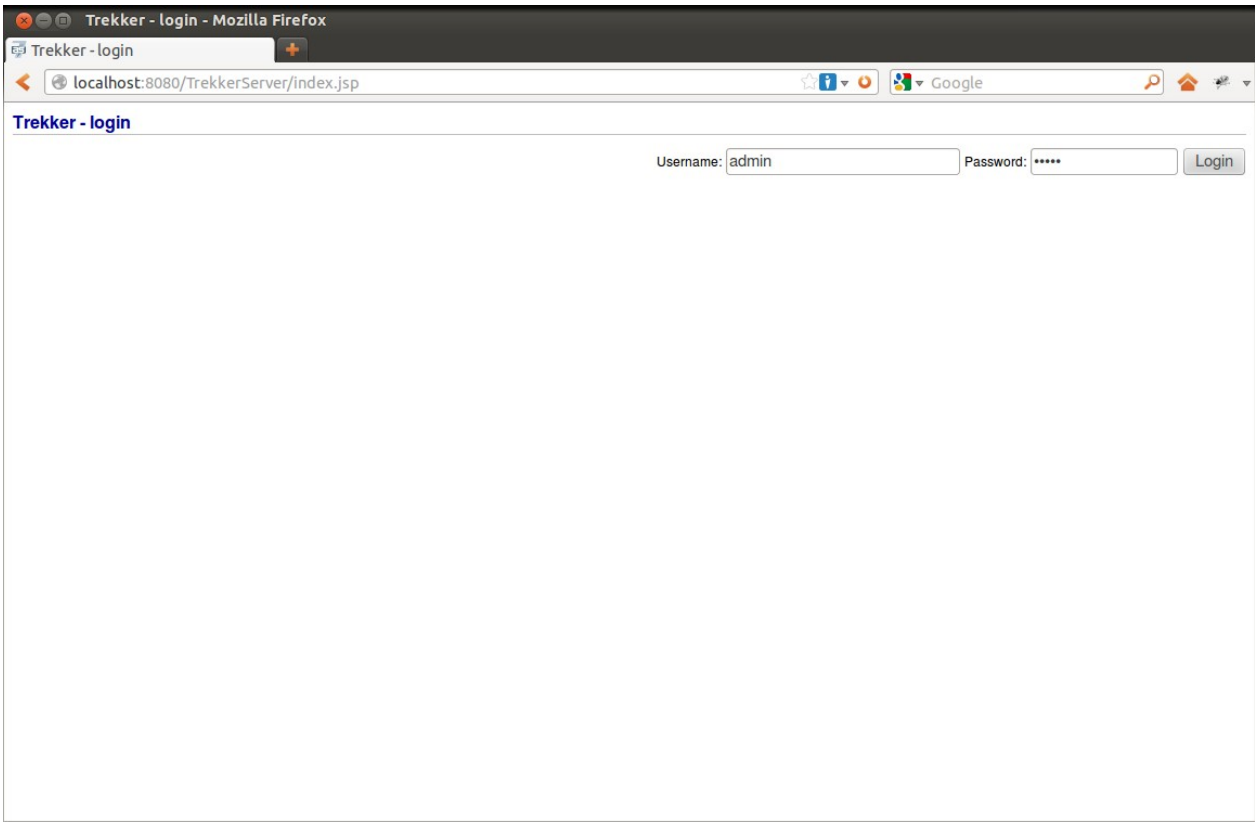

Fig. 4.6: Pagina di login

A questo punto non resta che configurare i Session Bean in modo che rispondano soltanto agli utenti autorizzati. Per fare questo è sufficiente inserire delle annotazioni di classe all'interno

dell'implementazione dei Session Bean del tipo:

```
@SecurityDomain("TrekkerRealm")
@RolesAllowed("viewer")
```

Queste annotazioni definiscono il dominio di protezione ed il ruolo che gli utenti devono ricoprire per poter accedere ai Session Bean.

Ora è necessario configurare *JBoss* perchè possa eseguire correttamente il meccanismo di autenticazione voluto. Per farlo è sufficiente aggiungere il seguente codice al file di configurazione `standalone-full.xml` nella cartella `$JBOSS_HOME/standalone/configuration`, all'interno della sezione `<subsystem xmlns="urn:jboss:domain:security:1.1">`:

```
<security-domain name="TrekkerRealm">
	<authentication>
		<login-module code="Database" flag="required">
			<module-option name="dsJndiName" value="java:/TrekkerDS"/>
			<module-option name="principalsQuery" value="select passwd from
User u where u.username=?"/>
			<module-option name="rolesQuery" value="select role, 'Roles'
from User u where u.username=?"/>
			<module-option name="unauthenticatedIdentity" value="guest"/>
		</login-module>
	</authentication>
</security-domain>
```

Questo codice consente a *JBoss* di reperire le informazioni necessarie al meccanismo di autenticazione direttamente dal database dell'applicazione.

Capitolo 5: Test sperimentali

In questo capitolo vengono presentati i test effettuati sulla parte server di questo progetto per valutarne funzionamento e prestazioni. Il componente dedicato alla generazione automatica delle richieste è *JMeter* [JMET].

I test hanno l'obiettivo di evidenziare il comportamento del server all'interno di due scenari di lavoro:

1. *localhost*: le richieste vengono generate dalla stessa macchina che ospita il server in questione (Fig. 5.1);
2. *LAN*: più realistico, in cui le richieste sono generate da un host (`host2`) diverso da quello che ospita l'applicazione server (`host1`) ed appartenente alla stessa rete locale (Fig. 5.2: il protocollo di comunicazione tra i due host è TCP over IP, quello tra i componenti è HTTP).

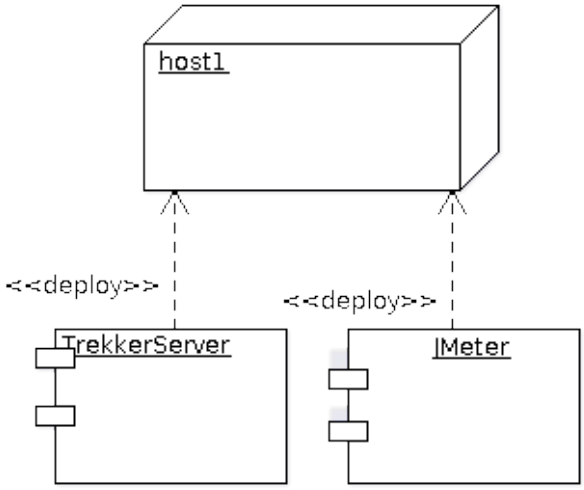

Fig. 5.1: Deployment per il test su localhost

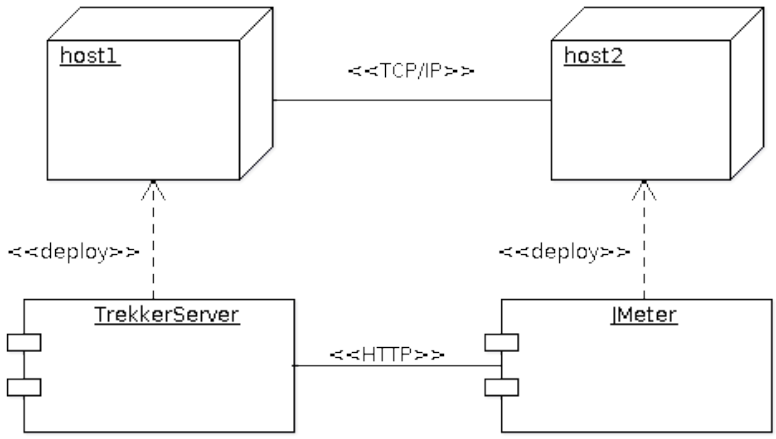

Fig. 5.2: Deployment per il test in rete locale

Tutti i test di entrambi gli scenari sono stati eseguiti successivamente ad un certo numero di richieste dopo l'avvio del server, per dar modo all'applicazione di istanziare tutte le risorse necessarie per rispondere a carichi di lavoro elevati. In altri termini, ogni test di ogni scenario è stato eseguito seguendo i seguenti passaggi:

1. avvio del server;
2. invio di 50 richieste di esempio senza scopo di raccogliere statistiche;
3. avvio del piano di test vero e proprio con raccolta delle statistiche;
4. spegnimento del server.

Il passo 2 consente al server di istanziare un primo pool di risorse necessarie a rispondere alle varie richieste. In questo modo, l'applicazione server, avrà a disposizione (già istanziati) i session beans necessari a soddisfare le richieste inviate al passo 3, dandoci quindi modo di simulare una situazione di lavoro a regime.

5.1 Configurazione del benchmark

Come già detto, i test sono stati effettuati in due differenti scenari. La configurazione delle macchine utilizzate per i test è riportata di seguito.

- La macchina che ospita il server (`host1`) ha le seguenti caratteristiche in entrambi gli scenari:
  - *Tipo*: Notebook
  - *Modello*: HP Pavilion dv5-1210el
  - *CPU*: Intel Pentium Dual-Core T4200, 2.0 GHz, 1MB cache L2, FSB 800 MHz
  - *RAM*: 4096 MB DDR2
  - *Hard Disk*: 250 GB SATA 5400 rpm
  - *Interfacce di rete*: WiFi 802.11b/g, Lan Gigabit Ethernet 10/100/1000Mbps
  - *Sistema operativo*: Ubuntu Linux 12.04 LTS, con desktop environment GNOME senza effetti grafici (per una maggiore "leggerezza")
- La macchina (`host2`) dedicata alla generazione di richieste verso il server nello scenario 2 (*LAN*) ha le seguenti caratteristiche:
  - *Tipo*: Notebook
  - *Modello*: Fujitsu Siemens AMILO PI 1505
  - *CPU*: Intel Core 2 Duo T5500, 1.66 GHz, 2MB cache L2, FSB 667 MHz
  - *RAM*: 1024 MB DDR2 533MHz (2 moduli da 512MB)
  - *Hard Disk*: 120 GB SATA 5400rpm
  - *Interfacce di rete*: 10/100/1000Mbps LAN, Wireless LAN (WLAN) solution 802.11a/b/g
  - *Sistema operativo*: Windows XP SP2
- Il router utilizzato per la realizzazione della rete LAN dello scenario 2 è un router Ethernet/Wi-Fi *Sitecom WL-118* IEEE 802.11a/b/g (max 24 Mbps)
- L'application server che ospita l'applicazione server da noi sviluppata è *JBoss "Brontes" 7.1.1 final* [JBOSS]
- Il software utilizzato per la generazione automatica delle richieste verso il server è Apache *JMeter 2.7* [JMET]

5.1.1 Configurazione del software JMeter

Apache JMeter è un software open source scritto in java per il testing e l'analisi delle performance di svariati servizi, con un orientamento particolare verso i servizi web.

Con JMeter è possibile realizzare piani di test che consentono la generazione di richieste specifiche, la ricezione delle relative risposte e la loro analisi. Per questo motivo si rivela necessario sviluppare un piano di test progettando richieste mirate a testare la funzionalità chiave del nostro server: il prefetch dei dati (paragrafo 4.2.2).

Il test progettato si basa su due tipologie di richieste:

- richiesta di dati sicuramente non precaricati (non presenti in cache)
- richiesta di dati sicuramente precaricati (presenti in cache)

Il nostro scopo in questa sede, è quello di analizzare separatamente e confrontare i tempi di risposta del server a seguito di richieste delle due tipologie sopra indicate. Prerequisito necessario allo svolgimento corretto del test è il popolamento del database delle escursioni con dati adeguati alla realizzazione dei nostri obiettivi: devono essere presenti escursioni geograficamente vicine tra loro (in modo che vengano precaricate in cache) e lontate (in modo da non trovarle già in cache). Il test prevede la generazione di `n` client distribuiti in un intervallo temporale di `m` secondi (il sistema impiega `m` secondi a generare gli `n` client), ognuno dei quali invia quattro richieste al server (due prevedono il ritrovamento dei dati in cache, e due obbligano il server ad una ricerca nel database). Tutte le richieste vengono generate facendo intercorrere, tra una e l'altra, un tempo calcolato in modo casuale e compreso in un intervallo tra 0 e 100 millisecondi (estremi inclusi).

I parametri `n` (numero di client) ed `m` (periodo di creazione degli `n` client) assumeranno valori differenti nei vari test, determinando così diversi carichi di lavoro a cui sottoporre il nostro server. I vari test (sia di scenario 1 che di scenario 2) sono stati effettuati con le seguenti configurazioni di `n` ed `m`:

`n` (numero di client)	**`m` (periodo di generazione degli `n` client) [secondi]**
100	50
100	25
100	10
100	0
200	50

200	25
200	10
200	0

Con queste configurazioni siamo in grado di realizzare test che variano sia nel numero dei client che nel tempo di ricezione delle richieste da parte del server.

La figura Errore: sorgente del riferimento non trovata mostra la struttura del piano di test.

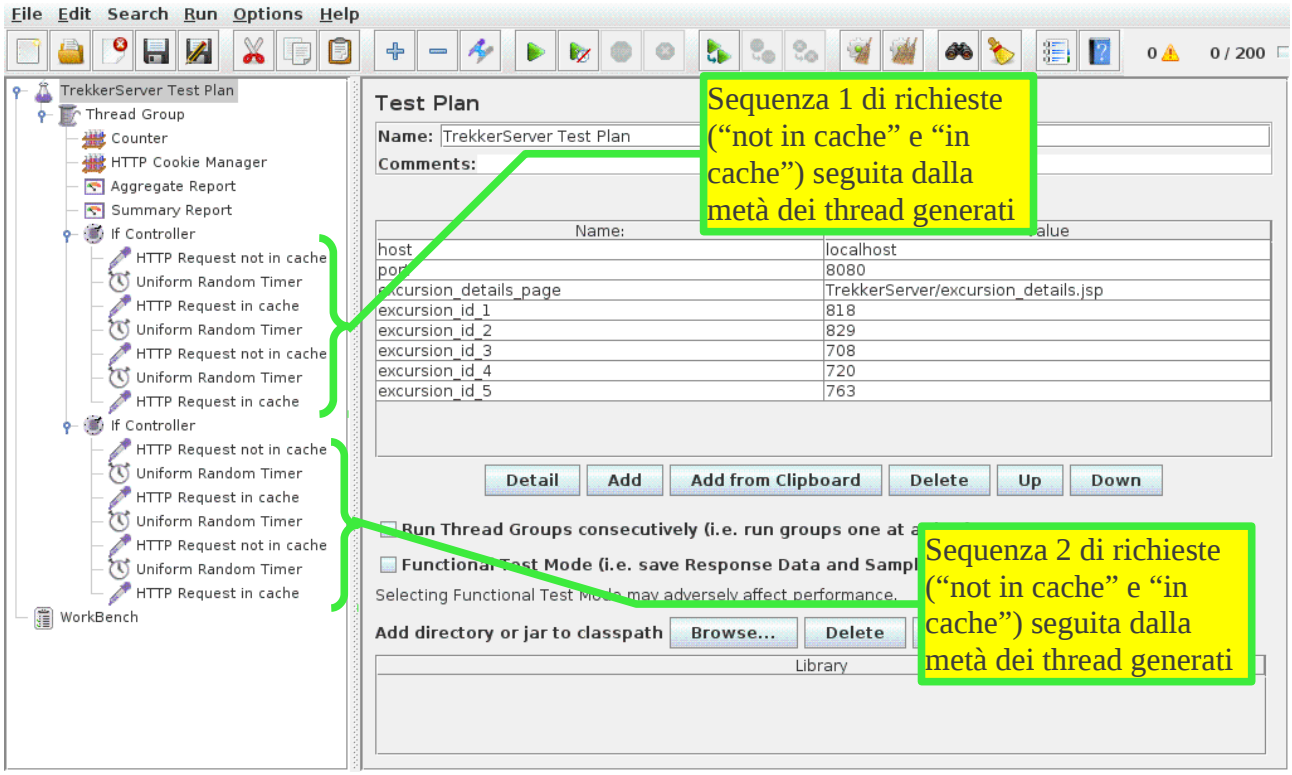

Fig. 5.3: Piano di test

Le sequenze 1 e 2 differiscono per l'ordine di richiesta delle escursioni, consentendo quindi di variare le attività dei thread generati. I thread vengono generati sequenzialmente quindi possono essere numerati con valori consecutivi crescenti: i thread “pari” eseguono la sequenza 1, i thread “dispari” eseguono la sequenza 2.

5.1.2 Configurazione del server

Ogni test è stato effettuato con due configurazioni differenti del server in modo da evidenziare il funzionamento del meccanismo di prefetch:

1. prefetch disabilitato
2. prefetch abilitato

La configurazione del server avviene in maniera immediata: è sufficiente modificare il parametro di contesto `cache_enabled` (con valore `true` o `false`) all'interno del file di deployment `web.xml`, situato in `$JBOSS_HOME/standalone/deployments/TrekkerServer.ear/TrekkerServer.war/WEB-INF`.

Per avere la cache abilitata e quindi il prefetch attivo:

```
<context-param>
	<param-name>cache_enabled</param-name>
	<param-value>true</param-value>
</context-param>
```

Per avere la cache disabilitata e quindi il prefetch non attivo:

```
<context-param>
	<param-name>cache_enabled</param-name>
	<param-value>false</param-value>
</context-param>
```

5.2 Esecuzione ed analisi dei test

In questo paragrafo analizziamo i risultati prodotti dai test eseguiti con le configurazioni di client e server descritti nel paragrafo precedente, sia all'interno dello scanario 1 (*localhost*) che dello scenario 2 (LAN).

5.2.1 Interpretazione dei risultati

I risultati dei test sono riportati in tabelle composte da diverse righe e colonne. Ogni riga si riferisce ad una particolare tipologia di richiesta generata, ogni colonna contiene il valore di un determinato parametro statistico di interesse. Le colonne riportate in ogni tabella sono le seguenti:

- **Request type**: tipo di richiesta; i valori possibili sono:
  - *HTTP Request not in cache*: richiesta di dati che sicuramente non sono in cache;
  - *HTTP Request in cache*: richiesta di dati che, nel caso in cui il server abbia la cache abilitata (prefetch attivo), sono sicuramente in cache; se la cache è disabilitata i dati non sono, ovviamente, in cache;
  - *TOTAL*: totale delle richieste generate, senza distinzione della tipologia; riassume i

risultati totali di ogni tipologia di richiesta.

- **Count**: numero di richieste del tipo specificato;
- **Average**: tempo medio di risposta dal server (in millisecondi);
- **Std. Dev**: deviazione standard del tempo di risposta (in millisecondi); stima la variabilità dei dati raccolti;
- **Median**: mediana del tempo di risposta dal server (in millisecondi), indica cioè il tempo di risposta massimo ottenuto dal 50% delle richieste;
- **90% Line**: 90° percentile del tempo di risposta dal server (in millisecondi), indica cioè il tempo di risposta massimo ottenuto dal 90% delle richieste;
- **Min**: tempo minimo di risposta ottenuto dalle richieste (in millisecondi);
- **Max**: tempo massimo di risposta ottenuto dalle richieste (in millisecondi);
- **Error %**: percentuale di errori ottenuti (risposte non ricevute o risposte di errore);
- **Throughput**: numero di richieste servite dal server in un dato intervallo temporale (numero di richieste per secondo);

Il valore medio del tempo di risposta (Average) è da considerare con il giusto peso, in quanto molto influenzato da ritardi di consegna dei messaggi dovuti alla rete. Meno soggetti ad occasionali ritardi dovuti alla rete, e quindi più attendibili rispetto alla media, sono la mediana (Median) ed il 90° percentile (90% Line).

5.2.2 Scenario 1: *localhost*

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	32	6,05	31	39	25	60	0.0	4,01
HTTP Request in cache	200	32	8,35	30	38	25	104	0.0	4,01
TOTAL	400	32	7,30	30	39	25	104	0.0	8,00

Tabella 5.1: Scenario 1 (localhost), n=100, m=50, cache disabilitata

Il primo test che abbiamo effettuato prevede un basso carico di lavoro da parte del server. Le richieste generate provengono da `n`=100 client distribuiti in un intervallo temporale di `m`=50 secondi (in sostanza viene generato un client al secondo, il quale invia quattro richieste HTTP GET: due richiedono dati in cache, due richiedono dati non in cache). Il test è stato eseguito prima con prefetch disabilitato (e quindi cache disabilitata) (Tabella 5.1), poi con prefetch abilitato (e quindi

cache abilitata) (Tabella 5.2).

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	31	6,87	29	41	23	59	0.0	4,00
HTTP Request in cache	200	18	5,12	18	23	12	47	0.0	4,01
TOTAL	400	25	8,90	25	38	12	59	0.0	7,99

Tabella 5.2: Scenario 1 (localhost), n=100, m=50, cache abilitata

Notiamo la differenza dei tempi di risposta tra il caso di cache disabilitata ed il caso di cache abilitata. Media, mediana e 90° percentile evidenziano tempi di risposta minori per le richieste di dati in cache (HTTP Request in cache) rispetto a richieste di dati non in cache (HTTP Request not in cache) nel caso di meccanismo di prefetch abilitato.

Provando a diminuire la dimensione dell'intervallo di tempo in cui vengono generate le richieste (`m`=25 secondi, `n`=100 client) otteniamo i seguenti risultati riportati in tabella, prima nel caso di cache disabilitato (Tabella 5.3) e poi nel caso di cache abilitata (Tabella 5.4).

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	37	11,42	33	56	25	92	0.0	7,92
HTTP Request in cache	200	37	12,27	33	54	25	106	0.0	7,91
TOTAL	400	37	11,86	33	55	25	106	0.0	15,75

Tabella 5.3: Scenario 1 (localhost), n=100, m=25, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	37	10,40	35	54	23	67	0.0	7,90
HTTP Request in cache	200	22	7,62	19	33	13	52	0.0	7,91
TOTAL	400	29	11,88	27	46	13	67	0.0	15,71

Tabella 5.4: Scenario 1 (localhost), n=100, m=25, cache abilitata

Anche in questo caso possiamo concentrarci principalmente su mediana e 90° percentile, notando che, nel caso di prefetch disabilitato, i tempi di risposta per richieste in cache e fuori cache si equivalgono, mentre diminuiscono per richieste in cache nel caso di prefetch abilitato.

Mostriamo ora i risultati raccolti dai test effettuati con i parametri così configurati: `n`=100, `m`=10.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	81	36,56	75	137	26	182	0.0	19,09
HTTP Request in cache	200	78	36,04	77	127	26	179	0.0	18,91
TOTAL	400	80	36,33	76	132	26	182	0.0	37,30

Tabella 5.5: Scenario 1 (localhost), n=100, m=10, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	344	150,35	344	535	77	851	0.0	18,67
HTTP Request in cache	200	213	109,15	201	352	28	533	0.0	18,71
TOTAL	400	279	146,71	253	485	28	851	0.0	36,63

Tabella 5.6: Scenario 1 (localhost), n=100, m=10, cache abilitata

Le considerazioni sono analoghe alle precedenti, possiamo cioè notare una maggiore rapidità di risposta a richieste di dati in cache nel caso di prefetch abilitato.

L'ultimo test effettuato mantiene fisso il numero di client (`n`=100) e prevede la generazione di tutte le richieste in un intervallo temporale limitatissimo (`m`=0), ovvero i client vengono creati contemporaneamente e le richieste sono inviate attendendo soltanto un tempo random tra 0 e 100 millisecondi tra una e l'altra.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	1781	278,54	1849	2061	39	2274	0.0	28,86
HTTP Request in cache	200	1885	150,99	1905	2062	1461	2257	0.0	23,85
TOTAL	400	1833	230,02	1890	2061	39	2274	0.0	46,33

Tabella 5.7: Scenario 1 (localhost), n=100, m=0, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	2960	728,24	2769	4116	1354	4613	0.0	18,09
HTTP Request in cache	200	2369	639,20	2616	2871	125	3357	0.0	18,62
TOTAL	400	2665	746,19	2685	3742	125	4613	0.0	31,75

Tabella 5.8: Scenario 1 (localhost), n=100, m=0, cache abilitata

Anche in questo caso, notiamo un miglioramento nei tempi di risposta quando il prefetch è abilitato. Questo comportamento è evidenziato soprattutto dal 90° percentile (90% Line).

I prossimi risultati mostrano il comportamento del server ai test che generano richieste da parte di `n`=200 client, con parametro `m` (intervallo di tempo in cui viene distribuita la creazione dei client) che varia da 50 a 0 millisecondi.

Il primo test prevede un intervallo di tempo pari a `m`=50 millisecondi (Tabella 5.9, Tabella 5.10).

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	36	10,19	33	52	25	87	0.0	7,94
HTTP Request in cache	400	38	12,79	34	54	25	123	0.0	7,94
TOTAL	800	37	11,59	33	53	25	123	0.0	15,79

Tabella 5.9: Scenario 1 (localhost), n=200, m=50, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	37	12,91	34	54	21	96	0.0	7,95
HTTP Request in cache	400	22	10,19	19	33	12	101	0.0	7,95
TOTAL	800	29	14,02	27	48	12	101	0.0	15,85

Tabella 5.10: Scenario 1 (localhost), n=200, m=50, cache abilitata

Anche raddoppiando il numero di client rispetto ai test precedenti, i test mettono in evidenza la maggiore rapidità di risposta a richieste di dati in cache quando il prefetch è abilitato.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	54	22,14	51	87	25	133	0.0	15,75
HTTP Request in cache	400	57	24,30	51	91	25	144	0.0	15,78
TOTAL	800	55	23,28	51	89	25	144	0.0	31,34

Tabella 5.11: Scenario 1 (localhost), n=200, m=25, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	50	21,77	46	82	22	160	0.0	15,74
HTTP Request in cache	400	29	15,77	25	49	12	145	0.0	15,73
TOTAL	800	40	21,75	36	69	12	160	0.0	31,26

Tabella 5.12: Scenario 1 (localhost), n=200, m=25, cache abilitata

Le tabelle sottostanti riportano i risultati dei test con `n`=200 ed `m`=10. Valgono le stesse considerazioni dei casi precedenti.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	1858	871,38	1959	2997	37	3299	0.0	23,09
HTTP Request in cache	400	1907	816,50	1970	2999	29	3264	0.0	22,08
TOTAL	800	1883	844,73	1967	2998	29	3299	0.0	43,84

Tabella 5.13: Scenario 1 (localhost), n=200, m=10, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	1322	634,82	1362	2068	32	2396	0.0	27,06
HTTP Request in cache	400	1043	570,10	1051	1746	12	1896	0.0	26,79
TOTAL	800	1182	619,32	1235	1965	12	2396	0.0	53,22

Tabella 5.14: Scenario 1 (localhost), n=200, m=10, cache abilitata

I test successivi (Tabella 5.15, Tabella 5.16) sono stati realizzati impostando n=200 ed m=0. Notiamo un leggero degrado delle prestazioni rispetto ai casi precedenti: il throughput si abbassa ed i tempi di risposta si alzano andandosi a triplicare, pur mantenendosi leggermente più bassi per le richieste di dati in cache quando il prefetch è abilitato.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	4134	566,65	4208	4676	1024	5331	0.0	25,65
HTTP Request in cache	400	4294	386,81	4251	4827	3269	5269	0.0	22,47
TOTAL	800	4214	491,69	4222	4781	1024	5331	0.0	42,14

Tabella 5.15: Scenario 1 (localhost), n=200, m=0, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	5166	1.457,33	4891	6888	270	7269	0.0	21,52
HTTP Request in cache	400	4631	884,85	4642	5679	503	7078	0.0	18,38
TOTAL	800	4898	1.234,80	4721	6654	270	7269	0.0	35,99

Tabella 5.16: Scenario 1 (localhost), n=200, m=0, cache abilitata

Da tutti i test precedenti emerge un dato interessante. Possiamo notare che la deviazione standard è sempre più bassa per richieste di dati in cache quando il prefetch è abilitato rispetto agli altri casi. Questo ci suggerisce una minore variabilità dei tempi di risposta raccolti, spiegabile con il fatto che i dati richiesti si trovano già in cache e quindi la costruzione dei risultati non è soggetta a ritardi (che possono essere molto variabili) dovuti all'accesso al database.

5.2.3 Scenario 2: *LAN*

Le configurazioni dei test effettuati nello scenario 2, cioè con macchina client diversa da quella che ospita il server ed entrambe inserite nella stessa rete locale, seguono lo stesso ordine di quelli svolti nello scenario 1. Di seguito riportiamo le tabelle contenenti i risultati raccolti durante i test, mantenendo valide le stesse considerazioni ricavate dallo scenario 1.

Test con `n`=100, `m`=50.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	118	71,29	95	193	68	517	0.0	3,98
HTTP Request in cache	200	124	78,32	95	192	71	476	0.0	4,02
TOTAL	400	121	74,95	95	193	68	517	0.0	7,92

Tabella 5.17: Scenario 2 (LAN), n=100, m=50, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	40	31,81	33	45	26	254	0.0	4,00
HTTP Request in cache	200	31	32,85	24	32	18	267	0.0	3,99
TOTAL	400	36	32,64	30	42	18	267	0.0	7,95

Tabella 5.18: Scenario 2 (LAN), n=100, m=50, cache abilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	116	97,32	94	157	67	1071	0.0	7,89
HTTP Request in cache	200	109	35,14	97	161	71	243	0.0	8,21
TOTAL	400	113	73,24	95	161	67	1071	0.0	15,62

Tabella 5.19: Scenario 2 (LAN), n=100, m=25, cache disabilitata

Test con `n`=100, `m`=25.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	36	9,90	33	49	25	102	0.0	7,94
HTTP Request in cache	200	26	8,10	23	35	18	77	0.0	7,93
TOTAL	400	31	10,47	30	44	18	102	0.0	15,78

Tabella 5.20: Scenario 2 (LAN), n=100, m=25, cache abilitata

Test con `n`=100, `m`=10.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	51	19,74	46	80	26	112	0.0	19,36
HTTP Request in cache	200	32	13,12	28	53	17	87	0.0	19,53
TOTAL	400	42	19,27	37	70	17	112	0.0	38,46

Tabella 5.22: Scenario 2 (LAN), n=100, m=10, cache abilitata

Test con `n`=100, `m`=0.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	2959	616,28	3126	3581	147	3827	0.0	17,58
HTTP Request in cache	200	2620	1.050,29	2643	3657	74	4449	0.0	15,06
TOTAL	400	2790	877,58	2981	3597	74	4449	0.0	28,69

Tabella 5.23: Scenario 2 (LAN), n=100, m=0, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	200	1816	586,14	1861	2592	50	2899	0.0	26,81
HTTP Request in cache	200	1469	526,74	1628	2077	182	2412	0.0	24,03
TOTAL	400	1642	583,53	1727	2254	50	2899	0.0	46,94

Tabella 5.24: Scenario 2 (LAN), n=100, m=0, cache abilitata

Test con `n`=200, `m`=50.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	102	69,48	80	136	65	609	0.0	7,93
HTTP Request in cache	400	106	77,93	83	155	69	907	0.0	7,92
TOTAL	800	104	73,87	81	143	65	907	0.0	15,76

Tabella 5.25: Scenario 2 (LAN), n=200, m=50, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	85	47,84	70	124	54	599	0.0	7,87
HTTP Request in cache	400	70	35,53	57	98	48	364	0.0	7,95
TOTAL	800	78	42,81	65	116	48	599	0.0	15,68

Tabella 5.26: Scenario 2 (LAN), n=200, m=50, cache abilitata

Test con `n`=200, `m`=25.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	77	37,52	70	128	31	294	0.0	15,66
HTTP Request in cache	400	81	37,86	75	134	32	225	0.0	15,67
TOTAL	800	79	37,73	72	131	31	294	0.0	31,08

Tabella 5.27: Scenario 2 (LAN), n=200, m=25, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	88	89,22	78	134	25	1248	0.0	15,00
HTTP Request in cache	400	78	114,26	63	118	19	1257	0.0	14,96
TOTAL	800	83	102,63	70	129	19	1257	0.0	29,73

Tabella 5.28: Scenario 2 (LAN), n=200, m=25, cache abilitata

Test con `n`=200, `m`=10.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	2247	1781,84	1881	4636	36	9467	0.0	19,49
HTTP Request in cache	400	2410	1791,45	2099	5136	47	7840	0.0	19,46
TOTAL	800	2328	1788,51	1970	4745	36	9467	0.0	38,49

Tabella 5.29: Scenario 2 (LAN), n=200, m=10, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	465	195,77	476	711	31	914	0.0	34,59
HTTP Request in cache	400	326	146,67	323	515	24	730	0.0	34,34
TOTAL	800	395	186,40	377	659	24	914	0.0	67,69

Tabella 5.30: Scenario 2 (LAN), n=200, m=10, cache abilitata

Test con `n`=200, `m`=0.

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	3568	2257,45	3048	6777	73	9966	0	21,62
HTTP Request in cache	400	3346	2279,42	3082	6355	76	10221	0	22,38
TOTAL	800	3457	2271,17	3058	6606	73	10221	0	42,49

Fig. 5.4: Scenario 2 (LAN), n=200, m=0, cache disabilitata

Request type	Count	Average	Std. Dev.	Median	90% Line	Min	Max	Error %	Throughput
HTTP Request not in cache	400	3667	1170,00	3583	5493	55	6507	0.0	26,06
HTTP Request in cache	400	3021	995,56	3023	4208	258	7200	0.0	24,29
TOTAL	800	3344	1133,24	3333	4784	55	7200	0.0	48,02

Tabella 5.31: Scenario 2 (LAN), n=200, m=0, cache abilitata

In questo secondo scenario notiamo che, quando il prefetch è disabilitato, il server sembra più lento a rispondere, sia per richieste di dati in cache che fuori cache, rispetto al caso con prefetch abilitato.

Capitolo 6: Conclusioni

Il progetto realizzato in questa sede ha consentito l'approfondimento di una moltitudine di tecnologie in ambito mobile e a livello enterprise. Tali tecnologie sono piuttosto mature e destinate a crescere ulteriormente, grazie al successo riscontrato e alle esigenze sempre maggiori degli sviluppatori.

I capitoli raccolti in questo documento mostrano una panoramica ed un possibile utilizzo degli strumenti a supporto delle sviluppo di applicazioni mobili ed enterprise, ponendo l'accento su prodotti come Android e JBoss. Il capitolo 3 ha mostrato un esempio di applicazione in grado di sfruttare diverse funzionalità dei moderni smartphone, sviluppata su piattaforma Android. Il capitolo 4 ha messo in luce alcuni aspetti della tecnologia Java EE, mostrando un esempio di implementazione di un'applicazione enterprise con architettura 4-tier in cui persistenza, Session Beans e RESTful Web Services sono al centro dell'attenzione.

Indice delle figure

Indice delle tabelle

Fonti

CDD: Google Inc, Android Compatibility, http://source.android.com/compatibility/index.html

LOCAPI: Google Inc, Location Strategies, http://developer.android.com/guide/topics/location/strategies.html

RESTTH: Roy Thomas Fielding, Architectural Styles andthe Design of Network-based Software Architectures, , , http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

JAXRS: Oracle, JSR-000311 JAX-RS: The JavaTM API for RESTful Web Services 1.0 Final Release , http://download.oracle.com/otndocs/jcp/jaxrs-1.0-fr-eval-oth-JSpec/

RESTEASY: JBoss, RESTEasy, http://www.jboss.org/resteasy

JBOSS: JBoss, JBoss AS 7, http://www.jboss.org/as7

AND: Google Inc, Android, http://www.android.com/

GMAP: Google Inc, Google APIs Add-On, https://developers.google.com/android/add-ons/google-apis/?hl=it

GSON: Google Inc, Google Gson, http://code.google.com/p/google-gson/

CLING: 4th Line.org, Cling - Java/Android UPnP library and tools , http://4thline.org/projects/cling/

WIFID: WiFi Alliance, WiFi Direct, http://www.wi-fi.org/discover-and-learn

DOCCL: 4th Line.org, Cling Core Documentation, http://4thline.org/projects/cling/core/manual/cling-core-manual.html

STEPC: Ms. Najme Zehra Naqvi, Dr. Ashwani Kumar, Aanchal Chauhan, Kritka Sahni, Step Counting Using Smartphone-Based Accelerometer, International Journal on Computer Science and Engineering (IJCSE), 2012, Vol. 4, N. 5 (Maggio), pag. 675 - 681

HTTPCOM: Apache Software Foundation, HttpComponents, http://hc.apache.org/

GMAPSAPI: Google Inc, Google Maps JavaScript API v3, https://developers.google.com/maps/documentation/javascript/

JAAS: Oracle, Java Authentication and Authorization Service (JAAS), http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html

JMET: Apache, JMeter, http://jmeter.apache.org/