ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria Elettronica, Informatica e Telecomunicazioni

Titolo dell'elaborato:

Impiego combinato di GPS, BLE e riconoscimento di immagini per individuare entità nella realtà aumentata

Elaborato in:

Fondamenti di Informatica A

Relatore:

Prof. Mirko Viroli

Presentata da:

Giovanni Ciatto

Correlatore:

Prof. Alessandro Ricci

Sessione II

Anno Accademico 2013/2014

Ai miei genitori,
che mi hanno permesso
di arrivare a questo traguardo.
Ai miei amici e mia sorella
che per esso hanno
sopportato la mia assenza.

Indice

Capitolo 1

Introduzione

Viviamo in un'epoca in cui i dispositivi che utilizziamo ogni giorno sono in grado di assimilare informazioni in una varietà di maniere ancora non del tutto esplorata. Non solo il "cosa" ma anche il "dove" assume una rilevanza non più trascurabile: il dispositivo diventa spesso "punto di accesso" per l'utilizzatore nei confronti dei servizi utilizzati e viceversa, e sempre di più questi ultimi richiedono l'accesso alla posizione con l'intento di ampliare la propria comprensione del contesto dell'utente con lo scopo di un proprio raffinamento. E se il "dove" fosse l'informazione principale?

Visto il sempre crescente numero di tecnologie con cui i produttori equipaggiano i nostri device di uso quotidiano, l'ipotesi di sfruttarne più di una per fornire a questi ultimi la coscienza della propria posizione e quindi del contesto in cui si trovano acquista concretezza e fattibilità. Se poi si pensa che ormai quasi ogni macchina è capace di accedere alla rete e/o interagire con altre anche diverse, gli scenari immaginabili diventano tantissimi. Per fare un esempio banale lo smarrimento del cellulare potrebbe destare molte meno preoccupazioni se fossimo in grado, tramite un altro, di individuarlo con esattezza e/o essere assistiti nella sua ricerca: il dispositivo smarrito potrebbe comunicare la propria

posizione a quello con il quale si effettua la ricerca o, in alternativa, la propria prossimità ad altri punti di riferimento. Si potrebbe pensare di marchiare, in una città turistica, i luoghi di interesse cosicché i curiosi possano, ad esempio inquadrando una parte della città con la fotocamera, avere un'idea di quanti e quali luoghi visitare, ricevere un'anteprima delle informazioni utili, come quelle sul sito, distanza, posizione relativa etc, essere guidati dal dispositivo stesso verso il sito selezionato partendo dal luogo in cui ci si trova e con un mappa bidimensionale simile a quella di Google Maps e con l'aggiunta di indicazioni sulla porzione di mondo inquadrata. Si potrebbe uscire in gruppo in luoghi affollati disperdendosi a cuor leggero consci del fatto che, telefono alla mano, sia sempre possibile, inquadrando la folla, individuare gli altri componenti del gruppo; quest'ultimo scenario acquisisce ulteriore interesse nel caso di un'uscita familiare: uno strumento in più per fornire ai genitori maggiore controllo dei bambini. In situazioni catastrofiche, con molti feriti distribuiti su un'area di dimensioni non troppo estese, si potrebbe pensare di marchiare ognuno con un device che ne monitori posizione e stato di salute, cosicché gli addetti al soccorso, muniti ad esempio di dispositivi come i Google Glass, possano avere costantemente una rapida idea della dislocazione dei feriti da lontano e un dettaglio dei parametri vitali da vicino, in ogni caso mantenendo le mani libere.

Tutti gli scenari descritti hanno in comune il concetto di realtà aumentata: il dispositivo fornito di videocamera guarda il mondo ma da questo non carpisce mere immagini bensì molte altre informazini con le quali «aumenta» i contenuti mostrati all'utente. In comune è anche il concetto di posizionamento: stando sul pianeta, qualunque oggetto è individuabile da una terna di coordinate, od anche, in maniera più o meno precisa, da un riferimento a qualcos'altro di cui sia nota o conoscibile la posizione. Lo scopo della tesi è sondare quanto sia utile combinare le tre tecnologie citate nel titolo, GPS, BLE e riconoscimento di immagini per fornire un'informazione continua sulla posizione e sufficientemente precisa da permettere l'impiego della realtà aumentata, con particolare

attenzione alla seconda, il BLE, nella fattispecie i bluetooth beacon, tecnologia emergente le cui potenzialità non sono ancora state approfondite fino in fondo. Gli interrogativi a cui si vuole contribuire a dare una risposta sono:

- quanto è preciso e fino a che punto è effettivamente utile il GPS?
- quanto è efficace e a partire da che punto è effettivamente utile il riconoscimento di immagini?
- è possibile stimare una posizione impiegando i bluetooth beacon?
- è possibile impiegarli in qualche maniera per migliorare l'informazione del GPS?

Si parla di «miglioramento» in quanto tutte e tre le tecnologie hanno una sorta di «raggio d'azione»: il GPS funziona bene da lontano, in quanto la posizione fornita è affetta da un errore di almeno 5-10 metri, il BLE viene percepito fino a 40-50 metri di distanza e permette di avere una stima accettabile della distanza fino a 15-20 metri, il riconoscimento di immagini rende bene nelle distanze molto brevi come 2-3 metri: i bluetooth beacon possono essere utili in quella fascia di distanze interdetta alle altre due tecnologie? E se sì, in che modo e con che accuratezza?

Per cercare di far chiarezza sui dubbi appena esposti si è sviluppato un sistema distribuito in cui le entità in gioco sono composte da più dispositivi e possono giocare il ruolo di osservatori ed osservabili. L'interazione è mediata da un server che funge da base di dati. Ogni entità si occupa ricavare la propria posizione eventualmente combinando GPS e BLE e la posizione delle altre dal server. Inoltre, se un'entità capisce di essere in prossimità di un'altra, avvia una scansione della realtà alla ricerca di un'immagine che identifichi la seconda.

Nello sviluppo, pur avendo scelto come sistema target Android e pur essendo il sistema nato per il supporto alla sperimentazione, si è cercato comunque di

dare un certo rigore alle fasi di progettazione ed implementazione in vista di un potenziale riuso per fini analoghi e/o mantenimento del software; in particolar modo l'impiego nello specifico di GPS e BLE è visto sin da subito come dettaglio implementativo: si parla spesso, più che altro, di posizioni assolute (quelle fornite, ad esempio, dal GPS) e distanze relative (dal BLE), se un domani dovessero essere disponibili altre tecnologie equivalenti o migliori, il passaggio dovrebbe essere abbastanza indolore. Per quanto riguarda il riconoscimento di immagini ci si è affidati ai codici QR per motivazioni sia tecniche che pratiche spiegate nel seguito. Gli esperimenti svolti tramite l'infrastruttura realizzata hanno verificato che i limiti nominali di GPS e riconoscimento di QR siano poi quelli effettivi: nei casi migliori l'errore era di «appena» 5 metri ed era possibile riconoscere un codice stampato a partire dai 2 metri. Il bluetooth è utile per guidare l'osservatore verso il target fornendogli la nozione di avvicinamento / allontanamento ma, a causa principalmente dell'instabilità del segnale, fornisce distanze dedotte con valori altalenanti, il che rende non troppo precisa la stima della posizone, cosa molto evidente nella realtà aumentata. Questo fatto non deve però scoraggiare: il bluetooth è comunque in grado di fornire una posizione, seppur un poco imprecisa, e potrebbe ugualmente essere d'ausilio in assenza di GPS e/o in quelle situazioni in cui un piccolo errore nella localizzazione non sia evidente come nella realtà aumentata.

1.1 Panoramica dei contenuti

1.1.1 Capitolo 2

Nel capitolo 2 vengono descritte le tecnologie su cui si andrà a lavorare, ovvero GPS, BLE e AR, con brevi accenni alla storia ed alle caratteristiche e, successivamente, le librerie impiegate. In particolare sono descritti i dettagli tecnici dei bluetooth beacon e la struttura dell'advertisement.

1.1.2 Capitolo 3

Nel capitolo 3 vengono descritti i requisiti dell'infrastruttura che si andrà a sviluppare, successivamente vengono analizzati estrapolando requisiti espliciti ed impliciti, come ad esempio la necessità di un server. Alla luce di ciò, viene fissato un glossario che identifica i termini che verrano usati nel seguito e la loro definizione, ovvero la maniera in cui lo scrivente intende ognuno. Segue una visione a grandi linee del sistema da costruire che sarà il filo conduttore di progettazione e sviluppo.

1.1.3 Capitolo 4

Il capitolo 4 descrive l'infrastruttura sviluppata. In primis è spiegato il dominio ovvero l'insieme di interfacce e concetti sui quali poi si andrà a costruire tutto il resto. La sezione seguente riguarda la parte del progetto platform-independent: viene spiegato come le principali funzionalità del sistema siano affidate a dei componenti chiamati «manager» che a loro volta delegano i compiti platform-dependent ai cosiddetti «handler». Questi ultimi sono descritti successivamente nella sezione apposita, insieme agli altri dettagli specifici di Android. È presente anche una sezione relativa al layer di presentazione ed una che descrive come interfacciarsi al server cerntrale. Infine viene spiegata l'interfaccia del prototipo, ovvero l'App Android utilizzata negli esperimenti.

1.1.4 Capitolo 5

Nell'ultimo capitolo vengono presentate le conclusioni raggiunte al termine del lavoro svolto, pregi e difetti del sistema sviluppato e gli eventuali sviluppi futuri riguardo l'infrastruttura. Si cerca inoltre di dare una risposta agli interrogativi posti nell'introduzione.

Capitolo 2

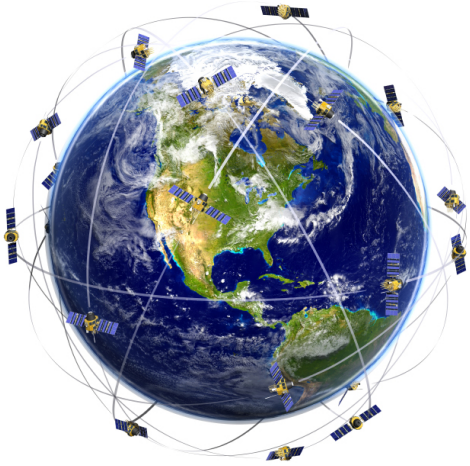
Le tecnologie

2.1 GPS - Global Positioning System

Il GPS[28], sigla che sta per Global Positioning System, è un sistema di posizionamento satellitare sviluppato dal Dipartimento della Difesa Statunitense per scopi civili e militari. Esso sfrutta una rete di satelliti artificiali dedicata capace di fornire ad un ricevitore informazioni sulla propria posizione geografica, indipendentemente dalle condizioni meteorologiche e dall'ubicazione di quest'ultimo a patto che esista un percorso diretto privo di ostacoli tra esso ed almeno quattro satelliti. Il ricevitore calcola la propria distanza dai satelliti deducendola dal ritardo di propagazione del segnale radio da essi inviato: note almeno quattro distanze è possibile multilaterare senza ambiguità ottenendo così una stima tanto più accurata quanto migliore è la conoscenza della distanza.[31]

Nei moderni devices si sfrutta un sistema di geolocalizzazione più complesso: l'AGPS, Assisted GPS. Il limite del GPS ordinario è il tempo di setup: è necessario, al primo avvio, caricare la lista dei satelliti visibili in quel momento nell'

Figura 2.1: Rappresentazione del sistema di satelliti dedicati del GPS



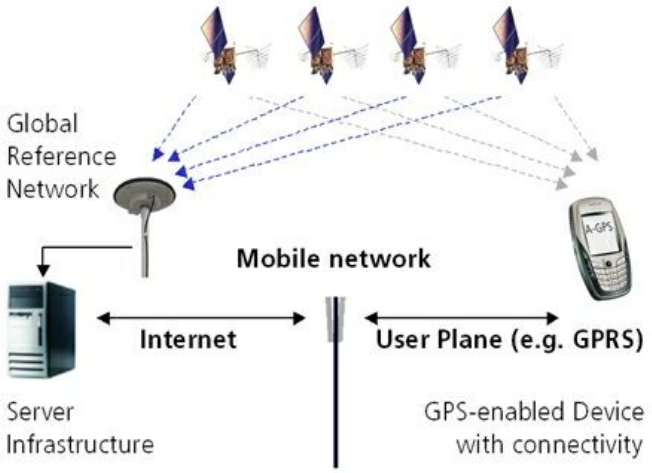
area. La miglioria introdotta non è altro che una sorta di caching: sapendo che le celle di telefonia mobile hanno necessariamente una posizione fissata si è pensato di far in modo che siano quest'ultime a fornire al device la lista necessaria, recuperandola da un apposito server, detto Assistance Server, e fornendogliela tramite la rete mobile. L'assunzione di fondo è che se un dispositivo comunica con una cella telefonica, verosimilmente entrambi si trovano in una zona coperta dagli stessi satelliti.[29]

Ormai la quasi totalità degli smatphones è fornita di un modulo GPS. Questo permette di avere una stima abbastanza accurata della posizione del dispositivo sul pianeta. Con un dispositivo equipaggiato col GPS è possibile conoscere:

- *L*atitudine, *L*ongitudine (espresse in gradi rispettivamente dall'equatore e dal meridiano fondamentale) e *A*ltezza (espressa in metri sul livello del mare), tripletta nota con il nome di *coordinate LLA;*
- Velocità e Orientamento (espresso in gradi rispetto alla direzione del Polo Nord *geografico*, non magnetico).

Le infomazioni ottenute hanno un'accuratezza variabile seppur stimabile: gli esperimenti svolti l'hanno vista oscillare da un minimo di circa 5m (all'aper-

Figura 2.2: Schema di funzionamento dell'AGPS



to in assenza di nubi) ad un massimo di 30-40m (al chiuso, in una giornata nuvolosa). Decàde quindi ogni eventuale congettura sull'utilizzo del solo GPS per l'individuazione di entità, esso rimane comunque una valida risorsa per le medie-lunghe distanze ovvero quando un errore dell'ordine delle decine di metri diventa poco significativo per i calcoli e trascurabile per la user-experience.

2.2 BLE - Bluetooth Low Energy

Il Bluetooth[28] è uno standard per lo scambio di dati wireless sulle brevi distanze sviluppato prima da Ericsson e successivamente gestito dal Bluetooth Special Interest Group. Dalla prima versione del 1999 ne sono state rilasciate altre in un crescendo di affidabilità, robustezza e risparmio energetico fino a giungere ad oggi alla 4.0 ed all'annuncio, risalente a fine 2013, della versione 4.1[25]. Esso possiede un raggio d'azione nominale di circa 60 metri ed è impiegato per comunicazioni CO[1] punto-punto, ovvero non è possibile il broadcasting di un'informazione.

[1] «Connection Oriented»

La versione 4.0, d'interesse ai fini di questa tesi, presenta un'interessante novità: il BLE o, in ambito commerciale Bluetooth Smarth, interessante perchè permette di avere, come si evince dal nome, un basso consumo di potenza senza tuttavia ridurre il raggio d'azione. Trattasi all'atto pratico di un nuovo standard: l'architettura non è più peer-to-peer ma permette ai dispositivi di interpretare quattro ruoli:

Broadcaster: trasmette segnali in broadcast senza tuttavia la possibilità di ricevere

Observer: riceve i segnali di un broadcaster senza tuttavia la possibilità di trasmettere

Peripheral: può sia trasmettere che ricevere e può connettersi ad altri dispositivi in modalità slave («ricevendo» la connessione)

Central: può sia trasmettere che ricevere e può connettersi attivamente ad altri dispositivi in modalità master (richiedendo la connessione)

I device dotati di modulo bluetooth sono configurabili in tre fasce:

Bluetooth: compatibili con il bluetooth «classico»

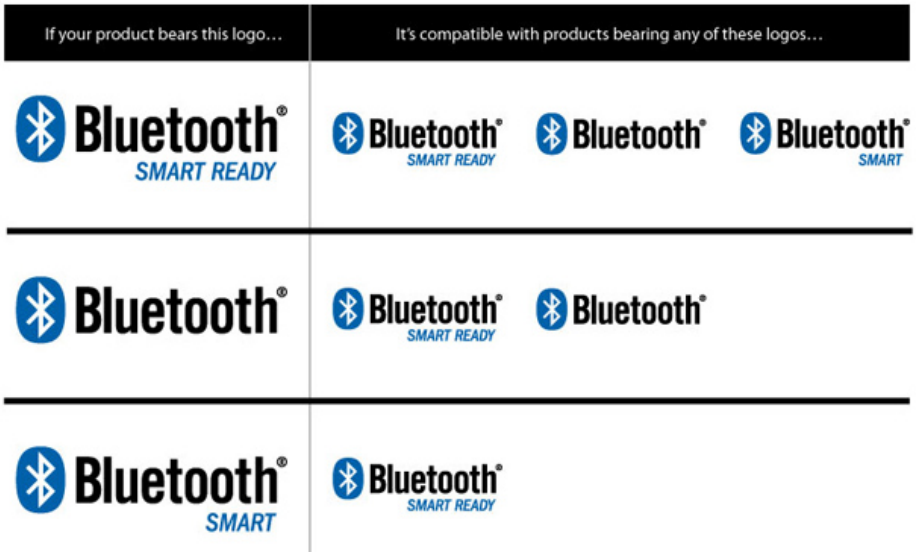
Bluetooth Smart: compatibili solo con il BLE

Bluetooth Smart Ready: compatibili con entrambi

Tra i dispositivi BLE che si è pensato di impiegare ci sono, come già detto, i bluetooth beacon[2]i quali, come a breve sarà più chiaro, nascono con lo scopo di inviare periodicamente un "fascio di informazioni" in broadcast nell'ambiente circostante che serve a notificare ad ogni eventuale ascoltatore la presenza del

[2]beacon in inglese significa "faro"

Figura 2.3: Intercompatibilità delle fasce di dispositivi bluetooth



beacon stesso. Purtroppo la modalità peripheral non è supportata dalle attuali versioni di Android[4, 8], il che impedisce, ad esempio, di creare un beacon virtuale, limite che si è fatto sentire in fase di progettazione. Potrebbe anche essere interessante approfondire com'è fatto il "fascio" di cui sopra, il cui nome effettivo è BLE Advertising e la struttura in termini di campi di bytes, così come standardizzato da Apple, promotrice della tecnologia, è la seguente[26, 28]:

UUID [16 byte]: la sigla sta per "identificativo univoco universale" ed è un numero usato per marchiare tutti i beacon appartenenti alla stessa organizzazione / applicazione;

Major [2 byte]: è un numero usato per distinguere sottoinsiemi di beacon diversi all'interno della stessa organizzazione / applicazione;

Minor [2 byte]: è un numero usato per identificare i beacon a parita di UUID e Major;

TxPower [1 byte]: è un campo contenente la calibrazione del beacon ovvero la RSSI[3] misurata ad un metro di distanza dal beacon. Questo campo,

[3]Received Signal Strength Indicator: indicatore della potenza del segnale ricevuto espressa in dBm

Figura 2.4: Fotografia degli iBeacons prodotti da Accent Advanced Systems

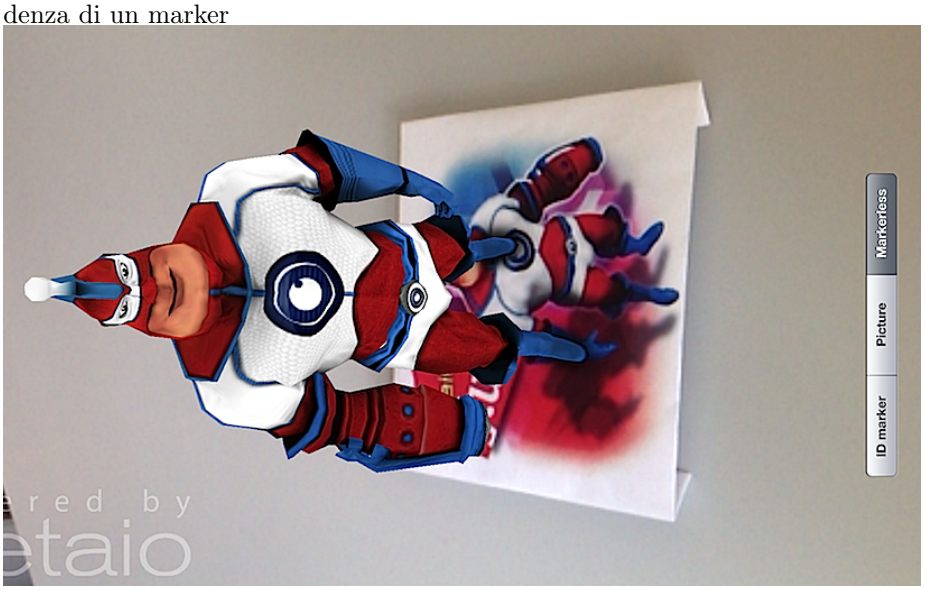
così come la sua accuratezza, è *indispensabile* per avere una stima della distanza dal beacon.

Questi campi solitamente possono essere settati a piacimento.

Ovviamente, essendo un dispositivo bluetooth, il beacon possiede anche un indirizzo MAC.

È importante notare come il beacon venga concepito come sensore di prossimità, tuttavia, come si vedrà nel seguito, con opportuni algoritmi è possibile impiegarlo come sensore di distanza deducendo quest'ultima dall'attenuazione subita dal segnale. Gli esperimenti in tal senso hanno evidenziato come le distanze calcolate in questa maniera siano abbastanza accurate (in condizioni ottimali), tuttavia, data la natura del segnale è facile che quest'ultimo sia poco stabile, così come ogni informazione da esso dedotta: la presenza di ostacoli nella linea retta fra trasmettitore e ricevitore fa variare e anche sensibilmente i valori percepiti.

Figura 2.5: Esempio di AR in cui viene mostrato un modello 3D in corrispondenza di un marker



2.3 Realtà aumentata

La realtà aumentata[30] è una vista diretta o indiretta di un ambiente del mondo fisico i cui elementi sono «aumentati» da dati sensoriali solitamente generati per mezzo di un elaboratore come ad esempio video, suoni, grafica o dati GPS. L'aumento della realtà avviene convenzionalmente in tempo reale e le informazioni fornite sono legate al contesto dell'ambiente osservato: l'esempio più banale è la visualizzazione del punteggio durante una partita in televisione. Le moderne tecnologie permettono di realizzare un aumento interattivo della realtà: mediante l'uso di sensori, i motori di realtà aumentata in esecuzione sugli smat-device sono in grado di fornire informazioni differenti a seconda di ciò che si sta inquadrando: e.g. è possibile mostrare un'immagine 2D od un modello 3D in corrispondenza di un'immagine o un oggetto riconosciuto. I sensori solitamente impiegati sono molteplici: fotocamera e/o sensore ottico, giroscopio, accelerometro, bussola, GPS, etc.

Figura 2.6: Esempio di AR in cui vengono mostrati elementi in base alla loro posizione

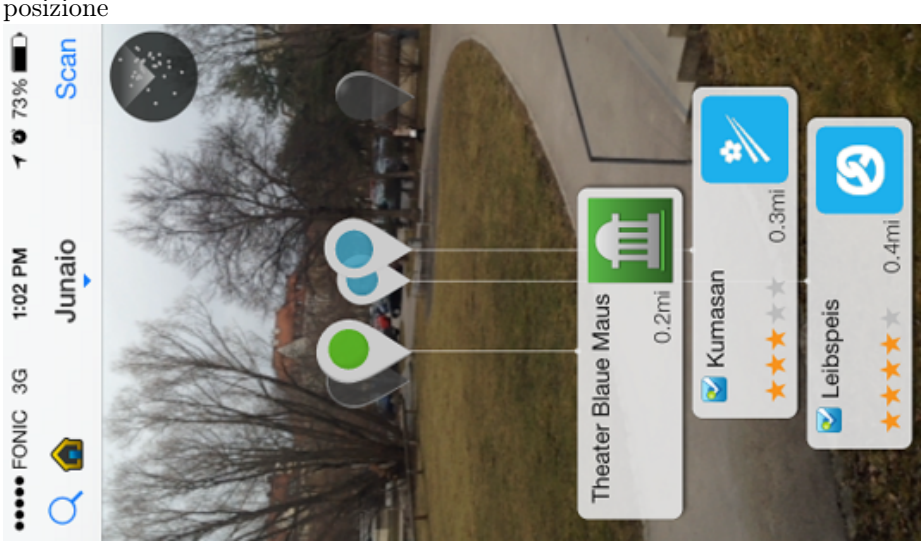


Figura 2.7: Esempio di AR in cui l'ambiente inquadrato dallo smartphone viene arricchito di informazioni relative al contesto

2.4 Librerie a disposizione

2.4.1 Android SDK

Le librerie di Android permettono allo sviluppatore di utilizzare il GPS con lo stile "a callback" tipico del sistema operativo mobile. Nella pratica si tratta di richiedere aggiornamenti sulla posizione ad un "Location Manager" specificando[15]:

1. Un fornitore ("*Location Provider*"): solitamente la scelta avviene tra NETWORK_PROVIDER e GPS_PROVIDER. Il primo fornisce una posizione approssimativa dedotta dal Wi-Fi, così come fa Google quando deduce la posizione di un computer fisso sprovvisto di GPS; il secondo fornisce un'informazione molto più accurata dedotta dall'interazione coi satelliti;

2. Una frequenza di aggiornamento: è possibile indicare il numero minimo di millisecondi che il sistema attende tra un aggiornamento di posizione ed il successivo;

3. Una distanza di aggiornamento: è possibile specificare la variazione minima di distanza in metri sufficiente per un nuovo aggiornamento;

4. Un ascoltatore di callbacks relative ad aggiornamenti sulla posizione e/o sullo stato del provider.

Già nella guida alla API di Android è consigliato di combinare i due Location Provider: la fase di inizializzazione del GPS è assai più lenta, l'altro provider sopperisce per la durata iniziale della "scoperta" della posizione.

Inoltre è importante notare come la ricerca della posizione sia un'attività completamente asincrona (difficile immaginare la cattura sincrona di un dato sensoriale): sarà importante tener conto di questo fatto in fase di progetto.

2.4.2 AltBeacon

AltBeacon[1] è una specifica aperta per un formato del messaggio di advertisement dei BLE beacon. Per supportare tale specifica su Andorid viene fornita una libreria, Android Beacon Library[3], che favorisce l'interazione tra un device Android ed i beacon circostanti. Tre sono i nodi d'interesse per quanto riguarda questo software:

- Il concetto di Beacon Parser[6]: trattasi di un componente che "spiega" alla libreria come interpretare i byte che compongono il messaggio di advertisement proveniente da un beacon. Quando un pacchetto BLE è compatibile (in gergo "matcha") con il pattern specificato in un'istanza di Beacon Parser, i suoi campi possono essere interpretati ed il beacon può essere usato dalla libreria.
  - Il pattern è costituito da una stringa a sua volta composta da una sequenza di moduli, separati da virgola, ognuno dei quali è nella forma:

    <prefisso>:<byte-iniziale>-<byte-finale>[=<valore_esadecimale>]

    [4]I prefissi accettati sono[6]:

    m: sequenza di riconoscimento ("*M*atching sequence") per il tipo di beacon da riconoscere (una sola ammessa)

    i: *I*dentificatore (più di uno ammessi)

    p: campo di calibrazione della *P*otenza (uno solo ammesso)

    d: campo di *D*ati (più di uno ammessi)
  - Questo concetto è importante perché la libreria, se utilizzata così come fornita sul sito principale, "trova", come impostazione di default, solo beacon il cui messaggio abbia la struttura dettata dalla specifica AltBeacon.

[4]Senza parentesi angolari.
Le parentesi quadre indicano opzionalità.

  - Si è posta quindi la necessità di creare un pattern che non imponga filtri di sorta:

    m:2−3=0215,i:4−19,i:20−21,i:22−23,p:24−24

    * I byte 2 e 3 costituiscono la sequenza di riconoscimento
    * Segue un identificatore di 16 byte, ovvero l'UUID del beacon
    * Seguono due identificatori da 2 byte ciascuno, ovvero, rispettivamente, Major e Minor
    * Segue un campo di dimensione 1 byte che contiene la potenza calibrata

- Il concetto di *Region[19]*: posto che inizialmente la libreria filtra i beacon sulla base della struttura del loro pacchetto, ad un livello più alto, diciamo applicativo, potrebbe porsi la necessità di filtrare tutti i beacon con un certo UUID, o Major, o Minor, o combinazioni dei precedenti a seconda della semantica associata a tali numeri. Una "regione" fa proprio questo: applica un filtro a tutti i beacon percepiti dal device.
- Il concetto di *distanza*: la libreria stima la distanza sulla base di una serie di considerazioni:

2.4.3 Metaio

Metaio[16] è un framework per la realtà aumentata sviluppato dall'omonima azienda, locomotiva della ricerca in tale ambito. Per quanto concerne gli scopi di questo documento, ciò che importa approfondire è l'approccio all'SDK. Metaio in Android espone le sue funzionalità per mezzo di un'Activity ed ha più modalità di funzionamento o, come vengono chiamate all'interno della libreria, *Tracking Configuration*s, per citarne solo alcuni:

- Localization-based AR: la realtà viene aumentata per mezzo di quelli che nel seguito verranno chiamati segnaposti. Il motore di AR è conscio della

posizione del dispositivo su cui è in esecuzione e, facendo un uso combinato dei vari sensori, riesce a capire quale "pezzo di mondo" viene inquadrato dall'utente, di conseguenza, note le posizioni LLA delle entità da mostrare, esse possono essere correttamente ubicate nella scena. Viene messo a disposizione anche un *radar*, ossia una rappresentazione bidimensionale della disposizione delle entità aumentanti vista dall'alto. A patto dunque che le coordinate LLA dei vari segnaposti siano coerenti con le posizioni effettive dei facenti parte il sistema, l'utilizzatore ha sempre o quasi una percezione relativamente precisa della posizione relativa di ciò che osserva rispetto a sé.

- Tracking Pattern: la realtà viene aumentata mostrando un modello 3D oppure un'immagine bidimensionale in corrispondenza del punto della scena in cui viene riconosciuto un certo marker. Il motore si occupa di comprendere la disposizione nello spazio reale del marker e di ridimensionare e ruotare opportunamente l'oggetto virtuale corrispondente. I marker da riconoscere devono essere noti a priori, l'efficienza del riconoscimento dipende proprio dal fatto che il motore "sa già" quali sono gli elementi da ricercare. Ne consegue che tale tipo di riconoscimento, che ben si confà ai desiderata, ponga delle limitazioni sull'apertura del sistema da realizzare: si imporrebbe un vincolo sull'ingresso nello stesso da parte di nuove entità. Inoltre ci sono delle considerazioni da fare per quanto riguarda la natura dei marker[23]:

  - Metaio considera le immagini in scala di grigi, quindi, contrariamente a quanto si potrebbe pensare a primo impatto, meglio avere molti colori nello stesso marker ed un elevato contrasto, mentre, immagini tendenzialmente monocromatiche sono sconsigliabili;
  - È bene che il rapporto tra le dimensioni dell'immagine non si allontani troppo dall'unità;
  - Immagini troppo buie sono sconsigliabili;

  - La dimensione minore del marker è bene che superi comunque i 150-200 pixel.

- [QR] Code Reader: Metaio offre la possibilità di scansionare la scena alla ricerca di codici QR[9] (ma all'occorrenza anche codici a barre o simili): seppure simili ai marker, l'approccio è molto diverso. Nel primo caso il sistema sa che cosa sta cercando, nel secondo caso sa che sta cercando un codice con una certa struttura ma non ha ipotesi su come questa sarà concretamente. Questa differenza, che potrebbe sembrare trascurabile mentre è semplicemente sottile, ha (come tutto d'altronde) vantaggi e svantaggi: un codice QR è facile da generare, relativamente facile da riconoscere, può codificare una mole non indifferente di informazioni, è una tecnologia consolidata che trascende i confini di questa specifica applicazione (banalmente, se si marchia qualcosa con un codice QR, posso decodificarne l'informazione anche senza Metaio); tuttavia il riconoscimento è comunque più oneroso di quello di un marker sia in termini computazionali, sia in termini pratici (non bisogna essere troppo distanti, la telecamera non deve muoversi troppo, etc).

È bene notare quello che è probabilmente il fattore più limitante di Metaio: nella stessa Activity *non possono coesistere più Tracking Configurations.* Ciò implica ad esempio che non è possibile mostrare i segnaposti e *contemporaneamente* scansionare la scena alla ricerca di marker e/o codici di qualunque genere e viceversa. Questo fattore sarà tenuto in forte considerazione in fase di progetto.

Capitolo 3

Requisiti

Ciò che si vuole approfondire è la maniera in cui dispositivi moderni come smartphone, tablet et similia possano aiutare l'utente nell'individuazione di entità che esistono e si spostano nello stesso ambiente dell'utilizzatore, ovvero quello normalmente inteso con la parola "realtà". La metafora con la quale si desidera fornire tale supporto è quella della "realtà aumentata", ossia l'aggiunta di contenuti informativi a quella che è la normale percezione dell'essere umano data dai cinque sensi. Nello specifico si vorrebbe che il dispositivo diventi in grado di *guardare* la realtà, *individuare* le entità presenti nell'ambiente, quandunque sia possibile conoscerne la posizione, e/o assistere l'utente nella ricerca, quando invece non sia possibile. Posta questa finalità, per raggiungerla, si vuole provare a combinare, come da titolo, tre diverse tecnologie:

- il GPS, Global Positioning System, utile per ottenere le coordinate sferiche di un punto sulla (o in prossimità della) superficie terrestre;
- il BLE, Bluetooth Low Energy, incarnato nei *beacon* bluetooth, dispositivi nati come sensori di prossimità ma impiegabili anche per avere una stima della distanza Ricevitore $\leftrightarrow$Beacon;

- il Riconoscimento di immagini: si ipotizza sin da ora di "marchiare" le entità da individuare con un simbolo, riconoscibile dai dispositivi, utile per un riconoscimento "certo" nelle corte distanze.

Lo scopo ultimo è quindi, in generale, "guidare" l'essere umano verso le entità che potrebbero interessarlo. Dovrebbe a questo punto essere chiaro il motivo per cui si tenta di combinare tecniche eterogenee: ognuna di quelle elencate ha diversi "raggi d'azione" garantendo accuratezze differenti a seconda della distanza tra utente ed entità.

Sarebbe inoltre gradito *non* vincolare più del dovuto il sistema che si andrà a realizzare alle tecnologie di cui sopra o, per lo meno, non chiudersi alla possibilità di integrarne di altre. Si sceglie di impiegare come piattaforma Android per motivazioni pratiche (maggiore flessibilità, maggiore disponibilità di dispositivi economici già equipaggiati di tutti i sensori necessari, maggiore disponibilià di sorgenti con licenze aperte) ma, al tempo stesso, di non legarsi ad essa.

3.1 Analisi dei requisiti

3.1.1 Requisiti Espliciti

Il sistema che si andrà a realizzare si dovrà occupare sostanzialmente di tre cose, come possiamo dedurre dai requisiti:

- localizzare le entità nello spazio reale;
- identificare le entità nello spazio aumentato;
- presentare le informazioni richieste all'utente con un livello di dettaglio consono alla sua distanza da ciò che sta osservando.

Da notare che la localizzazione *non* è un'informazione binaria: tra il "non so dove si trova" ed il "so esattamente dove si trova" esiste un continuo di possibilità. Il sistema non deve necessariamente essere conscio sempre e comunque di tutte le posizioni assolute: avendo a disposizione anche sensori di prossimità / distanza deve essere prevista la possibilità che esistano momenti in cui l'informazione sulla posizione non sia completa bensì parziale e tale incompletezza può (e deve) comunque essere utile all'utilizzatore. Un'informazione del tipo "qui vicino c'è...", "ti stai avvicinando a...", "ti stai allontanando da..." è sempre meglio di niente!

Viene inoltre naturale pensare alle entità come dotate di un'identità propria: nel momento in cui un osservatore percepisce un'entità vicina è facile immaginare che sia interessato a conoscere anche di chi si tratti; tuttavia, anche in questo caso, l'informazione parziale è meglio dell'assenza di informazione.

3.1.2 Requisiti Impliciti

La natura del sistema, così come si sta delineando, impone che questo sia un sistema distribuito: è evidente che gli attori in gioco necessitano di una comunicazione tra macchine, ergo una rete. Serve un mediatore che tenga traccia, ad esempio, delle posizioni assolute delle entità e che possa essere interrogato da una per ricevere tale informazione rispetto ad un'altra. Il mediatore è pertanto necessario ma non deve vincolare le entità più del dovuto, queste hanno un margine di autonomia: ad esempio non è necessario alcun mediatore per percepire la prossimità. La presa di coscienza della distribuzione del sistema aggiunge a sua volta altre considerazioni da fare come ad esempio il layer di presentazione. Tali considerazioni verranno fatte in dettaglio man mano.

Inoltre c'è un requisito che permea tutti i ragionamenti fatti sinora: si è genericamente parlato di entità anche se dovrebbe essere chiaro, a questo punto, che esistono almeno due ruoli che queste si troveranno ad assumere:

- Osservatore: ovvero l'entità che è interessata a cercarne altre e che è quindi attiva
- Osservato: ossia l'entità che è passivamente osservata

Altri requisiti impliciti sono banalmente la capacità dei device di conoscere la propria posizione (sensore GPS), di percepire il BLE (bluetooth smart capability), di comunicare sulla rete (Wi-Fi) e di "vedere" il mondo reale (fotocamera). In quanto segue vige sempre l'ipotesi di fondo che i dispositivi considerati abbiano tutte queste abilità.

3.2 Glossario

Sono già stati usati più volte una serie di termini ricorrenti ed altri ancora si aggiungeranno all'insieme di parole impiegate nello studio descritto in questo documento. Onde evitare ambiguità, la sezione corrente espone la definizione dei concetti indispensabili per la comprensione del progetto.

Dispositivo: (o *device*) termine generico con il quale ci si riferisce ad una qualunque macchina / componente elettronico che sia partecipe del sistema con un ruolo qualsivoglia (tablet, smartphone, pc, beacon, etc.). I device hanno solitamente una serie di attributi che ne definiscono le proprietà fondamentali: nel caso di un beacon ad esempio il MAC è un attributo caratteristico.

Entità: qualunque attore del sistema sufficientemente importante da avere un identificativo. Essa ha delle proprietà osservabili da altre entità tra cui, in primis, la posizione assoluta. Ad un'entità sono associati uno o più dispositivi fisici. A seconda del contesto, con tale termine, ci si può riferire sia all'entità in senso astratto, sia allo specifico dispositivo che la incarna, sia all'eventuale utente di cui il dispositivo diventa punto di accesso.

Identificativo: (o *id*, *identificatore*) sequenza di caratteri che identificano univocamente un'entità all'interno del sistema.

Self: in locale, relativamente ad un dispositivo, l'entità a cui il dispositivo stesso è associato è riferita come Self.

Osservatore: ruolo interpretato da un'entità in una certa fase di funzionamento del sistema. Un'entità osservatrice è interessata a individuare le altre e conoscerne alcune od ogni proprietà. A meno di casi particolari, essa è guidata dall'utente umano che quindi, per estensione, viene chiamato anch'esso osservatore.

Osservato: ruolo interpretato da un'entità in una certa fase di funzionamento del sistema. Un'entità osservata (o meglio, *osservabile*) esiste passivamente. L'unica sua responsabilità è essere conscia di tutto ciò che riguarda se stessa e fornire tali informazioni agli osservatori richiedenti.

Landmark: particolare tipo di entità che funge da "punto di riferimento" per le altre, essendo caratterizzata dal fatto che la sua posizione è nota e non varia o varia con frequenza trascurabile. Ogni landmark è a conoscenza della o può conoscere la distanza tra esso ed ogni altro landmark del sistema.

Posizione assoluta: conoscenza completa di dove si trova un punto sulla Terra espressa in un sistema di coordinate qualunque (solitamente latitudine - longitudine - altezza).

Posizione relativa: conoscenza di dove si trova un punto rispetto ad un altro. Se è nota la posizione assoluta del secondo, allora tale informazione permette di conoscere la posizione assoluta anche del primo.

Distanza relativa: definisce una distanza che ha come estremo "qualcosa". Se è noto anche il secondo estremo allora la distanza relativa rappresenta un segmento, altrimenti un'area circolare.

Distanza relativa ad un'entità: caso particolare di distanza relativa in cui il riferimento è dato da un'entità.

Trilaterazione: procedimento che porta alla conoscenza della posizione di un punto D a patto che siano conosciute le sue distanze da altri 3 punti noti A, B e C.

Pattern: in questo documento è inteso come "qualcosa da riconoscere": descrive la struttura di ciò che va riconosciuto permettendo che ciò avvenga.

Marker: in questo documento è inteso nel senso italiano di "marchio" apposto su un'entità reale per identificarla. N.B. Metaio usa la parola marker con una sfumatura di significato, più specifica.

Area di interesse: area in cui si concentrano la maggior parte delle entità del sistema, dove ha quindi senso collocare dei landmark per coadiuvare la localizzazione.

Livello di dettaglio: (o *LOD*) rispetto ad un'entità, ogni altra entità del sistema ha un livello di dettaglio ovvero un tag che suggerisce quali informazioni è conveniente mostrare e quali no. Esiste una correlazione tra LOD e distanza: intuitivamente entità distanti hanno un basso livello di dettaglio al contrario di entità vicine.

Collezione: generico raggruppamento di elementi, eventualmente presenti anche più volte, senza ipotesi sull'ordine.

Lista: raggruppamento ordinato di elementi.

Insieme: raggruppamento non ordinato di elementi unici.

Manager: componente software cui spetta la responsabilità di una macro-attività del sistema.

Handler: componente software cui spetta la responsabilitò di una micro-attività del sistema.

Contesto: (o *Contesto di sistema*, *System Context*, da non confondersi col il concetto di ApplicationContext in Android) l'insieme dei manager in esecuzione sul sistema in un dato momento.

3.3 Visione

Ho immaginato due ruoli per i dispositivi: l'osservatore e l'osservato comunicanti per mezzo di un server ubicato su internet, non sulla rete locale (così da permettere una più ampia scala di movimento dei dispositivi se dotati di connessione dati). L'osservato non é altro che un tablet o simile con attaccato un beacon. Lo schermo dell'osservato ha la sola funzione di mostrare il pattern ad esso associato mentre in background viene recuperata la posizione GPS ed inviata al server. Contemporaneamente il beacon lancia il proprio segnale con una frequenza mediamente elevata (una/due volte al secondo) . Il segnale comprende almeno le seguenti informazioni: il MAC del beacon, associato all'osservato e la potenza di trasmissione necessaria per fare delle speculazioni sulla distanza dal beacon stesso. L'osservatore interroga continuamente il server per sapere la posizione GPS delle varie entità mostrando dei segnaposti nella realtà aumentata che permettono all'utente di individuare le entità più distanti ed essere "conscio" della loro presenza. Anche l'osservatore sincronizza la propria posizione con il server, dunque quest'ultimo potrebbe mettere in atto politiche basate sulla distanza, fornendo ad esempio solo le informazioni sulle entità in un certo raggio d'azione. A tal proposito ho immaginato che, dal punto di vista di un osservatore, tutte le altre entità abbiano un livello di dettaglio, variabile in funzione della distanza, che indica quante e quali informazioni mostrare. Il dispositivo osservatore svolge anche altre funzioni in background: "migliora" il posizionamento delle entità più vicine ma non ancora visibili per mezzo di eventuali segnali beacon e, quando si rende conto di essere abbastanza vicino ad un'entità e/o su richiesta dell'utente, avvia una scansione alla ricerca di marker.

Per quanto riguarda i beacon ho immaginato ipotesi differenti in base alla configurazione di questi ultimi:

Hp0: Ogni dispositivo osservato ha un beacon associato che invia esclusivamente l'informazione relativa al proprio MAC con una potenza di trasmissione nota a priori: in questa ipotesi non si possono fare molte speculazioni sulla distanza effettiva tuttavia, in assenza di informazioni GPS, l'osservatore potrebbe comunque essere conscio del fatto che ad una distanza massima teorica si trova l'entità associata al beacon e anche capire se si stia avvicinando/allontanando da essa basandosi sul miglioramento/peggioramento del segnale.

Hp1: Hp0 + il beacon è configurato correttamente ed invia anche informazioni utili per il computo della distanza. Ciò permette all'osservatore di stimare la distanza da ogni entità sufficientemente vicina: non è ancora un posizionamento vero e proprio ma, studiando adeguatamente la user-experience, si può assistere l'osservatore nella ricerca dell'osservato sino a che questo non rientra nella visuale (momento nel quale viene riconosciuto con precisione). Rispetto all'Hp0 c'è un miglioramento dell'informazione: mentre in Hp0 è "binaria" (mi avvicino/mi allontano), qui l'informazione è quasi continua (so all'incirca quanto sono distante in ogni momento).

Hp2: Hp1 + impiego di $N \geq 3$ beacon "fissi" (nel seguito landmark) distribuiti in un'area di interesse con posizioni relative note. A questo punto ci sono delle sotto-ipotesi:

Hp2.1: L'osservato è un dispositivo mobile capace sia di emettere che di ricevere segnali beacon (ipotesi falsa a priori per Android sino al rilascio di Android L[17] e per tanto non approfondita più di tanto). In tal caso l'osservato trilatera la propria posizione rispetto ai landmark e la trasmette nel segnale di beacon (o comunque la comunica agli osservatori senza passare per il server). L'osservatore che riesca

a percepire almeno un landmark e anche l'osservato può calcolare la posizione relativa di quest'ultimo rispetto a se stesso.

Hp2.2: L'osservato emette il segnale beacon grazie ad un emettitore appositamente attaccato ad un dispositivo mobile. In tal caso il suddetto beacon ha la sola funzione di sensore di prossimità come alle hp 0 e 1. Il dispositivo dell'osservato si occupa invece di trilaterare la propria posizione rispetto ai landmark e comunicarla al server. Il dispositivo osservatore può quindi calcolare la propria posizione rispetto ai landmark, conoscere la posizione dell'osservato rispetto ai landmark tramite il server e, pertanto, dedurre la posizione dell'osservato rispetto ad esso.

Indipendentemente da *come* vengono calcolate distanze e posizioni, se viene individuato uno spostamento è possibile fare speculazioni se trattasi di avvicinamento o allontanamento e/o applicare politiche di conseguenza: ad esempio si può avviare una breve scansione se viene rilevato un avvicinamento ad un'entità oltre una certa soglia.

In ogni caso, anche in temporanea assenza di informazioni posizionali, deve essere possibile per l'utente che si accorga della presenza di un marker avviare una scansione manualmente.

Capitolo 4

L'infrastruttura

4.1 Dominio

Segue una descrizione del dominio, ovvero dell'insieme dei concetti, delle rispettive interfacce ed implementazioni, sui quali si andrà poi a costruire prima il progetto e poi il sistema e di cui la figura 4.1 fornisce un'overview.

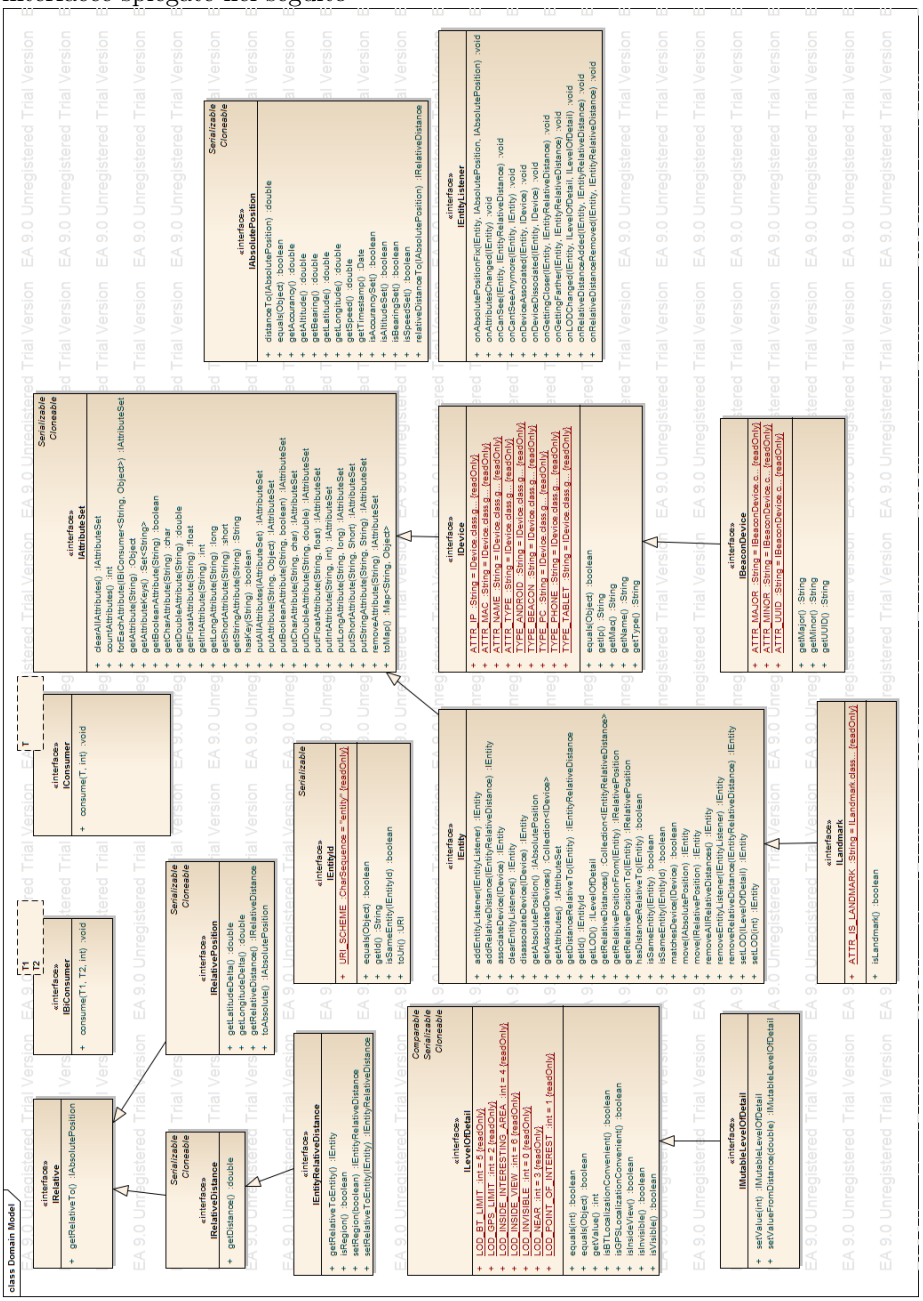
Il linguaggio target è Java, quindi le interfacce sono descritte secondo le sue regole, tuttavia si è scelto di non ricorrere a concetti tipici di Java, limitandosi ad usare tutt'al più classi di cui si trovano equivalenti in qualunque altro linguaggio Object Oriented (Stringhe, Liste, Collezioni, Mappe, Set, non di più).

4.1.1 Convenzioni

Ecco un rapido elenco di convenzioni auto-imposte nella stesura del codice:

- I nomi delle interfacce iniziano tutti con una "I" maiuscola.
- Eventuali costanti tipiche del concetto vengono definite nell'interfaccia.

Figura 4.1: Diagramma delle classi del dominio: rappresenta tutte le principali interfacce spiegate nel seguito



- Eventuali stringhe costanti sono nella forma

  <package>.<nome_modulo>.<valore_costante>

- La struttura dei package (e quindi delle cartelle) è sempre:

  it.unibo.gciatto.thesis.<nome_concetto>[.impl]

  il sotto-package "impl" contiene le implementazioni delle interfacce contenute nel padre.

- I modificatori ritornano il riferimento all'oggetto su cui sono stati chiamati così da poter fare più modifiche inline sullo stesso oggetto.

- I tipi di dato di base spesso estendono Serializzable per far sì che si integrino meglio con Java e con Android.

- I tipi di dato immutabili spesso estendono Cloneable, quindi le relative implementazioni espongono un metodo pubblico *clone()* che ne genera una copia.

- Metodi con una semantica asincrona vengono marchiati con il prefisso *async* nella signature. In caso di ambiguità, anche i metodi sincroni vengono marcati con il prefisso *sync*.

4.1.2 Tipi di dato

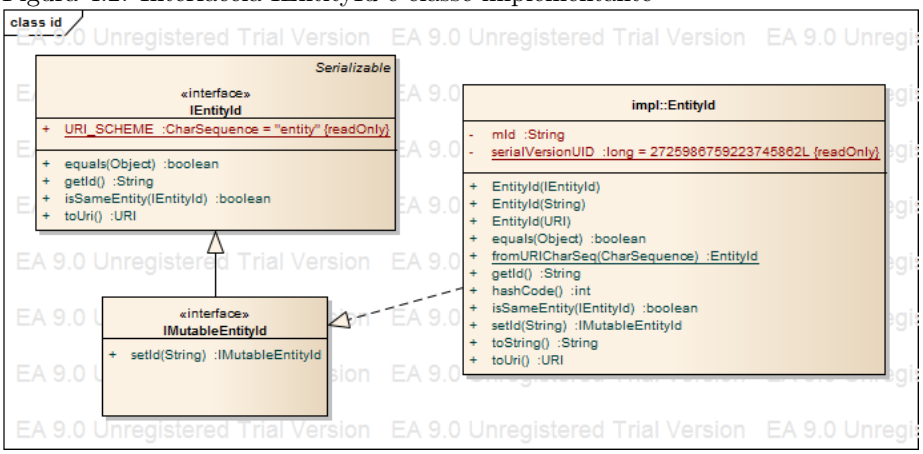
4.1.2.1 Identificatore di entità

(*it.unibo.gciatto.thesis.entity.id.IEntityId*)

Un identificatore di entità è una sequenza di caratteri, univoca all'interno del sistema, associata ad una ed una sola entità.

Trattasi di un concetto *passivo*, *atomico* e *immutabile*.

Figura 4.2: Interfaccia IEntityId e classe implementante



- Metodi (vedi fig. 4.2)
  - Il metodo *getId()* permette di ottenere la sequenza di caratteri di cui sopra.
  - Il metodo *isSameEntity()* confronta esplicitamente due identificatori per verificare se si riferiscono alla stessa entità o meno.
  - Il metodo *toUri()* genera una rappresentazione dell'identificatore nella forma di URI con la seguente struttura:

    entity:<id>

- Costanti

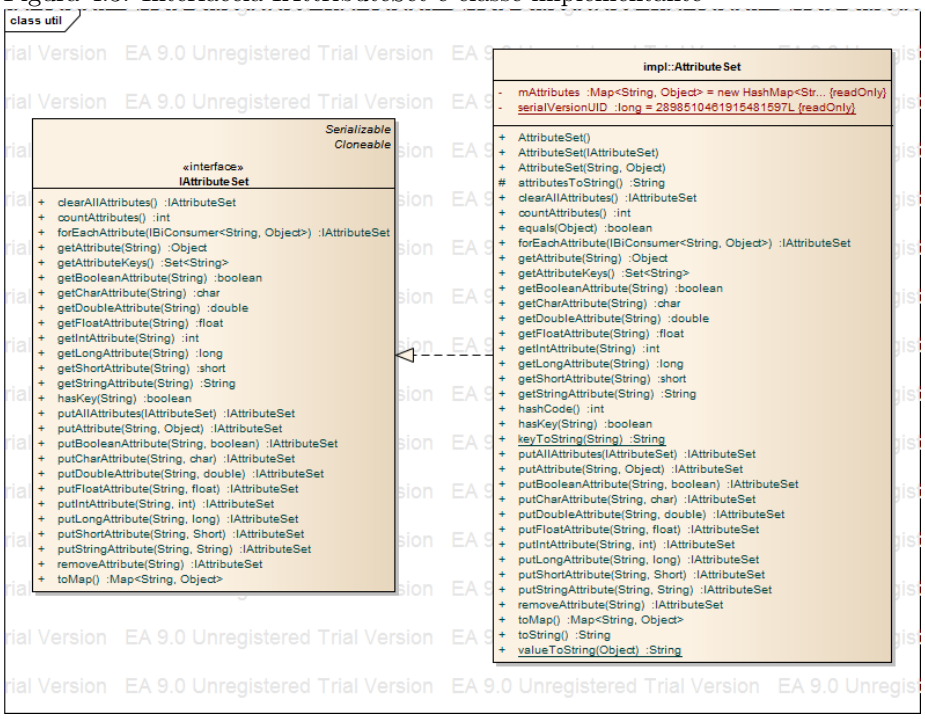
  URI_SCHEME: fissa la parte "scheme" dell'URI generato dal metodo *toUri()*. Il suo valore è "*entity*".

4.1.2.2 Insieme di attributi

(*it.unibo.gciatto.util.IAttributeSet*)

Un insieme di attributi non è altro che una collezione di coppie chiave-valore in cui ogni chiave può essere presente 0 o 1 volta. Le chiavi sono stringhe mentre i valori possono essere di qualunque tipo

Figura 4.3: Interfaccia IAttributeSet e classe implementante



Trattasi di un concetto *passivo*, *composto* e *mutabile*.

- Metodi (vedi fig. 4.3)
  - Il metodo *clearAllAttributes()* svuota l'insieme e ritorna l'oggetto su cui è stato invocato.
  - Il metodo *countAttributes()* ritorna il numero di attributi attualmente presenti nell'insieme.
  - Il metodo *getAttribute()* cerca l'attributo registrato con la chiave in ingresso, in assenza del quale ritorna *null*.
  - Il metodo *getAttributeKeys()* ritorna un set di stringhe contenente tutte le chiavi attualmente registrate nell'insieme.
  - Il metodo *hasKey()* controlla se la chiave in ingresso è registrata oppure no.

  - Il metodo *putAllAttributes()* permette di aggiungere all'insieme tutte le coppie chiave-valore presenti nell'insieme in ingresso e ritorna l'oggetto su cui è invocato.
  - Il metodo *putAttribute()* permette di registrare la coppia chiave-valore in ingresso nell'insieme e ritorna l'oggetto su cui è invocato. Se la chiave è già registrata viene sovrascritto il valore.
  - Il metodo *removeAttribute()* rimuove la coppia corrispondente alla chiave in ingresso e ritorna l'oggetto su cui è invocato.
  - Il metodo *toMap()* genera una mappa di coppie *String-Object* a partire dalle coppie chiave-valore registrate nell'insieme. Modifiche apportate a tale mappa non intaccano lo stato dell'oggetto su cui è stato invocato il metodo.

N.B. Sono presenti anche metodi di appoggio per inserire / rimuovere agilmente i tipi primitivi e le stringhe.

4.1.2.3 Dispositivo

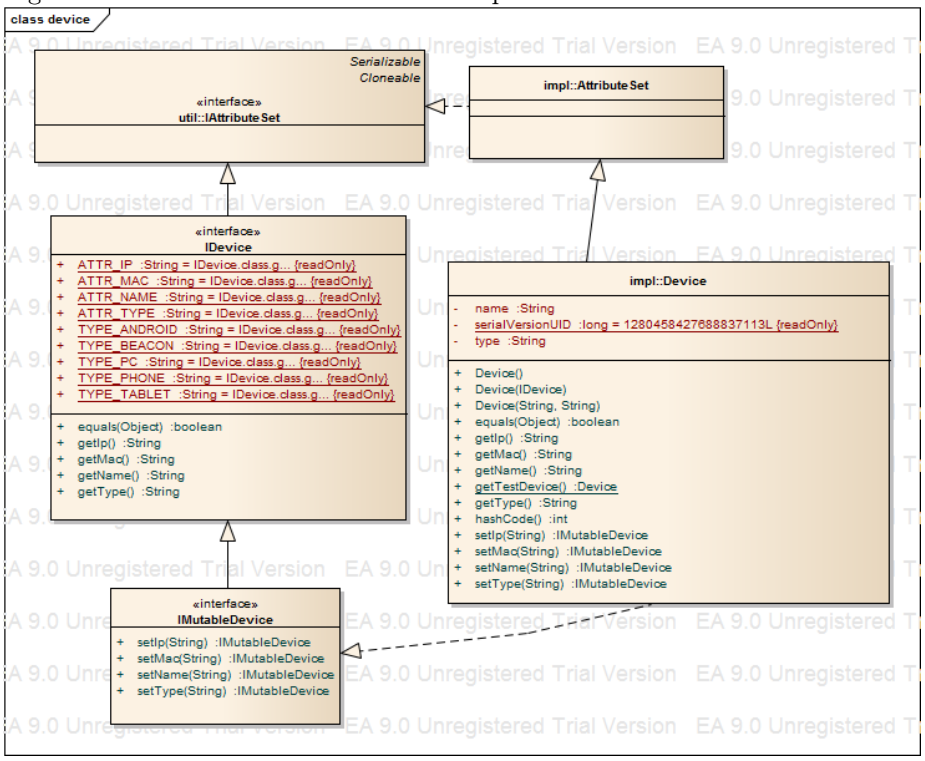
(*it.unibo.gciatto.thesis.device.IDevice*)

Un dispositivo è un insieme di attributi predefiniti che definiscono le caratteristiche principali di un device da associare ad un'entità. Si è scelto di estendere il concetto di insieme di attributi in quanto device di natura diversa potrebbero dover esporre caratteristiche differenti inoltre il medesimo device, in momenti diversi, potrebbe avere la stessa necessità.

Trattasi di un concetto *passivo*, *composto* e *mutabile*.

- Metodi (vedi fig. 4.4)
  - Il metodo *getIp()* ritorna, se presente, l'indirizzo IP del dispositivo nella decimal dotted notation.

Figura 4.4: Interfaccia IDevice e classe implementante



  - Il metodo *getMac()* ritorna, se presente, l'indirizzo MAC del dispositivo nella forma di cifre esadecimali separate da due punti.
  - Il metodo *getName()* ritorna, se presente, il nome del dispositivo.
  - Il medoto *getType()* ritorna, se presente, il tipo del dispositivo.
  - Il metodo *equals()* confronta il device con un altro IDevice.
    Esso ritorna *true* se l'oggetto in ingresso è lo stesso su cui è invocato il metodo oppure se, chiamanto A il device con più campi e B quello con meno campi, ogni campo di B è presente anche in A ed ha lo stesso valore.

- Costanti
  - Definizione campi predefiniti

    ATTR_TYPE = "it.unibo.gciatto.thesis.device.IDevice.type"

    ATTR_NAME = "it.unibo.gciatto.thesis.device.IDevice.name"

    ATTR_MAC = "it.unibo.gciatto.thesis.device.IDevice.mac"

    ATTR_IP = "it.unibo.gciatto.thesis.device.IDevice.ip"
  - Definizione valori predefiniti per il campo type

    TYPE_PC = "it.unibo.gciatto.thesis.device.IDevice.pc"

    TYPE_ANDROID = "it.unibo.gciatto.thesis.device.IDevice.android"

    TYPE_BEACON = "it.unibo.gciatto.thesis.device.IDevice.beacon"

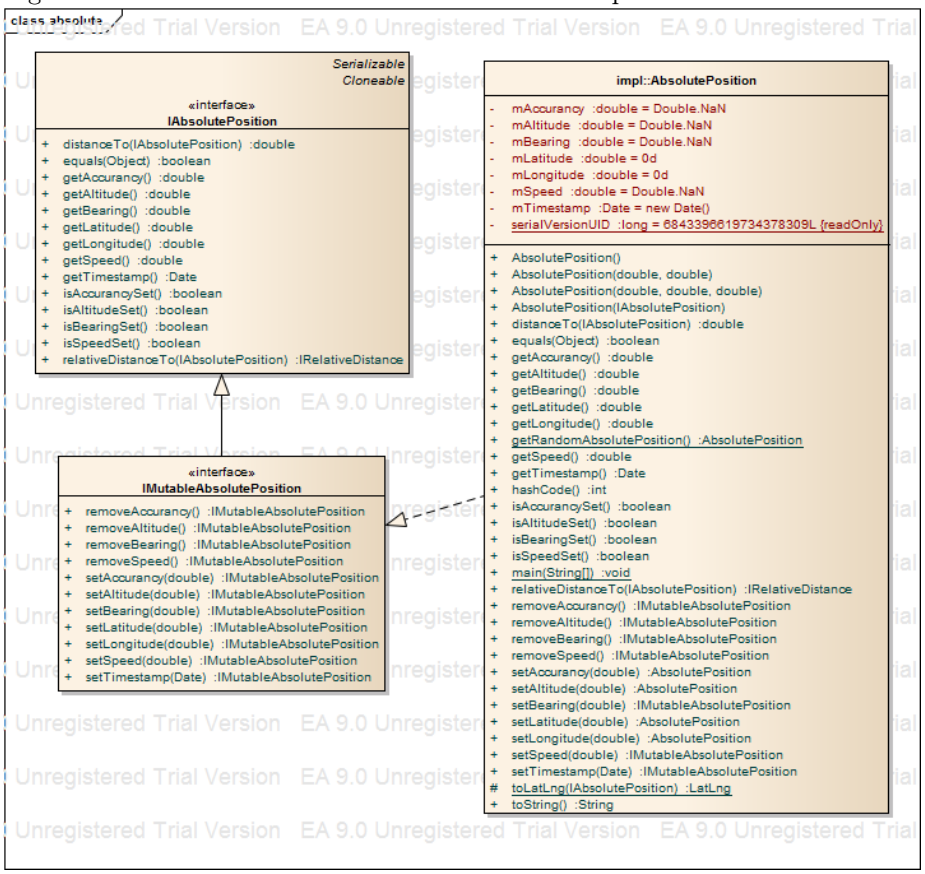
4.1.2.4 Posizione assoluta

(*it.unibo.gciatto.thesis.position.IAbsolutePosition*)

Una posizione assoluta è un punto sulla o in prossimità della superficie terrestre.
Al tempo stesso rappresenta il pacchetto informativo fornito dal sistema GPS.

Trattasi di un concetto *passivo*, *composto* e *immutabile*.

Figura 4.5: Interfaccia IAbsolutePosition e classe implementante



- Metodi 4.5

  - Il metodo *distanceTo()* recupera la distanza in metri tra la posizione su cui è invocato il metodo ed un altro punto. Vengono considerate solo latitudine e longitudine.

  - Il metodo *equals()* confronta l'oggetto corrente con un altra IAbsolutePosition considerando solo latitudine e longitudine.

  - Il metodo *get*()* recupera una proprietà della posizione:

    Accuratezza: l'errore in metri della localizzazione (definisce un'area circolare centrata nel punto con le coordinate date). Double.NaN[1] indica assenza di valore.

    Altitudine: espressa in metri sul livello del mare. Double.NaN indica assenza di valore.

    Bearing: orientamento in gradi rispetto al nord geografico ($0° \div 360°$). Double.NaN indica assenza di valore.

    Latitude: angolo in gradi rispetto all'equatore ($\pm 90°$).

    Longitudine: angolo in gradi rispetto al meridiano fondamentale ($\pm 180°$).

    Velocità: velocità. Double.NaN indica assenza di valore.

    Timestamp: istante in cui è stato generato il dato. Utile per confrontare cronologicamente i dati sensoriali.

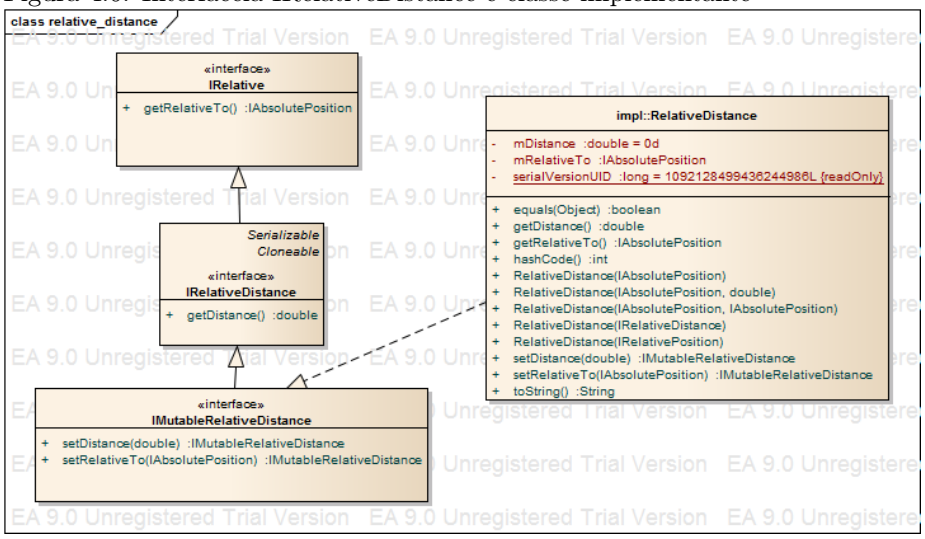
  - Il metodo *is*Set()* verifica se il campo corrispondente è settato o meno i.e. se è o no uguale a Double.NaN.

4.1.2.5 Distanza relativa

(*it.unibo.gciatto.thesis.position.IRelativeDistance*)

[1] Not a Number, si veda IEEE Standard 754

Figura 4.6: Interfaccia IRelativeDistance e classe implementante



Una distanza relativa rappresenza un segmento di cui siano noti almeno lunghezza e posizione di un estremo. Nessuna ipotesi è fatta sulla disposizione nello spazio di tale segmento indi per cui la distanza relativa può anche intendersi come area circolare.

Trattasi di un concetto *passivo*, *atomico* e *immutabile*.

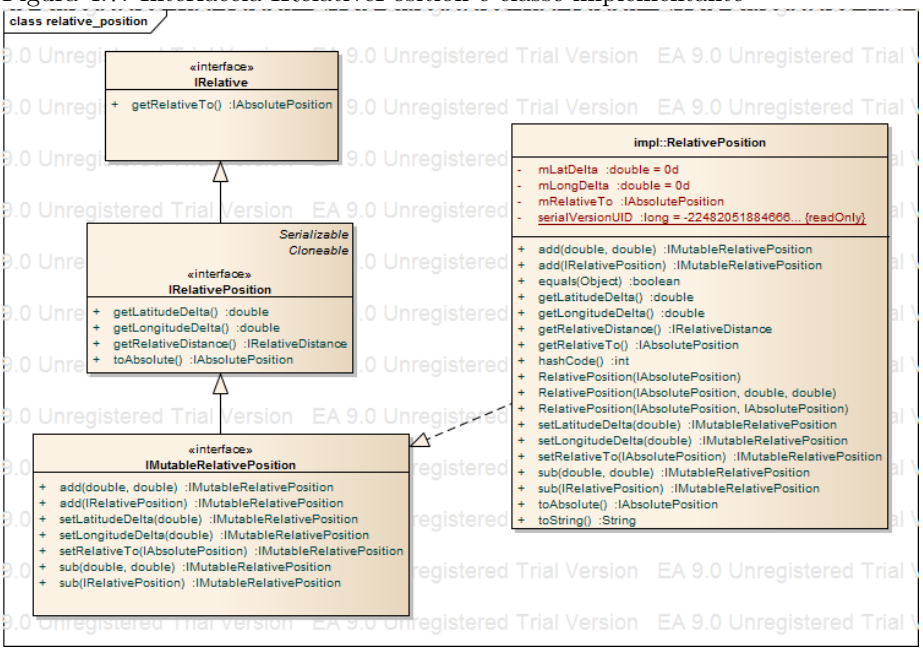
- Metodi (vedi fig. 4.6)
  - Il metodo *getRelativeTo()* restituisce la posizione assoluta dell'estremo noto.
  - Il metodo *getDistance()* restituisce la lunghezza in metri.

4.1.2.6 Posizione relativa

(it.unibo.gciatto.thesis.position.IRelativePosition)

Una posizione relativa è una struttura dati d'appoggio: mantiene le informazioni sulla posizione di un punto nella forma di variazioni dei valori delle coordinate terresti rispetto ad un altro punto di cui si conosce la posizione assoluta.

Figura 4.7: Interfaccia IRelativePosition e classe implementante



Trattasi di un concetto *passivo*, *atomico* e *immutabile*.

- Metodi (vedi fig. 4.7)

  - Il metodo *getRelativeTo()* restituisce la posizione assoluta dell'estremo noto.

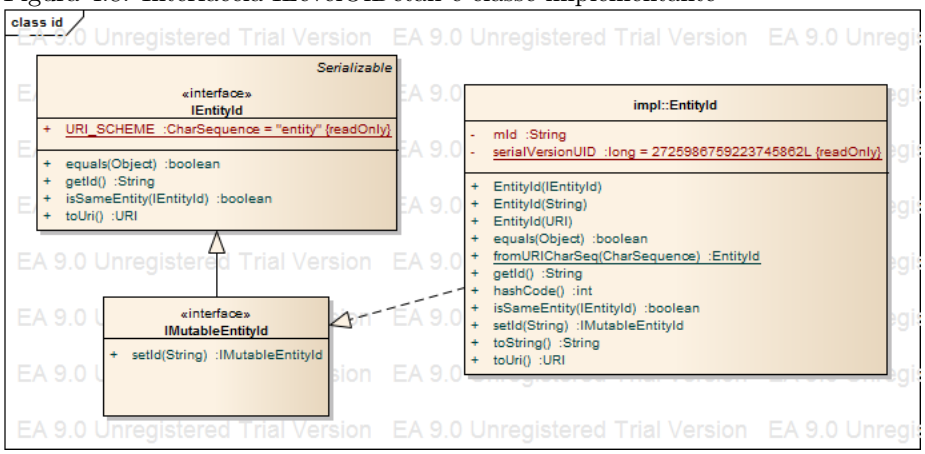
  - Il metodo *getLatitudeDelta()* restituisce la variazione di latitudine rispetto al punto di riferimento.

  - Il metodo *getLongitudeDelta()* restituisce la variazione di longitudine rispetto al punto di riferimento.

  - Il metodo *getRelativeDistance()* restituisce la distanza relativa dal punto di riferimento.

  - Il metodo *toAbsolute()* calcola la posizione assoluta.

Figura 4.8: Interfaccia ILevelOfDetail e classe implementante



4.1.2.7 Livello di dettaglio[2]

(*it.unibo.gciatto.thesis.lod.ILevelOfDetail*)

Rispetto ad un'entità, un'altra ha un livello di dettaglio che dipende da fattori come la distanza e come quest'ultima viene calcolata. A seconda del livello di dettaglio possono essere messe in atto politiche differenti su e.g. la quantità / qualità di informazioni da mostrare etc.

Trattasi di un concetto *passivo*, *atomico* e *immutabile*.

- Costanti

  LOD_INVISIBLE = 0

  L'entità viene percepita ma non va mostrata.

  LOD_POINT_OF_INTEREST = 1

  L'entità viene percepita e si è interessati solo alla posizione assoluta.

  LOD_GPS_LIMIT = 2

  Come LOD_POINT_OF_INTEREST, tuttavia l'entità si trova ad una distanza tale per cui il GPS comincia ad essere poco accurato.

[2] spesso abbreviato LOD

LOD_NEAR = 3

L'entità è vicina e viene percepita direttamente, non è detto però che se ne possa conoscere con esattezza la posizione.

LOD_INSIDE_INTERESTING_AREA = 4

Come LOD_NEAR, tuttavia la percezione non è diretta ma permessa dai landmark presenti nell'area di interesse.

LOD_BT_LIMIT = 5

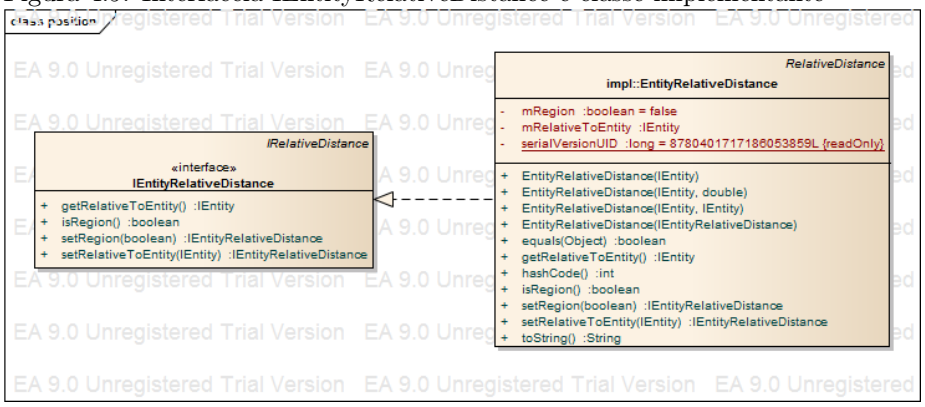
Come LOD_INSIDE_INTERESTING_AREA, tuttavia l'entità si trova ad una distanza tale per cui anche il BT comincia ad essere poco accurato.

LOD_INSIDE_VIEW = 6

L'entità è talmente vicina da poter essere vista.

- Metodi (vedi fig. 4.8)
  - Il metodo *equals()* confronta il livello di dettaglio corrente con un altro o direttamente con un valore numerico.
  - Il metodo *getValue()* fornisce il valore numerico corrispondente al livello di dettaglio.
  - Il metodo *isVisible()* verifica che il livello di dettaglio non sia LOD_INVISIBLE.
  - Il metodo *isInvisible()* verifica che il livello di dettaglio sia LOD_INVISIBLE.
  - Il metodo *isInsideView()* verifica che il livello di dettaglio sia LOD_INSIDE_VIEW.
  - Il metodo *isGPSLocalizationConvenient()* verifica che il livello di dettaglio sia minore o uguale a LOD_GPS_LIMIT.
  - Il metodo *isBTLocalizationConvenient()* verifica che il livello di dettaglio sia minore o uguale a LOD_BT_LIMIT.

Figura 4.9: Interfaccia IEntityRelativeDistance e classe implementante



– Il metodo *compareTo()* confronta il livello di dettaglio corrente con un altro sulla base dei rispettivi valori numerici riportando 0 se sono uguali, un numero positivo se il primo ha un dettaglio maggiore, un numero negativo altrimenti.

4.1.2.8 Distanza relativa ad un'entità

(*it.unibo.gciatto.thesis.entity.position.IEntityRelativeDistance*)

Estensione del concetto di distanza relativa. Qui la distanza non è semplicemente relativa ad una posizione assoluta, bensì ad un'entità (la quale possiede a sua volta una posizione assoluta, quindi l'estensione è abbastanza trasparente). Tale concetto può essere usato anche per riferirsi ad un'area circolare centrata su un'entità.

Trattasi di un concetto *passivo*, *atomico* e *mutabile*.

- Metodi (vedi fig. 4.9)

  – Il metodo *getRelativeToEntity()* restituisce l'entità a cui si riferisce la distanza.

- Il metodo *isRegion()* indica se bisogna considerare la distanza come regione o come distanza effettiva. Se *isRegion()* == *true* bisogna considerare la distanza come un valore approssimato da non interpretarsi in valore assoluto: essa indica semplicemente un'area particolare di raggio pari a *getDistance()*. Questa distinzione è importante perchè potrebbe nascere la necessità di considerare distanze "nominali": se per qualunque motivo non si riuscisse a calcolare la distanza da un beacon, ma questo fosse comunque visibile, se ne potrebbe dedurre una distanza massima (a partire dalla potenza di trasmissione) che è un'informazione incompleta e va differenziata da quella più precisa.
- Il metodo *setRegion()* decide se la distanza è da considerarsi una regione o meno.
- Il metodo *setRelativeToEntity()* imposta l'entità relativa a questa distanza.
- Il metodo *getRelativeTo()* restituisce la posizione assoluta dell'entità di cui sopra.

4.1.2.9 Ascoltatori di entità

(*it.unibo.gciatto.thesis.entity.listener.IEntityListener*)

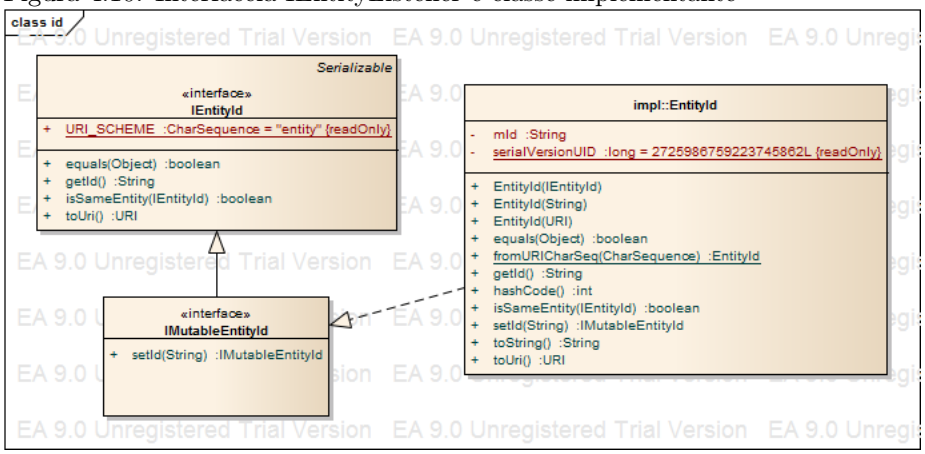
Permettono di intercettare il cambiamento di stato delle entità e applicare delle particolari politiche di conseguenza.

- Eventi intercettabili (vedi fig. 4.10)

  onAbsolutePositionFix invocato quando cambia la posizione assoluta di un'entità. Fornisce un riferimento all'entità su cui è stato invocato, uno alla nuova posizione ed uno alla vecchia.

  onAttributesChanged invocato quando avviene un cambiamento negli attributi di un'entità. Fornisce un riferimento all'entità.

Figura 4.10: Interfaccia IEntityListener e classe implementante



onCanSee invocato quando un'entità comincia a percepirne un'altra e può avere una stima della distanza relativa. Fornisce un riferimento all'entità che percepisce e uno alla distanza relativa all'entità percepita. (Quindi in pratica un riferimento ad entrambe le entità ed alla distanza).

onCantSeeAnymore invocato quando un'entità smette di percepirne un'altra. Fornisce un riferimento all'entità che percepisce ed uno alla percepita.

onDeviceAssociated invocato quando ad un'entità viene associato un dispositivo. Fornisce un riferimento all'entità ed al dispositivo.

onDeviceDissociated invocato quando da un'entità viene dissociato un dispositivo. Fornisce un riferimento all'entità ed al dispositivo.

onGettingCloser invocato quando un'entità si avvicina ad un'altra. Fornisce un riferimento all'entità che percepisce lo spostamento e uno alla distanza relativa all'entità che si avvicina. (Quindi in pratica un riferimento ad entrambe le entità ed alla distanza).

onGettingFarther invocato quando un'entità si allontana ad un'altra. Fornisce un riferimento all'entità che percepisce lo spostamento e uno

alla distanza relativa all'entità che si allontana. (Quindi in pratica un riferimento ad entrambe le entità ed alla distanza).

onLODChanged invocato quando cambia il livello di dettaglio di un'entità. Fornisce un riferimento all'entità, al nuovo livello di dettaglio ed anche al vecchio.

onRelativeDistanceAdded invocato quando viene aggiunta una distanza relativa all'entità. Fornisce un riferimento all'entità ed alla distanza.

onRelativeDistanceRemoved invocato quando viene rimossa una distanza relativa dall'entità. Fornisce un riferimento all'entità ed alla distanza.

4.1.2.10 Entità

(*it.unibo.gciatto.thesis.entity.IEntity*)

L'entità è il concetto cardine dell'intero sistema. A livello di interfaccia essa espone: un identificativo, un insieme di attributi, eventualmente anche vuoto; una collezione di dispositivi associati, eventualmente anche vuota; una collezione di distanze relative ad altre entità, eventualmente anche vuota; una posizione assoluta; un livello di dettaglio; la possibilià di aggiungere / rimuovere ascoltatori di callback relative ai cambiamenti di stato dell'entità stessa. Sono presenti in oltre molte funzionalità d'appoggio, costruite a partire dalle più elementari, al fine di rendere più agevole la stesura del codice.

L'entità è un concetto *attivo*, *composto* e *mutabile*.

(Il seguito fa riferimento alla fig. 4.11)

- Metodi relativi all'identità
  - Il metodo *getId()* restituisce l'identificativo dell'entità.

Figura 4.11: Interfaccia IEntity e classe implementante

  - I metodi *isSameEntity()* sono scorciatoie il confronto tra entità.

- Metodi relativi agli ascoltatori

  - I metodi *addEntityListener()* e *removeEntityListener()* servono rispettivamente per aggiungere e togliere un ascoltatore dall'entità.
  - Il metodo *clearEntityListener()* rimuove indiscriminatamente tutti gli ascoltatori.

- Metodi relativi alla distanza

  - Il metodo *addRelativeDistance()* aggiunge una distanza relativa. Se è già presente una distanza relativa alla stessa entità, viene semplicemente aggiornato il valore corrispondente alla distanza.
  - Il metodo *removeRelativeDistance()* rimuove la distanza relativa alla stessa entità di quella passata in ingresso.
  - Il metodo *hasDistanceRelativeTo()* controlla che esista una distanza relativa per l'entità in ingresso.
  - Il metodo *getRelativeDistances()* recupera tutte le distanze relative presenti nell'entità corrente.
  - Il metodo *getDistanceRelativeTo()* recupera l'eventuale distanza relativa all'entità in ingresso, *null* altrimenti.

- Metodi relativi ai dispositivi

  - I metodi *associateDevice()* e *dissociateDevice()* servono rispettivamente per associare e dissociare un dispositivo all'entità. Associare un dispositivo (ad esempio un beacon) ad un'entità significa che quandunque venga percepito il dispositivo (ad esempio tramite MAC) il sistema interpreti la cosa come la percezione dell'intera entità.
  - Il metodo *getAssociatedDevices()* la collezione di dispositivi associati all'entità.

  - Il metodo *matchesDevice()* controlla che il dispositivo in ingresso sia associato all'entità.

- Metodi relativi agli attributi

  - Il metodo *getAttributes()* recupera l'insieme di attributi dell'entità. *N.B.:* IEntity estende IAttributeSet: tutti i metodi ereditati da tale interfaccia, se invocati, hanno lo stesso effetto dell'invocazione degli stessi sull'oggetto restituito da *getAttributes()*.

- Metodi relativi alla posizione

  - Il metodo *getAbsolutePosition()* restituisce la posizione assoluta dell'entità.
  - Il metodo *getRelativePositionTo()* restituisce la posizione relativa dell'entità in ingresso rispetto a quella su cui è invocato il metodo.
  - Il metodo *getRelativePositionFrom()* restituisce la posizione relativa dell'entità su cui è invocato il metodo rispetto a quella in ingresso.
  - I metodi *move()* cambiano la posizione assoluta dell'entità, uno usando una posizione relativa, uno usando una posizione assoluta.

- Metodi relativi al livello di dettaglio

  - Il metodo *getLOD()* recupera il livello di dettaglio dell'entità.
  - I metodi *setLOD()* impostano il livello di dettaglio dell'entità.

- L'implementazione Entity (*it.unibo.gciatto.thesis.entity.impl.Entity*)

  - È thread-safe: l'entità è il nucleo del sistema, essa verrà facilmente manipolata da più thread concorrenti.
  - Si occupa di gestire le callback riguardanti le distanze relative:

* L'invocazione di *removeRelativeDistance()* su un'istanza provoca la chiamata delle callback *onRelativeDistanceRemoved()* e *onCantSeeAnymore()* su tutti gli ascoltatori registrati a patto che l'operazione di rimozione vada a buon fine;
* L'invocazione di *addRelativeDistance()* su un'istanza provoca la chiamata della callback *onRelativeDistanceAdded()* su tutti gli ascoltatori registrati. Successivamente, se *non* era già presente una distanza relativa alla stessa entità viene chiamata la callback *onCanSee()*, altrimenti viene valutata la variazione rispetto al valore precedentemente presente: a patto che sia in valore assoluto superiore alla costante DIST_THRESHOLD[3], se si tratta di una variazione positiva viene è chiamata la callback *onGettingFarther()* altrimenti *onGettingCloser()*.
* Con queste callback è possibile intercettare gli avvicinamenti / allontanamenti ed applicare politiche in merito agli spostamenti relativi.

4.1.2.11 Note sul dominio

- Esiste anche l'interfaccia ILandmark che modella un landmark. Non è altro che un'entità con in più un metodo, isLandmark(), che ritorna sempre *true*. Un entità landmark è inoltre vincolata ad avere sempre l'attributo it.unibo.gciatto.thesis.entity.landmark.ILandmark.isLandmark settato a *true*.
- Esiste anche l'interfaccia IBeaconDevice che modella un beacon. Esso espone, oltre ai metodi ereditati da IBeaconDevice anche dei metodi per recuperare UUID, Major e Minor.
- Esistono anche delle interfacce IMutable* per rendere mutabili i corrispondenti concetti immutabili aggiungendo dei modificatori, così da rendere più

[3] Valore attuale 1 cm

pratica la programmazione. Queste sono: IMutableDevice, IMutableBeaconDevice, IMutableEntityId, IMutableLevelOfDetail, IMutableAbsolutePosition, IMutableRelativeDistance, IMutableRelativePosition.

- Per ogni concetto esiste un'implementazione di base nel corrispondente package *".impl"*. Le implementazioni sono mutabili.
- L'implementazione della classe Entity è thread-safe.
- Per i landmark esiste un decoratore, LandmarkDecorator, che permette di estendere le normali entità.

4.1.2.12 Classi d'utilità

Esistono anche una seria di classi *Utils per rendere più agevole lo sviluppo:

- La classe LandmarkUtils contiene metodi statici per manipolare le collezioni di entità contenenti landmark e l'utilissimo isLandmark che deduce da un'oggetto di tipo IEntity se è o meno un landmark.
- La classe MathUtils.geo contiene utilities per la conversione da coordinate terrestri a metri e viceversa. Esse si basano sulla libreria SimpleLatLng 1.3.0[22] distribuite sotto licenza Apache 2.0[5].
- La classe MathUtils.rand contiene utilities per generazione di numeri casuali interi e a virgola mobile.
- La classe MathUtils.alg contiene una serie di funzioni statiche che effettuano operazioni vettoriali su n-uple di numeri reali.
- La classe MathUtils.triangles contiene utilities per quando riguarda i triangoli, nella fattispecie:
  - La funzione MathUtils.triangles.trilaterate() effettua la trilaterazione: conoscendo le coordinate xyz di tre punti P1, P2 e P3 e le loro

distanze da un punto ignoto P, è possibile ottenere le coordinate di quest'ultimo. La funzione assume che le distanze così come le coordinate possano essere imprecise e quindi tenta sempre di restituire un risultato, anche quando geometricamente non ci sarebbe soluzione.

– La classe MathUtils.triangles.carnot sfrutta il Teorema di Carnot.

4.2 Progetto platform-independent

In quanto segue, come preannunciato, non si fanno ipotesi sulla piattaforma sulla quale il codice verrà eseguito. Prima di usare una funzionalità fornita dal framework di Java si è avuta cura di verificare che non fosse peculiarità di tale linguaggio.

In quanto segue è importante tenere presente che il sistema funziona per mezzo di un *server centrale* i cui dettagli saranno chiariti in seguito. L'interazione con il server è incapsulata dentro un'interfaccia, *IDataServerHandler*, descritta in seguito ed implementata nella parte platform-dependent. I componenti che desiderino interagire direttamente sul server devono fare riferimento a questa interfaccia e null'altro, così facendo possono mantenersi platform-independent.

4.2.1 Il concetto di Self

Ogni dispositivo che partecipa al sistema, server a parte, è associato ad un'entità. Tale entità, nell'ambito del dispositivo che "la mantiene in vita" è riferita come Self. Il Self è un'entità particolare in quanto, ad esempio, riceve le informazioni su posizione assoluta e distanze dai sensori e non dal server. Nel caso di un dispositivo osservatore, il Self non viene visualizzato. L'utilizzatore umano può intervenire solo sugli attributi del self, non su quelli di altre entità. Per il Self il concetto di livello di dettaglio è irrilevante. Il concetto di Self verrà approfondito nella sezione relativa al progetto platform-dependent.

4.2.2 Knowledge

Tutta la "conoscenza del sistema", espressione con cui si intende l'insieme di costanti e parametri di configurazione vari che governano il funzionamento del sistema, è contenuta nella classe *it.unibo.gciatto.thesis.knowledge.K.*

Essa contiene una serie di classi innestate, una per ogni categoria di costanti:

K.addresses contiene gli indirizzi degli eventuali componenti tra cui il server.

K.env contiene costanti d'ambiente come il nome del software, l'UUID dell'applicazione, etc.

K.numbers contiene costanti numeriche

K.timeouts contiene i vari timeout che regolano l'interazione sulla rete.

4.2.3 I manager

Il sistema svolge sostanziamente quattro macro-attività:

- Tenere traccia degli identificativi di tutte le entità partecipanti al sistema stesso e dei vari dispositivi ad esse associate.
- Tenere traccia delle posizioni assolute e delle distanze relative delle varie entità.
- Tenere traccia dei vari attributi di tutte le entità e fornirli in base alla necessità.
- "Scannerizzare" la realtà alla ricerca di marker che palesino la presenza di entità.

Si è pensato di delegare ogni macro-attività ad un apposito sotto-sistema. I sotto-sistemi vengono chiamati *manager*.

L'insieme dei principali manager definisce un *contesto*.

Tale impostazione presenta molteplici vantaggi: all'interno delle implementazioni dei manager si possono applicare varie politiche che rimangono trasparenti alle entità. Anche l'interazione con un server centrale è trasparente alle entità: in linea teorica è perfettamente possibile creare una versione del sistema che sfrutti tutt'altra maniera di scambiare dati. La struttura appena descritta è riassumibile nel seguente schema:

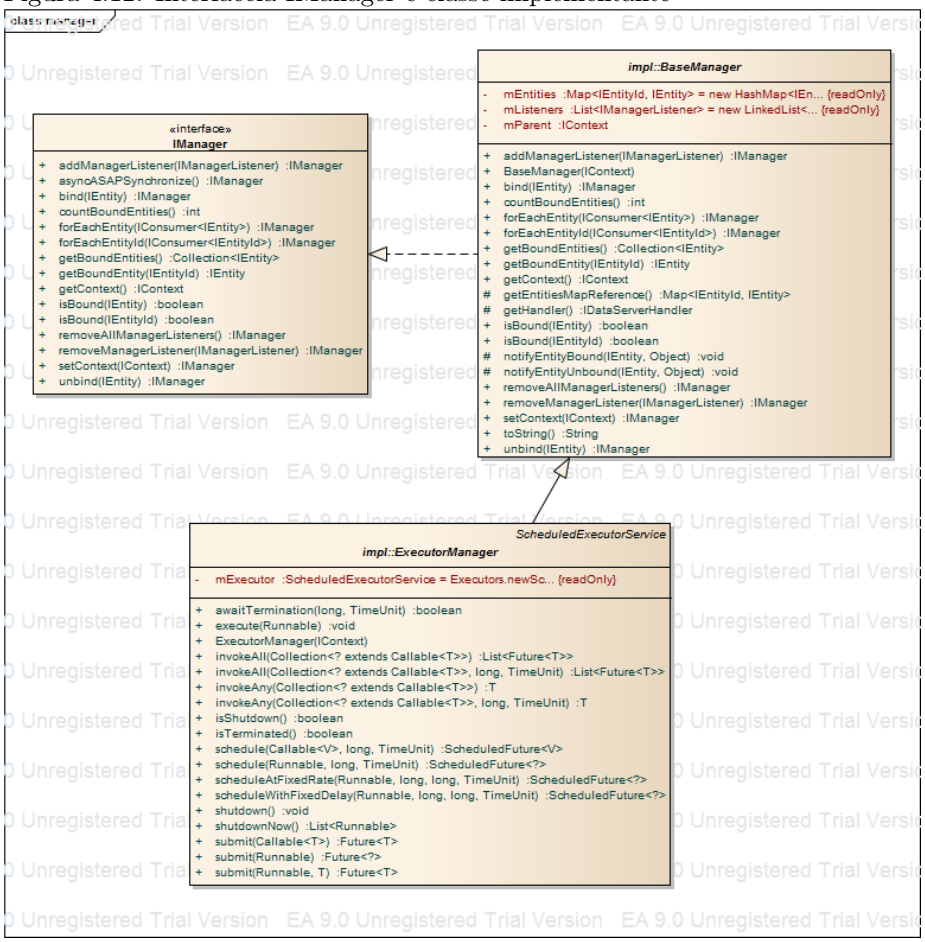
4.2.3.1 IManager

(*it.unibo.gciatto.thesis.manager.IManager*)

I manager specializzano tutti la stessa interfaccia IManager che raggruppa alcune funzionalità in comune.

- Metodi (vedi fig. 4.12)
  - I metodi *addManagerListener()* e *removeManagerListener()* servono rispettivamente per aggiungere e togliere ascoltatori per le callback relative all'attività del manager.
  - Il metodo *removeAllManagerListeners()* rimuove tutti gli ascoltatori.
  - Il metodo *asyncASAPSynchronize()* ha senso solo in quei manager che per la loro attività necessitano di un'interazione col server centrale: esso serve a richiedere che avvenga un aggiornamento il più presto possibile.
  - Il metodo *bind()* lega un'entità al manager: i manager si occupano solo delle entità precedentemente bindate.

Figura 4.12: Interfaccia IManager e classe implementante



  - Il metodo *unbind()* libera il manager da ogni responsabilità nei confronti dell'entità.
  - Il metodo *countBoundEntities()* restituisce il numero di entità bindate al manager.
  - Il metodo *getBoundEntities()* restituisce la collezione delle entità bindate.
  - Il metodo *getBoundEntity()* restituisce, se bindata, l'entità con l'id in ingresso.
  - Il metodo *getContext()* restituisce un riferimento al contesto padre del manager corrente, mentre il metodo setContext() permette di cambiarlo.
  - Il metodo *isBound()* serve a verificare che un'entità sia bindata al manager.

- IManagerListener permette di intercettare il binding e l'unbinding delle entità.

4.2.3.2 BaseManager

(*it.unibo.gciatto.thesis.manager.impl.BaseManager*)

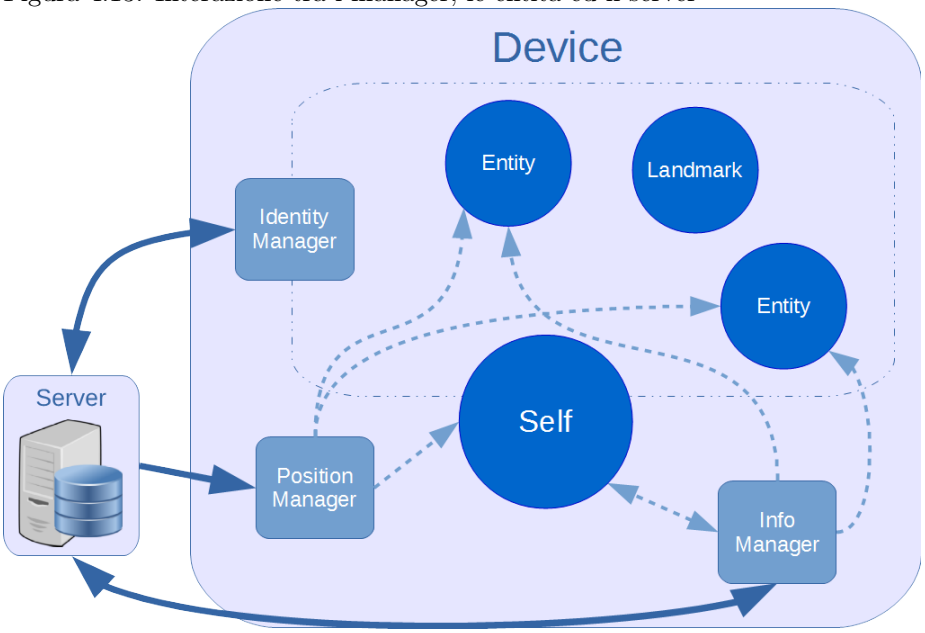
La classe astratta BaseManager implementa tutte le funzionalità di base esposte nell'interfaccia IManager. Gli accessi alle entità sono thread-safe.

4.2.3.3 ExecutorManager

(*it.unibo.gciatto.thesis.manager.impl.ExecutorManager*)

La classe astratta ExecutorManager estende BaseManager ed implementa ScheduledExecutorService[20] delegando le funzionalità esposte da quest'ultima ad uno

Figura 4.13: Interazione tra i manager, le entità ed il server



ScheduledThreadPoolExecutor[21]. Ciò implica che i manager che estendono tale classe possono eseguire dei task in maniera asincrona e/o metterne in coda per un'esecuzione futura o periodica.

4.2.3.4 L'architettura dei manager

Nel seguito sono descritti i tre manager principali. La loro interazione è schematizzata nella figura 4.13.

Il significato dello schema viene chiarito nei paragrafi successivi.

4.2.3.5 L'Identity Manager

L'Identity Manager è per molti versi il manager più importante. Esso è responsabile di aspetti cruciali del sistema quali:

- mantenersi informato quali entità stiano attualmente partecipando al sistema e quali siano invece uscite;

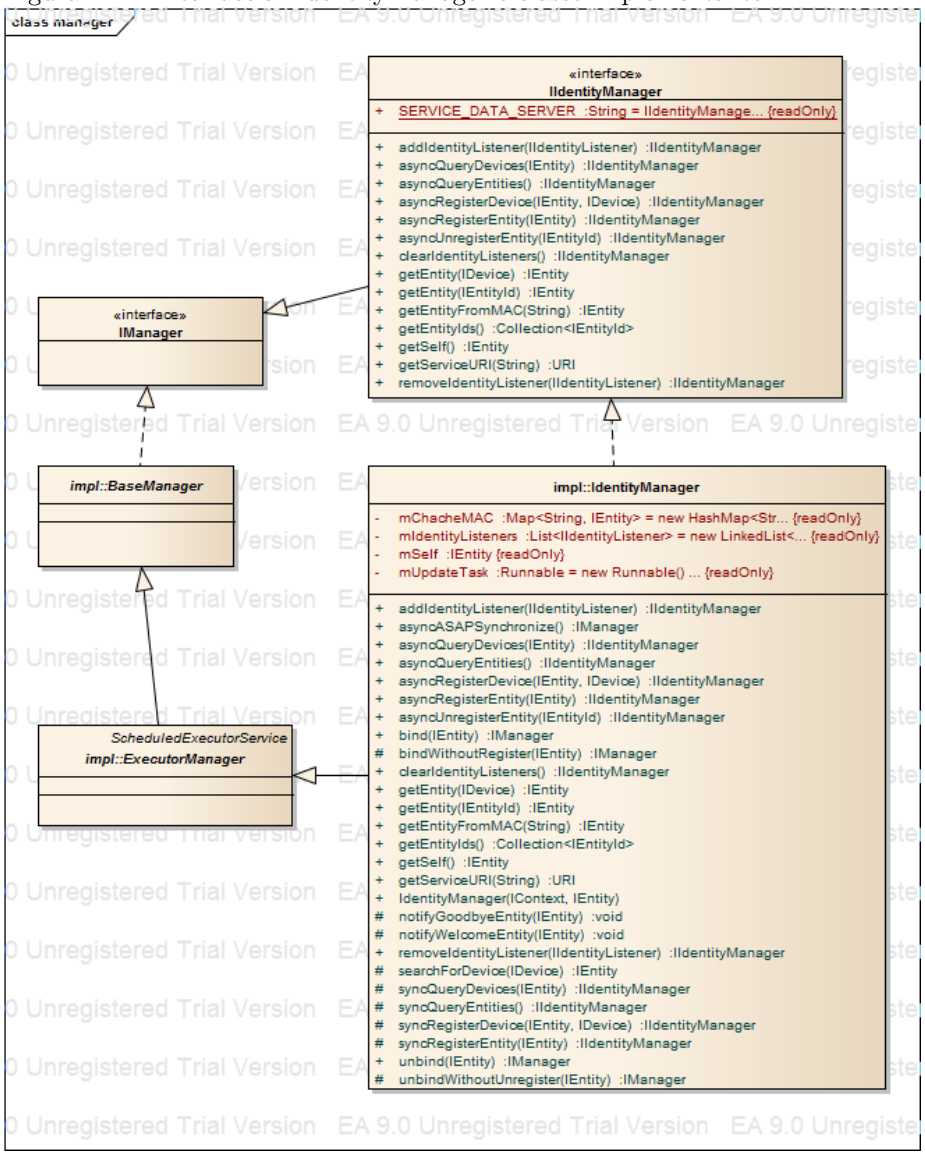
- mantenersi informato sui dispositivi associati alle varie entità;
- mantenere un riferimento al Self e fornire il principale punto di accesso ad esso;
- esporre la possibilità di recuperare il riferimento ad un'entità dato il suo identificativo;
- esporre la possibilità di recuperare il riferimento ad un'entità dato il suo MAC;
- esporre la possibilità di recuperare il riferimento ad un'entità dato un suo dispositivo;
- mantenersi informato sull'URI più corretto per il server centrale ed esporlo per chi desiderasse conoscerlo;
- permettere di intercettare l'entrata e l'uscita di un'entità nel sistema.

  Da notare che l'Identity Manager è un punto critico del sistema: gestisce le informazioni di base sulle quali il resto del sistema va ad operare. Queste non necessitano di una frequenza di aggiornamento troppo elevata: è lecito immaginare che a meno di una fase di setup iniziale, l'entrata / uscita delle entità nel sistema non avvenga continuamente né tantomeno la modifica dei device associati ad una di esse. Le informazioni scambiate sono a grana abbastanza larga: si è interessati il più delle volte all'intero blocco di id, meno spesso ad uno solo.

Un Identity Manager deve implementare l'interfaccia IIdentityManager in figura 4.14.

- Metodi (vedi fig. 4.14)
  - I metodi *addIdentityListener()* e *removeIdentityListener()* rispettivamente aggiungono e tolgono ascoltatori di callback che permettono

Figura 4.14: Interfaccia IIdentityManager e classe implementante



di intercettare l'ingresso e l'uscita di entità nel sistema (Notare che si tratta di concetti diversi rispetto al binding / unbinding di entità rispetto alla specifica istanza di IIdentityManager).

  - Il metodo *asyncQueryEntities()* asincronamente contatta il server e scopre gli id delle entità attualmente "online", aggiorna i dati del Self sul server, se ci sono nuove entità entrate nel sistema si informa anche sui rispettivi dispositivi.
  - Il metodo *asyncQueryDevices()* contatta asincronamente il server per informarsi sulla lista di device associati ad un'entità, aggiornando di conseguenza l'entità locale.
  - I metodi *asyncRegisterDevice()* e *asyncRegisterEntity()* registrano rispettivamente un device ed un'entità sul server.
  - Il metodo *asyncUnregisterEntity()* fa uscire un'entità dal sistema.
  - I metodi *getEntity*()* permettono di recuperare il riferimento ad un'entità data un'informazione relativa (id, device, MAC)
  - Il metodo *getSelf()* permette di ottenere un riferimento al Self. Tale metodo è cruciale per il corretto funzionamento del sistema.
  - Il metodo *getServiceURI()* fornisce l'URI() del server.
  - Il metodo *bind()*, oltre a bindare l'entità al manager, ne provoca la registrazione sul server. Analogamente il metodo *unbind()* provoca anche l'uscita dell'entità dal sistema. Questo comportamento particolare dei due metodi è peculiarità dell'Identity Manager.

- L'implementazione IdentityManager (*it.unibo.gciatto.thesis.manager.impl.IdentityManager*):
  - estende ExecutorManager;
  - delega l'interazione ultima con il server ad un IDataServerHandler;
  - l'istanza appena creata schedula un task con lo stesso compito di *asyncQueryEntities()* con

* un ritardo di K.timeouts.TIMEOUT_IDS_UPDATE_INITIAL_-DELAY secondi[4],
* periodicità di K.timeouts.TIMEOUT_IDS_UPDATE_PERIOD secondi[5].

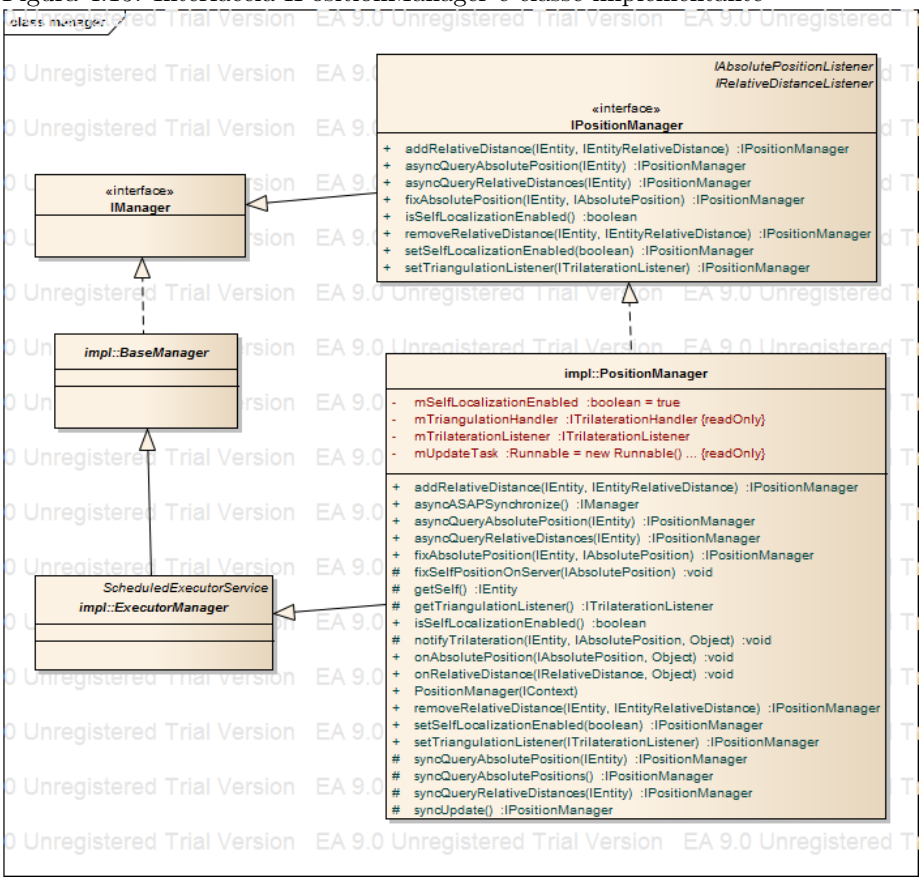
4.2.3.6 Il Position Manager

Non meno importante è il Position Manager, certamente quello con più responsabilità:

- gestisce l'interazione col GPS;
- gestisce l'interazione coi sensori di distanza / prossimità;
- deduce la posizione assoluta del Self applicando internamente le politiche che ritiene opportune:
  - possibilmente trilaterando la posizione in presenza di sufficienti informazioni
    N.B.: il fatto che la posizione venga dal GPS o sia dedotta in altri modi è del tutto trasparente per il Self;
- deduce le distanze relative del Self dalle altre entità applicando internamente le politiche che ritiene opportune;
- mantiene il server informato sulla posizione del Self e sulle distanze di questo dalle altre entità;
- si informa continuamente sulla posizione delle entità partecipanti al sistema e sulle mutue distanze.

[4]Valore attuale: 1s
[5]Valore attuale: 30s

Figura 4.15: Interfaccia IPositionManager e classe implementante



Da notare che il Position Manager è un punto critico del sistema. Gestisce delle informazioni a grana fine con una frequenza di aggiornamento tendenzialmente elevata ed una velocità di invecchiamento ancora maggiore.

Un Position Manager deve implementare l'interfaccia IPositionManager in figura 4.15.

- Metodi (vedi fig. 4.15)
  - Il metodo *addRelativeDistance()* aggiunge una distanza relativa all'entità in ingresso, posto che questa sia bindata al manager, il metodo removeRelativeDistance() la rimuove.

  - Il metodo *asyncQueryAbsolutePosition()* avvia una query che porterà all'aggiornamento della posizione per l'entità in ingresso.
  - Il metodo *asyncQueryRelativeDistances()* avvia una query che porterà all'aggiornamento delle distanze per l'entità in ingresso.
  - Il metodo *fixAbsolutePosition()* modifica la posizione assoluta di un'entità, posto che questa sia bindata al manager.
  - Il metodo *isSelfLocalizationEnabled()* verifica che sia abilitato il posizionamento per il Self mentre setSelfLocalizationEnabled() lo abilita / disabilita.
  - Il metodo *setTrilaterationListener()* imposta un ascoltatore di callback relative alla trilaterazione: trattandosi di una feature sperimentale si vuole delegare al livello più alto la fruizione dell'informazione "posizione trilaterata".
  - I metodi *onAbsolutePosition()* ed *onRelativeDistance()* sono callback invocate rispettivamente da un Absolute Position Handler e da un Relative Distance Handler, componenti a grana più fine, di cui si discuterà nel seguito, cui sono delegate le interazioni con GPS e BLE (e che sono per loro natura platform-dependent). Servono per intercettare la ricezione dei dati sensoriali ed applicare le politiche del caso.

- L'implementazione PositionManager (*it.unibo.gciatto.thesis.manager.impl.PositionManager*)
  - estende ExecutorManager;
  - delega l'interazione ultima con il server ad un IDataServerHandler;
  - l'istanza appena creata schedula un task con il compito di aggiornare le posizioni assolute e le distanze relative delle entità bindate con:
    * un ritardo di K.timeouts.TIMEOUT_POSITION_UPDATES_INITIAL_DELAY secondi[6],

[6] Valore attuale: 1s

* periodicità di K.timeouts.TIMEOUT_POSITION_UPDATES_-*PERIOD* secondi[7];

– delega l'interazione con il GPS ad un IAbsolutePositionHandler e da questo viene informato di una nuova posizione del Self tramite il metodo *onAbsolutePosition()* che, nella specifica implementazione, aggiorna la posizione del Self ed informa il server;

– delega l'interazione con i beacons ad un IRelativeDistanceHandler e da questo viene informato dell'aggiunta / rimozione di una distanza tra Self ed un'altra entità tramite il metodo *onRelativeDistance()* che, nella specifica implementazione, aggiunge / rimuove la distanza al /dal Self, informa il server della variazione e se possibile avvia una trilaterazione.

– delega il computo della posizione trilaterata ad un TrilaterationHandler.

– l'interazione del Position Manager con gli gli handler suoi sottoposti è riassunta in figura 4.16

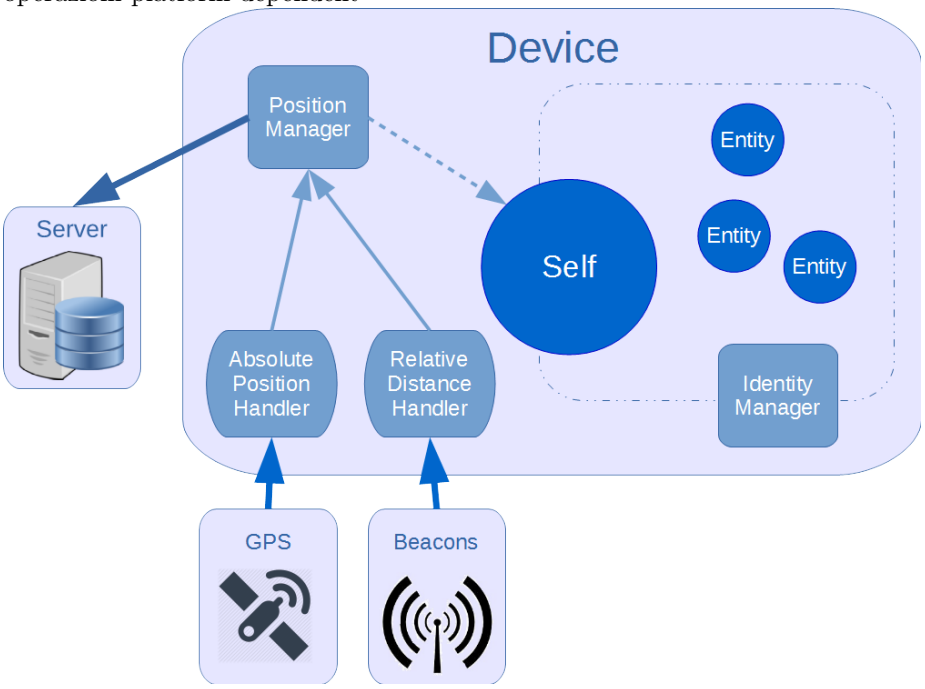
4.2.3.7 L'Information Manager

L'Information manager è il terzo ed ultimo componente che necessita di interazione con il server, certamente quello con le responsabilità meno critiche. Esso deve:

- mantenersi informato sugli attributi delle varie entità partecipanti al sistema;
- informare il server delle variazioni degli attributi del Self.

[7] Valore attuale: 2s

Figura 4.16: Interazione tra il Position Manager e gli handler cui delega le operazioni platform-dependent

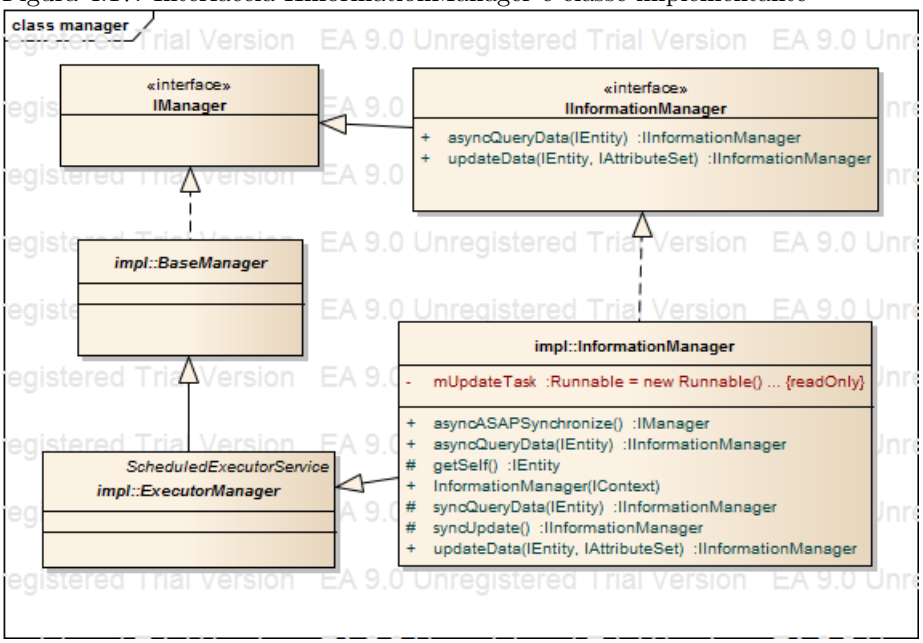


Da notare che l'Information Manager gestisce informazioni che, a meno di applicazioni particolari, hanno grana potenzialmente grossa, non richiedono aggiornamenti frequentissimi e non invecchiano troppo velocemente.

Un Information Manager deve implementare l'interfaccia IInformationManager in figura 4.17.

- Metodi (vedi fig. 4.17)
  - Il metodo *asyncQueryData()* avvia una query che porterà all'aggiornamento degli attributi dell'entità in ingresso.
  - Il metodo *updateData()* aggiorna gli attributi dell'entità in ingresso, a patto che questa sia bindata al manager.
- L'implementazione InformationManager (*it.unibo.gciatto.thesis.manager.InformationManager*)

Figura 4.17: Interfaccia IInformationManager e classe implementante



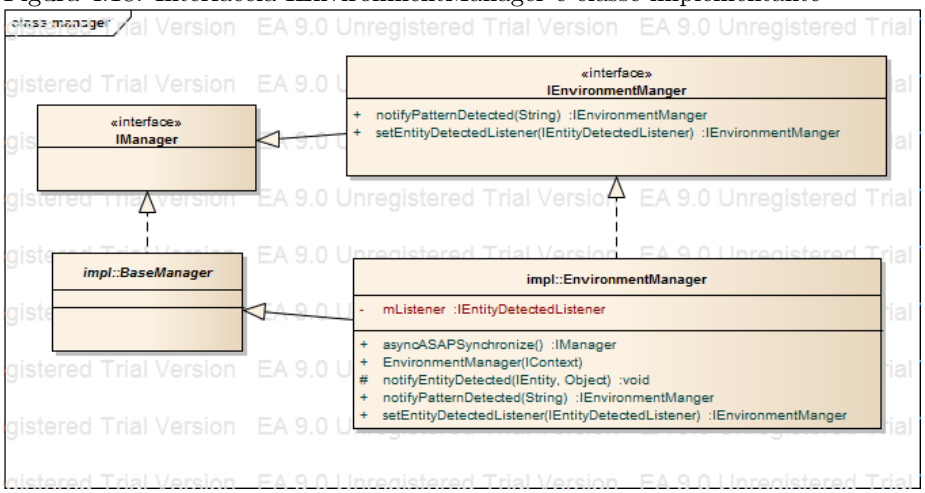
- estende ExecutorManager;
- delega l'interazione ultima con il server ad un IDataServerHandler;
- l'istanza appena creata schedula un task con il compito di aggiornare gli attributi delle entità bindate con:
  - un ritardo di K.timeouts.TIMEOUT_ATTRIBUTES_UPDATE_INITIAL_DELAY secondi[8],
  - periodicità di K.timeouts.TIMEOUT_ATTRIBUTES_UPDATES_PERIOD secondi[9];

4.2.3.8 L'Environment Manager

L'Environment Manager uniforma la ricerca di pattern nella realtà aumentata all'impostazione mager-mediated del resto del sistema. Esso ha la responsabilità di:

[8] Valore attuale: 5s
[9] Valore attuale: 5 min = 300s

Figura 4.18: Interfaccia IEnvironmentManager e classe implementante

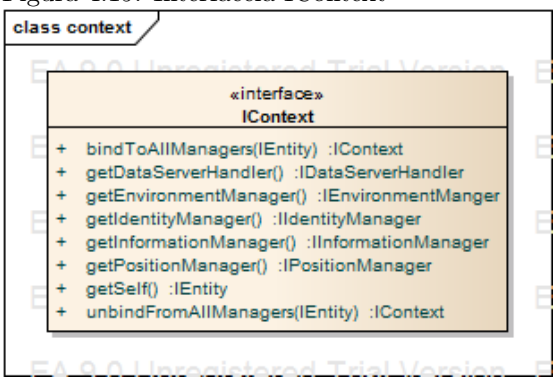


- gestire la scansione della scena alla ricerca di pattern da riconoscere;
- se viene riconosciuto un pattern associato ad un'entità bindata, notificare la cosa al livello superiore.

Un Environment Manager deve implementare l'interfaccia IEnvironmentManager in figura 4.18.

- Metodi (vedi fig. 4.18)
  - Il metodo *notifyPatternDetected()* è invocato per notificare al manager l'individuazione di un pattern il cui contenuto è rappresentato in forma di stringa.
  - Il metodo *setEntityDetectedListener()* permette di impostare un ascoltatore di callback invocate quando viene riconosciuta un'entità bindata a seguito di una chiamata a *notifyPatternDetected()*.
- L'implementazione EnvironmentManager
  - estende BaseManager;
  - delega ricerca effettiva di pattern al motore di realtà aumentata sottostante;

Figura 4.19: Interfaccia IContext



  - si aspetta di ricevere, tramite il metodo *notifyPatternDetected()*, l'URI contenente l'id di un'entità codificato in forma di stringa.

4.2.4 Il Contesto

Come anticipato, a run-time l'insieme dei manager costituisce un contesto. Per l'esattezza un contesto è un oggetto che si occupa di:

- creare le istanze dei manager e configurarle;
- referenziare tutti i manager per un accesso rapido e condiviso (si ricorda che ogni manager ha un riferimento al contesto-padre, quindi può, indirettamente, accedere alle funzionalità degli altri manager);
- fornire delle scorciatoie per operazioni ricorrenti col solito scopo di permettere uno stile di programmazione più scorrevole.

Il contesto è definito nell'interfaccia IContext in figura 4.19.

- Metodi (vedi fig. 4.19)
  - I metodi *get*Manger()* forniscono i riferimenti ai manager.
  - Il metodo *getSelf()* è una scorciatoria per getIdentityManager().getSelf().

  - I metodi *bindToAllManagers()* e unbindFromAllManagers() permettono di bindare a / unbindare da tutti i manager l'entità in ingresso con una sola istruzione.
  - Il metodo *getDataServerHandler()* recupera un riferimento all'oggetto responsabile dell'interazione con il server, operazione necessaria per eventualmente "scavalcare" i manager ed interagire direttamente con il server.

- L'implementazione è platform-dependet e verrà analizzata nell'apposita sezione, tuttavia è stata realizzata un'implementazione parziale platform-independent nella classe Context:
  - Essa implementa tutti i metodi descritti nell'interfaccia, senza aggiungere politiche di sorta.
  - L'entità Self viene fornita tramite costruttore.
  - Tutti i manager vengono istanziati.
  - L'Identity Manager è configurato affinché ogni nuova entità che entra a far parte del sistema tramite l'interazione con il server venga automaticamente bindata agli altri manager.

4.2.5 Gli Handler

Così come le macro-attività sono affidate a dei manager, questi ultimi dividono il carico di lavoro tra i cosiddetti *handler*, ovvero entità cui è affidata una singola micro-attività. Le micro-attività del sistema sono:

- interazione con il server
- interazione col GPS, o, in generale, con un componente in grado di fornire la posizione assoluta

- interazione coi bluetooth beacon, o, in generale, con un componente in grado di fornire delle distanze relative

- trilaterazione.

Le prime tre sono attività intrinsecamente platform-dependent, tuttavia le interfacce possono essere definite sin da ora.

4.2.5.1 IHandler

(*it.unibo.gciatto.thesis.handler.IHandler*)

Tutti gli handler devono implementare l'interfaccia IHandler in figura 4.20.

Come si nota non è richiesto molto ad un oggetto per dirsi un handler. L'implementazione di base, *BaseHandler*, implementa semplicemente questi due metodi.

4.2.5.2 IAbsolutePositionHandler & IRelativeDistanceHandler

Sono definite a questo livello le interfacce di base per i componenti che si occuperanno di fornire al position manager posizioni assolute e distanze relative (vedi fig. 4.21)

Essi permettono soltanto di aggiungere degli ascoltatori di callback. Si ricorda che l'interfaccia IPositionManager estende sia IAbsolutePositionHandler che IRelativeDistanceHandler. Il funzionamento di questi handler dovrebbe essere quindi chiaro: recuperano i dati sensoriali e li forniscono al manager di competenza.

Figura 4.20: Interfaccia IHandler

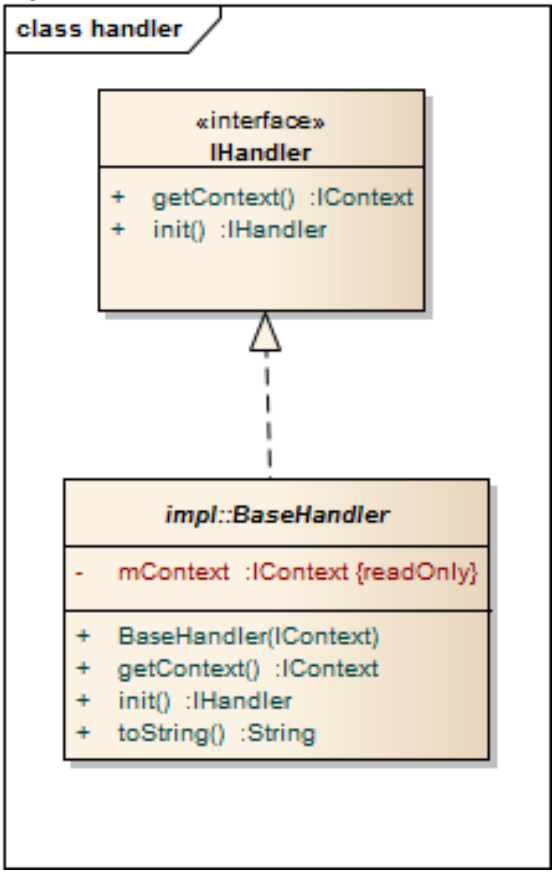


Figura 4.21: Interfacce IAbsolutePositionHandler e IRelativeDistanceHandler

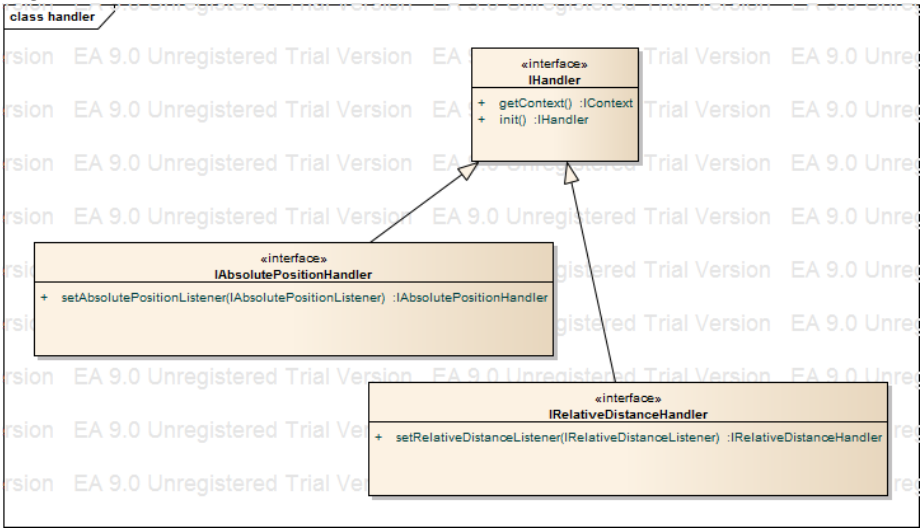
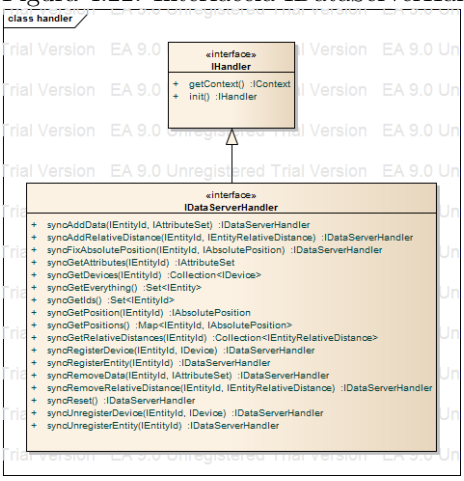


Figura 4.22: Interfaccia IDataServerHandler



4.2.5.3 DataServer Handler

Passiamo ad analizzare un altro componete critico del sistema: il DataServer Handler. Esso è il componente responsabile di interagire con il server mediante un meccanismo di RPC. Trascendendo dai dettagli implementativi platform-dependent, in questa fase ci si limita a descrivere l'interfaccia e l'implementazione astratta.

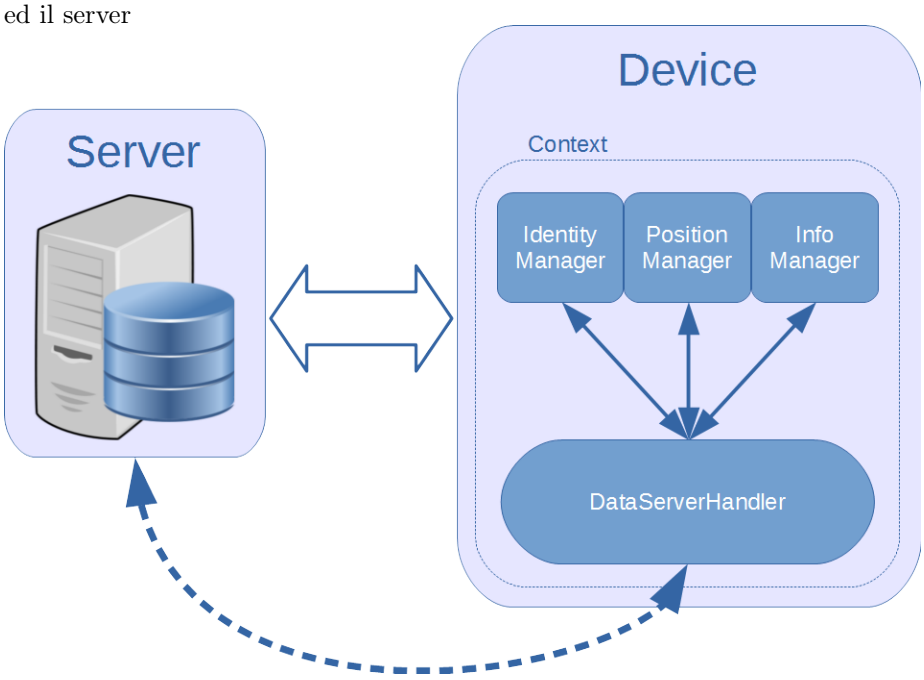
- Eccezioni
  - L'eccezione ProtocolException è generata dai metodi di cui sotto a fronte di un qualunque problema di comunicazione.
- Metodi

  N.B.: I metodi sono tutti marchiati col prefisso *sync*: sono metodi sincroni e pertanto bloccanti. L'impiego di semantiche asincrone è lasciata agli utilizzatori di tale classe.
  - Il metodo *syncAddData()* registra sul server gli attributi per l'entità con l'id in ingresso, mentre *syncRemoveData()* è il suo duale.

  - Il metodo *syncAddRelativeDistance()* registra sul server la distanza relativa per l'entità con l'id in ingresso, mentre *syncRemoveRelativeDistance()* è il suo duale.
  - Il metodo *syncFixAbsolutePosition()* registra sul server la distanza relativa per l'entità con l'id in ingresso.
  - Il metodo *syncGetAttributes()* recupera dal server gli attributi per l'entità con l'id in ingresso.
  - Il metodo *syncGetDevices()* recupera dal server i dispositivi associati all'entità con l'id in ingresso.
  - Il metodo *syncGetEverything()* recupera tutte le informazioni su tutte le entità registrare sul server.
  - Il metodo *syncGetIds()* recupera tutti gli id di tutte le entità registrate sul server.
  - Il metodo *syncGetPosition()* recupera la posizione assoluta dell'entità con l'id in ingresso.
  - Il metodo *syncGetPositions()* recupera tutte le posizioni assolute di tutte le entità registrate sul server e le restituisce sotto forma di mappa IEntityID-IAbsolutePosition.
  - Il metodo *syncGetRelativeDistances()* recupera tutte le distanze relative dell'entità con l'id in ingresso.
  - Il metodo *syncRegisterDevice()* associa, sul server, un dispositivo all'entità con l'id in ingresso, mentre *syncUnregisterDevice()* è il suo duale.
  - Il metodo *syncRegisterEntity()* registra l'id in ingresso sul server, mentre *syncUnregisterEntity()* è il suo duale.
  - Il metodo *syncReset()* cancella tutte le informazioni dal server.

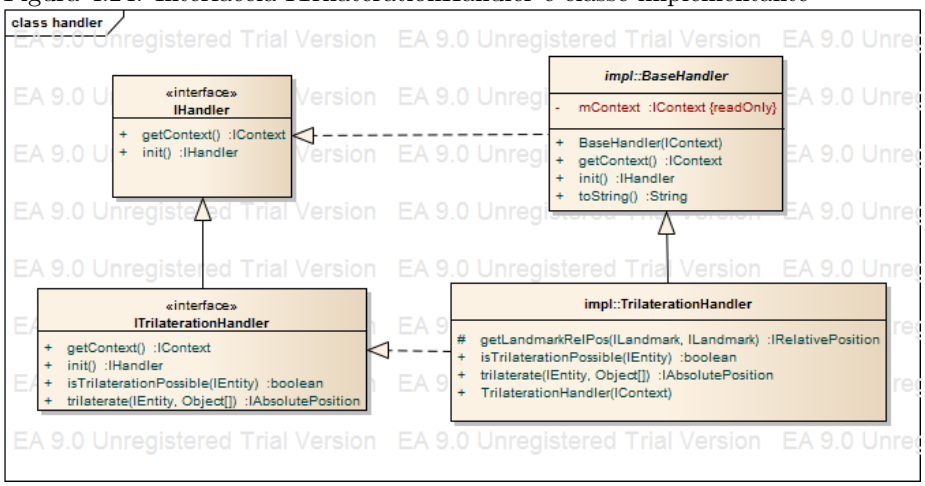
- L'implementazione DataServerHandler
  (*it.unibo.gciatto.thesis.handler.impl.DataServerHandler*)

Figura 4.23: Il DataServerHandler media l'interazione tra i componenti locali ed il server



- Estende BaseHandler
- Ipotizza di interagire con un server HTTP che funzioni come descritto in seguito.
- Si rende indipendente dalla specifica implementazione del client HTTP usanso le interfacce wrapper IHttpRequest ed IHttpResponse.
- Delega il marshalling e l'unmarshalling ai metodi astratti *readJsonArray()* e *readJsonObject()*.
- Delega la generazione e l'invio della richiesta ai metodi *generateRequest()* e *executeRequest()*.
- Le implementazioni finali devono pertanto solo preoccuparsi di implementare questi quattro metodi.
- La maniera in cui questo componente si interpone tra gli altri ed il server è schematizzato in figura 4.23.

Figura 4.24: Interfaccia ITrilaterationHandler e classe implementante



4.2.5.4 Trilateration Handler

(*it.unibo.gciatto.thesis.handler.ITrilaterationHandler*)

L'interfaccia ITrilaterationHandler non espone molte funzionalità (vedi fig. 4.24).

- Metodi (vedi fig. 4.24)
  - Il metodo *isTrilaterationPossible()* verifica se è possibile triangolare la posizione dell'entità in ingresso, verificando che, tra le entità da cui siano note le sue distanze relative vi siano almeno 3 landmark.
  - Il metodo *trilaterate()* restituisce la posizione assoluta trilaterata per l'entità in ingresso e prova a copiare nel parametro di output il riferimento ai landmark impiegati per trilaterare.

N.B.: Il vantaggio di triangolare a questo livello, ovvero usando distanze relative ed entità anziché valori *double* e beacons, è evidente: si può trilaterare indipendentemente da come vengono fornite le distanze relative, poco importa se queste provengono dal server, dai beacon o da altre ed ancora inesplorate sorgenti sensoriali.

- L'imlementazione TrilaterationHandler
  (*it.unibo.gciatto.thesis.handler.impl.TrilaterationHandler*)
  - Estende BaseHandler
  - Svolge la trilaterazione utilizzando una versione leggermente modificata della procedura descritta in Python nella pagina «Trilateration using 3 latitude and longitude points and 3 distances»[24] a sua volta basato sulla descrizione che Wikipedia dà della trilaterazione alla pagina Trilateration[32].
  - La computazione si basa sulla conversione di coordina LLA in coordinate cartesiane ECEF[10] e viceversa. Le procedure per tali conversioni sono descritte nel documento «Datum Transformations of GPS Positions»[10] che fa a sua volta riferimento al sistema WGS84[11] il quale descrive:
    * Il modello del pianeta: un ellissoide di cui sono noti semiasse maggiore[12], semiasse minore[13], schiacciamento, eccentricità etc.
    * La terna cartesiana di riferimento: zero degli assi nel centro di massa della Terra, asse z rivolto verso il polo Nord, assi x e y giacenti sul piano equatoriale, x passante per il meridiano fondamentale, y rivolto verso l'Asia.
  - La parte algoritmica è contenuta nella classe MathUtils e nelle sue classi innestate, segue un estratto:

```
public static double[] trilaterate(
                final double[] p1, final double[] p2,
                final double[] p3, final double dP1,
                final double dP2, final double dP3) {
```

[10] Earth-Centered, Earth-Fixed
[11] World Geodetic System 1984
[12] volgarmente "raggio equatoriale"
[13] volgarmente "raggio polare"

```
// sottrazione
final double[] p12 = alg.sub(p2, p1);
final double[] p13 = alg.sub(p3, p1);

// norma euclidea
final double d = alg.norm(p12);

// moltiplicazione per lo scalare 1/d
final double[] ex = alg.div(p12, d);

// prodotto scalare
final double i = alg.dot(ex, p13);
final double[] temp = alg.sub(p13, alg.mult(ex, i));

// ottiene un versore
final double[] ey = alg.vers(temp);

// prodotto vettoriale
final double[] ez = alg.cross(ex, ey);
final double j = alg.dot(ey, p13);
final double x = (dP1 * dP1 - dP2 * dP2 + d * d) / (d * 2);
final double y = (dP1 * dP1 - dP3 * dP3 + i * i + j * j) /
        (j * 2) - ((i / j) * x);
final double absZ = Math.sqrt(
        Math.abs(dP1 * dP1 - x * x - y * y));

// prodotto per uno scalare
final double[] xex = alg.mult(ex, x);
final double[] yey = alg.mult(ey, y);
final double[] zez = alg.mult(ez, absZ);
```

```
        // sommatoria
        final double[] res = alg.sum(p1, xex, yey, zez);
        return res;
}
```

4.3 Il layer di presentazione

Com'è stato già anticipato il sistema richiede che dei dati attraversino la rete. Si pone quindi la neccessita di serializzare e deserializzare i dati. Si è scelto di utlizzare JSON[12] per la presentazione e HTTP[11] per il trasporto. Le motivazioni sono molteplici. L'impiego di un protocollo testuale, com'è noto, aumenta la visibilità e la semplicità di (de)serializzazione. Inoltre e soprattutto sono tecnologie "di uso comune" per cui effettuare il porting su piattaforme diverse non richiede stravolgimenti del progetto; praticamente tutti i dispositivi mobile sono compatibili con entrambe le tecnologie e esiste grande abbondanza di librerie già fatte, il che, dato che lo sviluppo di un sistema distribuito è un requisito ma non uno scopo di questo progetto, ha permesso di abbattere i tempi di sviluppo.

Le librerie impiegate sono le sequenti:

- org.json con licenza open-source[14]. Trattasi della stessa libreria integrata nelle SDK di Android: per poter usare la versione più aggiornata si è reso necessario rinominare il package in org.json2.
- Android Asynchronous Http Client[2], un client Http asincrono basato un meccanismo di callback analogo a quello di Android. Viene fornito anch'esso sotto i termini della licenza Apache 2.0[5].

Per ogni concetto del dominio per cui si è posta la necessità di (de)serializzazione è stata creata un'apposita implementazione JSON-compatibile.

Sono state introdotte due interfacce base:

```
public interface IJsonArrayConvertible {
        JSONArray toJsonArray();
}

public interface IJsonObjectConvertible {
        JSONObject toJsonObject();
}
```

ed è stata fissata la convenzione che tutte le classi che estendano IJsonArrayConvertible abbiano un costruttore che accetti un JSONArray così come tutte le classi che estandano IJsonObjectConvertible devono avere un costruttore che accetta un JSONObject.

Ecco un elenco delle classi del layer di presentazione. Esse hanno tutti i costruttori delle rispettive classi base più quello per deserializzare:

JsonAttributeSet estende *AttributeSet* e implementa *IJsonObjectConvertible*

JsonAbsolutePosition estende *AbsolutePosition* e implementa *IJsonObjectConvertible*

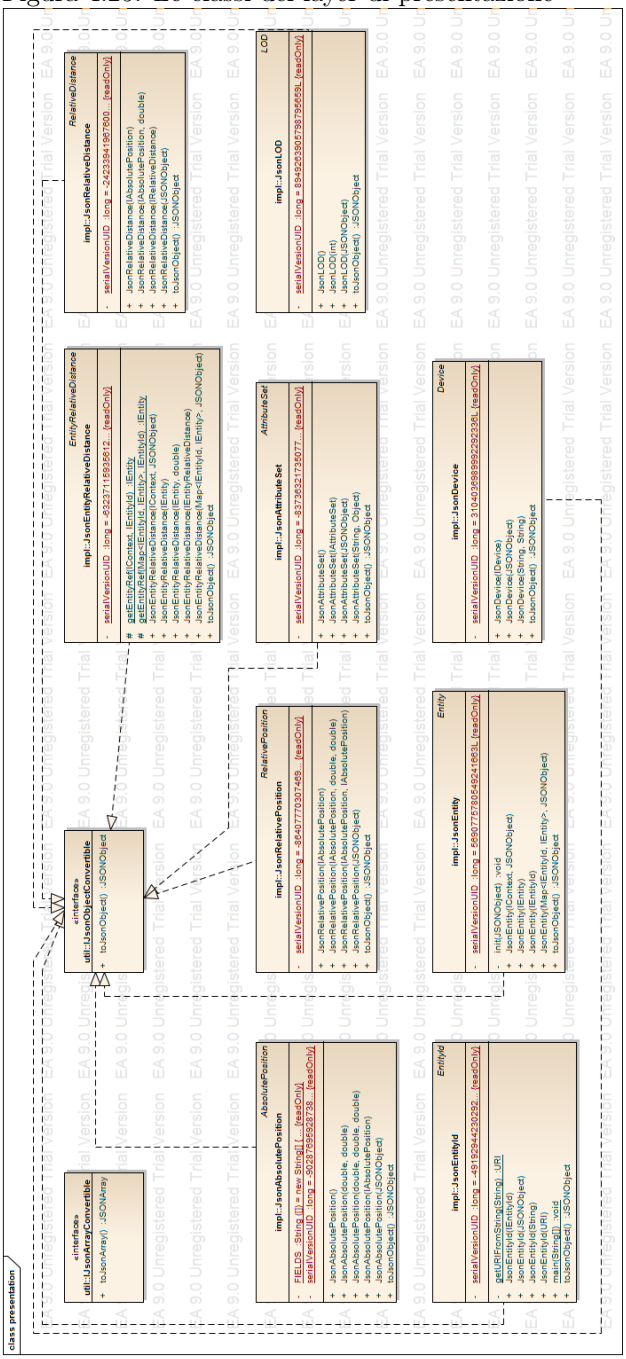
JsonRelativePosition estende *RelativePosition* e implementa *IJsonObjectConvertible*

JsonRelativeDistance estende *RelativeDistance* e implementa *IJsonObjectConvertible*

JsonEntityId estende *EntityId* e implementa *IJsonObjectConvertible*

JsonDevice estende *Device* e implementa *IJsonObjectConvertible*

Figura 4.25: Le classi del layer di presentazione



JsonEntityRelativeDistance estende *EntityRelativeDistance* e implementa *IJsonObjectConvertible*. Il costruttore per la deserializzazione richiede anche una mappa IEntityId-IEntity. Questo perchè il metodo toJSONObject() della stessa classe serializza solo l'id dell'entità cui la distanza è relativa, onde per cui, in fase di deserializzazione, occorre recuperare il riferimento all'entità dato il suo id.

JsonEntity estende *Entity* e implementa *IJsonObjectConvertible*. Il costruttore per la deserializzazione richiede anche una mappa per eventualmente deserializzare le *JsonEntityRelativeDistance* interne.

Notare che, grazie all'ereditarietà, una volta deserializzati, gli oggetti si comportano esattamente come se fossero stati creati localmente.

4.4 Il server

Il server centrale è, come abbiamo visto, un componente fondamentale del progetto. Fin'ora si è stabilito che si tratta di un server HTTP e che i client vi interagiscono per mezzo di un'interfaccia fissata. Passiamo ora ad analizzare il comportamento del server: esso ricerve richieste secondo il protocollo HTTP e ad esse fornisce delle risposte. L'interazione è senza stato: ogni messaggio HTTP è indipendente dagli altri e nella stessa maniera viene gestito e servito. Ciò ovviamente non significa che le richieste HTTP non modifichino lo stato interno del server o dei client ma semplicemente che la singola richiesta HTTP contiene tutte le informazioni per essere servita.

Il server risponde a richieste HTTP strutturate in una certa maniera. Si guarda prima al nome del metodo (i metodi di interesse sono GET, POST e DELETE), poi si cercano i parametri "What" e "Entity". Quest'ultimo, se presente, è seguito da un URI che rappresenta un identificativo di entità solo quando la

combinazione valore-di-What e metodo lo rendono necessario: infatti è proprio questa combinazione a stabilire il funzionamento del server.

- Richieste GET (What e Entity entrambi opzionali)
  - Assenza del parametro What

    Viene generata una pagina HTML che mostra tutte le entità registrare sul server ed i rispettivi dati.
  - What = "Data"

    Se Entity è l'id di un'entità registrata viene inclusa nella risposta una rappresentazione come JSON Object degli attributi dell'entità, altrimenti viene generato un errore 404 not found.
  - What = "Everything"

    Viene inclusa nella risposta una rappresentazione (al più vuota) come JSON Array di tutte le entità registrate sul server, in cui ogni elemento è un JSON Object codificante un'entità.
  - What = "Ids"

    Viene inclusa nella risposta una rappresentazione (al più vuota) come JSON Array di tutti gli id di tutte le entità registrate sul server, in cui ogni elemento è un JSON Object codificante un id.
  - What = "Position"

    Se Entity è l'id di un'entità registrata viene inclusa nella risposta una rappresentazione come JSON Object della posizione assoluta dell'entità, altrimenti viene generato un errore 404 not found.
  - What = "Positions" (al plurale)

    Viene inclusa nella risposta una rappresentazione (al più vuota) come JSON Array di tutti gli id di tutte le entità registrate sul server e delle corrispettive posizioni assolute: ogni elemento è a sua volta un JSON Array di due elementi: il primo è un JSON Object codificante l'id, il secondo la posizione assoluta.

  - What = “Distances”

    Se Entity è l’id di un’entità registrata viene inclusa nella risposta una rappresentazione come JSON Array delle distanze relative dell’entità in cui ogni elemento è un JSON Object codificante una distanza relativa, altrimenti viene generato un errore 404 not found.

  - What = “Devices”

    Se Entity è l’id di un’entità registrata viene inclusa nella risposta una rappresentazione come JSON Array dei dispositivi associati all’entità in cui ogni elemento è un JSON Object codificante un device, altrimenti viene generato un errore 404 not found.

  - Qualunque altro valore del parametro What

    Viene generato un errore 400 bad request

- Richieste POST (What e Entity entrambi obbligatori)

  Quando non esiste sul server un’entità con id uguale al valore di Entity ma la richiesta ha comunque senso, viene generata e registrata automaticamente una nuova entità così che la richiesta non fallisca.

  - Assenza del parametro What

    Viene generato un errore 417 expectation failed.

  - What = “Data”, il valore di Entity è un id valido ed il body della richiesta è la codifica JSON di un IAttributeSet

    Gli attributi vengono decodificati e aggiunti all’entità sul server.

  - What = “Distance”, il valore di Entity è un id valido ed il body della richiesta è la codifica JSON di una IEntityRelativeDistance

    La distanza viene decodificata ed aggiunta all’entità sul server.

  - What = “Device”, il valore di Entity è un id valido ed il body della richiesta è la codifica JSON di una IDevice

    Il device viene decodificato ed aggiunto all’entità sul server.

  - What = "Position", il valore di Entity è un id valido ed il body della richiesta è la codifica JSON di una IAbsolutePosition
    La posizione viene decodificata ed aggiunta all'entità sul server.
  - What = "Id", il valore di Entity è un id valido
    Viene creata una nuova entità con l'id in ingresso ed aggiunta al server.
  - What = "Distance",
  - Qualunque altro valore del parametro What
    Viene generato un errore 400 bad request

- Richieste DELETE (What obbligatorio e Entity opzionale)
  - Assenza del parametro What
    Viene generato un errore 417 expectation failed.
  - What = "Everything"
    Viene cancellata ogni entità dal server.
  - What = "Id", il valore di Entity è un id valido
    Viene cancellata l'entità con l'id in ingresso.
  - What = "Device", il valore di Entity è un id valido ed il body della richiesta è la codifica JSON di un IDevice
    Viene cancellato dall'entità con l'id in ingresso qualunque device faccia match con quello decodificato dal body.
  - What = "Data", il valore di Entity è un id valido ed il body della richiesta è la codifica JSON di un IAttributeSet
    Viene cancellato dall'entità con l'id in ingresso qualunque attributo sia contenuto anche nell'insieme decodificato dal body.
  - What = "Distance", il valore di Entity è un id valido ed il body della richiesta è la codifica JSON di una IEntityRelativeDistance
    Viene cancellato dall'entità con l'id in ingresso la distanza relativa alla stessa entità di quella decodificata.

- Qualunque altro valore del parametro What

  Viene generato un errore 400 bad request

4.4.1 L'implementazione del server

L'interfaccia appena descritta non impone molti vincoli sulla struttura interna del server. Quest'ultimo potrebbe usare, per mantenere le informazioni, un database, un file di testo o la semplice memoria prima senza che la cosa riguardi minimamente il resto del progetto.

Dato che come si è detto questo sistema è un mezzo e non un fine, si è cercato di non spendere troppo tempo nell'implementazione effettiva del server.

Ne è stata realizzata una versione minimale tramite una servlet che utilizza la memoria primaria per mantenere le informazioni. Java EE permette infatti di gestire agilmente richieste e risposte HTTP. Avendo già implementato il layer di presentazione lato client, la costruzione del server è stata molto rapida.

L'ultima versione di questa implementazione è in esecuzione all'indirizzo:

`http://thesis-gciatto.rhcloud.com/server/data`

4.5 Progetto platform-dependent

Arrivati a questo punto bisogna integrare la ferrea impostazione di interfacce e classi minuziosamente definita nei punti precedenti con le librerie che permettono l'interazione effettiva con server e sensori, ovvero la parte platform-dependent del progetto.

In quanto segue l'ipotesi di fondo è di trovarsi in ambiente Android e di avere accesso quindi alle SDK del sistema.

4.5.1 Il Self

Il Self viene implementato nell'omonima classe del package *it.unibo.gciatto.thesis.android.entity.impl*. Esso non si differenzia particolarmente dalle normali entità se non per il fatto che:

- è un singleton: può esserci una sola istanza per device;
  - l'id è automaticamente settato con il seriale del device (Build.SERIAL),
  - viene automaticamente aggiunto un attributo ATTR_USER settato con il nome dell'utente di Android (Build.USER),
  - viene automaticamente aggiunto un SelfDevice di tipo TYPE_ANDROID, nome pari al seriale del dispositivo, ip e mac settati in base ai valori attuali sulla macchina.

4.5.2 Gli handler e l'interazione coi sensori

Implementando gli handler si chiude il cerchio.

4.5.2.1 BaseAndroidHandler

(*it.unibo.gciatto.thesis.android.handler.impl.BaseAndroidHandler*)

Trovandosi su Android, gli handler possono estendere la classe BaseAndroidHandler, che, rispetto a BaseHandler, ha in più il riferimento all'Application Context (qui inteso nella semantica data da Android) necessario per interagire con i servizi del sistema operativo.

4.5.2.2 AbsolutePositionHandler

(*it.unibo.gciatto.thesis.android.handler.impl.AbsolutePositionHandler*)

Estende BaseAndroidHandler ed implementa IAbsolutePositionHandler. Il funzionamento è molto semplice: nel costruttore viene richiesto ad Android un'istanza di LocationManager. Il metodo *init()* avvia due richieste di aggiornamento sulla posizione, una che usi come provider la rete e l'altra il GPS:

```
mLocationManger.requestLocationUpdates(
        LocationManager.NETWORK_PROVIDER,
        K.timeouts.TIMEOUT_NW_MIN_PERIOD,
        K.numbers.NUM_NW_MIN_DISTANCE,
        mLocationListener
);
mLocationManger.requestLocationUpdates(
        LocationManager.GPS_PROVIDER,
        K.timeouts.TIMEOUT_GPS_MIN_PERIOD,
        K.numbers.NUM_GPS_MIN_DISTANCE,
        mLocationListener
);
```

Le due richiesete usano lo stesso ascoltatore, hanno un periodo minimo di 1 s, ed una distanza minima di 0 m.

Il listener ha un solo compito: inoltrare la Location ricevuta dai provider all'eventuale IAbsolutePositionListener registrato nell'handler, previa conversione in IAbsolutePosition che avviene per mezzo della classe-adattatore *AndroidAbsolutePosition*.

Con questa semplice classe si permette al Position Manager di conoscere la posizione assoluta effettiva del dispositivo a patto che l'utente abiliti la Geolocalizzazione. Se così non dovesse essere semplicemente il LocationListener non

riceverebbe aggiornamento alcuno lasciando il sistema privo di informazioni sulla posizione ma perfettamente funzionante.

4.5.2.3 RelativeDistanceHandler

(*it.unibo.gciatto.thesis.android.handler.impl.RelativeDistanceHandler*)

Estende BaseAndroidHandler ed implementa IRelativeDistanceHandler. Il funzionamento anche in questo caso è abbastanza semplice: nel costruttore viene richiesta un'istanza di BeaconManager, nel metodo init() l'handler viene bindato al beacon manager di cui prima. Ciò si traduce nell'avvio del cosiddetto "*Ranging*": il sistema attiva la ricerca di beacon advertisements; colleziona tutti i beacon percepiti nell'arco di un secondo, stimando la distanza dagli stessi e notificando di volta in volta i beacon visibili al RelativeDistanceHandler. Quest'ultimo, internamente, tiene traccia dei beacon visibili: quando uno inizia ad essere percepito diventa visibile e, se è associato ad un'entità esistente, è notificata una nuova distanza relativa a tale entità al position manager. Analogamente, quando un beacon non viene più percepito, è notificata al position manager la rimozione di un'entità.

In ogni istante è possibile recuperare la collezione di beacon attualmente visibili mediante l'apposito metodo *getVisibileBeacons()*. Ciò permette ai livelli superiori di vedere anche beacons non associati ad alcuna entità e di appliare politiche di conseguenza (ad esempio, in una fase di setup del sistema, si potrebbe associare in maniera automatica al Self di un tablet il beacon più vicino).

4.5.2.4 AndroidDataServerHandler

(*it.unibo.gciatto.thesis.android.handler.impl.AndroidDataServerHandler*)

Estende DataServerHandler e ne completa l'implementazione occupandosi dell'invio di richieste e della ricezione di risposte HTTP. Si appoggia ad un oggetto

di tipo SyncHttpClient che, come si deduce dal nome, funziona in maniera sincrona: l'invio della richiesta provoca la sospensione del thread fino alla ricezione della risposta. Se l'interazione va a buon fine si tenta di ricostruire un JSON Object o JSON Array a partire dal testo eventualmente contenuto nel body della risposta. Sarà poi responsabilità dell'oggetto che ha fatto la richiesta decodificare il JSON.

4.5.3 Il contesto di sistema

Il contesto di sistema, realizzato nella classe SystemContext, è il punto di accesso a tutte le funzionalità sin qui descritte. La classe SystemContext estende Context e si occupa di:

- istanziare il Self, i manager e configurarli come farebbe la classe Context,
- istanziare gli handler, configurarli e avviarli

Il SystemContext è un singleton, il che porta alcuni vantaggi, tra cui in primis il fatto che una rotazione dello schermo in Android (con conseguente riavvio dell'activity) non provoca particolari complicazioni.

4.5.4 Observed

Quando un dispositivo "gioca" il ruolo di observed il suo scopo è quello di lasciarsi localizzare ed individuare. A tal fine, come spiegato, sincronizza la propria posizione con il server e mostra un marker su schermo così che questo possa essere identificato. A livello implementativo si è scelto di utilizzare i codici QR per i motivi già evidenziati in fase di analisi: maggiore apertura del sistema, facilità di implementazione, tecnologia consolidata.

Il codice QR visualizzato dall'osservato deve contenere la codifica testuale dell'URI che rappresenta l'id dell'entità osservata *e null'altro*. Questo vincolo permette un'implementazione semplice ed agevole: i codici QR vengono generati on-demand per mezzo della libreria QRGen[18], distribuita sotto licenza Apache 2.0[5] e basata sul progetto open-source ZXING[27]. L'immagine è generata sulla base codificando l'id del Self come codice QR e viene mostrata al centro dell'Activity. Quest'ultima mantiene un riferimento al SystemContext che, in background, si occupa di mantenere la sincronia col resto del sistema.

4.5.5 Observer

Quando un dispositivo "gioca" invece il ruolo di observer, la situazione si complica molto dato che bisogna mettere in campo la realtà aumentata.

Come accennato Metaio si occupa da solo di capire come il dispositivo osservatore sia disposto nello spazio (i.e. quale "porzione di mondo" è inquadrata) e di ubicare correttamente nella scena le Geometrie[14] associate alle entità partecipanti. La gestione delle geometrie non è banalissima, richiede infatti che queste vengano create con l'ingresso di un'entità nel sistema e distrutte con la loro l'uscita; è importante mantenere un riferimento facilmente accedibile ad ognuna in quanto, essendo modelli 3D, devono subire trasformazioni in risposta al movimento della telecamera, oltre al fatto che risentono del movimento delle entità corrispondenti e del cambiamento del loro livello di dettaglio.

La forte impostazione "a callback" che si è delineata sinora semplifica di molto queste attività, tuttavia si è sentità la necessità di definire un altro manager per la gestione delle geometrie:

```
public interface IGeometryManager extends IManager {
        IManager bind(IEntity entity,
```

[14] In Metaio una geometria, interfaccia IGeometry, è un qualunque modello 3D da ubicare nella scena. La posizione è settabile, tra l'altro, per mezzo di coordinate LLA.

```
            IGeometry geometry);
        List<IGeometry> getGeometries();
        IGeometry getGeometry(IEntity entity);
        IGeometry getGeometry(IEntityId id);
        IGeometryManager setGeometry(IEntity entity,
            IGeometry value);
        IGeometryManager setGeometry(IEntityId id,
            IGeometry value);
}
```

(*it.unibo.gciatto.thesis.android.observer.manager.IGeometryManager*)

- Metodi
  - Il metodo *bind()* con due parametri serve a bindare un'entità ed associarle una geometria pre-esistente in una sola istruzione.
  - Il metodo *getGeometries()* restituisce una lista di tutte le geometrie registrare nel manager.
  - I metodi *getGeometry()* restituiscono la geometria associata all'entità o all'id in ingresso.
  - I metodi *setGeometry()* associano una geometria ad un'entità.
- L'implementazione GeometryManager
  - Estende BaseManager
  - È thread-safe

L'introduzione di un nuovo manager porta inevitabilmente all'estensione del concetto di contesto che lo comprenda:

```
public interface IARContext extends IContext {
        IGeometryManager getGeometryManager();
}
```

La classe implementate ARContextDecorator non è altro che un decorate di IContext il cui metodo *getGeometryManager()*:

- se il contesto da decorare è già un IARContext, restituisce il Geometry Manager originale;
- altrimenti un Geometry Manager appositamente creato in fase di inizializzazione del decoratore.

Ciò detto, è possibile descrivere il funzionamento della classe ObserverActivity. All'avvio essa si occupa di:

- far partire il motore di realtà aumentata in modalità "Localization-based AR";
- creare un SystemConcext e decorarlo affinché diventi un IARContext;
- configurare il contesto affinché l'ingresso di una nuova entità si traduca nella generazione di una nuova IGeometry e conseguente aggiunta al "pool" contenuto nel Geometry Manager e l'uscita di un'entità nella rimozione della suddetta geometria;
- configurare il Self affinché gli avvicinamenti verso / allontanamenti da altre entità vengano notificati all'utente;
- configurare il Self affinchè un'avvicinamento ad un'altra entità sotto una certa soglia provochi l'innesco di una scansione alla ricerca di codici QR;
- configurare il contesto affinché un'eventuale identificazione di codici QR si traduca nella comunicazione del contenuto all'Environment Manager;
- configuare l'Environment Manager affinché il riconoscimento di un'entità venga notificato all'utente;

- configuare il contesto affinché lo spostamento delle entità (inteso come variazione di posizione assoluta) si traduca nel conseguente aggiornamento delle posizioni delle geometrie;
- predisporre un pulsante con il quale l'utente umano possa manualmente avviare la scansione per ricercare codici QR.

4.6 Il prototipo

Un prototipo è distribuito per mezzo di un'unica app, descritta nel seguito, che permette di scegliere se essere un osservatore od un osservato. Essa richiede Android 4.3 (API level 18) o superiore, versione in cui è stato introdotto il supporto al BLE. Inoltre, per fornire un testing valido, un dispositivo deve essere equipaggiato di Wi-Fi, possibilmente connessione dati, GPS, supporto hardware al BLE, fotocamera posteriore, e sensori vari correttamente funzionanti. Segue una descrizione delle Activity principali dell'app, si omette tuttavia la descrizione tecnica della view del sistema su Android data l'estraneità dell'argomento ai fini del documento.

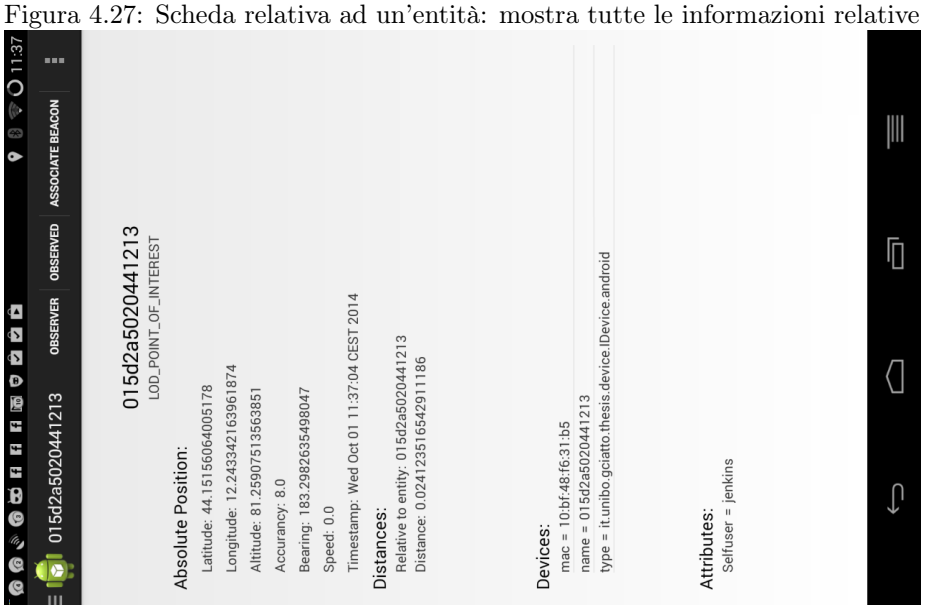
4.6.1 Dashboard Activity

L'Activity Dashboard è la prima cosa con la quale l'utente entra in contatto. Nata con il compito di essere d'aiuto nel testare l'interazione con il server, è passata sostanzialmente inalterata nelle fasi dello sviluppo: al primo avvio essa crea un'istanza di SystemContext e, quando questa si sincronizza con il server, l'Activity mostra nella barra laterale tutte le entità attualmente partecipi del sistema in un elenco verticale.

La selezione di una voce dell'elenco porta alla visualizzazione della scheda relativa: vengono mostrate tutte le informazioni relative all'entità in "tempo reale", nel senso che se essa si muove si possono osservare le coordinate variare.

Figura 4.26: Barra laterale: mostra tutte le entità facenti parte il sistema in un dato istante

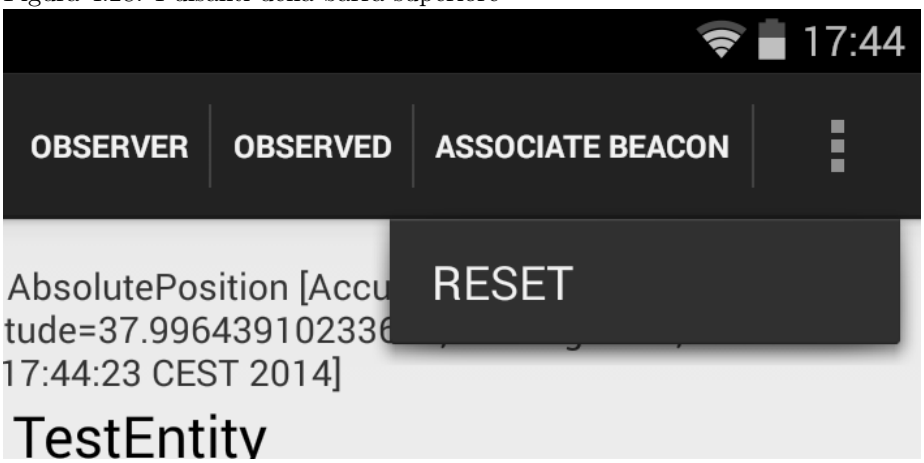
Figura 4.27: Scheda relativa ad un'entità: mostra tutte le informazioni relative



La percezione di entità vicine tramite il BLE si manifesta nella comparsa delle entry corrispondenti nella lista “Distances”.

Nella barra superiore sono presenti quattro pulsanti:

Figura 4.28: Pulsanti della barra superiore



- *OBSERVER* e *observed* avviano le corrispondenti Activity (descritte in seguito) specializzando il ruolo del dispositivo
- *Associate beacon* avvia un’activity che permette di registrare le caratteristiche di un beacon all’entità Self del dispositivo
- *RESET* invia il corrispondente comando al server

4.6.2 Observed Activity

L’ObservedActivity a run-time presenta semplicemente un codice QR di dimensioni maggiori possibili.

4.6.3 Observer Actitivty

L’ObserverActivity a run-time mostra ciò che viene inquadrato dalla fotocamera. Non appena diventino disponibili vengono visualizzate anche le geome-

Figura 4.29: L'ObservedActivity: il QR codifica l'ID dell'entità osservata

trie rappresentanti i segnaposti delle entità partecipanti. In corrispondenza di ogni segnaposto è mostrato l'identificativo dell'entità. In sovrimpressione rispetto alla scena della fotocamera vi sono i componenti grafici che permettono l'interazione con l'utente:

1. In alto a destra si trova il pulsante che chiude l'Activity. La sua pressione, così come quella del pulsante Back di Android, ha l'effetto di terminare l'ObserverActivity e ritornare alla Dashboard.

2. In alto a sinistra si trova il *Radar*. Si tratta di un componente fornito da Metaio: tramite una rappresentazione semplice, efficace ed intuitiva il radar fornisce una rapida idea di come siano disposte le entità rispetto al Self. Il tap di un segnaposto nella scena colora di rosso il corrispondente pallino sul radar per un'individuazione ancora più rapida.

3. In alto al centro si trova uno spazio dedicato alle informazioni. Qui apparirà una messaggio in chiara sovrimpressione che comunica all'utente l'eventuale avvicinamento / allontanamento da altre entità.

Figura 4.30: La realtà aumentata nell'Observer Activity

4. In basso al centro si trova il pulsante "Scan" che permette all'utente di avviare una scansione manuale. In alternativa la scansione viene avviata in automatico quando viene captato un avvicinamento ad un'entità e la distanza è inferiore al mezzo metro. Nel primo caso la scansione durerà *K.TIMEOUT_USER_SCAN_DURATION*[15], nel secondo *K.TIMEOUT_AUTO_-SCAN_DURATION*[16]. Indipendentemente da come viene avviata la scansione rispetta sempre la stessa sequenza di step:

   (a) Un Toast notifica l'avvio della scansione

   (b) Mentre avviene la scansione in alto al centro vengono mostrati i secondi trascorsi su quelli totali. Durante la decodifica di un codice (o presunto tale) Metaio mostra a video dei simboli in prossimità dei 4 quadrati principali di un codice QR.

   (c) Se è riconosciuta un'entità, viene mostrato un AlertDialog che ne mostra gli attributi. La scansione continua regolarmente ma nessun

[15] Valore attuale 7s
[16] Valore attuale 2s

Figura 4.31: Una scansione in corso: il sistema prova a decodificare il QR, fallendo a causa del riflesso della luce

ulteriore messaggio è mostrato sino alla chiusura del primo.

(d) Al termine della sequenza di scansione viene mostrato un Toast che avvisa l'utente.

Figura 4.32: Una scansione andata a buon fine

Capitolo 5

Conclusioni e sviluppi futuri

Con il "sistema in mano" è possibile tirare delle somme.

5.1 Il GPS

Il GPS, proprio come ci si aspettava, si è rivelato utile solo sulle lunghe distanze: all'aperto, in una giornata senza nubi l'accuratezza raggiunge minimi di 5m: trattasi del caso ottimo. Nel caso medio, ovvero in presenza di nubi/pioggia o chiome di alberi, l'accuratezza si aggira intorno ai 10m. Stessa cosa non si può dire al chiuso o in situazioni in cui esista un ostacolo tra i satelliti ed il dispositivo: in-door il GPS risente anche del "piano" in cui ci si trova ad operare: all'ultimo piano il segnale è migliore ed è possibile ottenere un posizionamento ancora accettabile (accuratezza tra 10m e 20m), ma si degrada velocemente ai piani inferiori. Tutte imprecizioni che hanno peso solo osservando un'entità vicina: rimane comunque uno strumento utilissimo ed indispensabile per avere un'idea della posizione di entità lontante, e del loro movimento: man mano che ci si allontana dal target, l'imprecisione è sempre più trascurabile.

È importante ricordare che il GPS fornisce anche velocità e orientamento. Il prototipo non impiega in maniera utile tali informazioni limitandosi a non perderle. In futuro si potrebbe pensare di impiegare anche questi dati: se l'osservatore conosce velocità ed orientamento dell'osservato può spostare di conseguenza il segnaposto avendo buone probabilità che l'aggiornamento successivo non sia un punto poi così distante da quello ottenuto virtualmente, ovviamente nell'ipotesi che gli aggiornamenti siano sufficientemente frequenti.

Un'altra strada non sondata è quella della pulizia delle informazioni GPS: si potrebbe pensare, disponendo di aggiornamenti frequenti, di campionare N posizione GPS ed effettuare una media fornendo quindi una nuova posizione GPS ogni N effettivamente ricevute. In quest'ipotesi si dovrebbe anche stimare empiricamente il miglior valore N considerando che, intuitivamente, un numero troppo piccolo implica minore accuratezza, un numero troppo elevato implica troppo ritardo.

5.2 Il BLE

Il BLE è uno strumento molto interessante. Come preannunciato, scopo della tesi era anche sondarne le potenzialità.

Se usato propriamente, ovvero interpretando il segnale come semplice messaggio "esisto!", fornisce degli ottimi risultati: il rumore sul segnale diventa poco influente ed il beacon, da questo punto di vista, svolge egregiamente il proprio lavoro; si perdono tuttavia informazioni su distanza e posizione.

Se usato come sensore di distanza, ossia cercando di stimare la distanza dall'emettitore senza tuttavia impiegare tale informazione in combinazione con altre distanze, si ottengono comunque dei risultati incoraggianti: seppur il segnale e la conseguente distanza siano molto altalenanti, è comunque possibile capire se ci si sta allontanando / avvicinando, il che è utilissimo. Attenzione però che

il segnale beacon è facilmente influenzabile da una serie di fattori: distanza, conformazione dell'ambiente, ostacoli, esseri umani: la mano dell'utilizzatore, se posta di fronte all'antenna bluetooth, altera la ricezione del segnale. In condizioni ottimali, a distanze non troppo elevate e con dispositivo e beacon fermi, l'errore sulla distanza stimata è dell'ordine della decina di centimetri, il problema è che il valore oscilla con un'ampiezza di circa 2-3cm. Si potrebbe pensare di introdurre un filtro passa-alto che ignori le variazioni in modulo troppo piccole, come di fatto poi è stato fatto per rilevare gli avvicinamenti e gli allontanamenti. Inoltre, la nozione di avvicinamento potrebbe essere combinata con i sensori del device registrando in quale direzione l'osservatore dovrebbe continuare a muoversi.

In sostanza, quindi, sì: è possibile fornire delle informazioni a supporto del GPS per meglio localizzare un'entità.

Se usato per la trilaterazione, l'errore nella misura della distanza diventa molto meno trascurabile: in tal caso si potrebbe pensare di ovviare agli incovenienti intrinseci di questa tecnologia aumentando di molto il numero di landmark e disponendoli in maniera tale che sia alta la probabilità che in un dato momento un'entità abbia "campo libero" nei confronti di almeno tre landmark. Tale via non è stata sondata a causa dell'insufficiente quantità di beacon a disposizione e pertanto la soluzione proposta è più che altro una congettura. Bisogna tener presente inoltre che il setup dei landmark è un'operazione relativamente onerosa: richiede di conoscere le coordinate di tutti i punti di riferimento scelti. Risulta cruciale, per una corretta trilaterazione, avere un posizionamento molto accurato onde evitare di aggiungere imprecisione a delle informazioni già di per sé affette da errore. I test effettuati in tal senso hanno evidenziato come la trilaterazione sia comunque in grado di fornire una posizione anche se abbastanza stabile ed affetta da un'imprecisione di pochi metri.

Resta aperto il problema del rumore sull'RSSI del beacon advertisement: in presenza di molti landmark, si potrebbe pensare di fare una media delle posizioni

trilaterate rispetto a terne di beacon differenti, trovare algoritmi euristici che traggano vantaggio da una moltitudine di punti fermi etc. Più semplicemente si potrebbe pensare di aumentare al massimo la frequenza dell'advertisement e di calcolare la distanza "ripulendo" il segnale ad esempio facendo la media delle RSSI ad intervalli regolari. Una soluzione simile è adottata dalla libreria di Altbeacon, il che ha la sua manifestazione più immediata nel fatto che, a fronte dello spostamento di un beacon, la distanza stimata comincia in ritardo a variare e lentamente tendere al nuovo valore.

Ricapitolando, quindi, sì: è possibile ottenere una stima della posizione mediante BLE, seppure non in maniera efficace come inizialmente si era sperato. Questo non è un risultato negativo in quanto:

- l'imprecisione della posizione ottenuta tramite beacon risulta molto evidente nella realtà aumentata, certamente lo sarebbe di meno in applicazioni che fanno invece uso di mappe;
- in assenza di GPS viene comunque fornita una posizione ed i landmark si posso ubicare anche sotto un tetto senza che ne venga alterata la funzionalità.

5.3 Il riconoscimento di codici QR

Il riconoscimento di immagini si è rivelato meno flessibile di quanto sperato: seppure il riconoscimento sia molto rapido, esso richiede condizioni abbastanza statiche, il che non è comodo. Si è notato che il riconoscimento di codici stampati è molto più efficace di quello di immagini mostrate sul display di uno smart-device, probabilmente per merito delle maggiori dimensioni ottenibili e dell'assenza di riflessi. Se mostrando il codice sullo schermo di un Nexus 7, alla dimensione massima possibile, si è costretti ad avvicinarsi sino a 0.5m di distanza, stampando l'immagine su un foglio A4 tale vincolo si rilassa fino a 1.5m-2m.

È lecito presumere che immagini di dimensioni ancora maggiori possano permettere di riconoscere l'entità da ancora più lontano. Un fattore influente in tal senso è anche la risoluzione della fotocamera: aumentarla influisce positivamente sulla distanza minima di riconoscimento.

5.4 Il server

Il server e tutto ciò che lo concerne è stato realizzato con scarsa attenzione alle performance, come è facile evincere dall'impiego dell'HTTP. Sviluppi futuri potrebbero ottimizzare questo aspetto utilizzando magari un middleware più articolato e degno di questo nome che non necessariamente impieghi l'HTTP (l'interfaccia IDataServerHandler non lo prevede), il che indubbiamente renderebbe più agile la comunicazione, vero collo di bottiglia di questo prototipo. In quest'ottica si potrebbe immaginare di abbandonare il modello «a polling» in cui è il client a richiedere periodicamente aggiornamenti al server a volte anche non necessari, ma far sì che sussista una connessione tra i due e che sia il secondo a notificare gli aggiornamenti al primo solo quando si renda necessario; in questo scenario avrebbe senso immaginare che il sia il server a possedere la concezione di livello di dettaglio e che in base ad essa fornisca differenti quantità e qualità di informazioni ai client.

5.5 La user-interface

Come spesso accade nei prototipi, anche in questo caso l'interazione con l'utente non è stata una priorità. Gli sviluppi futuri potrebbero prevedere molte semplici migliorie: segnaposti più piccoli che non sforino dallo schermo, al posto di messaggi testuali si potrebbe pensare di mostrare simboli più intuitivi come frecce o simili, le informazioni essenziali potrebbero essere mostrate in corrispondenza

del segnaposto o con diverse versioni dello stesso, la scansione potrebbe essere avviata da una gesture più che da un pulsante, il riconoscimento di un codice QR potrebbe implicare un semplice ri-disegno del segnaposto in corrispondenza dello stesso piuttosto che l'apparizione di una finestra di dialogo, si potrebbe aggiungere la possibilità di visionare una mappa dell'intera area di interesse, etc.

Bibliografia

[1] Altbeacon. http://altbeacon.org/. [Online; in data 2-ottobre-2014].

[2] Android asynchronous http client. http://loopj.com/android-async-http/. [Online; in data 3-ottobre-2014].

[3] Android beacon library. https://github.com/AltBeacon/android-beacon-library. [Online; in data 2-ottobre-2014].

[4] Android ble peripheral mode. https://code.google.com/p/android/issues/detail?id=59693. [Online; in data 2-ottobre-2014].

[5] Apache license, version 2.0. http://www.apache.org/licenses/LICENSE-2.0.html. [Online; in data 3-ottobre-2014].

[6] Beacon parser. http://altbeacon.github.io/android-beacon-library/javadoc/org/altbeacon/beacon/BeaconParser.html. [Online; in data 2-ottobre-2014].

[7] Beaconparser.setbeaconlayout. http://altbeacon.github.io/android-beacon-library/javadoc/org/altbeacon/beacon/BeaconParser.html. [Online; in data 2-ottobre-2014].

[8] Bluetooth low energy. https://developer.android.com/guide/topics/connectivity/bluetooth-le.html. [Online; in data 2-ottobre-2014].

[9] Codici qr. http://www.qrcode.com/en/about/. [Online; in data 3-ottobre-2014].

[10] Datum transformations of gps positions. https://microem.ru/files/2012/08/GPS.G1-X-00006.pdf. [Online; in data 3-ottobre-2014].

[11] Http. http://tools.ietf.org/html/rfc2616. [Online; in data 3-ottobre-2014].

[12] Json. http://json.org/. [Online; in data 3-ottobre-2014].

[13] Json in java. http://www.json.org/java/. [Online; in data 3-ottobre-2014].

[14] Json license. http://www.json.org/license.html. [Online; in data 3-ottobre-2014].

[15] Location strategies. http://developer.android.com/guide/topics/location/strategies.html. [Online; in data 3-ottobre-2014].

[16] Metaio. http://www.metaio.com/. [Online; in data 3-ottobre-2014].

[17] New in android: L developer preview and google play services 5.0. http://android-developers.blogspot.it/2014/06/developer-preview-and-play-services-5.html. [Online; in data 3-ottobre-2014].

[18] Qrgen. https://github.com/kenglxn/QRGen. [Online; in data 3-ottobre-2014].

[19] Region. http://altbeacon.github.io/android-beacon-library/javadoc/org/altbeacon/beacon/Region.html. [Online; in data 2-ottobre-2014].

[20] Scheduledexecutorservice. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html. [Online; in data 3-ottobre-2014].

[21] Scheduledthreadpoolexecutor. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html. [Online; in data 3-ottobre-2014].

[22] simplelatlng. https://code.google.com/p/simplelatlng/. [Online; in data 3-ottobre-2014].

[23] Tips to create image trackable. https://dev.metaio.com/sdk/documentation/tracking-config/create-image-trackable/. [Online; in data 3-ottobre-2014].

[24] Trilateration using 3 latitude and longitude points, and 3 distances. http://gis.stackexchange.com/questions/66/trilateration-using-3-latitude-and-longitude-points-and-3-distances. [Online; in data 3-ottobre-2014].

[25] Updated bluetoothÂ® 4.1 extends the foundation of bluetooth technology for the internet of things. http://www.bluetooth.com/Pages/Press-Releases-Detail.aspx?ItemID=197. [Online; in data 2-ottobre-2014].

[26] What is ibeacon? a guide to beacons. http://www.ibeacon.com/what-is-ibeacon-a-guide-to-beacons/. [Online; in data 2-ottobre-2014].

[27] Zxing. https://github.com/zxing/zxing/. [Online; in data 3-ottobre-2014].

[28] Andrea Fortibuoni. Sviluppo di una infrastruttura location-based per l'auto-organizzazione di smart-devices, 2013/2014.

[29] Wikipedia. Gps assistito — wikipedia, l'enciclopedia libera. http://it.wikipedia.org/w/index.php?title=GPS_Assistito&oldid=56698415, 2013. [Online; in data 2-ottobre-2014].

[30] Wikipedia. Augmented reality — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Augmented_reality&oldid=627196306, 2014. [Online; accessed 3-October-2014].

[31] Wikipedia. Sistema di posizionamento globale — wikipedia, l'enciclopedia libera. http://it.wikipedia.org/w/index.php?title=Sistema_di_Posizionamento_Globale&oldid=68262202, 2014. [Online; in data 2-ottobre-2014].

[32] Wikipedia. Trilateration — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Trilateration&oldid=622732474, 2014. [Online; accessed 3-October-2014].

Ringraziamenti Lavorare a questo progetto è stato interessante sotto molteplici aspetti: per me è stato il primo vero approccio con un sistema pensato sin dall'analisi per essere distribuito e comprendere dispositivi per molti versi eterogenei. È stato il primo tentativo di interfacciamento effettivo con un hardware "reale" e non simulato e pertanto il primo scontro con le difficoltà tecniche che da ciò derivano. Concludendo: ritengo sia stata un'attività produttiva ed interessante e spero di avere tempo e modo di continuare la sperimentazione in questa direzione.

Desidero a questo punto ringraziare il relatore, il professor Mirko Viroli, ed il correlatore, il professor Alessandro Ricci che *anche* in questa esperienza hanno avuto una disponibilità esemplare; i miei amici Manuel e Andrea per avermi assistito nell'organizzazione degli ultimi giorni ed i miei genitori che mi hanno supportato[1] in questi anni di Università.

[1] all'occorrenza si può sostituire la «U» con la «O»