Hovering Information in Android Platform
Elaborato per il Corso di Sistemi Distribuiti 2012/2013

Lorenzo Perlini mat. 0000680521
Andrea Tomasello mat. 0000661064
Luca Ricchi mat. 0000679516

19 marzo 2013

Sommario

Il progetto realizzato si può suddividere in queste sezioni:

- *Analisi e studio del concetto di hovering information.*
- *Studio della libreria Jade-LEAP, con particolare attenzione all'implementazione rivolta a dispositivi Android.*
- *Studio della tecnologia di trasmissione tra device Wifi direct e analisi della situazione commerciale e diffusione sui dispositivi attualmente in commercio.*
- *Realizzazione di una libreria per lo sviluppo di applicazioni per Android utilizzando le due tecnologie appena citate; il suo scopo è facilitare lo sviluppo di app che avviino in automatico una piattaforma Jade-LEAP per device Android, quando si instaura una connessione tra dispositivi tramite il Wifi direct.*
- *Realizzazione di una app dimostrativa che supporta i concetti relativi alla Hovering information, sfruttando Wifi direct e Jadeleap Android.*

Indice

1 Hovering information

L'avvento di nuove tecnolgie, quali PDAs, Smartphone con accesso ad internet (3g, 4g, ecc...) e *mobile device* in generale, ha focalizzato l'attenzione sul tema dell'informazione geo-refernziata.
Con il termine *georeferenziazione* s'intende la localizzazione di un certo contenuto informativo in un'area geografica precisa e ben definita. Fino ad oggi questo concetto si traduceva in informazioni rese disponibili attraverso un server centrale, mentre la ricerca è oggi incentrata sulla completa decentralizzazione. Il tema di ricerca da noi preso come riferimento per lo sviluppo di questo progetto è l'**Hovering Information**.

1.1 Hovering information - Concetti base

L'**Hovering Information** (HoverInfo) rappresenta un nuovo paradigma, una nuova metodologia che affronta il problema di come sia possibile scambiare informazioni in un'area geografica ben definita e circoscritta, tra tutti quei dispositivi che vi rientrano. Non avere un server centrale in grado di monitorare e replicare l'informazione, pone non pochi problemi di gestione e interazione tra i dispositivi coinvolti nella comunicazione, i quali dovranno essere in grado di duplicare autonomamente il contenuto informativo e trasmetterlo ad altri device mobili senza l'ausilio di alcun meccanismo centralizzato.
La diffusione sempre maggiore di dispositivi mobili renderà questo nuovo modo di interagire con le informazioni ampiamente diffuso e impiegato su scala mondiale; i device assumeranno il ruolo di **nodi** in grado di scambiare, accedere e memorizzare un enorme numero di informazioni, le quali non verranno più trasmesse da un server ma saranno direttamente contenute nelle memorie dei dispositivi mobili, progressivamente più performanti.
In tal senso uno dei caposaldi del modello **Hovering**, è la capacità dell'informazione di essere completamente indipendente dall'hardware del device,

costituendo così un'entità autonoma in grado di *librarsi* nell'area geografica circostante e migrare negli altri device presenti.

1.2 Hovering Information - Definizioni Formali

- **Coordinata geografica**: definiamo **a=(lat,long)** un punto geografico con una certa latitudine *lat* e longitudine *long*. Non si considera il concetto di altitudine.
- **Area**: definiamo **A=(a,r)**, come l'insieme dei punti b (coord. geografiche) tali che la distanza fra il punto a e un punto b sia minore del raggio r, con coord. a al centro dell'area A, ovvero:

$$A(a, r) = (b \mid dist(a, b) < \ r) \tag{1}$$

- **Nodo**: si definsice nodo mobile una tupla del tipo n = (id, loc, speed, dir, rcomm), dove:
  - **id** è l'identificativo di un nodo;
  - **loc** è una coordinata geografica;
  - **speed** è la velocità in m/s;
  - **dir** è un vettore che rappresenta la direzione del movimento più recente;
  - **rcomm** è chiamato *range of communication* ed è la distanza massima in metri a cui può comunicare senza fili un nodo.

Con il termine nodo si denota un dispositivo mobile, dotato di storage, "sfruttato" dai pezzi di hovering information. Un insieme N è "ben formato" se - presi due nodi N1 e N2 - allora:

$$\forall(N1, N2) \in N(id(n1) = id(n2)) \rightarrow (n1 = n2) \tag{2}$$

In altre parole due nodi che hanno lo stesso *id*, sono lo stesso nodo.

Un pezzo di hovering information è una tupla h = (id, a, r, n, data, policies, size), dove:

- **id**: è l'identificativo del pezzo;
- **a**: è la locazione dell *anchor* (**Anchor location**). Questa àncora è definita come il centro dell'area A, a cui è associato un raggio r, chiamato *anchor radius*.
- **n**: è il nodo in cui al momento si trova il pezzo di HoverInfo.

- **policies**: sono le *hovering policies* di h (come e quando un pezzo di hovering information deve "fluttuare" ovvero "librarsi in volo");
- **size**: è la dimensione, espressa in termini di byte, della HoverInfo;

Consideriamo H un insieme di pezzi di Hovering information. Gli identificatori di un pezzo di hoverinfo sono univoci, ma le repliche - che trasportano gli stessi dati e *anchor information* - sono permesse su nodi diversi, quindi possiamo dire che H è ben formato se:

$$\begin{aligned}\forall h_1, h_2 \in \mathcal{H}, (h_1 \neq h_2) \Rightarrow \\ (id(h_1) \neq id(h_2)) \vee \\ ((id(h_1) = id(h_2)) \wedge (a(h_1) = a(h_2)) \wedge \\ (r(h_1) = r(h_2)) \wedge (data(h_1) = data(h_2)) \wedge \\ (n(h_1) \neq n(h_2))).\end{aligned}$$

Figura 1: *Condizioni poste affinché H sia ben formato*

La formula in figura 1 mostra come H sia ben formato se dati due pezzi di Hoverinfo, *H1* e *H2*, valgono i seguenti punti:

- i due pezzi di Hovering hanno *id* diversi;
- hanno stesso *id*, stessa *anchor location*, stesso raggio *r*, stesso contenuto informativo *data*, ma (n(H1) != (n(H2))) ovvero risiedono su nodi diversi.

Quindi, assumendo che h appartenente ad H è un pezzo di *hoverInfo*, una sua replica che chiameremo *hr* è un pezzo di hovering information tale che:

$$id(h) = id(hr) \wedge n(h)! = n(hr) \tag{3}$$

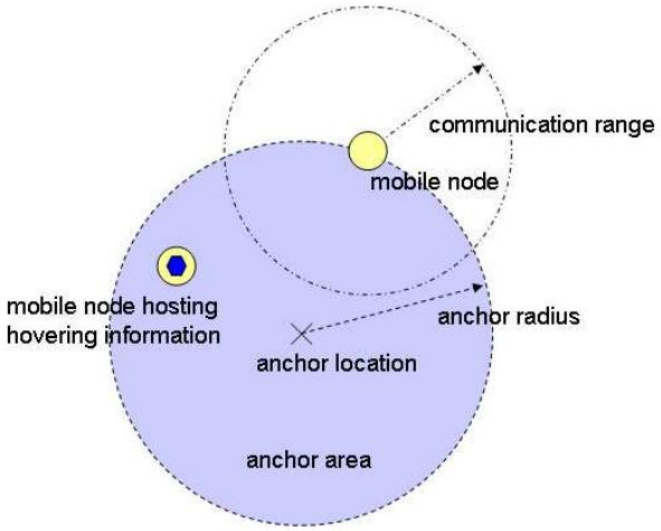

Figura 2: *Rappresentazione grafica di HoverInfo*

Un sistema hovering information all'istante t è definito come:
HoverInfot = (Nt , Ht), dove:

- *Nt* è un insieme ben definito di nodi;
- *Ht* è un insieme ben definito di hovering information su *Nt*, tali che

$$(\forall h \in Ht) \Rightarrow (n(h) \in Nt) \tag{4}$$

HoverInfot, in altre parole, è uno *snapshot* dello stato del sistema all'istante di tempo t, il sistema poi evolve con il procedere dei *tick*: t, t+1, etc...
Durante la sua evoluzione può accadere che:

- i nodi mobili possano cambiare la loro locazione;
- nuovi nodi mobili possano aggregarsi al sistema (nuovi dispositivi possono entrare nella anchor area);
- altri nodi possano abbandonare l'area;
- possano comprarire nuovi pezzi di HoverInfo, a seguito della creazione di nuovi contenuti;
- possano muoversi o scomparire pezzi di HoverInfo, a seguito dello spostamento da un nodo all'altro, oppure a causa dell'uscita dell'*HoverInfo piece* dall'anchor area.

Considerando *HoverInfot = (Nt , Ht)* un sistema di HoverInfo nel momento *t*, sia *h* un pezzo di HoverInfo e *n* appartenente a *Nt* un nodo al momento *t*, definiamo:

$$R_H(h,t) = \{k \in \mathcal{H}_t \mid id(h) = id(k)\}$$

Figura 3: *L'insieme delle repliche di pezzi di HoverInfo h, all'istante t.*

$$R_N(n,t) = \{h \in \mathcal{H}_t \mid n(h) = n\}$$

Figura 4: *L'insieme di pezzi di HoverInfo presenti nel nodo n,all'istante t.*

$$N_N(n,t) = \{m \in \mathcal{N}_t \mid (dist(loc(m), loc(n)) < r_{comm}(n)) \vee (dist(loc(m), loc(n)) < r_{comm}(m)),\}$$

Figura 5: *L'insieme dei nodi vicini ad n, all'istante t.*

1.3 Hovering information - Proprietà HoverInfo

In questo paragrafo verranno enunciate proprietà fondamentali dell'Hovering, quali: **Availability**, **Survivability**, **Accessibility**.

1.3.1 Availability

Con il concetto di **Availability**(disponibilità) s'intende che un certo contenuto informativo è disponibile in un momento t se c'è almeno un nodo nella sua anchor area che contiene una replica di questa informazione.
Riprendendo il concetto di *HoverInfot = (Nt , Ht)*, cioè un sistema Hover all'istante t, si dice che l'informazione h è disponibile al'istante t se, considerando un qualsiasi nodo n nell'anchor area di h, l'intersezione tra l'insieme delle repliche di h nel momento t e l'insieme di pezzi di HoverInfo nel momento t nel nodo n, non è vuota.
In altre parole, come si può facilmente vedere nella figura che segue, il nodo nel tempo t dev'essere nell'anchor area e possedere una replica dell'hovering information.

$$av_H(h,t) = \begin{cases} 1 & \textit{if } \exists n \in \mathcal{N}_t, (P_N(n,t) \in A_H(h)) \wedge \\ & (R_H(h,t) \cap R_N(n,t) \neq \emptyset) \\ 0 & \textit{otherwise.} \end{cases}$$

Figura 6: *Sistema che riassume la condizione di Availability*

1.3.2 Survivability

Con il concetto di **Survivability**(sopravvivenza) s'intende che un certo contenuto informativo h è vivo nell'istante t se l'intersezione tra l'insieme delle repliche di h nel momento t e l'insieme di pezzi di hovering information nel momento t in un qualsiasi nodo n, non è vuota. In altre parole, se ci sono dei nodi che possiedono il pezzo h.

$$sv_H(h,t) = \begin{cases} 1 & \textit{if } \exists n \in \mathcal{N}_t, R_H(h,t) \cap R_N(n,t) \neq \emptyset \\ 0 & \textit{otherwise.} \end{cases}$$

Figura 7: *Sistema che riassume la condizione di Survivability*

1.3.3 Accessibility

Si può distinguere la **Accessibility** dalla availability con questo esempio:
un pezzo di hovering information (o una sua replica), presente in un nodo localizzato nella anchor area, è detta available (disponibile). Tuttavia, lo stesso pezzo di potrebbe non essere accessibile da un nodo significativamente distante da quello in cui è attualmente memorizzata l'informazione (o una sua replica). Diremo quindi, che un'informazione è accessibile da un nodo n, all'istante t, se il nodo è in grado di ottenere tale informazione. In altri termini, se esiste un nodo m che è nel raggio di comunicazione (anchor radius) del nodo n ed ha con sé la hovering information richiesta da n.
Più formalmente, una hovering information h è accessibile da un nodo n all'istante t se, considerando un qualsiasi nodo m appartenente all'insieme dei nodi vicini ad n al tempo t, l'intersezione fra l'insieme delle repliche di *h(t)* e l'insieme di pezzi di hoverInfo all'istante t nel nodo m, non è vuota.

$$ac_H(h,n,t) = \begin{cases} 1 & \textit{if } \exists m \in \mathcal{N}_t, (m \in N_N(n,t)) \wedge \\ & (R_H(h,t) \cap R_N(m,t) \neq \emptyset) \\ 0 & \textit{otherwise.} \end{cases}$$

Figura 8: *Sistema che riassume la condizione di Accessibility*

La figura di seguito riassume i diversi casi relativi a sopravvivenza, disponibilità e accessibilità:

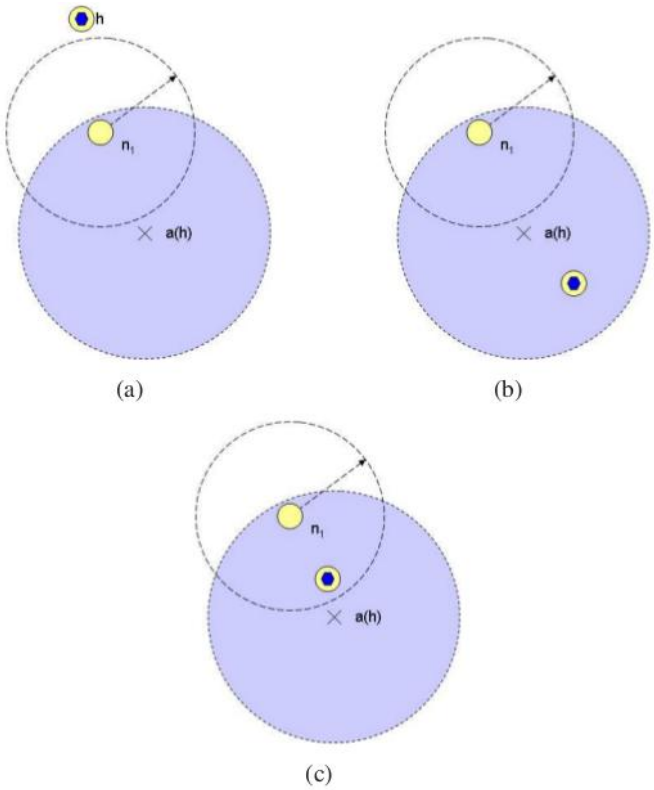

Figura 9: *a) Hoverinfo h è sopravvissuta ma non è né disponibile alla sua anchor area a(h) né accessibile b) Hoverinfo h è sopravvissuta e disponibile alla sua anchor area a(h), ma non è accessibile dal nodo n1. c) Hoverinfo h è sopravvissuta, disponibile alla sua anchor area a(h) e accessibile dal nodo n1.*

1.4 Hovering information - Safe, Risk, Relevant ed Irrelevant Areas

Come già specificato, le *hovering policies* sono relative ai singoli pezzi di hoverInfo. Per semplicità si consideri che tutti i pezzi di hover adottino le stesse politiche:

- **active replication**: possibilità di replicarsi in altri nodi;
- **hovering**: scambio d'informazioni, con il fine di rimanere all'interno dell'anchor area (si vuole garantire availibity e accessibility);
- **hovering e caching**: quando il pezzo di HoverInfo si distanzia troppo dall'anchor area (al fine di mantenere survivability)
- **cleaning**: quando la distanza del pezzo dall'anchor area è significativa (scomparsa dell'informazione).

La decisione di replicarsi o "fare hover" dipende dalla posizione corrente del dispositivo mobile in cui l'informazione è attualmente memorizzata.

1.4.1 SAFE AREA

Data una anchor area *A(a, r)*, la *safe area A(a, rsafe)* è lo spazio il cui centro è l'anchor location *a* e il cui raggio *rsafe* è positivo ma inferiore rispetto al raggio dell'anchor area *r*.

$$A(a, r_{safe}) = \{b \in \mathbb{E} \mid dist(a, b) < r_{safe}\}$$

Figura 10: *Definizione matematica del concetto di safe area. rsafe* $\prec$ *r*

Un pezzo d informazione che si trova in un nodo contenuto dalla safe area, può tranquillamente rimanere nel nodo stesso.

1.4.2 RISK AREA

Data una anchor area *A(a, r)*, una *risk area* è un anello centrato sull'anchor location, che si sovrappone con la anchor area ed è limitato dalla safe area. La risk area *R(a, rsafe, rrisk)* è data da:

$$R(a, r_{safe}, r_{risk}) = A(a, r_{risk}) \backslash A(a, r_{safe})$$

Figura 11: *Definizione matematica del concetto di risk area. rsafe* $\prec$ *r* $\prec$ *rrisk*

Un pezzo di hovering information localizzato nella risk area dovrebbe attivamente ricercare un nuovo nodo mobile contenuto nella safe area. E' in questa area (risk) che si replicano attivamente le informazioni, con il fine di rimanere disponibili e più vicine all'anchor location.

1.4.3 RELEVANT AREA

La *relevant area* è in grado di minare la survability, cioè la sopravvivenza, di un pezzo di hovering information. La *relevant area A(a, rrel)* è un disco il cui centro è l'anchor location *a* e il cui raggio *rrel* è maggiore del risk radius:

$$A(a, r_{rel}) = \{b \in \mathbb{E} \mid dist(a, b) < r_{rel}\}$$

Figura 12: *Definizione matematica del concetto di relevant area. rrisk* $\prec$ *rrelevant*

Rappresenta l'area in cui l'hovering information cerca di sopravvivere ma non si replica attivamente. Potrebbe avvicinarsi all'anchor area attraverso dispositivi mobili che vanno in quella direzione.

1.4.4 IRRELEVANT AREA

L'*irrelevant area* è tutta l'area *U (a, r)* posta al di fuori della relevant area, ed è data da:

$$U(a, r_{rel}) = \mathbb{E} \backslash A(a, r_{rel})$$

Figura 13: *Definizione matematica del concetto di irrelevant area.*

1.4.5 Visione d'insieme

Qui di seguito la figura che riassume le aree fino ad ora descritte:

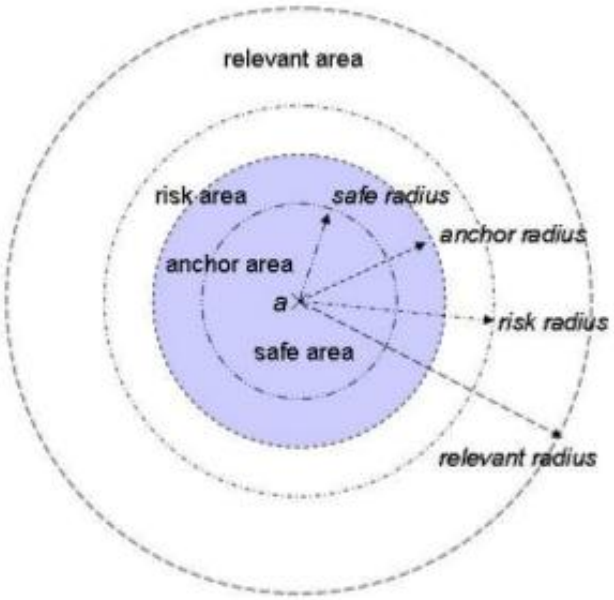

Figura 14: *Figura che evidenzia le aree fino ad ora descritte. Con a si evidenzia la anchor location.*

1.5 Hovering information - Algorithms

Si propone l'**Attractor Point algorithm**, che punta a tenere le hovering information vive e disponibili nella loro anchor area il più a lungo possibile. Un anchor location *a* funge da *attractor point*: tutti i pezzi di hovering information che hanno *a* come anchor location tendono a convergere verso di esso. Oltre all'attractor point algorithm, descriviamo il **Broadcast algorithm**, da cui ci si aspetta una performance relativa a survivability e availability migliore rispetto al precedente algoritmo, ma ad un costo maggiore in termini di risorse, di memoria e di rete. Infine, si proporrà anche un terzo algoritmo chiamato **Caching and Cleaning Modules Algorithm**
Assunzioni:

- **Unlimited memory**: Tutti i nodi hanno una quantità di memoria illimitata per memorizzare le repliche delle hovering information. L'algoritmo proposto non tiene conto della quantità di memoria rimasta o della dimensione dell'hovering information.
- **Unlimited energy**: Tutti i nodi hanno una quantità illimitata di energia. L'algoritmo proposto non considera eventuali errori di invio

messaggi causati da un basso livello di energia.

- **Instantaneous processing**: Il tempo di processazione dell'algoritmo in un nodo è 0. Non consideriamo quindi problemi di performance relativi al sovraccarico di operazioni del processore e tempi di esecuzione.
- **In-built geo-localization service** : I nodi hanno un servizio di geolocalizzazione come un GPS, in grado di fornire la posizione corrente. Assumiamo che questa informazione sia disponibile per i pezzi di hovering information.
- **Velocity vector service**: I nodi hanno un built-in *velocity vector service* che fornisce la velocità istantanea e la direzione di un nodo. Assumiamo che questa informazione sia disponibile per i pezzi di hovering information.
- **Neighbours discovering service**: I nodi hanno la possibilità di ricevere una lista di nodi vicini in ogni istante. La lista contiene posizione, velocità e direzione dei nodi. Anche questa informazione è disponibile per i pezzi di hovering information.

1.5.1 ATTRACTOR POINT ALGORITHM

L'anchor location di un pezzo di hovering information agisce costantemente come un punto di attrazione per quel pezzo di hovering information e tutte le sue repliche. Le repliche tendono a stare il più possibile nella loro anchor area spostandosi da un nodo all'altro. Periodicamente e per ogni nodo mobile, l'algoritmo verifica la posizione del nodo (linea 2) e carica l'elenco e le posizioni dei nodi presenti nel communication range (linee 3/4). L'algoritmo verifica poi se ci sono repliche di hovering information che sono nella risk area e necessitano di essere replicate (linea 8). Il numero dei nodi che compongono il multicast group è definito dalla costante kR (Replication Factor: è quindi un numero costante scelto a priori che dice quanti ne deve contattare per il multicast). La distanza tra ogni nodo mobile e l'anchor location viene calcolata nella linea 9. I kR nodi con la minor distanza vengono scelti come obiettivi per il multicast (linee 10/11). L'*information part* del pezzo corrente di hovering information viene "multicastata" ai kR nodi, all'interno del communication range, più vicini all'anchor location (linea 12).

Algorithm 1 Attractor Point Replication Algorithm

```
1:  procedure REPLICATION
2:      pos ← MY-POSITION
3:      N ← MY-NEIGHBOURS
4:      P ← POSITION(N)
5:      for all replica ∈ REPLICAS do
6:          anchor ← ANCHOR-LOCATION(replica)
7:          dist ← DISTANCE(pos, anchor)
8:          if (dist ≥ r_safe) and (dist ≤ r_risk) then
9:              D ← DISTANCE(P, anchor)
10:             D' ← SORT(D)
11:             M ← SELECT(D', 1, k_R)
12:             MULTICAST(info,M)
13:         end if
14:     end for
15: end procedure
```

Figura 15: *Pseudocodice Attractor Point Algorithm*

L'immagine che segue mostra il comportamento dell'Attractor Point algorithm. Considerando h un pezzo di hovering information nella risk area, si replica nei nodi all'interno del communication range che sono più vicini alla 'anchor location. Si assume, nell'immagine, che $kR = 2$, quindi i nodi *n2* ed *n3* riceveranno una replica, mentre tutti gli altri nodi non la riceveranno (nemmeno *n1*, *n4* ed *n5*, perché sono all'interno del communication range, ma non sono i due più vicini all'anchor location).

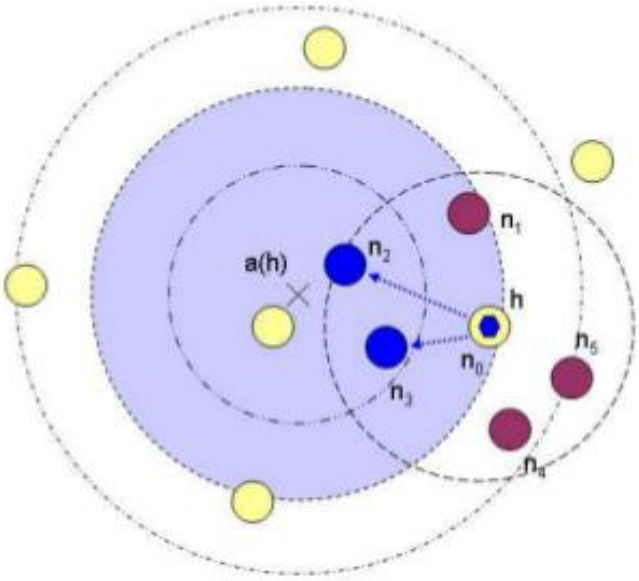

Figura 16: *Figura Attractor Point Algorithm*

1.5.2 BROADCAST ALGORITHM

Il Broadcast algorithm viene invocato periodicamente per ogni nodo mobile. Dopo aver verificato la posizione del nodo (linea 2), i pezzi di hovering information posizionati nella risk area (linea 6) sono replicati e inviati a tutti i nodi del communication range (linea 7). Ci si aspetta che questo algoritmo abbia migliori performance in termini di availability, ma peggiori in termini di consumo di risorse di rete e memoria.

Algorithm 2 Broadcast-based Replication Algorithm

```
1:  procedure REPLICATION
2:      pos ← MY-POSITION
3:      for all replica ∈ REPLICAS do
4:          anchor ← ANCHOR-LOCATION(replica)
5:          dist ← DISTANCE(pos, anchor)
6:          if (dist ≥ r_safe) and (dist ≤ r_risk) then
7:              BROADCAST(replica)
8:          end if
9:      end for
10: end procedure
```

Figura 17: *Pseudocodice Broadcast Algorithm*

Qui di seguito viene mostrata la figura che mostra l'algoritmo 'Broadcast':

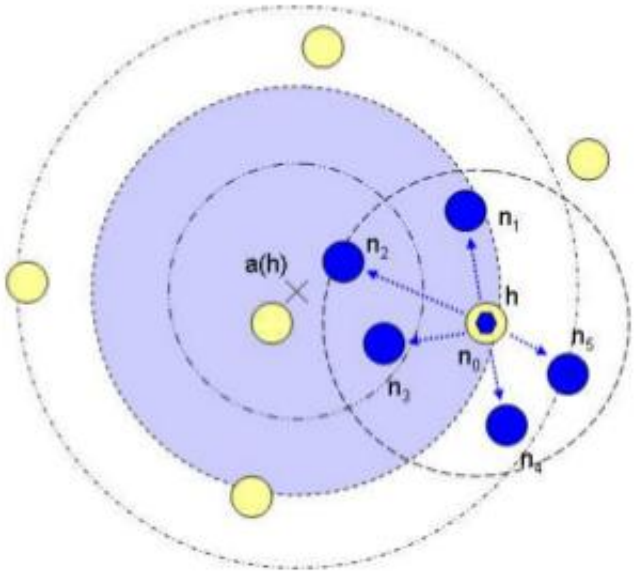

Figura 18: *Figura Broadcast Algorithm*

1.5.3 CACHING AND CLEANING MODULES ALGORITHM

Algorithm 3 Caching Algorithm

```
1: procedure CACHING(message)
2:     replica ← GET-REPLICA(message)
3:     if CONTAINS(replica, REPLICAS) = false
   then
4:         INSERT(replica, REPLICAS)
5:     end if
6: end procedure
```

Figura 19: *Pseudocodice Caching Algorithm*

Nell'algoritmo di caching si assume che ogni nodo abbia memoria illimitata perciò, ogni replica che viene inviata da un nodo all'altro, viene semplicemente memorizzata nel buffer dei nodi. Se un nodo riceve due o più repliche dello stesso pezzo di hovering information h, la prima replica che arriva verrà memorizzata e le altre ignorate (linee 3/4). Quindi al massimo una replica per ogni pezzo di hovering information è presente in un dato nodo n.

Algorithm 4 Cleaning Algorithm

```
1: procedure CLEANING
2:     pos ← MY-POSITION
3:     for all replica ∈ REPLICAS do
4:         anchor ← ANCHOR-LOCATION(replica)
5:         dist ← DISTANCE(pos, anchor)
6:         if (dist ≥ r_rel) then
7:             REMOVE(replica, REPLICAS)
8:         end if
9:     end for
10: end procedure
```

Figura 20: *Pseudocodice Cleaning Algorithm*

Il cleaning algorithm ogni TC secondi e per ogni nodo, rimuove le repliche che sono troppo distanti dall'anchor location, ad esempio le repliche che sono nella 'irrilevant area (linee 6/7). Sebbene la memoria sia illimitata e le repliche possano stare per un tempo indeterminato nella memoria del nodo,

rimuoviamo le repliche troppo distanti dall'anchor location, che rappresentano i casi in cui la replica è considerata troppo distante dalla anchor area e non più in grado di tornare indietro. Questo evita al meglio la situazione in cui tutti i nodi hanno delle repliche.

2 JADE LEAP - Android

JADE LEAP (Lightweight Extensible Agent Platform) è un add-on di JADE, sviluppato nel 2000 da un gruppo di aziende del settore di telecomunicazioni, di cui facevano parte Motorola, Siemens, Ericsson e due operatori telefonici: British Telecommunications e Telecom Italia. Il suo compito è sostituire alcune parti del kernel di JADE per permettere un suo corretto funzionamento su device con risorse hardware limitate, limitazioni sulla JVM e limitazioni di rete. I dispositivi che maggiormente presentano queste caratteristiche sono quelli mobili.
Più in dettaglio, JADE-LEAP fornisce un add-on di supporto per sviluppare applicazioni basate su JADE sui seguenti device:

- **pjava**: permette di eseguire applicazioni su dispositivi che supportano J2ME;
- **midp**: permette di eseguire applicazioni su dispositivi che supportano MIDP1.0 (o più recenti);
- **android**: permette di eseguire applicazioni su dispositivi che supportano Android 2.1 (o più recenti).
- permette infine l'esecuzione di applicazioni su sistemi che supportano l'ambiente di runtime **.NET**.

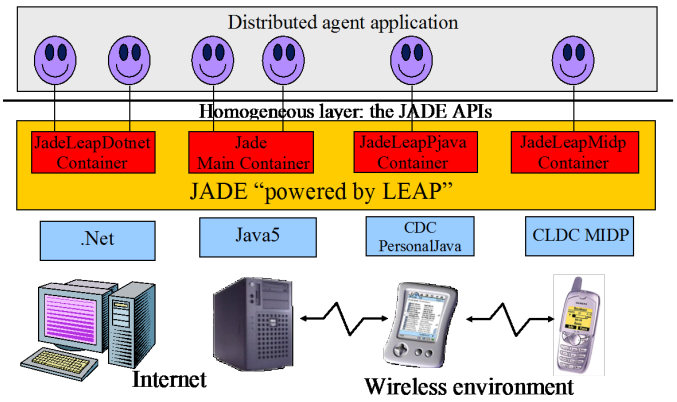

Figura 21: *JADE-LEAP runtime environment*

2.1 Differenze tra JADE e JADE-LEAP

JADE-LEAP fornisce esattamente le stesse API di JADE, ma possiede alcune differenze a *run-time*, che possiamo dividere in alcune macro sezioni:

- caratteristiche del **JAR** di JADE-LEAP e differenze relative alla **command line**;
- differenze implementative dell'**IMTP** (Internal Message Transport Protocol);
- il concetto **Container Execution Mode**.

2.1.1 JAR e Command line

Il jar di JADE-LEAP contiene già le classi relative ai tool di amministrazione (che in JADE sono in jadeTools.jar) e le classi relative all'MTPs (Message Transport Protocol) per HTTP e IIOP (in JADE rispettivamente in http.jar e iiop.jar).
Quando si vogliono caricare da linea di comando degli agenti, in JADE si utilizza come separatore il ';', mentre in JADE-LEAP si utilizza uno spazio ' '.
Ad esempio, per caricare due agenti (john e peter) di tipo myPackage.MyClass su JADE si utilizza il comando:

```
java jade.Boot -gui john:myPackage.MyClass;peter:myPackage.MyClass
```

mentre in JADE-LEAP:

```
java jade.Boot -gui john:myPackage.MyClass peter:myPackage.MyClass
```

Anche per specificare gli argomenti degli agenti da linea di comando si utilizza una sintassi diversa rispetto a JADE. Mentre in JADE, come separatore per indicare gli argomenti si utilizza lo spazio, in JADE-LEAP viene utilizzata una virgola.
Ad esempio, per specificare arg1 e arg2 come argomenti relativi all'agente john, in JADE si utilizzerà questa sintassi:

```
java jade.Boot -gui john:myPackage.MyClass(arg1 arg2)
```

Mentre in JADE-LEAP:

```
java jade.Boot -gui john:myPackage.MyClass(arg1,arg2)
```

Altre particolarità dell'utilizzo di JADE da linea di comando sono relativi alla specifica delle opzioni: '– nomobility' e '– dump' non sono disponibili; il modulo relativo all'IMPT può essere avviato definendo alcune opzioni che verrebbero ignorate su JADE (in 2.1.2 verrà approfondito questo aspetto).

2.1.2 IMPT

L'IMTP (Internal Message Transport Protocol) utilizzato per la comunicazione container-to-container, è completamente differente tra JADE e JADE-LEAP. Mentre JADE utilizza RMI, JADE-LEAP usa un protocollo proprietario chiamato JICP (Jade Inter Container Protocol). Le differenze tra questi due protocolli sono il motivo principale per cui un JADE-LEAP container non può registrarsi su un JADE Main-container e viceversa.
Lo strato che si occupa dell'IMTP in JADE-LEAP è composto da un 'Command Dispatcher' e uno o più Internal Communications Peers (ICPs). Il 'Command Dispatcher' è un singleton, la cui responsabilità sono:

- la serializzazione/deserializzazione dei comandi scambiati tra container;
- lo smistamento dei comandi ricevuti dallo strato sottostante che utilizza ICPs, ai container locali;
- l'invio dei comandi serializzati inviati da locale al corretto destinatario tramite l'ICP e la trasmissione in rete.

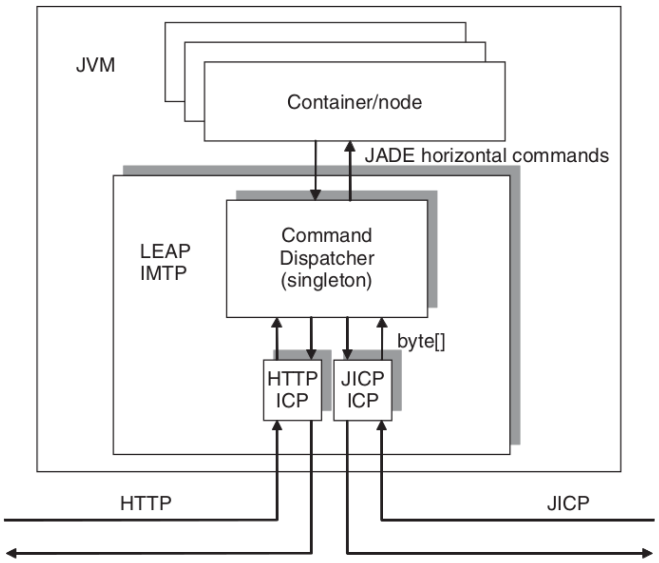

Figura 22: *LEAP IMTP*

LEAP include tre implementazioni di ICP: quella di default JICP-based, tramite la classe jade.imtp.leap.JICP.JICPPeer, la versione SSL di JICP, tramite jade.imtp.leap.JICP.JICPSPeer ed infine quella basata su HHTP, tramite jade.imtp.leap.http.HTTPPeer.
Questo significa che una JADE-LEAP platform può utilizzare JICP, JICP over SSL o HTTP come protocollo di comunicazione container-to-container. Potendo quindi specificare dei protocolli, sono disponibili alcuni parametri extra non presenti in JADE, ad esempio:

```
java jade.Boot -gui -icps jade.imtp.leap.JICP.JICPPeer;jade.imtp.leap.http.
    HTTPPeer(2000)
```

specifica che i protocolli di comunicazione supportati sono JICP standard e l'HTTP sulla porta 2000.

2.1.3 Container Execution Mode

Una JADE runtime viene implementata sempre tramite dei containers; JADE-LEAP ne fornisce uno alternativo chiamato SPLIT-container, che permette l'esecuzione di container in una split-execution mode, creata per andare incontro alle caratteristiche dei dispositivi mobili.
Quando JADE viene lanciato utilizzando la split execution mode, non viene

creato un normale container, ma un piccolo strato software chiamato front-end che fornisce agenti con le stesse caratteristiche di un container, ma implementa solo un piccolo sottoinsieme di queste, mentre delega le altre ad un processo remoto chiamato back-end. Mentre il front-end dal punto di vista degli agenti che vi risiedono sembra un normale container, lo stesso punto di vista avviene da parte degli altri container (compreso il Main) nei confronti del back-end.
Front-end e back-end quindi si può dire che formano un container diviso in due parti, la cui unione crea quello che viene chiamato *split execution mode*.

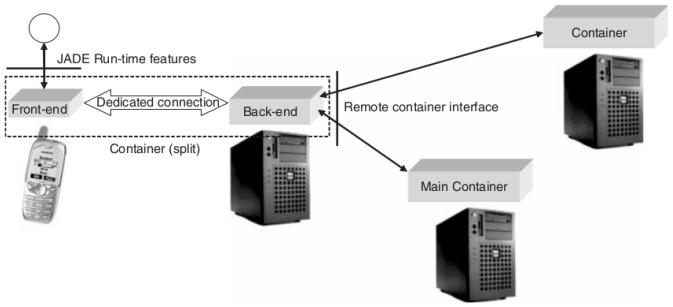

Figura 23: *Split execution mode*

Quando si crea quindi un container tramite JADE-LEAP, si può scegliere se creare un normale container, oppure uno split container.

2.2 JADE-LEAP Android

Abbiamo descritto l'add on JADE-LEAP, ora entreremo nel merito dell'implementazione relativa ai device Android.
L'implementazione di JADE-LEAP relativa ai device Android, si trova all'interno del package jade.android. E' interessante suddividere l'analisi di questo package in alcune macro sezioni, per riuscire ad analizzare in maniera più esauriente ogni aspetto.

2.2.1 JADE Android Service

Rispettando l'architettura di Android, la JADE runtime viene incapsulata in un Service. Più dettagliatamente, all'interno del jar jadeAndroid.jar, si possono trovare due classi: jade.android.RuntimeService e jade.android.MicroRuntimeService,

che rispettivamente fanno da wrapper per un full container ed uno split container.
L'utilizzo di questi Service è esattamente uguale ad un qualsiasi altro di Android, quindi comporta l'inserimento della definizione del Service nel manifest della app, all'interno del tag application:

```
<service android:name="jade.android.RuntimeService" />
```

All'interno di una Activity, per 'agganciare' un service, si utilizza una classe di tipo *Binder, la stessa cosa viene fatta per i Service di JADE-LEAP, tramite le classi già presenti nella libreria jade.android.MicroRuntimeServiceBinder e jade.android.RuntimeServiceBinder.
Queste sono le indicazioni per legare la Activity ai Service, mentre per inizializzare i container (full o split) si utilizzano semplicemente dei metodi presenti nei *Binder:

- **Full container**: espone metodi per la creazione di main agent container, tramite

```
public void createMainAgentContainer(RuntimeCallback<AgentContainerHandler>
    callback);
```

  e due metodi per creare semplici agent container

```
public void createAgentContainer(String host, int port, RuntimeCallback<
    AgentContainerHandler> callback);
```

  e

```
public void createAgentContainer(Profile profile, RuntimeCallback<
    AgentContainerHandler> callback);
```

  in entrambi i casi si può notare la natura asincrona del feedback di ritorno dopo la creazione dei container (verranno approfonditi in 2.2.2);
- **Split container**: nella split execution mode non avremo modo di avviare un main container, di conseguenza i metodi per avviare un container di frontend saranno semplicemente

```
public void startAgentContainer(String host, int port, RuntimeCallback<Void>
     callback);
```

  e

```
public void startAgentContainer(Properties properties, RuntimeCallback<Void>
     callback);
```

Internamente, ogni *Binder andrà ad invocare il rispettivo metodo del *Service.
Questi sono i metodi utilizzati per avviare i container, ma naturalmente sono presenti anche quelli per creare agenti, sospenderli, killarli, ecc...

2.2.2 JADE Android Callback

Ogni operazione effettuata sul *Service, dovrà necessariamente riportare dei feedback all'app di Android, per comunicare l'esito di ogni operazione. Per fare questo ci sono due opportunità: tramite la RuntimeCallback o attraverso i RuntimeServiceListener (da notare che nel caso dello split mode è possibile solo utilizzare le RuntimeCallback).
Per sfruttare la RuntimeCallback (nel caso dello split mode è possibile solo questa via) alla chiamata di ogni metodo, si deve sempre passare un'istanza della classe jade.android.RuntimeCallback tra i parametri di input; la classe appena citata è astratta e tipizzata, infatti il parametro 'Result' assume valori diversi in base al metodo che viene invocato e a cui si passa la sua istanza in input.

```
public abstract class RuntimeCallback<Result> {
        ...
}
```

I metodi da implementare sono:

```
public abstract void onSuccess(Result result);
```

e

```
public abstract void onFailure(Throwable throwable);
```

è facile intuire il motivo per cui verrà invocato un metodo piuttosto che l'altro.
Il parametro Result può assumere all'interno della libreria tre valori:

- **Void**: nel caso in cui non venga passato nessun parametro valorizzato in input al metodo;
- **AgentHandler**: quando è necessario passare alcuni parametri relativi ad un agente (ad esempio dopo averne creato uno). Ad esempio questa classe contiene l'istanza di una classe che implementa l'interfaccia AgentController, che permette di interagire con uno specifico agente.
- **AgentContainerHandler**: quando è necessario passare alcuni parametri relativi ad un container (ad esempio dopo averne creato uno). Al suo interno ha l'istanza di AgentContainer, con cui si può interagire con il container.

BUG-FIX Riguardo gli *Handler appena descritti, è importante far notare che durante la creazione del progetto, sono stati individuati e segnalati due bug presenti nella JADE-LEAP, relativi alla visibilità dei metodi getter di AgentController e AgentContainer ed erroneamente lasciati a visibilità default, invece di essere resi public[15].

La seconda metodologia con cui ricevere dei feedback è mediante l'interfaccia RuntimeServiceListener.

```
public interface RuntimeServiceListener {
        public void onAgentCreated(AgentHandler agentHandler);

        public void onAgentStarted(AgentHandler agentHandler);

        public void onAgentKilled(AgentHandler agentHandler);

        public void onAgentSuspended(AgentHandler agentHandler);

        public void onAgentActivated(AgentHandler agentHandler);

        public void onAgentContainerCreated(AgentContainerHandler
            agentContainerHandler);

        public void onAgentContainerDestroyed(AgentContainerHandler
            agentContainerHandler);
}
```

E' possibile creare classi che implementano questa interfaccia ed utilizzare il metodo addListener() delle classi RuntimeService e RuntimeServiceBinder, per inserire anche la propria classe tra quelle già memorizzate, così da ricevere dei feedback al termine delle operazioni descritte nei metodi.
Come già anticipato, questo metodo è utilizzabile SOLO nel full container.

2.2.3 Altre classi

Per reperire alcune informazioni standard su JADE-LEAP ed in particolare su Android, vengono utilizzate delle classi *Helper, contenenti dei metodi di utilità statici.
In dettaglio, la classe RuntimeHelper, possiede i seguenti metodi:

- createProfileProperties(String, int);
- completeProfileProperties(Properties);
- createMainProfile();
- completeProfile(Profile).

Tutti i metodi appena citati vengono utilizzati per compilare/completare alcuni campi di un oggetto di tipo jade.util.leap.Properties, utilizzando dei

valori di default.
La classe AndroidHelper invece possiede i seguenti metodi:

- isEmulator();
- getLocalIPAddress().

Ovviamente questi metodi sono indirizzati esclusivamente a device Android e permettono di individuare l'esecuzione dell'app su emulatore e di ottenere l'indirizzo IP del device su cui si sta eseguendo JADE-LEAP.

BUG-FIX All'interno della classe AndroidHelper, abbiamo rilevato un altro bug ed anche in questo caso, abbiamo provveduto a segnalarlo[14]. In dettaglio il metodo getLocalIPAddress() non teneva conto del fatto di poter incontrare degli indirizzi IPV6 definiti nel device Android.

Infine, la classe RuntimeUICallback, che estende RuntimeCallback ed è astratta, permette di ricevere notifiche tramite un android.os.Handler, classe utilizzata da Android, per effettuare variazioni alla UI da un thread diverso dal main thread[3].

3 Wifi Direct

Il Wi-Fi Direct è uno standard sviluppato dalla Wi-Fi Alliance e presentato ufficialmente nell'Ottobre 2010. I dispositivi Wi-Fi CERTIFIED Wi-Fi Direct™ hanno la possibilità di connettersi l'uno con l'altro senza il bisogno di un access point wireless. E' possibile effettuare connessioni uno-a-uno o formare dei gruppi dove i device sono connessi simultaneamente. Il Wi-Fi Direct è totalmente disgiunto dalla modalità detta Ad-Hoc, piuttosto può essere visto come un'estensione della più consueta Infrastructure Mode. Oltre alla non necessità di un access point dedicato, il Wi-Fi Direct estende questa modalità in quanto un device connesso tramite Direct, può continuare a mantenere una connessione simultanea ad una rete tradizionale Infrastructure.

Alla base di questa tecnologia vi è l'implementazione nei device, di un componente software in grado di agire come access point wireless e router; grazie a questo espediente è stato anche possibile ottenere retrocompatibilità, dando così la possibilità ad un dispositivo legacy di connettersi ad una rete Wi-Fi Direct. Inoltre la garanzia di avere ogni device Direct che può fungere da access point e router in maniera dinamica, porta ad un modello di rete

P2P; non è infatti necessario avere alcun tipo di infrastruttura di rete centralizzata e predefinita.
Ecco quindi i principali benefici che i consumers hanno dai propri device Direct[1]:

- **Mobilità e portabilità:** i dispositivi Wi-Fi Direct possono connettersi ovunque in qualsiasi momento.
- **Utilità immediata:** pur avendo un solo device munito di Direct, è possibile creare connessioni direct con dispositivi legacy già in possesso.
- **Facilità d'uso:** la possibilità di effettuare Device Discovery e Service Discovery - che tratteremo più avanti - permette agli utenti di effettuare connessioni solo con i dispositivi che effettivamente rispondono a determinate esigenze.
- **Semplicità nell'avere connessioni sicure:** il Wi-Fi Direct usa Wi-Fi Protected Setup™ per semplificare la creazione di connessioni sicure; tutto quello che l'utente ha bisogno di fare è premere un bottone o digitare un PIN.

3.1 Meccanismi chiave del Wi-Fi Direct

Di seguito ecco riassunti i meccanismi o fasi che i device che si attengono allo standard dovrebbero implmentare:

- **Device Discovery** - Meccanismo per cercare i dispositivi Wi-Fi Direct.
- **Serivce Discovery** - Meccanismo opzionale che permette di scoprire - prima di effettuare la connessione - i servizi offerti dai device (es. servizio di stampa).
- **Group Formation** - Meccanismo per determinare qual è il device responsabile del gruppo ovvero il Group Owner.
- **Invitation** - Meccanismo opzionale che permette un device di invitare un altro dispositivo direct di aggregarsi ad un gruppo esistente.
- **Client Discovery** - Meccanismo che permette un device direct di scoprire quali altri device fanno parte di un gruppo direct.
- **Power Management**
  - **P2P-PS and P2P-WMM-PS** - Adattamenti dei meccanismi Power Save e WMM-Power Save per essere sfruttati al meglio in dispositivi Wi-Fi Direct.

– **Notice of Absence** - Nuova tecnica che permette un device Group Owner di ridurre il consumo di energia comunicando un'assenza pianificata.
– **Opportunistic Power Save** - Permette ad un dispositivo Group Owner di ridurre il consumo energetico entrando in uno stato "dormiveglia", quando i membri del gruppo sono nel medesimo stato.

3.2 Scenario di funzionamento del Wi-Fi Direct

I device Wi-Fi Direct si connettono formando un Gruppo secondo una topologia uno-a-uno o uno-a-molti. Un dispositivo viene detto Group Owner ed è responsabile di decidere sia quali device possano far parte del Gruppo, sia quando avviare e terminare quest'ultimo. Il Group Owner è visto come un Access Point tradizionale dai device legacy. Quale device assumerà il ruolo di Group Owner viene deciso dinamicamente in fase di formazione del Gruppo. Inoltre è possibile per un dispositivo direct, creare un gruppo stand-alone.
La Figura 24 rappresenta lo scenario tipico [13]:

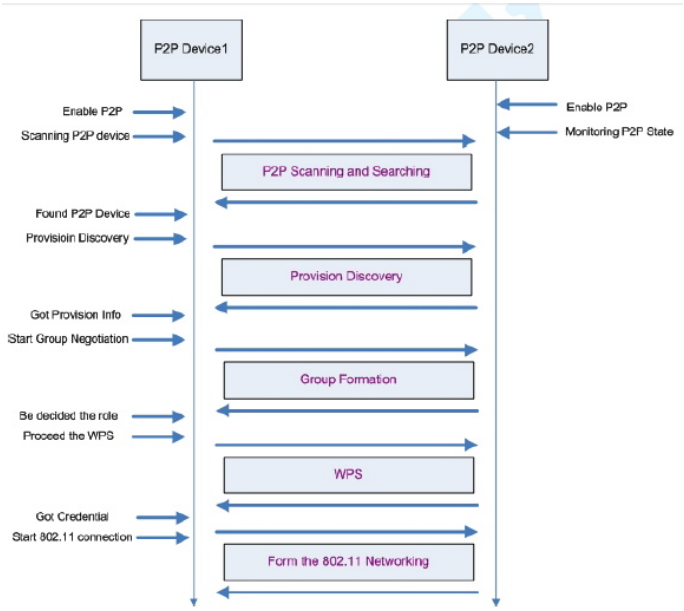

Figura 24: *Wi-Fi Direct: Scenario tipico di funzionamento*

3.3 Wi-Fi Direct in Android

A partire dalla versione Android 4.0 (API Level 14) o successive, i device con hardware appropriato hanno la possibilità di utilizzare tutti i vantaggi del Wi-Fi Direct. Vi è da specificare che - ad esempio - il Service Discovery è supportato solamente dalle API Level 17 ed inoltre, dopo aver sperimentato sul campo tali librerie, abbiamo riscontrato uno sviluppo ancora piuttosto acerbo del Direct, che ci ha obbligato all'implementazione di diversi workaround per cercare di minimizzare il più possibile i malfunzionamenti.
Le API Android del Wi-Fi Direct consistono dei seguenti componenti principali[12]:

- Metodi che permettono di effettuare il discover e la richiesta di connessione ai peers definiti nella classe WifiP2pManager
- *Listeners* che è possibile sfruttare per essere notificati del successo o del fallimento delle chiamate ai metodi di WifiP2pManager. E' infatti possibile passare implmentazioni di listeners come parametri di tali metodi.
- *Intents* che notificano specifici eventi catturati dal framework Wi-Fi Direct, come connessioni cadute o nuovi peers scoperti.

3.3.1 Esempio d'uso

Per poter interagire con il Wi-Fi Direct Framework di Android bisogna sempre eseguire dei passi base:

1. Richiedere determinati permessi inserendoli nel mandifest dell'app:

```
<uses-sdk android:minSdkVersion="14" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name="android.permission.CHANGE_NETWORK_STATE" />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

2. Creare un BroadcastReceiver che catturi gli eventi del framework:

```
/**
 * A BroadcastReceiver that notifies of important Wi-Fi p2p events.
 */
public class WiFiDirectBroadcastReceiver extends BroadcastReceiver {

    private WifiP2pManager mManager;
    private Channel mChannel;
    private MyWiFiActivity mActivity;
```

```
    public WiFiDirectBroadcastReceiver(WifiP2pManager manager, Channel
        channel,
            MyWifiActivity activity) {
        super();
        this.mManager = manager;
        this.mChannel = channel;
        this.mActivity = activity;
    }

    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();

        if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals(action)) {
            // Check to see if Wi-Fi is enabled and notify appropriate
                activity
        } else if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(
            action)) {
            // Call WifiP2pManager.requestPeers() to get a list of current
                peers
        } else if (WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals(
            action)) {
            // Respond to new connection or disconnections
        } else if (WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION.equals
            (action)) {
            // Respond to this device's wifi state changing
        }
    }
}
```

3. Ottenere nel metodo onCreate() dell'Activity un'istanza di WifiP2pManager e chiamare il metodo initialize() di tale istanza. initialize() restituirà un'istanza della classe Channel.

4. Creare un IntentFilter con gli eventi che si vorranno far catturare al BroadcastReceiver:

```
IntentFilter mIntentFilter;
...
@Override
protected void onCreate(Bundle savedInstanceState){
    ...
    mIntentFilter = new IntentFilter();
    mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION);
    mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION);
    mIntentFilter.addAction(WifiP2pManager.
        WIFI_P2P_CONNECTION_CHANGED_ACTION);
    mIntentFilter.addAction(WifiP2pManager.
        WIFI_P2P_THIS_DEVICE_CHANGED_ACTION);
    ...
}
```

5. Registrare il BroadcastReceiver con l'IntentFilter creato:

```
/* register the broadcast receiver with the intent values to be matched */
@Override
protected void onResume() {
    super.onResume();
    registerReceiver(mReceiver, mIntentFilter);
}
/* unregister the broadcast receiver */
@Override
protected void onPause() {
    super.onPause();
    unregisterReceiver(mReceiver);
}
```

6. Richiamare nei momenti opportuni i metodi discoverPeers() e connect() di WifiP2pManager.

Intent Un Intent in Android è una descrizione astratta di un'operazione che deve essere eseguita. Può essere usata per lanciare un'Activity o un Service in maniera esplicita od implicita. Inoltre tramite i BroadcastIntent è possibile comunicare con componenti chiamati BroadcastReceiver.

3.4 Un'alternativa: AllJoyn™

Fin dalle prime fasi di studio uno dei requisiti che avevamo fissato per il nostro progetto, è stato quello di voler realizzare una proximity-based P2P application, eliminando in toto l'uso di server centralizzati. Per questo, prima di optare per il Wi-Fi Direct, abbiamo eseguito delle ricerche ed in paticolare abbiamo effettuato un piccolo studio sulla piattaforma AllJoyn™.
Nel sito ufficiale è possibile trovare la seguente definizione[2]:
"AllJoyn is an open source peer-to-peer (P2P) software development framework that enables ad-hoc, proximity-based, device-to-device communication without the use of an intermediary server. It allows developers to easily write proximity-based P2P applications with a common set of APIs across hardware platforms and OS environments. AllJoyn manages the difficult connectivity, networking and security requirements of P2P communications and allows developers to focus on delivering new and exciting user experiences."
AllJoyn ci è quindi sembrato a prima vista rispondere in pieno alle nostre necessità. Approfondendone le caratteristiche abbiamo però rapidamente appreso che questa piattaforma offre ben più della sola possibilità di effettuare semplici connessioni P2P. AllJoyn può essere considerato un vero e proprio *middleware* con un supporto completo a tutte le funzionalità tipiche di tali

sistemi. E' stato quindi inutile approfondire ulteriormente l'argomento in quanto avremmo dovuto rinunciare all'uso di Jade, che dal punto di vista *middleware* ci offriva già tutto quello di cui avevamo bisogno. Inoltre abbinare due tecnologie di questo tipo, oltre che inutile ed oneroso, sarebbe stato estremamente complesso.

4 WifiDirect - Jade library

Dopo aver approfondito le questioni appena descritte, ci siamo resi conto che sviluppare una app di Android utilizzando queste tecnologie, ci portava ad affrontare molteplici problematiche di media complessità.
Abbiamo quindi deciso di informarci ulteriormente sulla possibilità di sviluppare componenti separati, ognuno responsabile di una sola di queste problematiche, ed unirli per comporre una libreria da utilizzare in seguito per lo sviluppo della app riguardante l'Hovering information.
Il framework è stato poi denominato 'Wifi p2p-Jade FWK' ed è importante analizzarlo a blocchi, per esporre in maniera esaustiva e chiara il funzionamento di ognuno di questi.

4.1 Android Library

Prima di cominciare lo sviluppo della libreria, è stato fondamentale capire come poterne creare una da riutilizzare nello sviluppo della app.
In Android creare una libreria non è semplicemente compilare delle classi Java in un jar e includere lo stesso in un altro progetto, ma è possibile ottenere un risultato analogo in due modi: attraverso Eclipse ed in particolare il suo plugin ADT (Android Development Tools)[10], oppure direttamente usando i tool messi a disposizione per creare e compilare progetti per Android da linea di comando[11].
Il metodo utilizzato da noi è il primo, che risulta molto più intuitivo ed immediato.
Per creare un progetto libreria si seguono questi passi:

- creare un nuovo progetto Android da Eclipse con il consueto: File >New >Project;
- selezionare quindi Android >Android Application Project, e click su Next;

- inserire i settaggi principali del progetto, quali il nome dell'applicazione, del progetto, del package e quali SDK sono supportate;
- nella pagina di configurazione del progetto, impostare il flag 'Mark this project as a library'.
- seguire con le altre impostazioni.

E' possibile anche convertire un progetto esistente in una library, basterà aprire le proprietà del progetto e abilitare il check su 'is Library'.

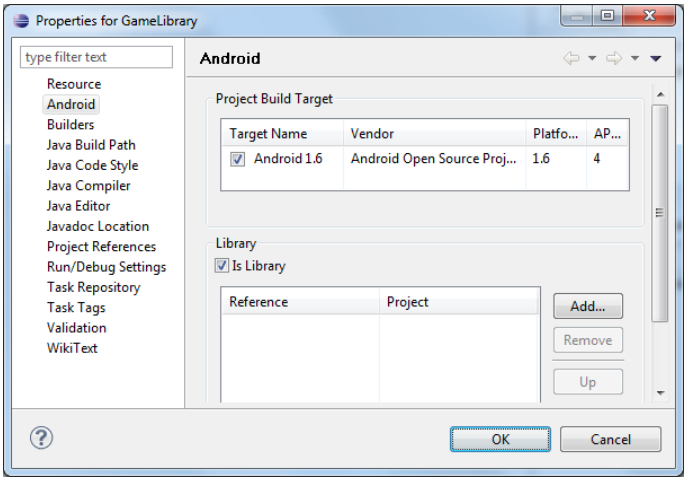

Figura 25: *ADT: come impostare il progetto 'libreria'*

Il risultato che si ottiene, seguendo queste indicazioni, è di avere un progetto completamente separato dagli altri che sarà la nostra libreria.
I progetti che invece lo vorranno utilizzare, dovranno seguire questi passi:

- innanzitutto assicurarsi che il progetto libreria sia nello stesso workspace di quello che si sta creando;
- aprire le proprietà del progetto;
- selezionare il gruppo di proprietà relative ad Android;
- nella sezione 'Library' cliccare su Add e indicare il progetto library che si intente includere.

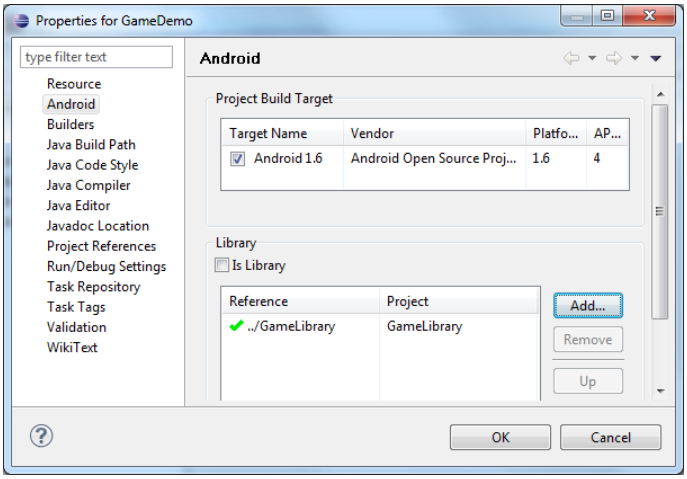

Figura 26: *ADT: come includere una 'libreria'*

4.2 Service

Risulta centrale nell'implementazione del nostro framework la classe JadeWifiDirectService, che è un Service Android. Abbiamo fatto ricorso a questo componente in quanto avevamo bisogno che la piattaforma Jade rimanesse in esecuzione anche in caso di terminazione dell'Activity principale della nostra app. L'idea è stata quella di avere un Service che fornisse supporto sia nella formazione di una connessione Wi-Fi Direct, sia nella successiva creazione ex-novo di una piattaforma Jade o aggancio di un container ad una piattaforma già in esecuzione.
Per meglio comprendere la logica di funzionamento del service, è meglio analizzare i commenti alle operazioni pubbliche che espone, definite nella classe JadeWifiDirectServiceBinder:

```
// Aggiunge un'implementazione di WiFiDirectInfoListener, per poter
    catturare
// tutti gli eventi relativi al Wi-Fi Direct
public void addWiFiDirectInfoListener(WiFiDirectInfoListener l)

// Rimuove un listener degli eventi Wi-Fi Direct
public void removeWiFiDirectFrameworkListener(WiFiDirectInfoListener l)

// Avvia il framework di gestione del Wi-Fi Direct implementato dalla classe
     WiFiDFrameworkManager
public void startWiFiDFramework()

// Restituisce le info sullo stato attuale del Wi-Fi Direct
public WifiDirectInfo getWiFiDirectInfo()
```

```
// Restituisce la lista di p2p peers rilevati
public WifiP2pDeviceList getWifiDPeerList()

// Restituisce info circa lo stato attuale di questo device relativamente al
     framework Wi-Fi Direct
public WifiP2pDevice getMyWiFiDirectDevice()

// Aggiunge un'implementazione di JadeListener, per poter catturare eventi
    sulla piattaforma Jade,
// in particolare sulla creazione di container ed agenti
public void addJadeListener(JadeListener l)

// Rimuove un listener degli eventi di Jade
public void removeJadeListener(JadeListener l)

// Avvia una nuova piattaforma Jade effettuando prima di tutto -
    trasparentemente -
// una connessione ad un device direct e poi avviando un main container o un
     container normale
public void startNewJadePlatform(WifiP2pDevice device)

// Verifica che la piattaforma Jade sia correttamente in esecuzione
public boolean platformReady()

// Crea un nuovo agente
public void createAgent(String agentName, String className, Object[] args)

// Restituisce un'interfaccia per comunicare con un agente da un'oggetto che
// non sia anch'esso un agente
public <T> T getO2AInterface(String agentName, Class<T> theInterface)
```

Vale la pena spendere qualche parola in più sul metodo startNewJadePlatform(). Come detto, prima di tutto ci si occupa di effettuare una connessione con il device direct passato come argomento. La connessione può avvenire con un device non ancora connesso oppure con un group owner, che per nostra decisione sarà sempre l'host di un main container. Una volta effettuata la connessione in maniera totalmente trasparente per il client, ci potranno essere due alternative:

1. Nel caso in cui i due device non fossero stati connessi, allora si verifica quale dei due abbia assunto il ruolo di group owner e su quest'ultimo viene eseguito un main container, mentre nell'altro un container normale.

2. Nel caso in cui ci si stesse connettendo ad un group owner ovvero ad un main container già in esecuzione, allora il device richiedente, una volta ottenuta la connessione direct, lancia un container normale che si aggancia al main.

I servizi offerti da JadeWifiDirectService non sono implementati direttamente da questa classe, ma da due classi principali: WiFiDFrameworkManager e JadeManager. Per i dettagli rimandiamo alle relative sezioni:4.3 4.4.

4.3 Wifi Direct

Come già detto in 3.3, la difficoltà maggiore nell'uso del Wi-Fi Direct, è stata sicuramente l'implementazione piuttosto approssimativa di tale tecnologia in Android. Per questo abbiamo incapsulato i vari workaround all'interno della classe WiFiDFrameworkManager, che svolge il ruolo centrale di "ponte" tra il framework Android del Wi-Fi Direct e la nostra app.
Le classi principali del nostro piccolo framework Wi-Fi Direct sono:

- **WiFiDFrameworkManager**
- **WiFiDirectBroadcastReceiver**
- **WifiDirectInfo**
- **WiFiDirectInfoListener**

4.3.1 WiFiDFrameworkManager

Questa è la classe centrale del framework. La classe si occupa di inizialiazzare un'istanza di WifiP2pManager e registrare un'istanza WiFiDirectBroadcastReceiver come receiver agli eventi generati da Android. Tale classe si occupa anche di mantenere il più possibile allineate ed aggiornate le info riguardanti lo stato attuale del Wi-Fi Direct, avvalendosi della classe WifiDirectInfo che incapsula tutti i dati necessari:

- **WifiDirectState**
- **WifiDirectConnectionState**
- **WifiDirectPeerListState**
- **WifiDirectFrameworkState**
- **WifiDirectDiscoveringState**
- **WiFiDirectDeviceState**

BroadcastReceiver Un BroadcastReceiver in Android è uno dei componenti principali grazie al quale è possibile ricevere particolari messaggi broadcast che possono essere generati dalla nostra app, da altre app o dal sistema stesso.

Per utilizzare la classe è sufficiente ottenerne un riferimento tramite il metodo statico getInstance() passandogli un'implementazione dell'interfaccia WiFiDi-

rectInfoListener, grazie alla quale si potranno catturare gli eventi generati dal Wi-Fi Direct. L'unico metodo dell'interfaccia da implementare è:

```
void onWifiDInfoChange(WifiDirectEvent event, WifiDirectInfo info, Object
extraInfo);
```

dove **event** definisce l'evento appena accaduto, **info** lo stato attuale del direct ed **extraInfo** eventuali dati aggiuntivi legati solo a particolari eventi. Una volta ottenura un'istanza di WifiP2pManager è necessario richiamare il metodo startFramework() per cominciare a ricevere le notifiche degli eventi. Inoltre WifiP2pManager fornisce il metodo connectToPeer(WifiP2pDevice device) con il quale ci si potrà connettere al device passato come argomento.

4.3.2 WiFiDirectBroadcastReceiver

La classe con cui WiFiDFrameworkManager cattura gli eventi del direct generati da Android è WiFiDirectBroadcastReceiver, che estende la classe Android BroadcastReceiver. WiFiDirectBroadcastReceiver comunica con WiFiDFrameworkManager mediante i seguenti metodi di callback:

```
void onAvailablePeerListChanged();
void onPersonalInfoChanged(WifiP2pDevice me);
void onWiFiDConnectionChanged(NetworkInfo netInfo);
void onWiFiDDiscoveryChanged(boolean isDicoveryStarted);
void onWiFiDStateChanged(WifiDirectState wDState);
```

La Figura 27 riporta il diagramma con la maggior parte delle classi appena descritte:

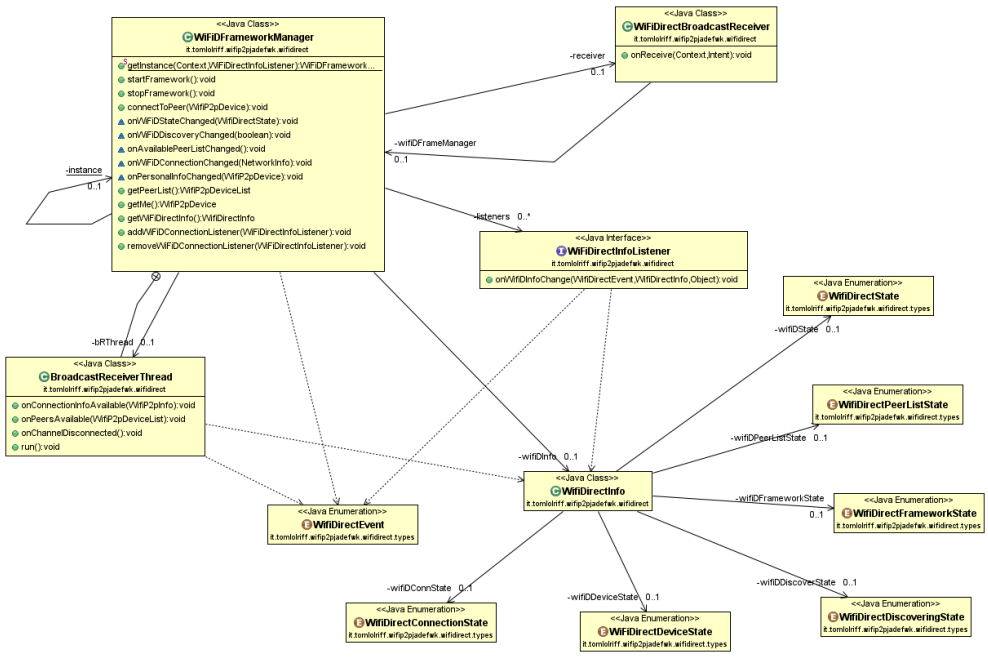

Figura 27: *UML: Wi-Fi Direct Framework*

4.4 JADE

Il Service descritto in precedenza 4.2 offre i servizi di Jade comunicando con la classe JadeManager, che è quella che effettivamente interagisce con la libreria Jade-LEAP Android 2.2. La classe è un singleton che presenta i seguenti metodi pubblici:

```
public void addJadeListener(JadeListener l)
public void createAgent(final String agentName, String className, Object[]
    args) throws AgentContainerNotStartedException
public void createAgentContainer(InetAddress mainContainerHostAddress)
public void createMainAgentContainer()
public <T> T getO2AInterface(String agentName, Class<T> theInterface) throws
     StaleProxyException, AgentContainerNotStartedException
public void killAgentContainer(final CountDownLatch latch)
public void removeJadeListener(JadeListener l)
```

Analizziamo per primi i metodi per la creazione di un main container e di un container normale.

4.4.1 Creazione dei container

In questa fase la scelta più importante è stata quella di decidere quale container execution mode 2.1.3 adottare. Durante la definizione dei requisiti ci siamo posti l'obiettivo di sviluppare un'architettura che fosse il più possibile decentralizzata. Sappiamo che Jade non offre la possibilità costruirne una "pura", a causa della necessità di avere un nodo con un main container; questo infatti porta a far assumere al nodo stesso un ruolo centrale, dato che il main container è responsabile di tutta la piattaforma Jade.
Al fine di non rendere ancora più "centrali" i device che avrebbero ospitato un main container, abbiamo quindi deciso di non far ricorso alla split container mode, ma ad una configurazione di container classica. Così gli host dei main container non devono gestire alcun back-end ed inoltre, il fatto stesso che il ruolo di main container potrebbe essere assunto dinamicamente da qualsiasi device 4.2, rende - almeno in parte - Jade più decentralizzato.
Un'ulteriore spinta verso questa decisione è stata la maturità tecnologica degli attuali device (smartphone e tablet), che hanno una potenza dell'hardware sempre maggiore, risolvendo così le problematiche per le quali gli sviluppatori di Jade LEAP hanno deciso di implementare la split execution mode 2.1.3.
Per la creazione dei container, main o normali, abbiamo fatto ricorso ad un'interfaccia chiamata JadeStarter che definisce le seguenti operazioni:

```
// Crea un nuovo Main Container
void createMainAgentContainer();

// Crea un container che si deve collegare al Main Container presente nell'
    host
// con indirizzo IP mainContainerHostAddress
// @param mainContainerHostAddress {@link String} - indirizzo IP dell'host
    che contiene il Main Container
void createAgentContainer(String mainContainerHostAddress);

// Notifica la creazione del container avveunta con successo
// @param result {@link AgentContainerHandler} - oggetto che espone metodi
// per interagire con il Container appena creato
// @param runtimeService {@link RuntimeService} - oggetto che espone metodi
// per interagire con il Container appena creato
void onSuccess(AgentContainerHandler result, RuntimeService runtimeService);

// Notifica un errore nella creazione del container
// @param error {@link Throwable} - dettagli sull'errore
void onFailure(Throwable error);
```

Il fine di questa interfaccia è stato quello di avere un meccanismo di avvio dei container personalizzabile. In particolare la nostra esigenza era quella di poter avviare un container, solo quando il main container era stato avvi-

ato e pronto a ricevere la registrazione di altri container. Abbiamo perciò realizzato la classe astratta JadeStarterImpl che implementa dell'interfaccia JadeStarter i soli metodi di creazione dei container, lasciando alle classi figlie il compito di realizzare i metodi di successo o fallimento delle operazioni. Per poter essere sicuri di avere la creazione di un container normale solo avendo già a disposizione un main container in esecuzione, abbiamo adottato una tecnica semplice ma efficace: prima di lanciare un'eccezione vengono fatti NUM_MAX_RETRY e ad ogni tentativo si attende un DELAY_MAX.
A questo punto JadeManager nei metodi createAgentContainer() e createMainAgentContainer(), non fa altro che fornire un'implementazione anonima di JadeStarterImpl, scrivendo inoltre le proprie versioni di onSuccess() e onFailure() per catturare messaggi di callback specifici.

4.4.2 Creazione degli agenti

Per la creazione di un agente, JadeManager invia messaggi direttamente ad un oggetto di tipo RuntimeService 2.2.1 di Jade LEAP Android, fornendo inoltre un'implementazione anonima di new RuntimeCallback <AgentHandler >(), già ampiamente discussa in 2.2.2. E' necessario specificare che il metodo createAgent(), in maniera trasparente, prima crea l'agente e poi - in caso di successo - lo avvia. Al successo dell'avvio dell'agente viene inoltre valorizzato un record in una mappa, la cui chiave è il nome dell'agente e il valore è un oggetto di tipo AgentController 2.2.2. Questa struttura verrà poi sfruttata dal metodo di JadeManager getO2AInterface(String, Class<T >), che dovrà restituire un'interfaccia grazie alla quale sarà possibile comunicare con l'agente da un normale oggetto java.

4.4.3 JadeListener

Le classi client di JadeManager possono essere informate degli eventi di Jade, realizzando l'interfaccia JadeListener e passandone il riferimento tramite il metodo getInstance(JadeListener listener) o addJadeListener(JadeListener l). JadeListner definisce le seguenti operazioni, il cui significato è facilmente deducibile dai lorno nomi:

```
void onMainContainerSuccessfullyStarted();

void onMainContainerFailureStarted(Throwable throwable);

void onContainerSuccessfullyStarted();
```

```
void onContainerFailureStarted(Throwable throwable);

void onAgentSuccessfullyStarted(AgentInfo agentInfo);

void onAgentFailureStarted(Throwable throwable);
```

4.4.4 Diagramma delle classi

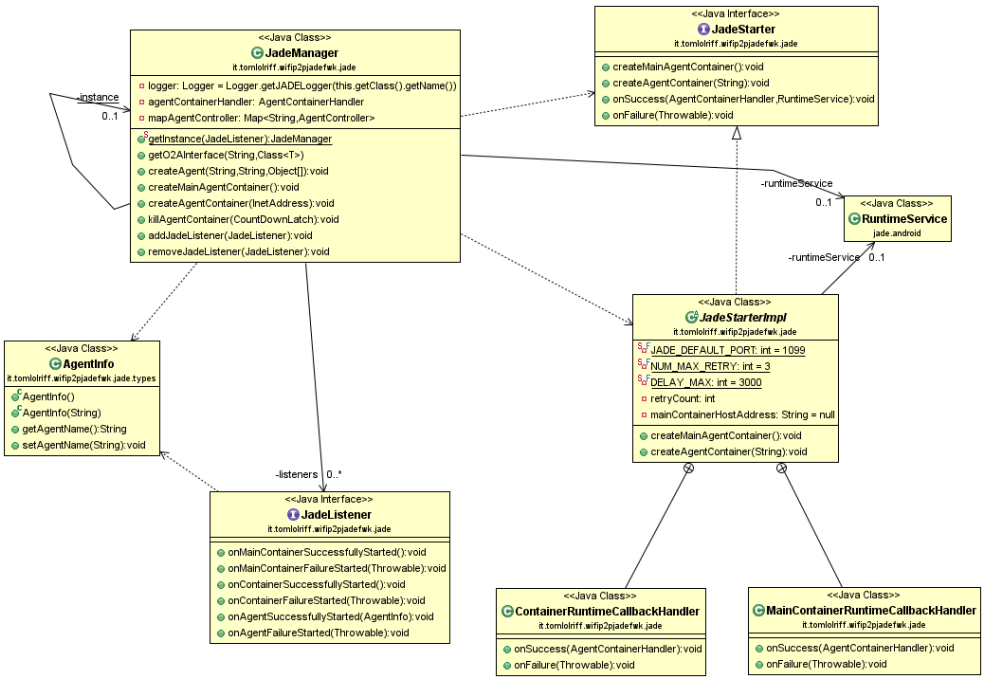

Figura 28: *UML: Jade Framework*

4.5 Activity

L'intero framework che abbiamo realizzato è pensato per essere usato dalle app semplicemente estendendo la classe JadeWifiDirectActivity che è un'Activity Android. E' necessario a questo punto spendere qualche riga che spieghi che cos'è un'Activity in Android.

4.5.1 Il componente Activity in Android

Citando la definizione che ci dà la documentazione Android[7] un'Activity è: *un'application component che fornisce una schermata con la quale l'utente può interagire per fare qualsiasi cosa.* Ad ogni Activity è data una finestra nella quale disegnare la UI. Un'app di solito consiste di attività multiple che sono non strettamente legate. Tipicamente in ogni app è presente una main Activity che è la prima visualizzata all'utente. Ogni Activity può lanciare un'altra Activity per portare a termine delle azioni ed ogni volta che una nuova activity viene eseguita, quella precedente viene terminata e salvata in uno stack; in questo modo l'utente premendo il tasto *indietro*, può tornare alla schermata precedente.
Centrale nella comprensione di questo componente, è la conoscenza del suo ciclo di vita ed in che modo vengano richiamati dal sistema Android, metodi specifici ad ogni cambiamento di stato dell'activity stessa. Questi metodi vengono poi sovrascritti dalle app che ne faranno uso. Un'Activity ha essenzialmente i seguenti stati[8]:

- Se un'activity è in foreground ovvero in cima allo stack, allora è nello stato *alive* o *running*.
- Se un'activity ha perso il focus - ma è ancora visibile - allora è nello stato *paused*. In questo stato è ancora completamente *alive*, ma potrebbe venire terminata dal sistema in situazioni di estrema necessità.
- Se un'activity è stata completamente oscurata da un'altra activity, allora è nello stato *stopped*. Sarà spesso terminata dal sistema in caso di necessità.
- Nel caso in cui un'activity, precedentemente terminata dal sistema, debba essere visualizzata nuovamente all'utente, dovrà essere completamente riavviata e ripristinato il suo stato precedente.

La Figura 29 esplica l'intero ciclo di vita e i metodi che vengono chiamati dal sistema durante le transizioni di stato:

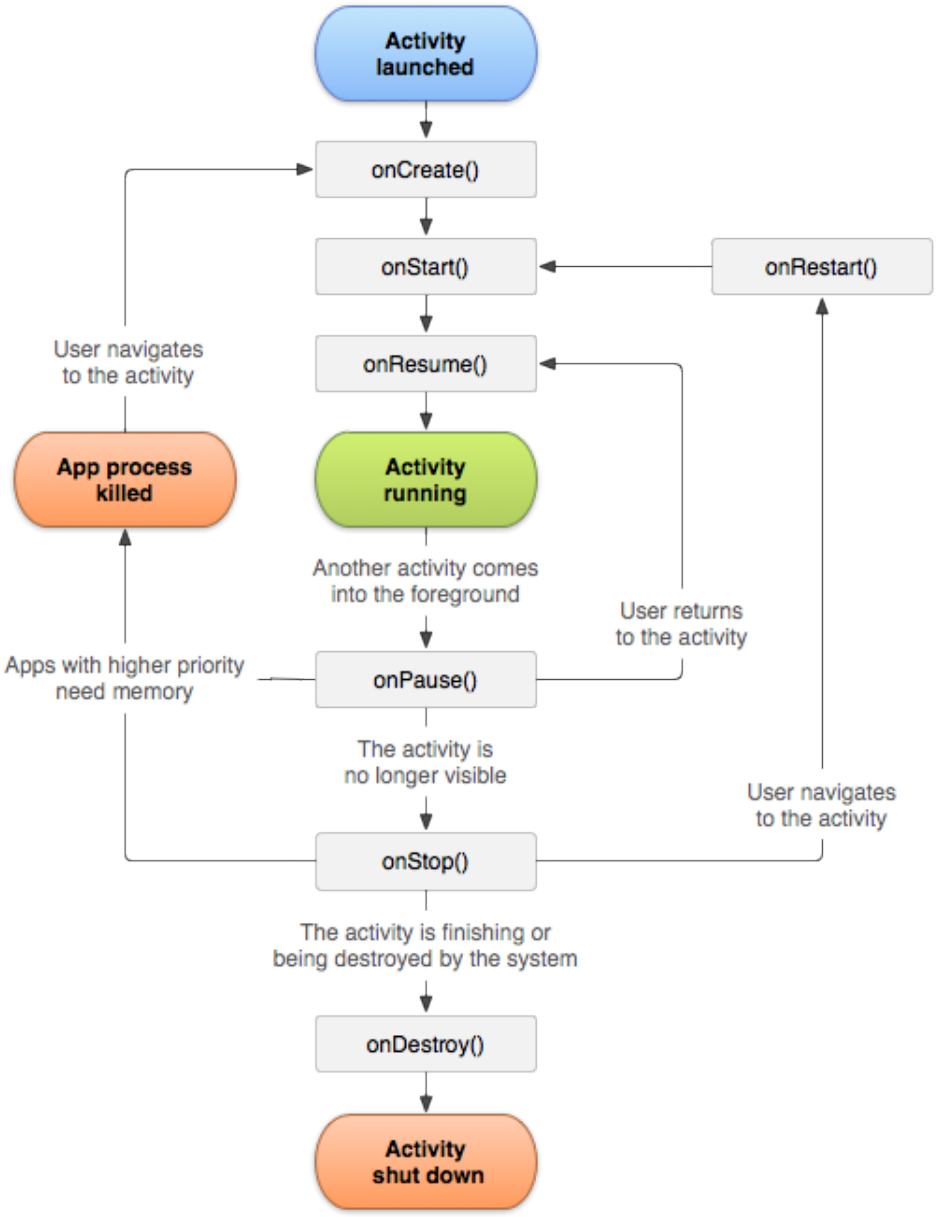

Figura 29: *Activity Lifecycle*

4.5.2 JadeWifiDirectActivity

Quello che abbiamo voluto realizzare è un'Activity astratta che offrisse alle classi figlie di qualsiasi app tutte le funzionalità per interagire da un lato con il Wi-Fi Direct, e dall'altro con Jade. JadeWifiDirectActivity infatti espone dei metodi per effettuare una connessione direct e per avviare una piattaforma jade. La classe sfrutta i servizi offerti da JadeWifiDirectService 4.2, effettuando il binding con questo nel metodo sovrascritto onResume(). JadeWifiDirectActivity definisce i seguenti metodi protected final:

```
/**
 * Connessione ad un Peer su piattafomra WiFi Direct + Jade
 * @param device {@link WifiP2pDevice} - peer a cui connettersi
 */
protected final void connectToPeer(WifiP2pDevice device)

/**
 * Resituisce la lista dei P2P Device raggiungibili
 * @return {@link WifiP2pDeviceList} - la lista dei P2P Device
 */
protected final WifiP2pDeviceList getCurrentWifiP2pDeviceList()

/**
 * Restituisce il nome Direct di questo dispositivo
 * @return {@link String} - nome del dispositivo
 */
protected final String getMyDirectName()

/**
 * Crea ed esegue un nuovo agente Jade
 * @param agentName {@link String} - nome del nuovo agente
 * @param className {@link String} - nome della classe che implementa l'
     agente
 * @param args {@link Object}[] - ulteriori argomenti da passare all'agente;
      puo' essere null
 */
protected final void createAndStartAgent(String agentName, String className,
     Object[] args)

/**
 * Permette di recuperare per l'agente <code>agentName</code>, l'interfaccia
      <code>theInterface</code>
 * @param agentName il nome dell'agente
 * @param theInterface l'interfaccia richiesta
 * @return l'interfaccia <code>theInterface</code> per l'agente <code>
     agentName</code>
 */
protected final <T> T getO2AInterface(String agentName, Class<T>
    theInterface)

/**
 * Indica se la piattaforma per eseguire le operazioni su jade e' pronta
 * @return <code>true</code> se la piattaforma per eseguire operazioni su
     jade e' pronta, <code>false</code> altrimenti
```

```
 */
protected final boolean isPlatformReady()
```

i cui commenti sono ampiamente esplicativi.
La classe inoltre definisce tutta una serie di metodi di callback protected che è possibile sovrascrivere e che verranno chiamati a seguito di determinati eventi. Anche in questo caso la loro elencazione dovrebbe essere sufficiente a capirne il significato:

```
/**
 * Richiamato quando ci si connette al Service
 * @param wifiDirectInfo {@link WifiDirectInfo} - info sullo stato attuale
     del WiFi Direct
 */
protected void onServiceConnectionConnected(WifiDirectInfo wifiDirectInfo)

/**
 * Richiamato quando si perde la connessione al Service
 */
protected void onServiceConnectionDisconnected()

/**
 * Richiamato quando e' avvenuto un evento WiFi Direct
 * @param event {@link WifiDirectEvent} - evento accaduto
 * @param info {@link WifiDirectInfo} - info sullo stato del direct a
     seguito dell'evento
 * @param extraInfo {@link Object} - ulterio dati extra legati a particolari
      eventi
 */
protected void doOnWifiDInfoChange(WifiDirectEvent event,
                WifiDirectInfo info, Object extraInfo)

/**
 * Richiamato quando un Main Container Jade e' stato creato con successo.
 * Se necessario e' possibile effettuarne l'override.
 */
protected void doOnMainContainerSuccessfullyStarted()

/**
 * Richiamato quando un Main Container Jade non e' stato creato
     correttamente.
 * Se necessario e' possibile effettuarne l'override.
 * @param throwable {@link Throwable} - dettaglio dell'errore
 */
protected void doOnMainContainerFailureStarted(Throwable throwable)

/**
 * Richiamato quando un Container Jade e' stato creato con successo.
 * Se necessario e' possibile effettuarne l'override.
 */
protected void doOnContainerSuccessfullyStarted()

/**
 * Richiamato quando un Container Jade non e' stato creato correttamente.
 * Se necessario e' possibile effettuarne l'override.
 * @param throwable {@link Throwable} - dettaglio dell'errore
```

```
 */
protected void doOnContainerFailureStarted(Throwable throwable)

/**
 * Richiamato quando un Agente Jade e' stato creato ed avviato con successo.
 * Se necessario e' possibile effettuarne l'override.
 * @param {@link AgentInfo} - info dell'agente appena creato
 */
protected void doOnAgentSuccessfullyStarted(AgentInfo agentInfo)

/**
 * Richiamato quando un Agente Jade non e' stato creato ed avviato
     correttamente.
 * Se necessario e' possibile effettuarne l'override.
 * @param throwable {@link Throwable} - dettaglio dell'errore
 */
protected void doOnAgentFailureStarted(Throwable throwable)
```

4.5.3 Diagramma delle classi

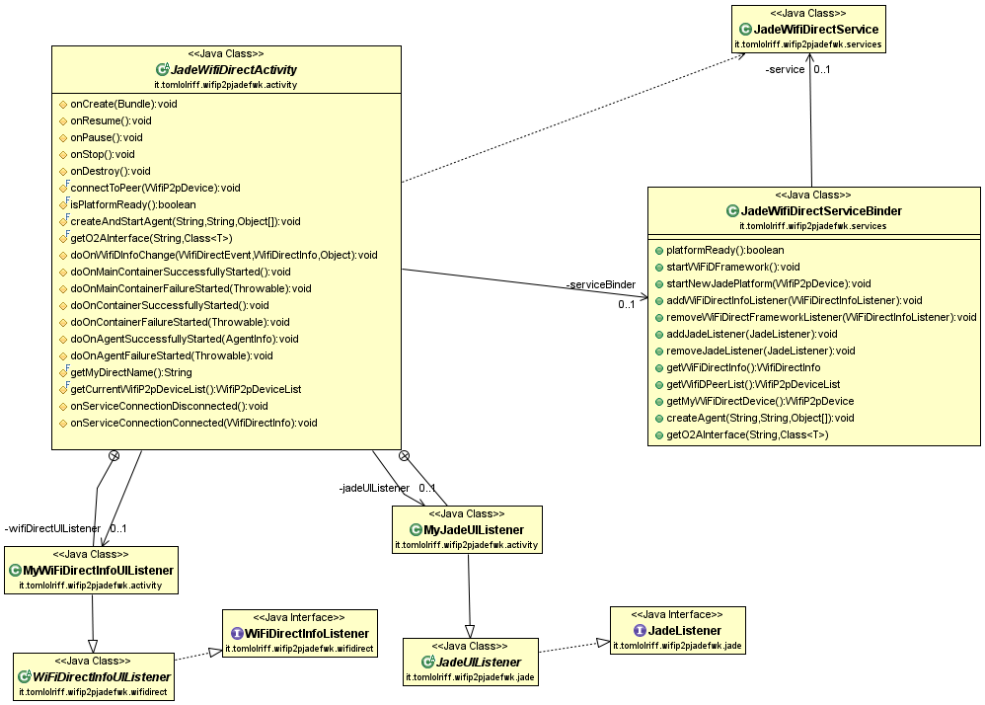

Figura 30: *UML: JadeWifiDirectActivity*

4.6 Location

In Android è possibile determinare la posizione geografica di un device optando per due modalità distinte: **GPS** e **Network Location Provider**. Il GPS, come sappiamo, è più accurato, ma non funziona all'interno di edifici, cosuma molta energia e non è rapidissimo nel determinare la location. Il Network Location Provider determina la posizione sfruttando le torri cellulari e il segnale Wi-Fi, funzionando anche negli interni e fornendo una risposta rapida consumando meno batteria. Di contro, il risultato che restituiscono non è precisio quanto quello rilevato dal GPS[9]. In più Android offre la possibilità di implementare dei 'Mock Provider' personalizzati al fine di rendere più agevole lo sviluppo e il debug, senza dover ricorrere a reali spostamenti fisici durante queste fasi; grazie ai mock provider è possibile simulare qualsiasi tipo di provider (network o gps), generando in qualsiasi momento Location fittizie.
Per le considerazioni appena dettagliate, la documentazione tecnica di Android suggerisce di fare molta attenzione quando si deve rilevare la location. Il giusto approccio è quello di effettuare un compromesso tra la precisione del dato e il cosumo della batteria, naturalmente senza tralasciare l'utente che si aspetta un risposta rapida. Tipicamente si procede secondo questi punti:

1. Avvio dell'applicazione.
2. In un momento successivo, inizio del rilevamento della posizione mediante i provider desiderati.
3. Mantenimento di una "Current Best Location", scartando i nuovo risultati qualora fossero meno accurati.
4. Fine del rilevamento.
5. Uso dell'ultima best location rilevata.

4.6.1 LocationStrategy

Seguendo i punti precedenti abbiamo deciso di implementare la classe LocationStrategy che rappresenta una strategia di geolocalizzazione, con la possibilità di decidere quali provider utilizzare: network, gps o mock. La classe è astratta e rende disponibili dei metodi da sovrascrivere:

```
protected void doOnGpsStatusChanged(int event)
```

per poter eseguire delle operazioni a seguito del cambiamento di stato del GPS.

```
protected void doOnLocationChanged(Location location)
```

per poter eseguire delle operazioni a seguito del cambiamento dell'attuale location.

```
protected void doOnStatusChanged(String provider, int status, Bundle extras)
```

per poter eseguire delle operazioni a seguito del cambiamento di stato del provider.

```
protected void doOnProviderEnabled(String provider)
```

per poter eseguire delle operazioni a seguito dell'abilitazione del provider.

```
protected void doOnProviderDisabled(String provider)
```

per poter eseguire delle operazioni a seguito della disabilitazione del provider.
Inoltre la classe definisce la seguente interfaccia:

```
interface Listener {
        void onCurrentBestLocationChange(Location
            currentBestLocation);
}
```

che i client possono implementare per essere informati quando la best location corrente è stata aggiornata.
Le classi che implementeranno una LocationStrategy dovranno passare al costruttore della classe padre, la modalità con cui vogliono gestire la propria strategy. E' suffciente mettere in or bit a bit i campi definiti in LocationStrategyMode: NETWORK, GPS, MOCK. Quindi la strategy può funzionare contemporaneamente secondo più modalità. In caso una delle opzioni fosse la modalità mock, mediante il metodo:

```
public final void addMockProvider(MockProvider
    mockProvider)
```

sarà possibile aggiungere tutti i mock provider personalizzati che si vogliono.
Per avviare la strategy e così rilevare la best location, i client dovranno chiamare il metodo:

```
public final void start()
```

analogamente per terminare la rilevazione basterà richiamare:

```
public final void stop()
```

4.6.2 DummyMockLocationStrategy

Data la natura del nostro progetto, ancora in fase prototipale, è stato per noi sufficiente implementare una semplicissima strategy singleton in modalità MOCK che si avvalesse di un unico mock provider. La classe sovrascrive semplicemente doOnLocationChanged():

```
@Override
protected void doOnLocationChanged(Location location) {
        setBestLocation(location);
}
```

che ad ogni cambiamento di location non fa nient'altro che modificare la current best location, senza fare altri controlli.

5 Hovering information App

L'app realizzata rappresenta una possibile soluzione implementativa del concetto di 'Hovering information' 1, ed in particolare ha lo scopo di unire le tecnologie descritte precedentemente e riuscire ad utilizzarle per creare una base software, su cui è possibile sviluppare algoritmi differenti relativi all'hovering information; infine propone due possibili implementazioni di algoritmi 5.7.
In maniera più approfondita, possiamo scroporare la applicazione innanzitutto in due macro sezioni: per quanto riguarda la trasmissione tra dispositivi Android si è utilizzato il Wifi Direct 3, mentre per implementare la logica dell'hovering information, si è utilizzato il paradigma di programmazione multi agente, ed in particolare la piattaforma JADE-LEAP orientata a dispositivi Android 2.
Seguiranno delle sezioni che andranno in dettaglio a definire l'applicazione realizzata dal punto di vista funzionale e da quello architetturale.

5.1 Funzionalità (casi d'uso)

Le funzionalità implementate dall'applicazione realizzata, sono le seguenti:

- **gestione connessione wifi direct:** l'utente si deve poter connettere ad una rete già esistente di dispositivi connessi tramite wifi direct o crearne una nuova con un secondo device;
- **visionare la lista dei piece:** vedere la lista dei 'piece' di hovering information attualmente sul device.

- **creare nuovi piece:** creare dei 'piece' di hovering information nuovi, scegliendo l'algoritmo di replication da utilizzare;
- **simulare lo spostamento:** deve poter simulare il cambio di 'zona', in modo da definire un comportamente significativo per l'hovering information;

Per ogni funzionalità verrà illustrato il caso d'uso relativo.

5.1.1 Gestione wifi direct

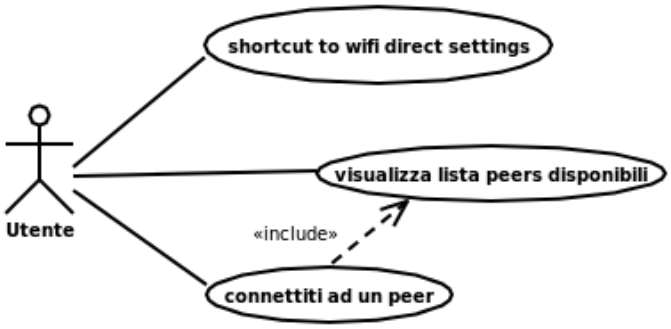

Figura 31: *Use case: gestione connessione device wifi direct*

L'utente, appena accede all'app, potrà visionare un tab chiamato CONNESSIONE, con una schermata contenente lo stato del wifi direct, il nome del proprio device visibile ai possibili peers e l'elenco dei possibili peers (vedi fig. 32).

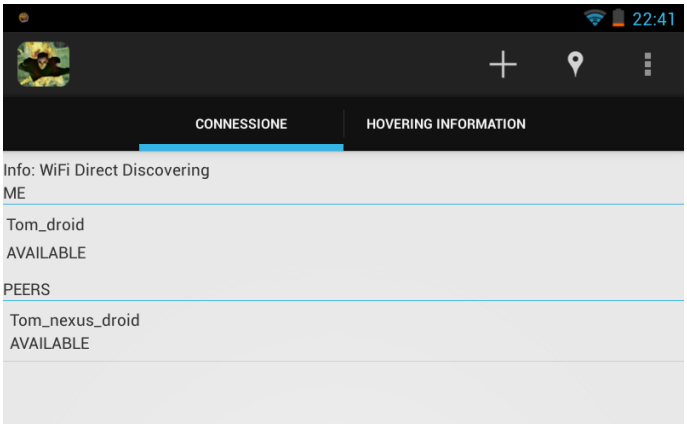

Figura 32: *Screenshot: gestione connessione*

Naturalmente, per poter utilizzare l'app, è necessario che l'utente abbia abilitato il wifi direct, in caso contrario deve entrare nelle impostazioni e farlo (è possibile sfruttare l'apposita opzione, che permette di andare direttamente nella pagina delle impostazioni).
Quando nell'elenco dei peers disponibili si vedranno altri device, con un tap in corrispondenza del peer, sarà possibile inviare una richiesta di collegamento tramite wifi direct. E' interessante notare che nell'elenco dei peers, in caso sia già stato creato un gruppo, viene indicato chi è il group owner, che nel nostro caso corrisponde esattamente a chi è main container in jade 4.2. In ogni istante sarà possibile vedere, tramite questa schermata, in che stato è il nostro device in termini di collegamento, quindi se non è connesso, se è connesso, se ha appena inviato un invito alla connessione.
Quando la connessione viene stabilita, sarà possibile passare alle altre funzionalità, altrimenti inibite.
Nella figura 33 è possibile vedere un device (Tom_droid), con wifi direct abilitato e una connessione ad un secondo device (Tom_nexus_droid).

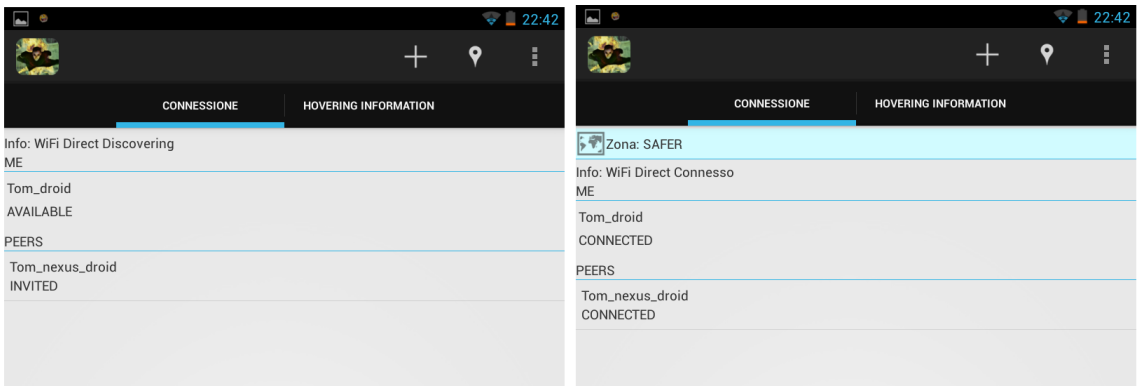

Figura 33: *Screenshot: invito alla connessione e connessione avvenuta*

5.1.2 Visualizzazione 'piece' presenti sul device e creazione

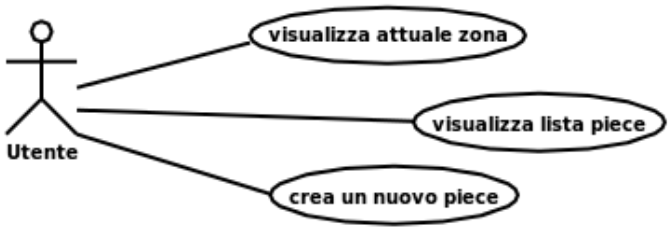

Figura 34: *Use case: visualizzazioni piece e creazione*

Quando il device è connesso ad una rete wifi direct, l'utente può cominciare ad utilizzare le funzioni specifiche dell'hovering information, passando alla apposita schermata, posizionata nel tab HOVERING INFORMATION.
Innanzitutto è possibile visionare la zona in cui si trova (da notare che ad ogni creazione di un nuovo piece e appena connesso, il device si imposta in SAFER zone) e questa informazione viene aggiornata ovviamente in tempo reale anche dopo gli spostamenti (questo dato è presente anche nel primo tab).
Nella stessa schermata si può visualizzare la lista dei 'piece' presenti sul device. Quella mostrata è una semplice lista, contenente la stringa inserita e l'identificativo del piece (è proprio l'AID dell'agente).

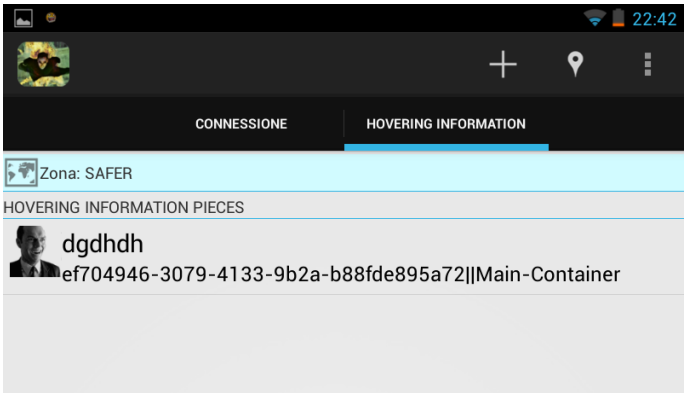

Figura 35: *Screenshot: lista di 'piece'*

La creazione di un nuovo piece è possibile tramite il pulsante di figura 36; cliccandolo si aprirà una popup in cui si può specificare la stringa che verrà contenuta nel 'piece' e l'algoritmo di replicazione scelto 5.7.

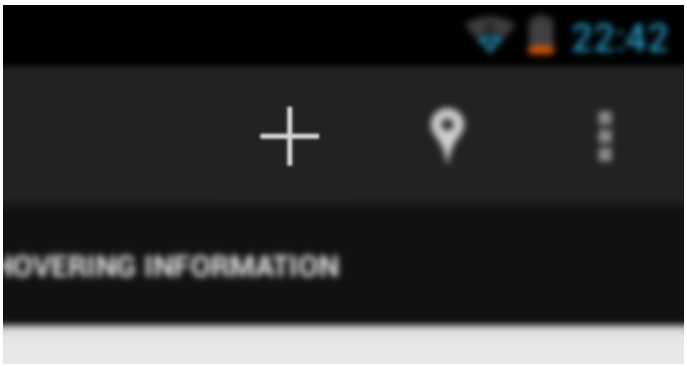

Figura 36: *Screenshot: pulsante inserimento nuovo 'piece'*

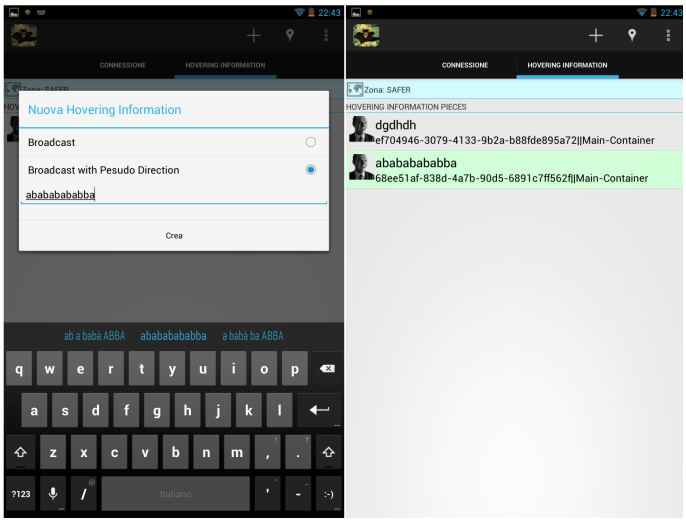

Figura 37: *Screenshot: inserimento nuovo 'piece'*

Come anticipato, ad ogni creazione il device viene 'spostato' in SAFE zone.

5.1.3 Spostamento di 'zone'

Figura 38: *Use case: simulazione spostamento*

Come già detto in 1, il concetto di hovering information si basa sulla localizzazione del nodo che possiede l'informazione, quindi è stato necessario implementare una funzionalità che permetta in maniera semplice ed immediata, tramite 4.6.2, di simulare degli spostamenti. Si è scelto direttamente di 'spostare' virtualmente i device da una zona ad un'altra, senza specificare le singole coordinate (che sono comunque state utilizzate, ma decise in fase di progettazione).
La funzionalità di *simulazione dello spostamento* si abilita tramite il pulsante in figura 39

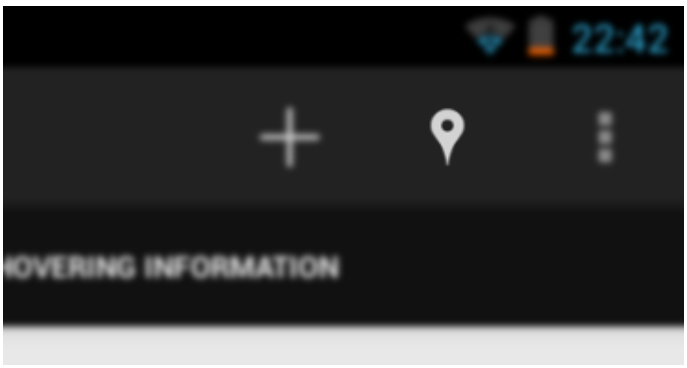

Figura 39: *Screenshot: pulsante simula spostamento*

quando viene cliccato si aprirà una popup come quella visibile in figura 40, in cui si sceglie semplicemente la zona nella quale si vuole 'andare', tramite un radio button. Ad ogni spostamento gli agenti agiranno in base all'algoritmo indicato in fase di creazione. Nella figura 40 è inoltre visibile, oltre alla popup, le conseguenze di un cambio di zona da SAFE a RISK: i piece del device si sono clonati sul secondo dispositivo.

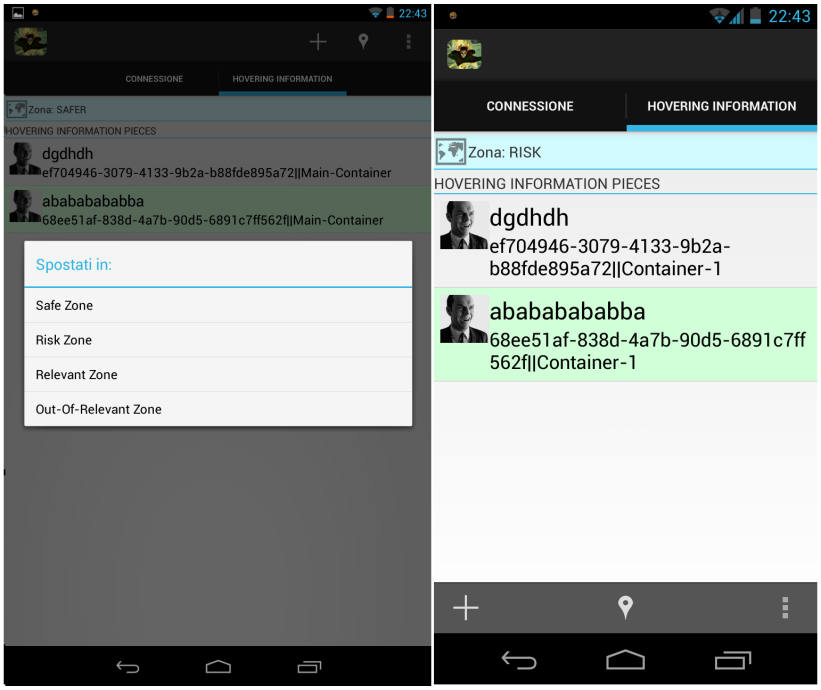

Figura 40: *Screenshot: cambio zona ed esito spostamento sul secondo device*

5.1.4 Terminazione service

Utilizzando il service, una volta usciti dall'applicazione, questo rimarrà in esecuzione e per terminarlo basta cliccare il pulsante Stop service dalla barra di notifica.

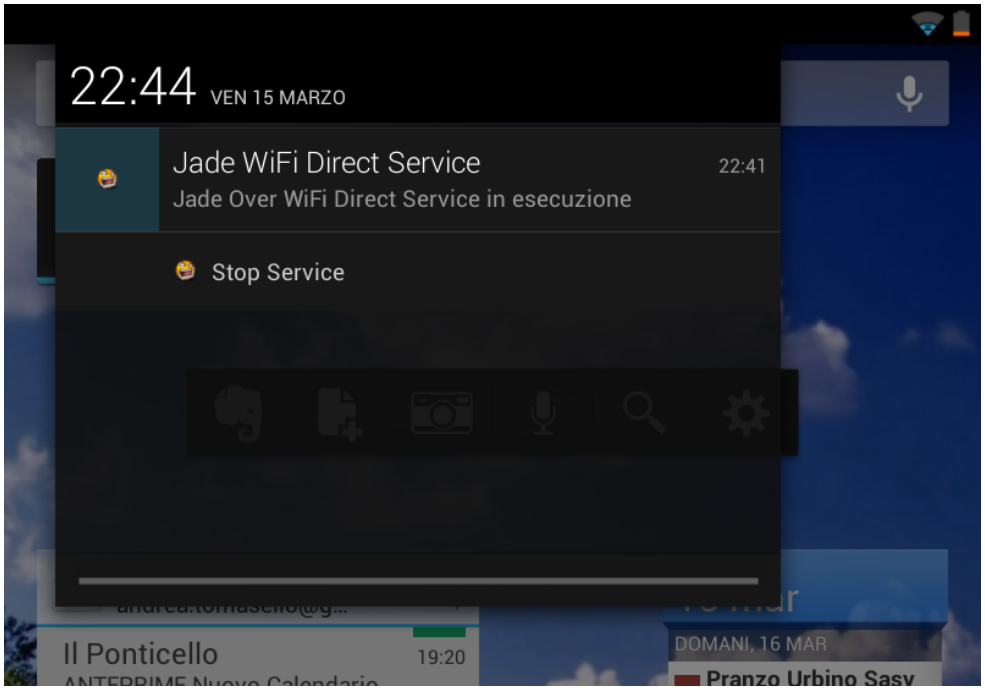

Figura 41: *Screenshot: stop service*

5.2 HoveringInformationActivity

In questo paragrafo descriveremo l'Activity principale della nostra app: HoveringInformationActivity. Evidentemente questa classe estenderà JadeWifiDirectActivity 4.5 e ne sfrutterà tutte le funzionalità di cui avrà bisogno. Un altro dei compiti principali di questa classe è quello di definire la struttura del layout della UI definendo nel metodo sovrascritto onCreate(), i seguenti componenti Android:

- un ViewPager che gestirà la transizione tra i Fragment mediante il meccanismo dello 'swipe';
- la Action Bar per la navigazione mediante Tabs e la visualizzazione di voci di menù;
- un TabsAdapter che fornirà dinamicamente i Fragment che implementano ognuna un sezione specifica della UI.

5.2.1 Logica di avvio degli agenti

La classe si appoggia a JadeWifiDirectActivity per eseguire uno dei suoi compiti fondamentali: il bootstrap degli agenti che andranno a formare la piattaforma Jade di HoverInfo. Per la logica con cui sono stati implementati gli agenti 5.3, è stato necessario adottare un ordine di avvio degli stessi ben definito:

1. Alla notifica di avvenuto avvio del container, nei metodi doOnContainerSuccessfullyStarted() e doOnMainContainerSuccessfullyStarted(), viene richiamato il metodo startVeryFirstAgents(); come il nome lascia intendere, questo avvia i primi agenti necessari: PieceHoverInformationCallbacksProviderAgent 5.6 e ContainerSubscriberAgent 5.5.

2. Nel successivo metodo doOnAgentSuccessfullyStarted(AgentInfo agentInfo) si possono distinguere due casi:
   - se l'agente appena creato è il ContainerSubscriberAgent, allora si avvia il LocalizatorAgent 5.4;
   - se l'agente appena creato è il LocalizatorAgent, allora ha inizio il rilevamento della posizione tramite il LocalizatorService

5.2.2 Creazione di un 'piece' di HoverInfo: UUID

Discorso a parte merita la creazione di un nuovo 'piece' di Hovering Information. L'utente ha a disposizione una voce di menù apposita la quale, una volta selezionata, manda in esecuzione il metodo createHoverInfoAgent(String payload), che come argomento accetta una String che al momento rappresenta il nostro payload. Il metodo esegue le seguenti operazioni:

1. Ci si sposta in automatico esattamente sull'unica anchor location che abbiamo definito.

2. Si genera un UUID da assegnare come nome all'agente, si veda 5.6.

3. Si crea e si avvia un PieceHoveringInformationAgent specifico in base alla scelta effettuta dall'utente: BROADCAST_ALGORITHM o BROADCAST_PSEUDO_DIR_ALGORITHM 5.7.

UUID L'identificativo univoco universale (universally unique identifier o UUID) è un identificativo standard usato nelle infrastrutture software, standardizzato dalla Open Software Foundation (OSF) come parte di un ambiente distribuito di computazione [18]. L'UUID è composto da 16-byte (128-bit). Nella sua forma canonica, l'UUID è rappresentato da 32 caratteri esadecimali, visualizzati in cinque gruppi separati da trattini, nella forma 8-4-4-4-12 per un totale di 36 caratteri (32 esadecimali e quattro trattini). Ad esempio:
550e8400-e29b-41d4-a716-446655440000

5.2.3 Diagramma delle classi

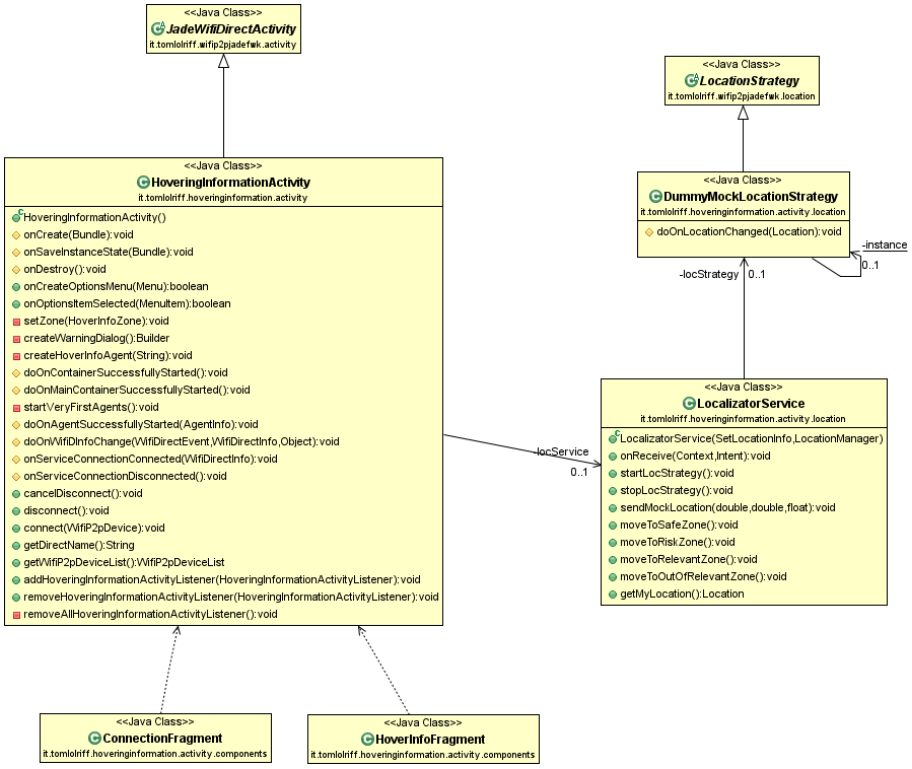

Figura 42: *HoveringInformationActivity: diagramma delle classi*

5.3 Agenti

Il problema dell'hovering information risulta molto complesso da implementare, ma il paradigma di programmazione multiagente ha delle caratteristiche che facilitano lo studio di una possibile soluzione. Tra queste caratteristiche, la 'mobility', ossia la possibilità per un agente di muoversi o clonarsi verso altri container, risulta essere di fondamentale importanza.
Per cercare di semplificare un po' il problema, è stato introdotto il vincolo di unica 'anchor location' nella app per ogni piece di hovering information che viene creato; questo vìola in parte quello che è stato definito da 1, secondo cui la anchor location è legata ad ogni singolo 'piece' di hovering information. Ciò comporterebbe la possibilità per ognuno di questi, di specificare una propria anchor location.
In dettaglio, la soluzione proposta, comporta la creazione dei seguenti agenti:

- **LocalizatorAgent:** il compito di questo agente è interrogare 'ciclicamente' il device in merito alla sua posizione corrente e comunicare al ContainerSubscriberAgent (del suo container) un eventuale cambio, dovuto quindi ad uno spostamento.
- **ContainerSubscriberAgent:** il suo compito è quello di mantenere memorizzate in locale informazioni sui container presenti nel sistema multiagente, indicando la loro posizione fisica in termini di 'zone' 1. Riceve inoltre le comunicazioni di cambio di zona dal LocalizatorAgent e dagli altri ContainerSubscriberAgent; nel primo caso, una volta aggiornate le informazioni sulla propria posizione fisica, comunica il cambio di posizione a tutti gli altri ContainerSubscriberAgent e tutti i PieceHoveringInformationAgent del suo stesso container.
- **PieceHoveringInformationAgent:** è una classe astratta che incapsula il payload da trasportare e fornisce gli strumenti per implementare la logica di spostamento/clonazione/cancellazione in base ad un algoritmo implementato dalle classi che lo estendono.
- **PieceHoverInformationCallbacksProviderAgent:** viene utilizzata per ottenere un 'canale di comunicazione' tra i PieceHoveringInformationAgent e il device su cui si trovano.

5.4 LocalizatorAgent

Come anticipato, il suo compito è interrogare il device in merito alla sua posizione fisica e comunicare un eventuale cambiamento al ContainerSub-

scriberAgent del suo stesso container.
Per fare questo è stato necessario approfondire la questione relativa alle chiamate da un agente JADE verso Android e viceversa, visto che le informazioni sulla posizione fisica del device è possibile reperirle solo effettuando richieste a cui deve rispondere un componente che può recuperare questi dati, rilevati dal sistema operativo.
Per recuperare le informazioni abbiamo utilizzato il componente LocalizatorService descritto in precedenza 4.6, mentre per permettere all'agente di inviare richieste al componente si è seguita questa logica:

- viene inviato come argomento dell'agente un'interfaccia di tipo RequestLocationInfo;
- nel LocalizatorAgent si è utilizzato un TickerBehaviour che ogni x secondi effettua una richiesta all'esterno tramite l'interfaccia;
- tramite il metodo registerO2AInterface si è registrata un'interfaccia implementata dallo stesso agente di tipo SetLocationInfo, in cui dall'esterno vengono inviati i valori delle nuove coordinate geografiche.

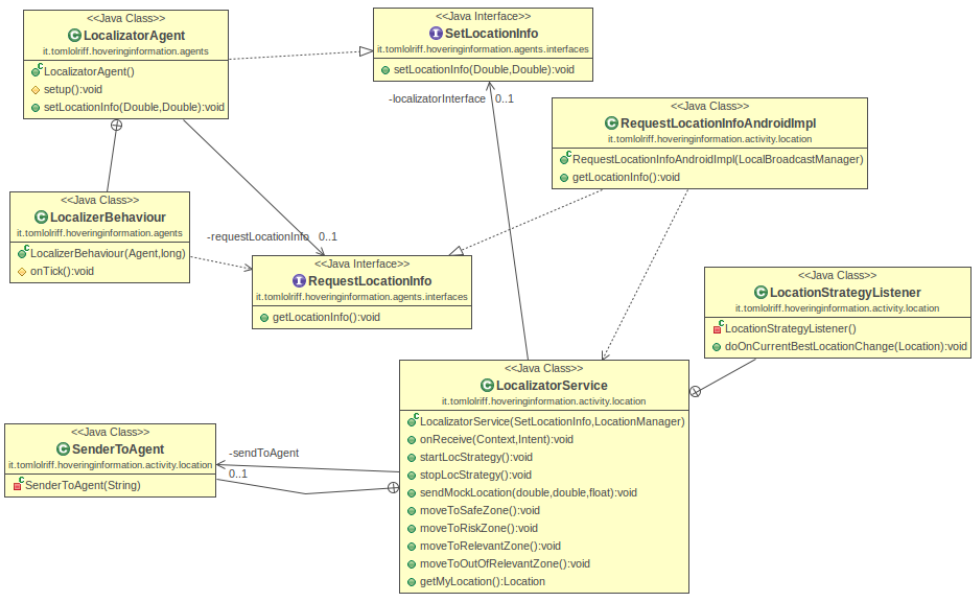

Figura 43: *UML: LocalizatorAgent*

Come si può vedere dal diagramma UML (fig.43), in realtà la situazione è un po' più complessa, a causa dei meccanismi di funzionamento di Android. In

particolare è vero che come argomento in fase di costruzione del LocalizatorAgent viene passata un'implementazione dell'interfaccia RequestLocationInfo, ma l'istanza (RequestLocationInfoAndroidImpl) non richiede direttamente ad Android la posizione geografica piuttosto - tramite il meccanismo dell'Intent - invia una action di tipo 'LocalizatorService.ACTION_REQUEST_LOCATION' a cui risponderà effettivamente un LocalizatorService 4.6 (si tenga presente che è un'estensione di un BroadcastReceiver a cui viene passato nel costruttore l'istanza di una classe che implementa l'interfaccia SetLocationInfo). Quando il LocalizatorService riceve l'intent, andrà a recuperare le coordinate geografiche che verranno comunicate all'agente tramite l'istanza passata al costruttore.
La comunicazione delle informazioni reperite al ContainerSubscriberAgent, consiste semplicemente nell'invio di un ACLMessage al ContainerSubscriberAgent, di cui viene passato l'AID come argomento; infatti la creazione di questi due agenti avviene contestualmente all'interno dell'Activity 5.2, quindi è decisamente sensato passare questa informazione in fase di creazione dell'agente.

HoverInfoZoneDetect: Questa classe si occupa del 'calcolo' della zona di appartenenza, in base alle coordinate passate e i parametri decisi in fase di creazione dell'agente, che in dettaglio sono:

- coordinate anchor location;
- dimensioni dei 'raggi' delle varie zone (SAFE, RISK, RELEVANT).

Avendo queste informazioni, tramite l'invocazione del suo metodo getFromCoordinate(), è possibile - passando delle coordinate geografiche - ottenere un Enum di tipo HoverInfoZone che indica la zona di appartenenza (oltre alle tipiche zone dell'hovering information contiene il valore UNKNOWN).

5.5 ContainerSubscriberAgent

Questo agente, come già anticipato, ha come compito fondamentale quello di fare da 'storage' delle informazioni fondamentali per un corretto funzionamento dell'hovering information, che si basa sulla posizione fisica corrente dei nodi ad ogni istante, da noi rappresentati dai device Android. La memorizzazione di queste informazioni si appoggia ad una struttura dati chiamata GeographicalDataStorage 5.5.1; per mantenere allineate queste informazioni, dovrà quindi implementare una rete di scambio di dati tra ContainerSubscriberAgent posti in device differenti 5.5.2.

Un'ultima attività è notificare cambi di posizione del device a tutti i 'piece' di hovering information presenti nel suo stesso device 5.5.3 e rispondere alle loro richieste relative al fornimento di un elenco di jade.core.Location per una data zona.

5.5.1 Ricezione cambio di posizione e aggiornamento dati

I ContainerSubscriberAgent necessitano quindi di una struttura dati che gli permetta di memorizzare le informazioni geografiche dei Container coinvolti. Questa struttura dati prende il nome di GeographicalDataStorage 5.5.1. La ricezione delle informazioni relative ai Container può arrivare in tre modi:

1. tramite il MainContainer, che in realtà non può comunicare informazioni geografiche, ma solamente quando un Container si 'unisce' alla MAS oppure se ne disconnette;

2. tramite il LocalizatorAgent, che gli comunica quindi un cambio di posizione del device in cui sono presenti;

3. tramite gli altri ContainerSubscriberAgent.

Il primo punto sfrutta un Behaviour di tipo AMSSubscriber, il cui compito è quindi ricevere dei feedback dal MainContainer quando scattano gli eventi IntrospectionVocabulary.ADDEDCONTAINER oppure IntrospectionVocabulary.REMOVEDCONTAINER.
Nel primo caso aggiungerà al suo GeographicalDataStorage un Container in zona UNKNOWN, nel secondo lo andrà a rimuovere. In questo modo avremo la garanzia di avere sempre all'interno di questa struttura dati, informazioni di tutti i Container.
Il secondo metodo invece, permette di aggiornare la zona in cui è correntemente posizionato il device. Nella sezione precedente 5.4 abbiamo già analizzato il messaggio che il LocalizatorAgent invia direttamente al ContainerSubscriberAgent ogni volta che cambia la posizione geografica del device; dal punto di vista del ContainerSubscriberAgent, non viene effettuato nient'altro che l'aggiornamento del proprio Container all'interno del GeographicalDataStorage. L'aspetto più interessante di questa operazione è ciò che avviene dopo l'aggiornamento del 'data storage': dopo aver allineato queste informazioni, tramite un SequentialBehaviour andrà ad inviare questo cambio di posizione a tutti i subscriber 5.5.2 e agli agenti di tipo PieceHoveringInformationAgent

presenti nel suo stesso Container 5.5.3.
Il terzo ed ultimo punto verrà approfondito in 5.5.2.

GeographicalDataStorage questa classe tiene memorizzate, all'interno del ContainerSubscriberAgent, le informazioni relative a tutti i Container presenti nel MAS. In termini strettamente tecnici consiste in un'implementazione dell'interfaccia Map<HoverInfoZone, List<GeographicalInfo>>nella quale la chiave è la zona 5.4, mentre il valore consiste in una lista di GeographicalInfo ossia una classe che contiene al suo interno i seguenti campi:

```
private HoverInfoZone zone;
private Location jade_location;
private Double x;
private Double y;
```

Contiene quindi la zona corrente, un'istanza di tipo jade.core.Location che permetterà poi agli agenti di clonarsi e spostarsi facilmente, oltre alle consuete coordinate geografiche.
Come si può notare, GeographicalDataStorage è una struttura dati fortemente sbilanciata dal punto di vista del reperimento dei Container presenti in una data zona (operazione effettuabile in O(1)), mentre l'aggiornamento di una zona necessita di una ricerca in tutte quelle possibili e in ogni elemento della List<GeographicalInfo >nel peggiore dei casi.

5.5.2 Scambio di informazioni tra ContainerSubscriberAgent

Per rendere 'reperibili' tutti i ContainerSubscriberAgent all'interno del MAS, è necessario utilizzare un Directory Facilitator. Ogni ContainerSubscriberAgent, nel proprio setup(), si registra al DF con Servizio 'hoverinfo-subscribe-locations'. In questo modo, ogni qualvolta ci sia la necessità di ottenere gli AID di questi tipi di agenti, basterà una semplice chiamata al componente DF di Jade.
Detto questo, risulterà abbastanza immediato il meccaniscmo che permette lo scambio di informazioni tra questi agenti, per mantenere i propri GeographicalDataStorage allineati il più possibile con l'attuale disposizione fisica dei device coinvolti.
L'immagine fig. 44 cerca di facilitare la spiegazione che permette di allineare i GeographicalDataStorage dei ContainerSubscriberAgent coinvolti.

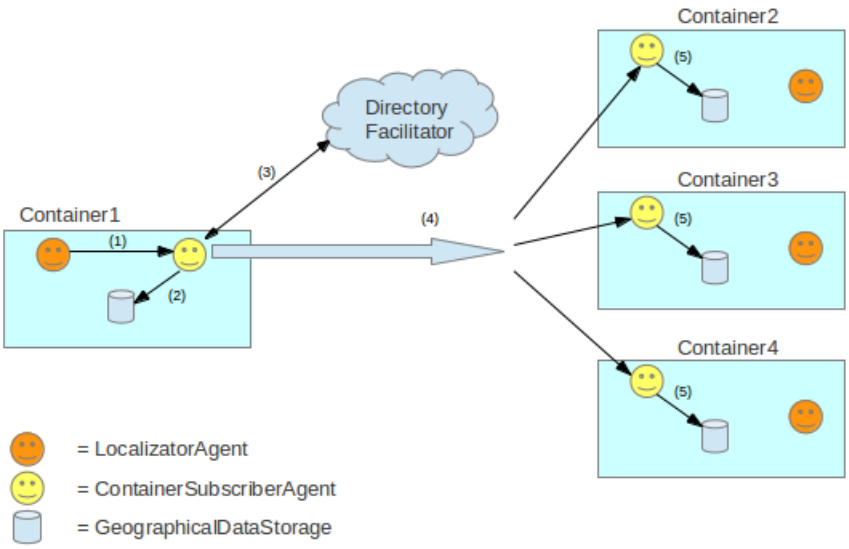

Figura 44: *Comunicazione tra ContainerSubscriberAgent*

Nel (1) il LocalizatorAgent riceve un cambio di posizione dall'esterno e invia un ACLMessage diretto al proprio ContainerSubscriberAgent con il cambio di posizione. Il ricevente aggiornerà quindi il proprio GeographicalDataStorage (2) e, dopo aver richiesto l'elenco dei ContainerSubscriberAgent al DF (3), invierà a tutti l'informazione (4). Infine, ogni ContainerSubscriberAgent aggiornerà il proprio GeographicalDataStorage (5).

5.5.3 Notifica cambio di posizione ai 'piece'

Le ultime operazioni effettuate dal ContainerSubscriberAgent, sono quelle relative alla comunicazione con i PieceHoveringInformationAgent del proprio Container, per segnalare un cambio di posizione a tutti i PieceHoveringInformationAgent e l'invio di un elenco di jade.core.Location per una data zona, in modo da dargli la possibilità di effettuare le opportune operazioni di 'replication', in base all'algoritmo che si vuole sviluppare.
Per comunicare il cambio di posizione, è stato necessario reperire tutti gli AID relativi ai PieceHoveringInformationAgent presenti nello stesso Container; si è scelto di utilizzare un Behaviour (poi riutilizzato anche in 5.6) che abbiamo

creato ad hoc: GetAgentsOnContainerAndReplyBehaviour 5.5.3, il cui compito è richiedere un elenco di tutti gli agenti presenti nel proprio container e l'invio di un messaggio contenente un'istanza di GeographicalInfo 5.5.1 contenente i dati aggiornati del proprio Container. Naturalmente questa soluzione non invia il messaggio solo ai PieceHoveringInformationAgent, ma con una breve valutazione si può stimare che questi, all'interno di un Container, siano: tutti gli agenti presenti ad eccezione degli 'agenti di servizio' di Jade, i due appena analizzati (ContainerSubscriberAgent e LocalizatorAgent) e un ultimo approfondito in 5.6.2; tali agenti non reperiranno il messaggio in quanto non sono in attesa di messaggi corrispondenti a quel MessageTemplate.
L'ultima operazione effettuata dal Container è quella di ricevere richieste di Location per una determinata zona da parte dei PieceHoveringInformationAgent; per questo è bastato mettere il ContainerSubscriberAgent in attesa di un messaggio opportunamente composto da quel tipo di richiesta, a cui viene risposto elencando una List<Location>.

GetAgentsOnContainerAndReplyBehaviour Questo Behaviour è stato realizzato per rispondere ad una richiesta frequente degli agenti che abbiamo creato: «reperisci tutti gli agenti correntemente presenti nel Container dello stesso Agente ed invia un messaggio». Il suo funzionamento si appoggia sull'invio di un messaggio di tipo QueryAgentsOnLocation (appartenente alla ontology JADE-Agent-Management), che permette di richiedere un elenco degli agenti del Container da cui viene effettuata la chiamata. Quando arriva l'elenco aggiunge gli AID presenti nella risposta dell'ACLMessage passato al costruttore ed effettua l'invio.

5.6 PieceHoveringInformationAgent

Questa classe incapsula la logica di un generico 'piece' di Hovering Information, senza entrare nel merito del comportamento da effettuare quando cambia la zona in cui si trova correntemente, operazioni delegate alle sottoclassi 5.7.

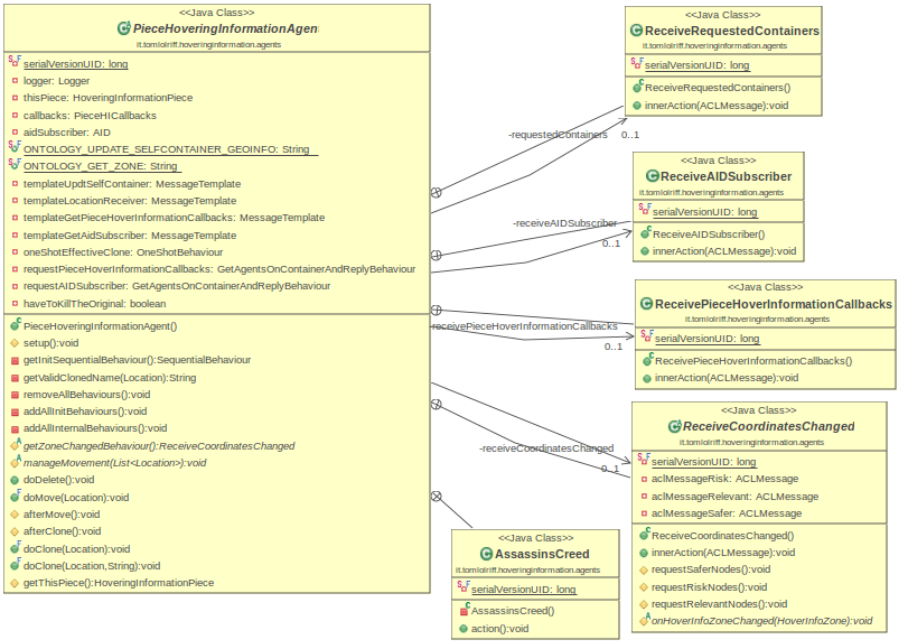

Figura 45: *UML: diagramma delle classi di PieceHoveringInformationAgent*

Dalla fig 45 è possibile visionare la struttura di questa classe, in particolare si può notare che possiede un campo contenente delle informazioni relative al 'piece', all'interno di una istanza di HoveringInformationPiece, chiamata thisPiece. All'interno di questa classe sono presenti i seguenti campi:

```
private UUID id;
private Object payLoad;
private GeographicalInfo geoInfo;
private GeographicalInfo geoInfoFrom;
```

Il campo id, di tipo UUID, è stato precedentemente descritto 5.2.2 ed è un identificativo del 'piece' generato in fase di creazione, che rimarrà costante indipendentemente dall'AID dell'agente (quindi anche dalla posizione fisica e del fatto che sia l''originale' o un clone).
Il payload è il dato contenuto; geoInfo incapsula le informazioni relative alla posizione geografica attuale del 'piece' 5.5.1, mentre geoInfoFrom incapsula le medesime informazioni ma relative alla precedente posizione. Questo campo permette di effettuare degli algoritmi di replication, che si basano sulla precedente posizione del 'piece', ad esempio per indicare una direzione indirettamente 5.7.2.

Nella costruzione di un agente di questo tipo, è necessario passare come argomenti le seguenti informazioni:

- l'UUID del 'piece' creato;
- il payLoad;
- le attuali coordinate geografiche.

Nel momento in cui viene creato l'agente, ci si ritrova a dover affrontare la problematica del *naming*; in dettaglio è opportuno studiare come poter chiamare l'agente in fase di creazione e soprattutto quando verrà effettuata una operazione di clone dell'agente, per farla andare a buon fine senza dover andare a scorrere tutti gli agenti già presenti nel Container di destinazione per capire se è possibile o meno 'accettare' l'agente.
La soluzione trovata fa leva su questi concetti:

- il nome dato all'agente in fase di creazione è ininfluente;
- il nome dell'agente dev'essere composto sempre da UUID + nome della Location in cui si trova.

Questi punti fermi permetterebbero di avere 'gratis' il controllo di presenza di un 'piece' in un Container di destinazione in fase di clone/move, infatti Jade, in caso di clash di nomi, non fa altro che non far terminare a buon fine l'operazione, che è esattamente ciò che ci serve.
Per implementare però questa logica è stato necessario forzare il comportamento dell'agente in fase di creazione tramite un SequentialBehaviour, in questo modo:

- l'agente appena creato effettuerà una operazione di doClone() nel suo stesso Container, componendo il nome dell'agente con i criteri stabiliti, quindi concatenanto l'UUID passato come argomento e il nome della location corrente;
- dopo aver effettuato l'operazione di doClone() l'agente appena creato richiamerà il metodo doDelete(), tramite il behaviour AssassinsCreed.

Questo ci assicura di avere sempre e solo agenti di tipo PieceHoveringInformationAgent che seguono i criteri di naming che abbiamo stabilito.
Dopo aver descritto come è stato affrontato il problema del naming, si descriveranno tutti i Behaviour che caratterizzano questo agente; le operazioni che deve effettuare sono:

- ricevere informazioni in merito al cambio di posizione del device in cui si trova;
- in base alla nuova zona decidere che operazioni effettuare;
- notificare al device le operazioni che sta effettuando, in modo da poter permettere alla applicazione (qualunque essa sia, non necessariamente

una app di Android), di effettuare operazioni di risposta ad eventi che accadono.

5.6.1 Azioni in risposta al cambio di posizione

Il PieceHoveringInformationAgent, per operare correttamente, deve innanzitutto ricevere dal ContainerSubscriberAgent le nuove coordinate (e quindi eventualmente la nuova zona in cui si trova); per fare questo si è semplicemente definito correttamente un MessageTemplate, in base ai parametri settati nell'ACLMessage inviato. Una volta ricevuto il 'cambio di posizione', verrà aggiornato il campo geoInfoFrom di thisPiece e, nel caso in cui la zona sia cambiata, viene invocato il metodo abstract

```
protected abstract void onHoverInfoZoneChanged(HoverInfoZone newZone);
```

Qui si deduce che, una volta notificato il cambio di zona, sarà la classe che estenderà PieceHoveringInformationAgent a decidere come deve comportarsi l'agente; nella sezione 5.7 vengono illustrati due esempi realizzati.
Le classi che estendono PieceHoveringInformationAgent avranno comunque bisogno di alcune funzionalità da combinare a piacere per implementare i diversi algoritmi, tra questi:

- l'opportunità di notificare al device in cui si trova alcune operazioni (verranno approfondite in 5.6.2);
- richiedere al ContainerSubscriberAgent del nodo in cui si trova un elenco di Location localizzati in determinate zone.

Per quanto riguarda la notifica al device, che verrà approfondita in 5.6.2, possiamo anticipare brevemente che è stata creata un'interfaccia, PieceHICallbacks, con cui l'agente può notificare all'esterno determinati eventi.
L'AID del ContainerSubscriberAgent invece, è necessario per realizzare il secondo punto.
Queste due informazioni non possono essere passate come argomenti in fase di costruzione dell'agente, in quanto questo permetterebbe di effettuare correttamente tali operazioni solo ed esclusivamente da agenti che non operano 'mobility', eventualità ovviamente da escludere nel caso di questi tipi di agenti. Per soddisfare queste necessità, si è ricorso ancora una volta al GetAgentsOnContainerAndReplyBehaviour5.5.3, sia per ottenere l'AID del ContainerSubscriberAgent, sia per ottenere un riferimento ad un'interfaccia di tipo PieceHICallbacks. Nel primo caso la spiegazione è logica e semplice: l'AID del ContainerSubscriberAgent si ottiene semplicemente inviando una richiesta a tutti gli agenti della stessa Location dell'agente, a cui risponderà

il ContainerSubscriberAgent inviandogli il suo AID. Da quel momento potrà inviare le richieste di Location per le zone di cui necessita.
La seconda parte invece necessita di un approfondimento che avverrà in 5.6.2.

5.6.2 Notifiche al device

Abbiamo già detto che ogni PieceHoveringInformationAgent necessita di comunicare col device alcuni suoi cambiamenti di stato, rappresentati dall'interfaccia PieceHICallbacks:

```
public interface PieceHICallbacks extends Serializable {
        void onPieceCreated(HoveringInformationPiece piece);
        void onPieceCloned(HoveringInformationPiece piece);
        void onPieceMoved(HoveringInformationPiece piece);
        void onPieceDeleted(HoveringInformationPiece piece);
}
```

La difficoltà maggiore, per rendere possibile la comunicazione tra il PieceHoveringInformationAgent e il device tramite questa interfaccia, è data dall'impossibilità di risolvere questo problema come si è fatto precedentemente con il LocalizatorAgent, quindi passando l'istanza di una classe che implementa l'interfaccia come argomento, perchè questo tipo di agente, dopo aver effettuato operazioni di mobility, deve necessariamente cambiare questo riferimento. La soluzione proposta, introduce un altro agente: PieceHoverInformationCallbacksProviderAgent. Anch'esso, come il ContainerSubscriberAgent e il LocalizatorAgent, viene creato ed avviato quando la connessione di un device tramite wifi direct va a buon fine e, come dice il nome, il suo unico compito è di poter ricevere richieste di interfacce di tipo PieceHICallbacks ed inviare l'istanza di una classe che la implementa.

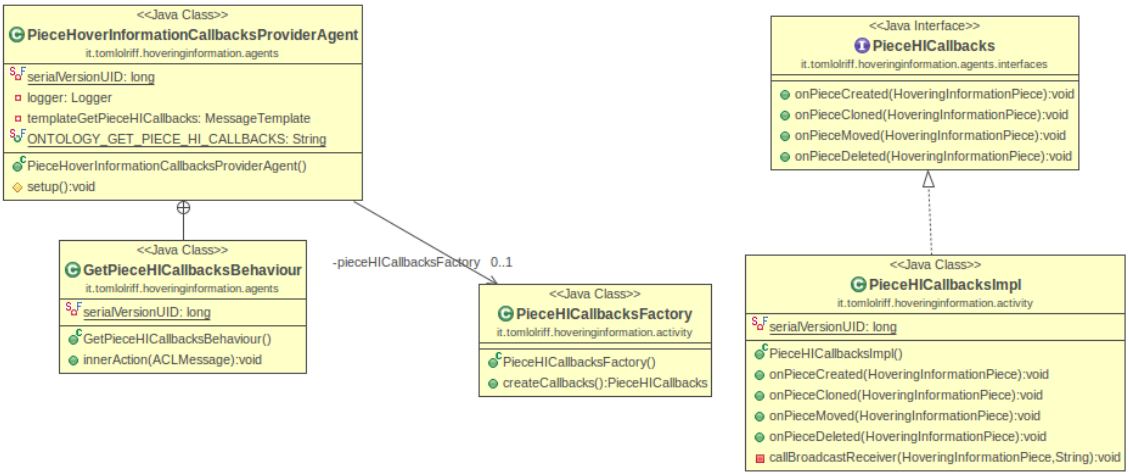

Figura 46: *UML: diagramma delle classi di PieceHoverInformationCallbacksProviderAgent*

Come si può vedere da 46, l'agente introdotto possiede un'istanza della classe PieceHICallbacksFactory, il cui compito è fornire proprio l'istanza di una classe che implementa l'interfaccia PieceHICallbacks; nel nostro caso è unica ed è PieceHICallbacksImpl.
La classe PieceHICallbacksImpl si appoggia a dei BroadcastReceiver per 'girare' le notifiche al device:

```
public class PieceHICallbacksImpl implements PieceHICallbacks {
        ....

        @Override
        public void onPieceCreated(HoveringInformationPiece piece) {
                callBroadcastReceiver(piece,
                    PieceHoveringInformationReceiver.ACTION_CREATED);
        }

        @Override
        public void onPieceCloned(HoveringInformationPiece piece) {
                callBroadcastReceiver(piece,
                    PieceHoveringInformationReceiver.ACTION_CLONED);
        }

        ....

        private void callBroadcastReceiver(HoveringInformationPiece piece,
            String action) {
                Intent intent = new Intent();
                intent.setAction(action);
                intent.putExtra(PieceHoveringInformationReceiver.
                    EXTRA_PARAM_NAME, piece);
```

```
                LocalBroadcastManagerExecutor.getInstance().sendBroadcast(
                    intent);
        }

}
```

Ora diventa chiaro il metodo utilizzato per ottenere dal PieceHoveringInformationAgent un 'canale di comunicazione' verso i device Android, infatti l'unico tassello mancante è la richiesta di un'interfaccia tramite GetAgentsOnContainerAndReplyBehaviour, esattamente come viene fatto per recuperare l'AID del ContainerSubscriberAgent.

5.7 Estensioni di PieceHoveringInformationAgent

Nel progetto sviluppato si sono proposti due tipi di algoritmi di 'replication', che chiameremo:

- Broadcast Replication;
- Broadcast Replication Pseudo Direction.

5.7.1 Broadcast Replication

Questo tipo di algoritmo di replication agisce seguendo questa logica:

- se l'agente è in zona SAFE non fa niente;
- se l'agente è in zona RISK si replica verso tutti i Container presenti in quel momento nella RISK zone;
- se l'agente è in zona RELEVANT si sposta verso il primo Container che trova presente nella RELEVANT zone;
- se l'agente è in zona OUT RELEVANT 'si uccide'.

In termini programmatici, la classe che estende PieceHoveringInformationAgent implementerà i metodi astratti in questo modo:

```
@Override
protected void onHoverInfoZoneChanged(HoverInfoZone newZone) {
        switch (newZone) {
        case RISK:
                // si vuole clonare, quindi richiede quelli in Risk ZONE
                requestRiskNodes();
                break;
        case RELEVANT:
                // si vuole muovere, quindi richiede quelli in Relevant ZONE
                requestRelevantNodes();
                break;
        case OUT_RELEVANT:
                // harakiri!
                doDelete();
                break;
```

```
        default:
                // non fa nulla
                break;
        }
}
```

quando la nuova zona è SAFE non farà nulla, se è RISK richiederà al ContainerSubscriberAgent le Location presenti nella RISK zone, se è nella RELEVANT analogamente richiederà le Location della RELEVANT zone, mentre se si trova oltre la RELEVANT effettuerà la delete di se stesso.
Ognuna delle invocazioni dei metodi request*Nodes() scatenerà, all'arrivo della risposta da parte del ContainerSubscriberAgent, l'invocazione del secondo metodo che dev'essere implementato:

```
@Override
protected void manageMovement(List<Location> locationsList) {
        for (Location location : locationsList) {
                if(getThisPiece().getGeoInfo().getZone() == HoverInfoZone.
                    RISK) {
                        doClone(location);
                } else if(getThisPiece().getGeoInfo().getZone() ==
                    HoverInfoZone.RELEVANT) {
                        doMove(location);
                }
        }
}
```

Una volta ricevute le Location, nel caso in cui si trovi nella RISK zone si clonerà in ognuno di queste, nel caso in cui si trovi invece nella RELEVANT zone, effettuerà uno spostamente verso una Location random. Da notare infatti che anche se è presente un ciclo, il doMove() verrà effettuato solo una volta, in quanto la classe padre PieceHoveringInformationAgent implementa a sua volta i metodi afterMove()/afterClone() e in entrambi i casi, la prima operazione che effettuerà sarà eliminare i behaviour, per non permettere ad agenti che provengono da operazioni di mobility, di inviare e ricevere messaggi da agenti posti nella Location di provenienza.

5.7.2 Broadcast Replication Pseudo Direction

Lo scopo del secondo algoritmo di replication è simulare la gestione di una possibile informazione relativa alla direzione presa dai nodi. In realtà, esempi di algoritmi che prendono in considerazione direzioni come l' 'Attractor Point', tengono in considerazione quelle dei nodi circostanti per valutare in quali di essi replicarsi/muoversi. Nel nostro caso invece, l'idea è di tenere conto del proprio movimento e non di quello dei nodi di destinazione, implementando una logica di questo tipo:

- se l'agente è in zona SAFE non fa niente;
- se l'agente è in zona RISK, nel caso in cui provenga dalla SAFE si clona in tutti i Container presenti in quel momento nella RISK zone e nella SAFE zone, nel caso provenga dalla RELEVANT si clona solo nei Container presenti nella RISK zone;
- se l'agente è in zona RELEVANT e proviene dalla RISK, si sposta verso nel primo Container che trova presente nella RELEVANT o RISK zone, se proviene dall'OUT RELEVANT non fa nulla, perchè non può avere piece;
- se l'agente è in zona OUT RELEVANT 'si uccide'.

Anche in questo caso mostriamo a livello programmatico, come sono stati implementati i metodi abstract della PieceHoveringInformationAgent:

```
@Override
protected void onHoverInfoZoneChanged(HoverInfoZone newZone) {
        switch (newZone) {
        case RISK:
                if(getThisPiece().getGeoInfoFrom().getZone().equals(
                    HoverInfoZone.SAFER)){ // se provenivo dalla safe
                        // mi clono nei safe e nei risk
                        requestSaferNodes();
                        requestRiskNodes();
                }
                else if(getThisPiece().getGeoInfoFrom().getZone().equals(
                    HoverInfoZone.RELEVANT)) { // provenivo dalla relevant
                        // mi clono nei risk
                        requestRiskNodes();
                }
                break;
        case RELEVANT:
                if(getThisPiece().getGeoInfoFrom().getZone().equals(
                    HoverInfoZone.RISK)) { // se provenivo dalla risk
                        // mi sposto nei risk e nei relevant
                        requestRiskNodes();
                        requestRelevantNodes();
                }
                // se provenivo dalla out non ho piece; non devo fare niente
                break;
        case OUT_RELEVANT:
                // harakiri!
                doDelete();
                break;
        default:
                // non fa nulla
                break;
        }
}
```

Una volta ricevute le Location effettuerà le seguenti operazioni:

```
@Override
protected void manageMovement(List<Location> locationsList) {
```

```
        for (Location location : locationsList) {
                if(getThisPiece().getGeoInfo().getZone() == HoverInfoZone.
                    RISK) {
                        //come nome del clone metto: ID||random
                        doClone(location);
                } else if(getThisPiece().getGeoInfo().getZone() ==
                    HoverInfoZone.RELEVANT) {
                        doMove(location);
                }
        }
}
```

6 Possibili sviluppi futuri

Il progetto che abbiamo realizzato si presta ovviamente ad una lunga serie di miglioramenti ed ottimizzazioni delle funzionalità già presenti, così come all'aggiunta di nuove di queste. Questo sia per quanto riguarda l'app di HoverInfo, sia per quanto riguarda il framework di supporto Wi-Fi Direct + Jade. Partendo proprio da quest'ultimo, uno degli aspetti fondamentali che è stato tralasciato, è la possibilità di restituire una lista di peers direct, filtrandoli tramite il Service Discovery 3.3, in base al servizio da loro offerto. Al momento si è supposto che i peers rilevati abbiano tutti in esecuzione l'app di HoverInfo; è chiaro che in uno scenario reale gli utilizzatori del framework necessitino di un meccanismo più evoluto. Un'altra evoluzione significativa del framework, sarebbe sicuramente la possibilità di poter gestire in maniera trasparente, una topologia della piattaforma Jade a due livelli, appoggiandosi all'inter-platform communucation. Si potrebbero così avere delle sottoreti ognuna formata da un main container ed n container; alzandosi di livello invece, ci sarebbe una rete globale nella quale tutti i main container effettuano comunicazioni inter-platform.
Passando all'app di HoverInfo è possibile trovare anche più spunti per interessanti evoluzioni future. Nell'elenco seguente proponiamo quelle che al momento ci sono sembrate più degne di nota:

- Gestione di una piattaforma multi-anchor: ogni piece dovrebbe avere la propria anchor location di riferimento, definita in fase di creazione del piece stesso rilevando la posizione attuale del device. Un'idea in tal proposito potrebbe essere quella di avere ContainerSubscriberAgent multipli, ognuno con la gestione dei container relativi ad un'unica anchor.

- Possibilità di consistenza dei piece nei device: i piece al momento non hanno nessun tipo di consistenza e vengono persi nel caso in cui l'app venga terminata. Sarebbe interessante avere per i domini applicativi che lo richiedano, la possibilità di salvare i piece in un database e poter così recuperare le informazioni in momenti successivi.
- Implementazione di algoritmi di replication più consoni ad una situazione reale: effettuare considerazioni su provider GPS e Network, considerando inoltre le componenti direzione e velocità.
- Possibilità di creare dei piece con payload generici di qualsiasi tipo: testo, immagini, suoni, video etc...

Riferimenti bibliografici

[1] Wi-Fi Alliance®. *Wi-Fi Direct Industry White Paper*. URL: `http://www.wi-fi.org/register.php?file=wp_Wi-Fi_Direct_20101025_Industry.pdf`.

[2] *AllJoyn™*. URL: `https://www.alljoyn.org/about`.

[3] *Android Guts: Intro to Loopers and Handlers*. URL: `http://mindtherobot.com/blog/159/android-guts-intro-to-loopers-and-handlers/`.

[4] Fabio Bellifemine, Giovanni Caire e Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. 2004.

[5] Giovanni Caire. *LEAP USER GUIDE*. Nov. 2011. URL: `http://jade.tilab.com/doc/tutorials/LEAPUserGuide.pdf`.

[6] Giovanni Caire et al. *Jade programming for Android*. Giu. 2012. URL: `http://jade.tilab.com/doc/tutorials/JadeAndroid-Programming-Tutorial.pdf`.

[7] Google. *Activity Android*. URL: `http://developer.android.com/guide/components/activities.html`.

[8] Google. *Activity Android - Javadoc*. URL: `http://developer.android.com/reference/android/app/Activity.html`.

[9] Google. *Location in Android*. URL: `http://developer.android.com/guide/topics/location/strategies.html`.

[10] Google. *Managing Projects from Eclipse with ADT*. URL: `http://developer.android.com/tools/projects/projects-eclipse.html`.

[11] Google. *Managing Projects from the Command Line*. URL: http://developer.android.com/tools/projects/projects-cmdline.html.

[12] Google. *Wi-Fi Direct*. URL: http://developer.android.com/guide/topics/connectivity/wifip2p.html.

[13] *Realtek Wi-Fi Direct Programming Guide*. URL: http://dishingtech.blogspot.it/2012/01/realtek-wi-fi-direct-programming-guide.html.

[14] Luca Ricchi et al. *JADE LEAP Android AndroidHelp.getLocalIPAddress() problem*. URL: http://avalon.tilab.com/pipermail/jade-develop/2013q1/018914.html.

[15] Andrea Tomasello et al. *JADE LEAP Android getO2AInterface() with Stand-Alone version*. URL: http://avalon.tilab.com/pipermail/jade-develop/2013q1/018957.html.

[16] M. Ughetti, T. Trucco e D. Gotta. "Development of Agent-Based, Peer-to-Peer Mobile Applications on ANDROID with JADE". In: *Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM '08. The Second International Conference on* (2008).

[17] Alfredo A. Villalba Castro, Giovanna Di Marzo Serugendo e Dimitri Konstantas. "Hovering Information - Infrastructure-Free Self-Organising Location-Aware Information Dissemination Service". In: *Sensor Networks, Ubiquitous and Trustworthy Computing, 2008. SUTC '08. IEEE International Conference on* (2008). URL: http://asg.unige.ch/publications/TR08/ASG2008-9.pdf.

[18] Wikipedia. *UUID*. URL: http://it.wikipedia.org/wiki/UUID.