ALMA MATER STUDIORUM · UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Smart Buses: Agent Aided Management (SBAAM)

Progetto di Sistemi Distribuiti

Realizzato da:

MICHELE AMATI
MATTEO CORFIATI
FEDERICO FUCCI
FABRIZIO MASINI

Anno accademico 2012/2013

Indice

Introduzione

L'elaborato mostra le principali fasi di analisi, progettazione ed implementazione di un sistema di supporto per la gestione delle corse degli autobus di una ipotetica città. La realizzazione del sistema distribuito ha come obiettivo principale il miglioramento dell'esperienza utente ed il coinvolgimento dello stesso nell'intero sistema.

Attualmente nelle città più importanti è possibile ritrovare sistemi simili, solitamente incentrati sul problema del ritardo, con opportuni display disposti ad ogni fermata che indicano il ritardo per ogni linea. In modo del tutto statico, è eventualmente riportato in formato elettronico il corrispettivo di un libretto degli orari cartaceo, solitamente su una piattaforma web.

Abbiamo seguito l'intero ciclo di creazione del sistema avendo come punto di riferimento una piattaforma che fornisse le utilità già presenti in quelle odierne e ne implementasse di nuove, come la gestione attiva dell'utente e l'eliminazione della componente statica sulle informazioni fornite, in modo da gestire meglio la *fault tolerance*.

In particolare, non ha più una importanza centrale il sito web perché le informazioni sugli orari e sulle tratte sono distribuite su ogni fermata: in questo modo, qualora si modificasse temporaneamente una linea, un utente in possesso di uno *smartphone* potrebbe essere sempre informato non basandosi sulla staticità del libretto degli orari. Inoltre, un'altra caratteristica importante è, come già detto, il coinvolgere gli utenti: questo viene fatto tramite l'invio semiautomatico di feedback (coordinate [x;y]) in modo da permettere alla direzione aziendale di effettuare analisi sulla localizzazione geografica delle fermate.

Capitolo 1
La realtà modellata

Fin dalle prime fasi di analisi, si sono delineati i soggetti, attivi e passivi, che entrano in gioco. Il sistema è quindi composto da diverse entità (molte delle quali, come mostrato nei successivi capitoli, assumeranno la forma di agenti) che interagiscono tra loro tramite scambio di messaggi.

La fermata è il soggetto che ha maggior valore informativo, quindi maggiore complessità: è preposto alla comunicazione con gli autobus, con gli utenti, con le altre fermate e con il deposito centrale. Quando un autobus "passa" davanti a una fermata, un sensore raccoglie l'informazione sull'orario corrente per valutare l'eventuale ritardo. Questi dati vengono quindi inviati a tutte le fermate successive della stessa linea, in modo tale che ognuna di queste possa essere sempre aggiornata sugli orari e sui ritardi.

L'autobus inizialmente era stato pensato come oggetto passivo, una sorta di evento che innescava tutto il processo di invio di messaggi sulle notifiche dei ritardi. In seguito il suo ruolo è stato rivisto attribuendogli maggior valore: è stato progettato un meccanismo secondo il quale gli utenti che salgono sull'autobus debbano attraversare un tornello (o un sensore) che, tramite l'interazione con lo *smartphone*, incrementa (o decrementa nel caso di discesa dal mezzo) un

contatore. Questo è utile sia a fini statistici, sia per la sicurezza, sia per il controllo di biglietti o abbonamenti.

Il deposito è la parte centralizzata del sistema distribuito: possiede l'intera conoscenza delle fermate, della loro posizione, degli orari delle diverse linee e dei tragitti degli autobus. Queste informazioni vengono utilizzate nella fase iniziale per effettuare il *bootstrap* del sistema e, periodicamente, per verificare (ed eventualmente ricreare) “entità morte”. L'altra caratteristica importante della centrale è quella di immagazzinare dati statistici che periodicamente vengono inviati dalle fermate. E' l'unico soggetto considerato sempre attivo e immune da errori (in una realtà ipotetica sarebbe installato *in-house* con le dovute precauzioni per preservarne la stabilità e la sicurezza).

Infine, l'entità più importante è l'utente che, tramite il suo dispositivo mobile, interagisce ed è parte integrante del sistema (non a caso ogni utente è un agente JADE). Oltre alle caratteristiche della normale utenza, che può richiedere informazioni su ritardi, linee e fermate, fornisce informazioni utili al miglioramento di tutta la piattaforma: tramite il meccanismo già citato del tornello indica la salita e la discesa dagli autobus e, per mezzo di *feedback*, invia alla fermata in cui è sceso le coordinate del punto in cui doveva effettivamente fermarsi, in modo tale che la direzione possa verificare se le fermate sono localizzate correttamente sul territorio.

La supposizione che esistano dei collegamenti cablati, wireless o satellitari tra le varie entità ci ha permesso di evitare di analizzare il problema fisico dello scambio di messaggi, preoccupandoci solo della progettazione di alto livello del sistema.

Capitolo 2

Scelta di JADE e JADE-Mobile

Introduzione alla progettazione e scelta della tecnologia

Il progetto vede coinvolte molteplici entità distribuite sia logicamente che fisicamente. Le entità comunicano le une con le altre formando nel complesso un sistema in grado di fornire le funzionalità che ci siamo preposti. Per quanto la cooperazione tra esse sia fondamentale, pensando al problema viene naturale concepirle come autonome, detentrici di un proprio flusso di controllo e questo ci ha spinto verso il paradigma di programmazione orientata agli agenti. Le principali tecnologie disponibili in questo ambito sono Jade e TuCSoN. Abbiamo deciso di utilizzare Jade perché fornisce una documentazione migliore anche se comunque carente per quanto riguarda molti dei suoi *add-on* tra i quali jade-leap. Quest'ultimo componente è il secondo motivo della nostra scelta, in quanto fornisce la possibilità di utilizzare jade su dispositivi mobili.

Integrazione con Jade Mobile

Per rendere disponibili le funzionalità del sistema anche su dispositivi mobili avevamo due possibilità:

- Utilizzare la versione mobile del *framework* jade, ovvero jade-leap.
- Utilizzare un approccio client server dove i client sarebbero stati gli smartphone degli utenti e il server un'entità ponte che comunicasse da un

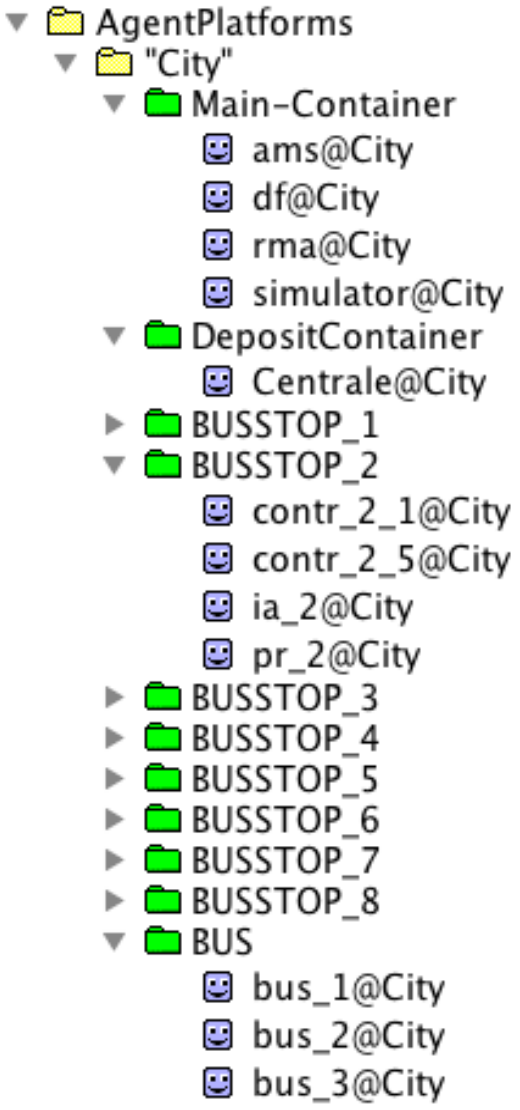
lato col sistema jade per acquisire informazioni e dall'altro esponesse web services restful interrogabili dai clients.

La strada intrapresa è stata la prima, nello specifico abbiamo utilizzato la versione jade-leap per dispositivi android. Il vantaggio principale è stato il mantenimento di un ambiente quasi totalmente uniforme che ha consentito di non complicare il sistema. Non ci sono però motivi particolari per escludere la seconda possibilità, molte applicazioni mobile con necessità di scambiare dati con un qualche sistema esterno usano l'approccio a web services restful perché molto adatto a devices con risorse limitate e soggette a connettività instabile.

Jade-leap in versione android mette a disposizione tutte le funzionalità di Jade, sconsiglia però l'uso di Container standard a favore dei così detti "Split-Container". Il vantaggio è un minor uso di risorse sia a livello di memoria/cpu sia a livello di rete, in quanto con questa modalità è possibile dividere (da qui, "split") il Container in due parti, Front-End e Back-End. La prima gira effettivamente sul device mentre la seconda su un server remoto. Le operazioni più *resource-inensive* quali ad esempio la comunicazione con il main container sono svolte sfruttando le risorse del server remoto lasciando quindi più scarico il device. Il principale svantaggio di questo approccio è che si perde il supporto per la mobilità degli agenti, funzionalità comunque non necessaria ai fini di questo progetto.

Capitolo 3

Progettazione dell'applicazione



Piattaforme, Container, Agenti

I "mattoni" messi a disposizione da Jade per la costruzione dei sistemi sono principalmente tre: Piattaforme, Containers ed Agenti. Non è scopo di questo documento entrare nei dettagli del loro funzionamento, di seguito verranno descritti solo in funzione del ruolo che ricoprono nel sistema. In linea di massima, le entità descritte nel capitolo due sono mappate in maniera molto diretta e intuitiva su Agenti, raggruppati poi in Container contenuti a loro volta in Piattaforme.

Container

Ad ogni città coperta dal nostro servizio corrisponde una piattaforma. Ogni piattaforma contiene i seguenti Container:

Main-Container

Contiene gli agenti Jade di base: AMS, DF, RMA e un agente Simulator che usiamo per simulare tutti gli eventi "fisici" sui quali si basa il sistema, come ad esempio il fatto che un autobus passi davanti a una certa fermata ad una tal ora. La parte di simulazione verrà trattata più approfonditamente nel capitolo 5.

DepositContainer

Rappresenta il deposito degli autobus, contiene l'agente che amministra la stazione centrale degli autobus.

Un container per ogni fermata

Codificato col formalismo: BUSSTOP_N dove N è il numero della fermata, contiene tutti gli agenti necessari per comunicare con la centrale, gli utenti e le altre fermate. In questo ordine quindi abbiamo: un agente ia_N (affari interni), un agente pr_N (pubbliche relazioni) e un agente per ogni linea servita da quella fermata contr_N_L (controllore) dove N è sempre il numero della fermata e L quello della linea servita.

BUS

Contiene gli agenti corrispondenti ai singoli autobus, nella forma: bus_N dove N è il numero dell'autobus.

Un container per ogni utente mobile

Questi ultimi container sono tutti Split-Container e vengono creati e distrutti dinamicamente all'apertura/chiusura dell'applicazione android. Ciascuno contiene un agente di tipo UserAgent il cui nome nel sistema è una stringa generata randomicamente ad ogni tentativo di connessione.

È stata scelta questa organizzazione sia perché mappa in maniera molto naturale il dominio del problema mantenendo quindi il progetto facilmente comprensibile, sia per motivi più pratici legati alla *fault tolerance*, alla necessità di isolare servizi di aree/città diverse e limitazioni della versione Jade-Android.

Agenti

Segue la descrizione del comportamento degli agenti del sistema.

CentralParkingAgent

Ha il compito di creare e, se necessario, ricreare tutti i componenti del sistema, containers e agenti, in base alle informazioni che legge da un data source. Le informazioni in questione quindi sono:

- Fermate identificate da nome e posizione.
- Controllori associati a una fermata e una linea assieme a informazioni di start-up quali i nomi dei controllori successivi e gli orari ai quali dovrebbe passare l'autobus.
- Bus attivi identificati dal loro numero.

Gli unici elementi del sistema che non sono di sua competenza sono i container e gli agenti degli utenti mobile. Oltre a questo, la centrale ha

anche il compito di ricevere i dati dei feedback elaborati e spediti dagli agenti addetti agli affari interni delle singole fermate; con questi dati chi eroga il servizio potrà tentare di ottimizzarlo. Come ultima funzionalità, non legata al dominio del problema, ma più al mantenimento delle funzionalità del sistema, la centrale controlla periodicamente la "*lifeness*" degli agenti da lei creati e li ripristina se ne trova di morti.

InternAffairsAgent

Presente obbligatoriamente in ogni container fermata, si occupa degli "affari interni" alla fermata stessa. Nello specifico, riceve dall'agente addetto alle pubbliche relazioni i dati di feedback mandati dall'utente e li processa usando le proprie coordinate e quelle presenti nel feedback per calcolare la distanza coperta dall'utente dopo che è sceso a quella fermata. I dati elaborati vengono aggregati man mano che arrivano e spediti alla centrale periodicamente.

PublicRelationAgent

Presente anch'esso in ogni fermata, si occupa delle pubbliche relazioni, facendo da collante tra utenti, controllori e affari interni. Con un abuso di terminologia si può dire che rappresenti l'interfaccia pubblica della fermata. Per quanto riguarda l'interazione con gli utenti mobile, si occupa di ridirigere le loro richieste ai controllori di competenza. Queste richieste sono: "elenco delle fermate della linea X, con il relativo ritardo." e "elenco delle linee che passano dalla fermata X, con il relativo ritardo.". Per quanto riguarda la prima, il PR inoltra la richiesta al controllore della propria fermata relativo alla linea specificata dall'utente. La seconda invece la inoltra a tutti i controllori, solo quelli che sanno di appartenere alla fermata del PR inoltrante risponderanno. In entrambi i casi, l'agente delle pubbliche relazioni rimbalza le risposte all'utente appena le ottiene.

Questo agente ha anche il compito di inoltrare i feedback degli utenti al CentralParkingAgent.

ControllerAgent

Anche se nulla vieta a livello pratico che esista una fermata senza controllori, è logicamente corretto pensare che ce ne sia almeno uno su ogni fermata. Questo agente detiene la conoscenza dello stato di una linea in relazione ad una precisa fermata. Conosce il tabellone degli orari della propria linea, il ritardo in un dato momento calcolato al passaggio dell'autobus tramite confronto tra l'orario teorico e quello effettivo, ed infine conosce un certo numero di controllori della propria linea in fermate successive, usando quest'ultima informazione per propagare lo stato di ritardo. Le sue interazioni con gli altri agenti dunque sono: ricevere l'informazione "sono passato" dai BusAgent di cui è competente, ricevere/inviare informazioni da/a controllori precedenti/successivi riguardo al ritardo della linea, rispondere alle richieste del PublicRelationAgent.

BusAgent

Ce n'è uno su ogni autobus in circolazione. Ha un certo numero di posti totali disponibili e tiene il conto di quanti passeggeri sta portando in un determinato momento. In linea teorica, un sensore dovrebbe generare un evento quando l'autobus si ferma ad una fermata di propria competenza, a seguito di questo evento l'agente dovrebbe comunicare il proprio ritardo alla fermata. Sempre in linea teorica, l'agente dovrebbe potersi accorgere in maniera semi-automatica della salita e discesa dei passeggeri, e gestire di conseguenza il suo stato interno. Per ovvi motivi pratici, gran parte del comportamento di questo agente è simulato, si vedano i capitoli sulla simulazione e l'implementazione per maggiori informazioni.

UserAgent

Ne nasce uno ogni volta che un'app mobile si connette al sistema, muore alla chiusura dell'applicazione. Il suo compito principale è quello di interrogare il sistema per ottenere informazioni relative alle corse degli autobus. Nello specifico le richieste disponibili sono due: "elenco delle fermate della linea X, con il relativo ritardo." e "elenco delle linee che passano dalla fermata X, con il relativo ritardo.". Entrambe sono rivolte all'agente addetto alle pubbliche relazioni della fermata desiderata, il quale risponderà nonappena entra in possesso dei dati richiesti. Ulteriore compito dello UserAgent è l'invio di un feedback contenente le coordinate geografiche del luogo di arrivo dell'utente, ovvero quel luogo al quale l'utente era intenzionato ad andare quando ha scelto quella determinata linea ed è sceso a quella determinata fermata. Anche questa comunicazione è diretta all'agente delle pubbliche relazioni, nello specifico quello delle fermata a cui l'utente è sceso. Ai fini della simulazione, l'utente è anche incaricato di segnalare in maniera attiva la salita e la discesa dall'autobus.

Gestione utenti e periferiche mobili

La modalità con cui gestiamo gli utenti mobile va forse un po' contro alcuni aspetti della buona programmazione ad agenti. Innanzi tutto, Split-Containers ed agenti di tipo UserAgent compaiono nel sistema con un nome non significativo, addirittura il nome degli agenti è una stringa generata a caso. Il motivo che ci ha spinto verso questa soluzione e che allo stesso tempo la rende viabile è che il ruolo dell'utente nel sistema è tale per cui sarà sempre lui a prendere l'iniziativa nelle comunicazioni. Il resto del sistema si limita a rispondere al mittente senza necessità di conoscere a priori il suo nome. Oltre a questo ci sono problemi pratici relativi al fatto che non possono esistere agenti

con stesso nome nella stessa piattaforma, il che ci riporta al problema del *naming*. Si sarebbe potuto architettare qualcosa, ma abbiamo deciso che in mancanza di una vera necessità non valeva la pena di complicare il sistema. Da notare che il problema non si sarebbe posto solo in presenza di più utenti fisicamente diversi allo stesso tempo. In caso di disconnessione irregolare dal sistema può capitare che l'agente continui a risultare vivo per un periodo di tempo anche lungo, questo porterebbe quindi alla possibilità che un futuro tentativo di connessione di quello stesso utente entri in conflitto con la versione precedente di se stesso. Generando una stringa ragionevolmente lunga evitiamo il problema ad ogni connessione. Gli agenti "zombie" rimasti nel sistema non costituiscono un problema rilevante visto che nessuno ha interesse a provare a contattarli se non sono loro a fare il primo passo.

Aspetto centralizzato

Per quanto il sistema sia distribuito, è comunque identificabile un punto chiave che potremmo definire "centrale". Si tratta appunto dell'entità "Centrale" che rappresenta la stazione principale degli autobus. La sua stessa natura rende difficile pensare di decentralizzarla. Va però notato che il suo ruolo è vitale per il sistema solo in fase di start-up, una volta che containers ed agenti sono attivi sulle fermate e sugli autobus, il sistema può continuare a funzionare senza problemi tranne per quanto riguarda l'accumulo di feedback.

Scalabilità

La scalabilità è certamente un aspetto importante nella progettazione di sistemi come questo che coinvolgano un numero variabile e potenzialmente molto alto di entità. Minori sono i punti di centralizzazione e la loro complessità, più sarà facile far crescere il sistema senza appesantirlo. Nel nostro caso abbiamo un solo principale punto di centralizzazione, ovvero quello rappresentato dalla centrale degli autobus. Il suo carico di lavoro è comunque modesto e non ha problemi a gestire un numero di entità verosimile per una città come Rimini.

Fault tolerance

Il sistema è progettato in modo da poter ammortizzare guasti di varie componenti senza perdere troppe funzionalità, l'unica cosa che non può permettersi di cadere è il Main-Container pena la caduta totale del sistema.

Segue un'analisi veloce delle conseguenze di alcuni dei possibili guasti:

- CentralParkingAgent, ammesso che il guasto si verifichi dopo lo start-up del sistema (questione di secondi), si perderà la funzionalità che registra i feedback e la funzione di check e ripristino di agenti morti.
- InternAffairsAgent, anche in questo caso la funzionalità persa sarà relativa all'elaborazione dei feedback degli utenti per la sola fermata alla quale l'agente appartiene.
- PublicRelationAgent, tutte le funzionalità di una singola fermata sono interrotte, eccezione fatta per la ricezione/propagazione di informazioni sullo stato delle linee servite.
- ControllerAgent, non ci sono grosse conseguenze finché la maggioranza dei ControllerAgent rimane funzionante. Più ne cadono, meno aggiornata sarà la conoscenza del sistema sullo stato della linea. Gli utenti non

riceveranno risposta se richiederanno informazioni riguardanti proprio quella linea a quella fermata.

- BusAgent, semplicemente il sistema smetterà di ricevere aggiornamenti sul suo stato, le altre linee non avranno problemi.
- UserAgent, l'unico ad avere problemi sarà l'utente stesso fino alla successiva connessione al sistema.

E' chiaro però che, pur non influenzando troppo il sistema nella sua totalità, ogni guasto va riparato in tempo ragionevole. A questo ci pensa l'agente CentralParkingAgent che controlla a intervalli di tempo regolari lo stato di vita di tutti gli agenti che ha generato in fase di start-up ed in caso qualcuno risulti morto, provvede a ricrearlo. Un problema pratico con l'utilizzo della tecnologia Jade ci impedisce di controllare lo stato di vita di un container, quindi se per qualche ragione tutto un container dovesse morire non saremmo in grado di accorgercene ed agire di conseguenza.

Capitolo 4

Implementazione

Scelte implementative

Conoscenza centrale

Abbiamo deciso di implementare un sistema distribuito con "conoscenza centralizzata", ovvero spostando tutte le informazioni e i dati relativi a autobus, percorsi e orari di linea verso la Centrale Operativa degli Autobus.

Questa scelta è stata attuata per due scopi principali:

- Il primo è relativo ad una questione "didattica", in quanto mantenendo tutta la conoscenza in un nodo centrale, tutte le richieste che vengono fatte da agenti periferici, vengono demandate ai vari agenti coinvolti nella richiesta fino ad arrivare infine alla centrale. Questa logica permette di gestire tutte le richieste sotto forma di scambio di messaggi, senza dare per scontato alcuna conoscenza intrinseca degli agenti periferici distribuiti.
- Il secondo è frutto di alcune considerazioni relative ad un aspetto "logico-implementativo". Ideando un sistema con conoscenza centralizzata, si cerca di semplificare gli agenti periferici (sia in termini di struttura hardware che software). Inoltre si riduce il carico di lavoro e la quantità di dati trasmessi allo "startup" del sistema, in quanto la **Centrale** fornisce solamente le informazioni

minime indispensabili per una corretta risoluzione delle richieste che vengono loro inoltrate (si vedano gli agenti **Controllori di Linea**, i quali conoscono i Controllori successivi sulla stessa linea, per poter comunicare con loro).

Agenti distribuiti Pro-Attivi

Gli agenti, ricordiamo, sono pensati come elementi Pro-Attivi, ciononostante nella nostra soluzione possiedono una componente di comportamento fondamentalmente "Reattiva", che permette loro di agire in risposta alla ricezione di messaggi di richiesta (si veda il paragrafo successivo). Alcuni di loro continuano a presentare una componente "Attiva", che permette all'agente di agire secondo proprie logiche temporali e regole di comunicazione con gli altri agenti. Per esempio, l'agente **Affari Interni** decide il momento in cui inviare le informazioni di feedback ricevute dagli utenti, senza dover reagire ad un particolare comportamento esterno. Allo stesso modo, sono stati gestiti comportamenti di alcuni agenti, quali gli **Autobus**, per la modellazione di fenomeni reali "simulati". In altri termini, sono stati inseriti dei "temporizzatori" interni agli agenti, per la gestione "pseudo-casuale" dei ritardi di linea, e per la comunicazione del passaggio degli autobus dalle fermate.

Ontologie comuni per lo scambio di messaggi

Per tutti gli agenti, si è scelto di definire una logica comune di ricezione dei messaggi. In particolare, ogni agente possiede un proprio *Behaviour (letteralmente "Comportamento")* dedicato alla lettura di messaggi dalla *mailbox (casella di posta)* interna agli agenti. Questo behaviour estende la classe di *Behaviour Ciclici* forniti dal linguaggio, in modo da poter ripetere l'operazione di lettura dei messaggi nel tempo.

L'identificazione dei messaggi ricevuti, avviene sfruttando la proprietà semantica delle *Ontology*, fornita dal sistema JADE. A priori è stato definito un set di ontologie che vengono utilizzate dai vari agenti per etichettare i messaggi inviati nelle richieste, e di conseguenza per identificarli all'atto di ricezione.

```
// *** RICEZIONE MESSAGGI ***
private class ProcessingReplyMessages extends
CyclicBehaviour {
@Override
     public void action() {
     // Definisco il template per identificare i
messaggi
     MessageTemplate template =
MessageTemplate.or(
     MessageTemplate.MatchOntology(PROVA),…);

ACLMessage msg = myAgent.receive(template);
     if (msg != null) {
          switch (msg.getOntology()) {
          // Ontology = "PROVA"
          case PROVA:
               myAgent.addBehaviour(
new AskControllersLineDelay
(msg.getSender()));
               break;
          }
     }
}
```

Cyclic Behaviour per la ricezione e identificazione di messaggi.

Comportamenti specifici degli agenti

Centrale degli autobus

La Centrale degli Autobus (***Central Parking Agent***) si occupa della procedura di **Startup** o *"Avvio"* del sistema. L'implementazione di questa procedura viene delegata ad un *OneShotBehaviour* che legge le stringhe di

configurazione contenute in un file appositamente creato per questa fase di "*Bootstrap"*.

```
// STOP {numero_fermata} {coord_x} {coord_y}
STOP 1 65.7 89.7
STOP 2 65.7 89.7
STOP 3 43.3 82.2

// CONTR_{num_stop}_{num_bus} pr_{num_stop}
// {num_contr_succ} {1_contr_succ} {2_contr_succ}
...
// {num_orari} {1_orario} {2_orario} ...
contr_1_21 pr_1 3 contr_2_21 contr_3_21
contr_3_21 2 7 8

// BUSCONTAINER
BUSCONTAINER

// BUS {num_bus}
BUS 1
BUS 2
BUS 3
```

Esempio di file di configurazione: db.txt

Sono state delineate alcune strutture sintattiche riconosciute dalla *Centrale*, in modo da permettere la configurazione e la creazione automatica degli agenti distribuiti (attraverso l'utilizzo di parametri specifici).

Di seguito viene mostrata la struttura implementativa della classe *Startup* dell'agente **Centrale**.

```
private class StartUp extends OneShotBehaviour {
  @Override
  public void action() {
    ... ...
    try {
      in = new BufferedReader(new
FileReader("dataStore/db.txt"));
      String line;
      while((line = in.readLine()) != null)
      {
        if (!line.startsWith("%") &&
!line.startsWith("//") && !line.equals("")) {
```

```
          String[] words = line.split(" ");
// LETTURA DELLE FERMATE (con coordinate)
          if (words[0].equals("STOP")) {
            Profile p = new ProfileImpl();

p.setParameter(Profile.CONTAINER_NAME,
"BUSSTOP_"+words[1]);

// CREAZIONE CONTAINER - FERMATE
            c = rt.createAgentContainer(p);

// CREAZIONE AGENTI - PR
            c.createNewAgent("pr_"+words[1],
PublicRelationAgent.class.getName(),
null).start();

          // … Lettura termini da stringa per
FERMATA …

// CREAZIONE AGENTI - INTERN_AFFAIR
            AgentController b = c.createNewAgent
("affari_interni_"+words[1],
InternAffairsAgent.class.getName(), obj);
            b.start();
          } else if
(words[0].startsWith("contr")) {
            // … Lettura termini da stringa per
CONTR …
          }

// CREAZIONE AGENTI - CONTROLLER
            c.createNewAgent(words[0],
ControllorAgent.class.getName(), obj).start();
          } else if
(words[0].equals("BUSCONTAINER")) {
            Profile p = new ProfileImpl();
            p.setParameter(
Profile.CONTAINER_NAME, "BUS");

// CREAZIONE CONTAINER - AUTOBUS
            c = rt.createAgentContainer(p);
          } else if (words[0].equals("BUS")) {
            c.createNewAgent("bus_"+words[1],
BusAgent.class.getName(), null).start();
          }
        }
      } in.close();
    } catch Exception ...
    ... ...
```

```
    } finally {
        addBehaviour(new CheckAlive(myAgent,
10000));
    }
  }
}
```

(Parte di codice) Classe Startup - CentralParkingAgent.java

Al termine della classe di Startup viene richiamato un *TickerBehaviour* che ha una funzione di controllo sugli agenti del sistema. Ogni 10 secondi, il *behaviour* controlla che tutti gli agenti siano ancora "vivi" e in caso contrario, provvede a ricrearli nell'apposito *Container*.

Di seguito viene presentata la soluzione implementativa scelta per risolvere questi problemi di *Fault Tolerance*, ed in particolare i due metodi che permettono di verificare se un Agente esiste (*existAgent*) e di ricrearlo in caso di risposta negativa (*createAgent*).

```
private class CheckAlive extends TickerBehaviour
{

    public CheckAlive(Agent a, long period) {
      super(a, period);
    }

    @Override
    protected void onTick() {... ...}

    private boolean existAgent(String name) {
      AMSAgentDescription[] agents = null;
      AMSAgentDescription ad = new
AMSAgentDescription();
      ad.setName(new AID(name, AID.ISLOCALNAME));
      try {
        agents = AMSService.search(myAgent, ad);
        if (agents.length > 0) {
          return true;
        }
      } catch (FIPAException e) {
        System.out.println("Errore,
"+e.getMessage());
      }
      return false;
    }
```

```
    private void createAgent(String agentName,
String containerName, String agentClass, Object[]
args) {
      Runtime rt = Runtime.instance();
      ContainerController c = null;
      try {
        if(containerName.equals
(busContainer.getContainerName())) {
          c = busContainer;
        } else {
          for (int i=0;
i<busstopContainers.size(); i++){
            if(containerName.equals
((busstopContainers.get(i).
getContainerName()))) {
              c = busstopContainers.get(i);
            }
          }
        }
        if (c != null)
// CREO L'AGENTE E LO AVVIO
          c.createNewAgent(agentName, agentClass,
args).start();
      } catch {... ...}
    }
  }
```

(Parte di codice) Classe CheckAlive - CentralParkingAgent.java

Affari Interni

L'agente **Affari Interni** deve principalmente occuparsi della gestione dei *feedback* degli Utenti. Questo agente infatti, riceve un messaggio proveniente dall'agente **Pubbliche Relazioni**, che lo ha a sua volta ricevuto da un **Utente**.

Il suo secondo scopo è quello di andare a calcolare la "distanza giornaliera" ottenuta come sommatoria delle diverse distanze percorse dai singoli utenti e comunicate sotto forma di coordinate x e y (formula delle distanze euclidee sul piano).

In maniera “Attiva” deve poi decidere quando inviare queste distanze di feedback alla **Centrale degli autobus**, utilizzando un *Behaviour temporizzato (TickerBehavioour)*.

```
public class InternAffairsAgent extends Agent {
  String myPR, central;
  double coordX, coordY, dailyDistance = 0;

  private class ProcessingReplyMessages extends
  CyclicBehaviour {
    final String PR_FEEDBACK = "PR_FEEDBACK";

    @Override
    public void action() {
      MessageTemplate mt =
MessageTemplate.MatchOntology(PR_FEEDBACK);
      ACLMessage msg = myAgent.receive(mt);
      if (msg != null) {
        switch (msg.getOntology()) {
          case PR_FEEDBACK:
              // ... ...
              // Lettura di coordX, coordY e
distance
              // ... ...

              dailyDistance += distance;
            }
            break;
        }
      } else {
        block();
      }
    }
  }

  private class FeedbackToCentral extends
  TickerBehaviour {
    final String INTAFF_FEEDBACK =
"INTAFF_FEEDBACK";

    public FeedbackToCentral(Agent a, long
period) {
      super(a, period);
    }

    @Override
    protected void onTick() {
```

```
      System.out.println("Spedisco");
      // ... ...
      //INVIO messaggio di INFORM alla Centrale,
con
      // Ontology=INTAFF_FEEDBACK e
Content=dailyDistance
      // ... ...
      send(msg);
      dailyDistance = 0;
    }
  }
}
```

(Parte di codice) InternAffairsAgent.java

Relazioni pubbliche

Il ***Public Relation Agent (PR)*** deve gestire diverse tipologie di richieste che gli giungono dall'**Utente**. Inoltre funge da tramite tra l'**Utente** e i **Controllori delle linee** degli autobus, e tra gli **Affari Interni** e la **Centrale degli autobus**.

Per questo motivo, l'agente **PR** possiede una struttura di *ontologie* più complessa rispetto ad altri agenti. Di seguito vengono mostrate in dettaglio le tre macro-categorie di *ontologie* che sono state ideate per gestire le 3 diverse tipologie di richieste che deve essere pronto ad accogliere.

```
// *** ONTOLOGY ***
  // Stabilite per la comunicazione tra AGENTI

  // Richiesta: Ritardi e Linee di una Fermata.
  //
**********************************************
***
     final String USER_ASK_LINE_DELAY =
"USER_ASK_LINE_DELAY"; // USER --> PR

     final String PR_ASK_LINE_DELAY =
"PR_ASK_LINE_DELAY";   // PR --> ALL_CONTR
```

```
     final String CONTR_REPLY_LINE_DELAY =
"CONTR_REPLY_LINE_DELAY"; // CONTR --> PR

     final String PR_REPLY_LINE_DELAY =
"PR_REPLY_LINE_DELAY";   // PR --> USER

  // Richiesta: Ritardo e Fermate di una precisa
Linea.
  //
**********************************************
***
     final String USER_ASK_LINE_STOPS =
"USER_ASK_LINE_STOPS";   // USER --> PR

     final String PR_ASK_LINE_STOPS =
"PR_ASK_LINE_STOPS";       // PR --> CONTR

final String CONTR_REPLY_LINE_STOPS =
"CONTR_REPLY_LINE_STOPS"; // CONTR --> PR

     final String PR_REPLY_LINE_STOPS =
"PR_REPLY_LINE_STOPS";    // PR --> USER

  // Richiesta: Invio di Feedback da Utente a AI.
  //
**********************************************
***
     final String USER_SEND_FEEDBACK =
"USER_SEND_FEEDBACK"; // USER --> PR

     final String PR_SEND_FEEDBACK =
"PR_SEND_FEEDBACK";   // PR --> IA
  //
**********************************************
***
```

Ontologie per la comunicazione gestita dal PR.

Il **PR** dunque, oltre a possedere il *Behaviour ciclico* per la ricezione dei messaggi, possiede differenti *Behaviour* per l'invio di messaggi tra i differenti agenti coinvolti.

In particolare, nella richiesta dei ritardi di linea, deve poter comunicare con tutti i **Controllori di linea** della sua fermata. Assecondando la scelta progettuale espressa nel primo paragrafo di questo capitolo (si veda *Conoscenza centrale*), il **PR** non possiede alcuna

informazione relativa agli altri agenti presenti nelle fermate, per questo deve richiedere le informazioni (aggiornate) dal *Servizio AMS* della centrale.

```
// *** INVIO ***

  // ***** AZIONE: RICHIEDI LINEE FERMATA e
RITARDI *****
  // *** AGENTI COINVOLTI: UTENTE --> PR   PR -->
CONTR
  //                        CONTR --> PR   PR -->
UTENTE

  private class AskControllersLineDelay extends
  OneShotBehaviour {
      private AID[] getControllerAgents(){
        // Ricerca i controllori, sfruttando
l'AMSService
        // Il PR non possiede conoscenza interna,
deve
        // richiedere tutto all'esterno.
      }
    // ... ...
  }

  // Appena il PR riceve il messaggio di un
Controller,
  //  invia Contr + Delay all'utente.

  private class SendControllersLineDelayToUser
extends
  OneShotBehaviour { ... ... }
  //
**********************************************
***

  // ***** AZIONE: RICHIEDI FERMATE SULLA MIA
LINEA *****
  // *** AGENTI COINVOLTI: UTENTE --> PR   PR -->
CONTR
  //                        CONTR --> PR   PR -->
UTENTE

  private class AskControllerLineStops extends
  OneShotBehaviour { ... ... }

  // Invio all'UTENTE l'elenco delle fermate
```

```
sulla LINEA
  //  RICHIESTA

  private class SendControllerLineStopsToUser
extends
  OneShotBehaviour { ... ... }
  //
**********************************************
***

  // ***** AZIONE: INVIA FEEDBACK UTENTE a AI
*****
  // *** AGENTI COINVOLTI: UTENTE --> PR   PR -->
CONTR
  //  CONTR --> PR   PR --> UTENTE

  private class SendUserFeedbackToAI extends
  OneShotBehaviour { ... ... }
```

(Parte di codice) PublicRelationAgent.java

Controllore delle linee

Per ogni fermata, e per ogni linea di autobus che passa da quella fermata, esiste un differente **Controllore di linea (CONTR)**. Ogni **CONTR** deve gestire le richieste che provengono dagli **Utenti** (inoltrate dai **PR**), e comunicare o richiedere i "ritardi di linea" ai **CONTR** delle altre fermate sulla linea.

Per questo è stato necessario definire 2 differenti gruppi di *ontology*, il primo relativo alle comunicazione "PR-CONTR" e "CONTR-PR", il secondo per le comunicazioni "CONTR-CONTR".

```
// *** ONTOLOGY ***
  // PR --> ALL_CONTR
  final String PR_ASK_LINE_DELAY =
"PR_ASK_LINE_DELAY";
  // CONTR --> PR
  final String CONTR_REPLY_LINE_DELAY =

"CONTR_REPLY_LINE_DELAY";

  // PR --> CONTR
```

```
  final String PR_ASK_LINE_STOPS =
"PR_ASK_LINE_STOPS";
  // CONTR --> PR
  final String CONTR_REPLY_LINE_STOPS =

"CONTR_REPLY_LINE_STOPS";

  // CONTR --> CONTR
  final String CONTR_INFORM_CONTR =
"CONTR_INFORM_CONTR";
  final String CONTR_LISTEN_CONTR =
"CONTR_LISTEN_CONTR";
//
**********************************************
```

Ontologie per la comunicazione gestita dal Controllore di linea.

Al *Setup* di questo Agente vengono reperiti dalla **Centrale** i valori delle seguenti variabili (unica conoscenza interna all'agente):

```
// ELENCO DELLE FERMATE SULLA LINEA
protected List<String> ;
// ELENCO DEI TEMPI DELLA LINEA
protected List<Integer>;
// IDENTIFICATIVO DEL PR (di riferimento)
protected String myPR;
// RITARDO ATTUALE DI LINEA (AGGIORNATO)
protected int lateInThisMoment = 0;  // Display
ritardi
```

Parametri reperiti al Setup dell'Agente.

Per calcolare il ritardo degli autobus, sono state effettuate alcune considerazioni:

- Si presuppone che un autobus non tardi tanto da passare dopo l'orario del successivo, e non passi mai in anticipo.
- Semplifichiamo la lista degli orari con un elenco di interi (es. 5,6,7,8,ecc).
- Il ritardo viene calcolato per differenza tra l'orario atteso (autobusInTimeTable) e l'orario di passaggio effettivo dell'autobus (timePassed).

- Il valore mostrato è un "ritardo stimato" quindi ottenuto come differenza tra (timePassed - autobusInTimeTable).

```
private int getLateTime(int timePassed) {
      int autobusInTimeTable = timeTable.get(0);
      boolean findAutobus = false;
      String late = "";
      for (int i = 1; i < timeTable.size() &&
                          !findAutobus; i++) {
        if (timePassed <= timeTable.get(i)) {
          autobusInTimeTable = timeTable.get(i);
          findAutobus = true;
        }
      }
      return (int)(timePassed -
autobusInTimeTable);
    }
  }
```

(Parte di codice) Funzione per il calcolo del ritardo di linea.

Al passaggio dell'**Autobus**, viene comunicato il ritardo ai **CONTR** successivi su tale linea.

```
private class WarnOtherController extends
OneShotBehaviour {
    private Integer late = 0;
    public WarnOtherController(int lateTime) {
      late = lateTime;
    }
    @Override
    public void action() {
      // ...
      // Ad ogni successiva fermata sulla linea,
      // INVIO un messaggio di INFORM al CONTR,
      // con Ontology=CONTR_INFORM_CONTR e
Content=late
      // ...
      }
    }
  }
```

(Parte di codice) Comportamento "Reattivo" – Comunico il ritardo agli altri Controllori sulla mia linea.

Infine per la gestione delle richieste del **PR**:

- *ReplyToPRForList* – CONTR restituisce il ritardo di linea e l'elenco delle fermate sulla linea.
- *ReplyToPRForLines* – Ogni CONTR della stessa fermata del PR richiedente, risponde al PR comunicando il suo nome.

```
private class ReplyToPRForList extends
OneShotBehaviour {
    // ...
    // INVIO un messaggio di INFORM al PR,
    // con Ontology=CONTR_REPLY_LINE_STOPS e
    // Content = late + elenco delle fermate
    // ...
  }

  private class ReplyToPRForLines extends
  OneShotBehaviour {
    // ...
    // INVIO un messaggio di INFORM al PR,
    // con Ontology= CONTR_REPLY_LINE_DELAY e
    // Content = nome del CONTR
    // ...
  }
```

(Parte di codice) ControllerAgent.java

Autobus

L'agente ***Autobus (BusAgent)*** deve gestire le interazioni con gli **Utenti** e inviare messaggi ai **Controllori,** al passaggio dalle fermate.

Di seguito vengono elencate le quattro tipologie di messaggi che devono essere gestite, e parte di codice utile per gestirle (contenuto nella classe *BusAgent*):

- In ricezione, la salita di un passeggero.
- In ricezione, la discesa di un passeggero.
- Inviare alla fermata le info di passaggio .
- Ricevere il segnale del "sensore" per cominciare a spedire info.

```
public class BusAgent extends Agent{
  private int passengers;
  private int totSpace;
  private HashMap<String, Integer> passengerUp;
  private HashMap<String, Integer> passengerDown;
```

```
  private class Listening extends CyclicBehaviour
{

    @Override
    public void action() {
      // Controllo il MessageTemplate per le
Ontology
        switch(incomingMsg.getOntology()){
        case "ON_NOTIFY_BUS_ON":
// ...
          // SALITA DI UN PASSEGGERO
          // ...
          break;

        case "ON_NOTIFY_BUS_OFF":
          // ...
// DISCESA DI UN PASSEGGERO
          // ...
          break;

        case "sensorActivation":
          // ...
          // PREPARO MESSAGGIO DI INFORM ALLA
FERMATA
   // Conosco il numero della fermata perché
   // seguo la mia linea
          // ...

          // FA SCENDERE TUTTI I PASSEGGERI CHE
HANNO
          // RICHIESTO DI SCENDERE
   // Controllando la fermata e aggiornando
   // la disponibilità di posti e il totale
   // (passengerDown)
// ...

          // FA SALIRE TUTTI I PASSEGGERI CHE
HANNO
          // RICHIESTO DI SALIRE ALLA FERMATA
   // Controllando disponibilità immediata di
   // posti e aggiornando il totale a bordo
   // (passengerUp)
// ...

          // ...
          // INVIO MESSAGGIO DI INFORM AL
CONTROLLORE
          // ...
```

```
        }
      }
   }
}
```

(Parte di codice) BusAgent.java

Implementazione con Jade-Mobile

Utente mobile

L'agente *Utente* o *UserAgent*, che è stato implementato su piattaforma Jade Leap Mobile, deve interagire con diversi agenti periferici del sistema.

Per questo motivo, analogamente a quanto fatto per le altre classi precedentemente introdotte, sono state definite le *ontologie* di comunicazione tra agenti:

```
// Restituisce all'utente l'elenco delle fermate
della
// linea X, con il relativo ritardo di linea.
  final String ONT_ASK_STOPS_AND_DELAY_FOR_LINE =
"USER_ASK_LINE_STOPS";
  final String ONT_RCV_STOPS_AND_DELAY =
"PR_REPLY_LINE_STOPS";

// Restituisce all'utente l'elenco delle Linee
che
// passano dalla fermata X, con il relativo
ritardo (di
// linea).
  final String ONT_ASK_LINES_DELAY_FOR_BUS_STOP =
"USER_ASK_LINE_DELAY";
  final String ONT_RCV_LINES_DELAY =
"PR_REPLY_LINE_DELAY";

// Permette all'utente di inviare feedback agli
AI,
// facendo inoltrare il messaggio dal PR della
fermata
```

```
// presa in considerazione per il feedback.
  final String ONT_FEEDBACK =
"USER_SEND_FEEDBACK";

  final String ONT_NOTIFY_BUS_ON =
"ONT_NOTIFY_BUS_ON";
  final String ONT_NOTIFY_BUS_OFF =
"ONT_NOTIFY_BUS_OFF";
```

Ontologie per la comunicazione gestita dall'Utente.

L'agente **Utente** deve infatti gestire le seguenti tipologie di messaggi:

- Richiesta lista delle linee servite da una fermata (con ritardi).
- Richiesta stato della linea passante per una fermata (la risposta contiene il ritardo e la lista di fermate che saranno raggiunte).
- Invio feedback agli Affari Interni.
- Notifica al BUS che l'utente sta salendo (Simulazione "realtà").
- Notifica al BUS che l'utente sta scendendo (Simulazione "realtà").

Analogamente a quanto visto per gli altri Agenti, anche in questa classe sono stati implementati un *Behaviour ciclico* per la gestione dei messaggi in ingresso (che per ragioni di spazio non verrà mostrato nel testo) e diversi *behaviour* per la gestione dei comportamenti di invio specifici.

Vengono di seguito presentate l'interfaccia della classe *(IUserAgent)*:

```
package it.sbaam.jade_leap.user.agent;

public interface IUserAgent {
  public void notifyBus(int bus, int busStop,
Boolean amIGoingUp);
  public void sendFeedback(int busStop,
String coordinates);
  public void getLinesDelayForBusStop(int
busStop);
  public void getBusStopsAndDelayForLine(
```

```
int busStopToAskTo, int line);
}
```

IUserAgent.java

E alcune parti significative di codice, della classe stessa *(UserAgent)*.

```
public class UserAgent extends Agent implements
IUserAgent{
  // ... ...

  // *** RICEZIONE ***
  private class HandleResponses extends
CyclicBehaviour { ... }

  // *** INVIO ***
  private class getLinesDelayForBusStopBehavior
extends
OneShotBehaviour { ... }
  private class
getBusStopsAndDelayForLineBehavior
extends OneShotBehaviour { ... }

  private class notifyBusBehavior extends
OneShotBehaviour {
    @Override
    public void action() {
      // ... ...
      //INVIO messaggio di INFORM all'Autobus,
con
      // Ontology= ONT_NOTIFY_BUS_ON o
ONT_NOTIFY_BUS_OFF
 // ... ...

 // ... Memorizzo messaggio sul LOG ...
    }
  }

  private class sendFeedbackBehavior extends
OneShotBehaviour {
    @Override
    public void action() {
      // ... ...
      //INVIO messaggio di INFORM al PR, con
      // Ontology= ONT_FEEDBACK e
Content=coordinateXY
 // ... ...

 // ... Memorizzo messaggio sul LOG ...
```

```
      }
   }

}
```

(Parte di codice) UserAgent.java

Applicazione Android

La struttura e il comportamento dell'agente *UserAgent* non sono influenzati dal fatto che esso debba essere eseguito sotto Android, tranne per quanto concerne l'input e l'output. Nello specifico, gli input sono i dati sotto forma di stringa che l'utente immette nella GUI dell'applicazione per specificare ad esempio il nome della linea di cui vuole richiedere informazioni. Gli output sono le risposte anch'esse testuali che il sistema fornisce a seguito delle domande.

Le richieste sono realizzate tramite alcuni *OneShotBehaviour* generati alla pressione di bottoni della GUI. Il meccanismo attraverso il quale è possibile far interagire il mondo ad oggetti della programmazione Android e il mondo ad agenti della programmazione Jade è quello delle *O2AInterfaces* (object to agent interfaces).

```
...
private IUserAgent userAgent;
...
userAgent =
MicroRuntime.getAgent(agentName).getO2AInterface(
IUserAgent.class);
... public class UserAgent extends Agent
implements IUserAgent
```

(Parte di codice) IUserAgent.java

Dopo l'esecuzione di queste istruzioni, è possibile invocare i metodi definiti nell'interfaccia *IUserAgent* sull'oggetto *userAgent* che è ora collegato all'agente il cui nome è contenuto nella variabile *agentName* e la cui definizione è:

```
public class UserAgent extends Agent implements IUserAgent
```

L'unico compito dei metodi dell'interfaccia è aggiungere all'agente un *behaviour* che effettuerà le proprie operazioni in un'ottica ad agenti piuttosto che in una orientata agli oggetti.

Seguono due *screenshot* che mostrano la minimale interfaccia grafica dell'applicazione:

A sinistra vediamo l'interfaccia principale con i bottoni per connettersi ed eseguire le varie interrogazioni sul sistema, i campi di testo dove inserire i dati ad esse necessari ed infine un log testuale dove viene riportato lo stato dell'applicazione e l'esito delle richieste. A destra la schermata delle impostazioni dove inserire i dati per la connessione alla piattaforma.

L’applicazione è stata sviluppata sulla falsa riga dell’esempio reperibile a questo indirizzo: http://jade.tilab.com/papers-examples.htm

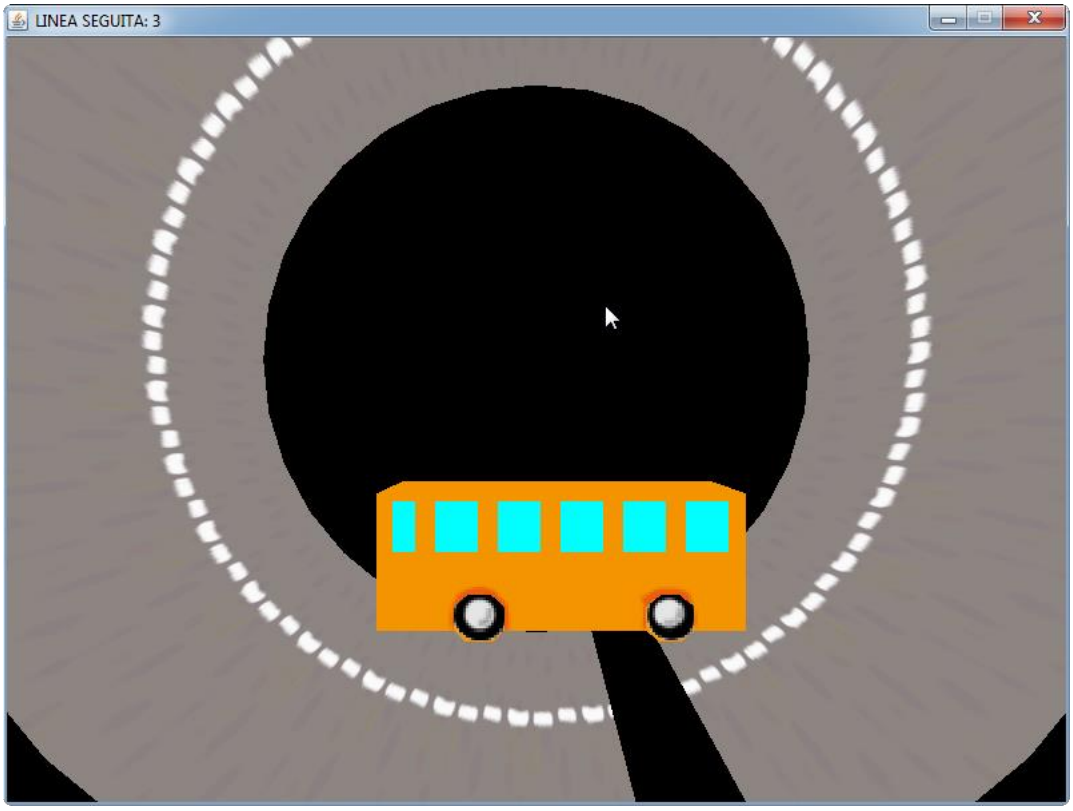
(Se ne consiglia la consultazione per approfondimenti sull’utilizzo di Jade-Leap Android).

Capitolo 5
GUI e Simulatore

Per testare l'efficacia e la correttezza del sistema è stato aggiunto un modulo ausiliario (non necessario alla diretta soluzione del problema) affinché un unico "cuore pulsante" potesse dare vita alla realtà modellata dall'intero sistema.

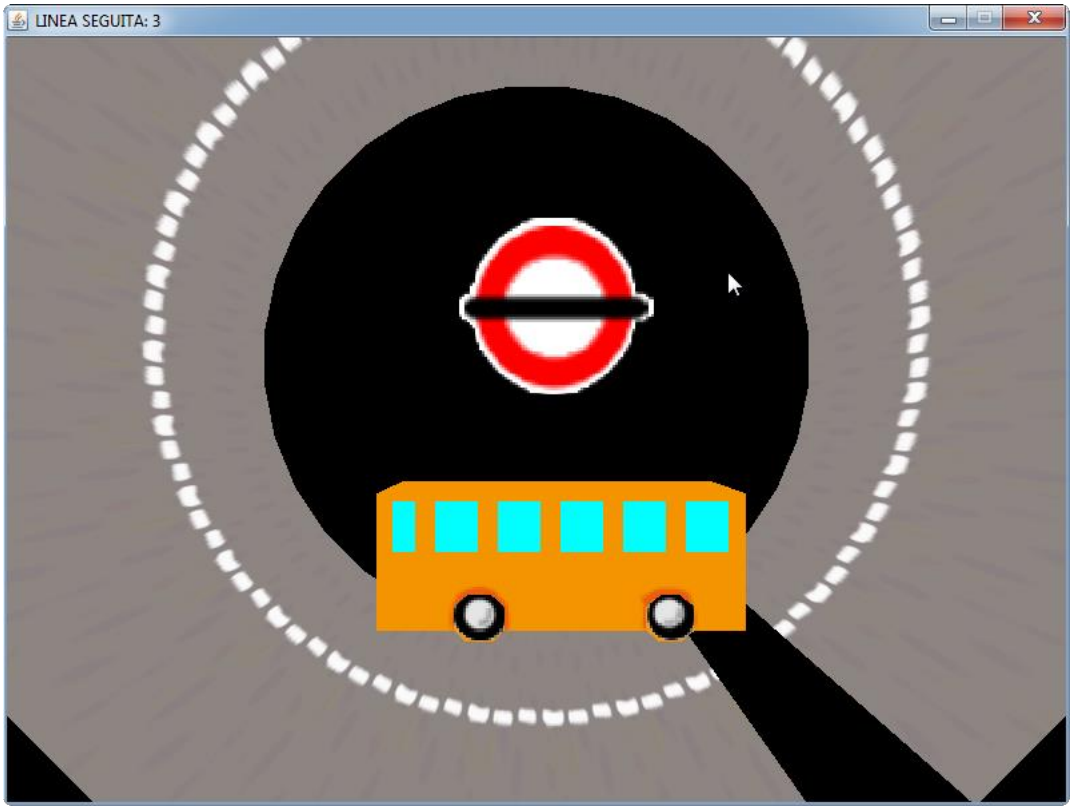
Un ulteriore agente, simulando quelli che potrebbero essere gli autisti di linea impiegati dall'azienda committente, guida i mezzi per la città, basandosi su stime di probabilità descriventi l'evento aleatorio "autobus in ritardo". Qualora l'autobus fosse in ritardo, un messaggio (idealmente avviato dal passaggio del veicolo in esame davanti ad un sensore in grado di riconoscerlo) viene inviato ad opera del Simulatore all'opportuna fermata; viceversa l'assenza di un atto comunicativo è in sé comunicazione implicita di ritardo.

Per rendere la visualizzazione più intuitiva ed accattivante si è corredato il sistema di una semplice visualizzazione grafica: un piccolo modello di autobus corre su un circuito infinito (a rappresentare la continuità del servizio di trasporto pubblico, garantita ora più di prima grazie all'ausilio della nostra applicazione!).



Indicazioni grafiche esplicite rappresentano lo stato attuale della corsa.

L'autobus si ferma all'apparire di una fermata:



E accelera la corsa - sormontato da un incombente punto esclamativo ("!") qualora sia in ritardo.

Per spostare il monitoraggio da una linea ad un'altra è sufficiente l'uso delle frecce direzionali: freccia sinistra per prendere visione della linea che precede la correntemente visualizzata nella lista, freccia destra per spostarsi a quella seguente.

Ovviamente è solo un semplice abbellimento grafico, con molteplici spunti per miglioramenti in probabili sviluppi futuri (per esempio potrebbe essere fra i *desiderata* del committente la possibilità di selezionare direttamente una data corsa per una linea specificata). È stata inoltre presa la decisione di non eliminare l'output testuale, per corredare la parte puramente visuale ad una stampa di informazioni utili e di interesse.

Non facendo direttamente parte della soluzione applicativa presentata per il problema individuato, non descriveremo nel dettaglio le soluzioni implementative utilizzate.

Sono comunque state realizzate due classi Java per la gestione di modelli tridimensionali, sia per quel che riguarda l'import degli stessi (CWavefront.java) che per gli aspetti inerenti alle relative texture (TGA.java); in supposti sviluppi futuri sarebbero quindi necessarie poche modifiche per la realizzazione di un modello simulante un'intera città, per eventuali analisi di varia natura su una realtà rappresentata in maniera decisamente più articolata.

Conclusioni

L'intero sistema creato soddisfa i requisiti prefissati. Riepilogando, rappresenta una modalità di gestione "intelligente" della viabilità pubblica di una ipotetica città. Nonostante, come già osservato, sia una tecnologia già presente in diverse località, SBAAM integra gli aspetti utili delle piattaforme già esistenti con la semplicità amministrativa di un sistema che, salvo gravi problemi, si autogestisce in svariati aspetti.

Da un lato, il paradigma orientato agli agenti ha semplificato la progettazione fornendoci delle basi su cui costruire l'ambiente (in primis la stessa entità agente, con tutte le sue caratteristiche).

Dall'altro, Jade ha aiutato la fase di implementazione: la scelta di questa tecnologia, oltre alle motivazioni spiegate nel capitolo 2 (possibile uso di librerie mobile integrate e *framework* orientato a una progettazione *agent-based*), ci ha permesso di non preoccuparci di problemi di basso livello (come scelta di collegamenti, protocolli e gestioni degli errori direttamente connessi), in modo da concentrarci maggiormente su problemi di alto livello (come la gestione dello scambio di messaggi o la *fault tolerance*).

In conclusione è utile fare un confronto tra il sistema realizzato e ciò che ancora potrebbe essere migliorato. La proprietà di *fault tolerance* è garantita ampiamente ma non completamente: come spiegato nei capitoli precedenti,

abbiamo ipotizzato che alcune parti della piattaforma siano immuni da problemi che ne potrebbero causare l'irraggiungibilità. Tra queste vi sono ovviamente il main-container, la cui "caduta" genererebbe il blackout totale del sistema, e il deposito centrale, che è l'addetto alla creazione iniziale degli agenti e al monitoraggio degli stessi. Se queste ipotesi continuassero a valere, l'unico problema sarebbe la morte di un container, caso ancora non gestito e che in un'eventuale miglioria potrebbe essere preso in considerazione.

La natura accademica del progetto ci ha permesso di contenere la complessità reale del problema, limitandola solo agli aspetti caratteristici dello stesso (es. non è gestita la corsa di più autobus contemporanei sulla stessa linea, il "meccanismo del tornello" è simulato tramite l'invio di un messaggio direttamente dall'utente, …).

Seppur progettato e implementato solo a scopi dimostrativi, anche il simulatore è un agente e, come tale, parte della piattaforma presentata.

Bibliografia

Documentazione e Applicazione Jade Leap Android. (s.d.). Tratto da Jade TiLab: http://jade.tilab.com/papers-examples.htm

JADE - Java Agent Development Framework. (s.d.). Tratto da http://jade.tilab.com/: http://jade.tilab.com/