

Relazione Sistemi Distribuiti 16-17

TraMa²

“Traffic MAnagement Multi-Agent system”

<https://bitbucket.org/edoardoantonini722/trama2>

Alessio Addimando

Edoardo Antonini

Indice

1. Introduzione	3
2. Aspetti fondamentali e richiami teorici	4
3. Analisi	8
3.1 Requisiti	8
3.2 Parti critiche.....	9
4. Architettura software	10
4.1 Architettura generale	10
4.2 Design dettagliato.....	11
4.2.1 CameraBehaviour	11
4.2.2 TrafficPoleBehaviour	12
4.3 Comunicazione e protocolli	13
4.3.1 Subscribe Interaction Protocol	13
4.3.2 Topic-based communication	14
5. Hardware	14
6. Sviluppo	15
6.1 Testing	15
6.2 Divisione dei compiti	15
6.3 Note di sviluppo.....	16
7. Commenti finali	17
7.1 Conclusioni e lavori futuri.....	17
Appendice A - Guida utente	18
Bibliografia.....	19

1. Introduzione

Il progetto TraMa² nasce dal desiderio di sviluppare un sistema multi-agente per la gestione “intelligente” di un incrocio stradale.

A tale scopo sono stati presi in considerazione, in fase di design, i vari componenti che formano l’incrocio (i.e. semafori, telecamere di monitoraggio, pulsanti di prenotazione pedonale etc.) valutandone l’interazione e considerando le varie situazioni possibili nel mondo reale. Lo scopo è quello di ottimizzare il flusso del traffico.

Il lavoro interessa un solo incrocio, ma il software può essere facilmente esteso in modo da gestirne molteplici.

La simulazione è stata fatta su 3 Raspberry Pi in cui è stata installata la piattaforma JADE.

2. Aspetti fondamentali e richiami teorici

Sistemi distribuiti

Solitamente un sistema distribuito viene definito come un insieme di computer indipendenti che comunica attraverso la rete e che appare all'utente come un singolo sistema coerente. Quest'ultimo svolge computazioni distribuite, cioè calcoli dove almeno due processi hanno contesti spaziali differenti. Come è ovvio, quindi, il concetto di "distribuito" è fortemente legato a quello di contesto spaziale.

IN MERITO AL PROGETTO: Il nostro progetto può essere definito un sistema distribuito, infatti i Raspberry comunicano tramite la rete svolgendo computazioni distribuite.

Agenti e sistemi multi-agente

L'aumento esponenziale della complessità del software che ha portato i sistemi computazionali ad assumere una natura pervasiva e autonoma, ha messo in crisi i tradizionali metodi di Software Engineering portando verso un cambio di paradigma.

La programmazione ad agenti va oltre quella ad oggetti ed incapsula, oltre allo stato e al comportamento, il flusso di controllo. Gli agenti quindi sono entità attive, autonome e che non possono essere invocate, al loro interno, a tal merito, è incapsulato il criterio per auto-governare il proprio flusso di controllo. Nel paradigma agent-oriented, durante lo scambio di messaggi, vengono trasmessi solo i dati e non il controllo, il data flow, quindi, va considerato indipendentemente da quest'ultimo.

Gli agenti comunicano scambiandosi informazioni; per esempio, tramite l'Agent Communication Language (ACL).

Questi possono essere considerati come singoli individui di un sistema multi-agente (MAS), dove rappresentano un singolo flusso di controllo tra una molteplicità; inoltre, possono essere distribuiti nell'ambiente.

Nei sistemi multi-agente l'interazione è una dimensione cardine, infatti, gli agenti devono comunicare tra loro e con l'ambiente in modo da ottenere le informazioni per portare a termine i loro compiti.

Inoltre, nei MAS sono presenti sia obiettivi individuali che globali e l'unico modo per gestirli entrambi, senza farli collidere, è usare le interazioni.

Riassumendo, un agente presenta le seguenti caratteristiche:

- Autonomia: incapsula il controllo, non ha interfacce e non può essere invocato.
- Interattività: in quanto componente di un MAS, deve interagire con il sistema scambiando informazione e conoscenza.
- Proattività: in quanto parte attiva e autonoma di un sistema rende possibile l'esecuzione di eventi piuttosto che aspettare passivamente.
- Reattività: il cambiamento dell'ambiente dove è *situato* deve provocare reazioni tempestive.
- Socialità: vive e interagisce nella sua "società di agenti" e nel MAS.

In aggiunta questo può: essere *intelligente, essere mobile e apprendere*.

IN MERITO AL PROGETTO: gli agenti JADE presentano le caratteristiche sopracitate e, inoltre, possono essere spostati da un container all'altro (mobilità). Non presentano goal o task ma definiscono un Behaviour che rappresenta un determinato comportamento dell'agente. Come spiegato più in dettaglio successivamente, durante l'esecuzione dei Behaviour gli agenti di TraMa2 si scambiano informazioni necessarie al funzionamento.

Middleware per sistemi distribuiti

Il middleware è generalmente definito come un software situato tra il sistema operativo e le applicazioni. Facilita lo sviluppo delle applicazioni e l'integrazione con il codice *legacy*, astruendo dal sistema operativo. Una caratteristica fondamentale è l'interoperabilità, ossia la capacità di far lavorare insieme più applicazioni che utilizzano, appunto, lo stesso middleware. In sostanza, la presenza di quest'ultimo cerca di definire un modello su un gruppo di applicazioni eterogenee.

Un'altra caratteristica sta nel favorire la comunicazione tra le entità del sistema introducendo, per esempio, protocolli di comunicazione. In particolare i *MoM* (Message-oriented Middleware) forniscono un servizio per lo storage dei messaggi che utilizza code sia per il mittente che per il destinatario garantendo un disaccoppiamento temporale tra i dialoganti.

Ulteriore possibile goal del middleware è occuparsi del processo di *naming* in modo scalabile ed efficiente preservando la natura predefinita dei sistemi distribuiti (es. location, openness, mobility).

IN MERITO AL PROGETTO: JADE è un agent-oriented middleware basato su Java che fornisce un framework per lo sviluppo multi-agente distribuito garantendo interoperabilità tramite gli standard FIPA.

Fornisce un sistema per il distributed message passing e per il distributed naming (AMS, "pagine bianche" e DF, "pagine gialle") oltre a diversi altri servizi utili allo sviluppo.

Il concetto di tempo nei sistemi distribuiti e la coordinazione

In un sistema non distribuito il tempo non è ambiguo, come al contrario succede in quelli distribuiti. Infatti, in questi, non è formalmente possibile fornire una nozione naturale di tempo.

Di conseguenza si può far uso del tempo fisico o, quando non necessario, è possibile svincolarsi totalmente dalla realtà introducendo una nozione di tempo logica. In particolare, quest'ultima è utile quando ciò che importa principalmente è l'ordine in cui accadono gli eventi.

In generale, tuttavia, ordinare gli eventi non è abbastanza. Bisogna, infatti, utilizzare delle policy più articolate come, per esempio, quelle per la gestione della mutua esclusione.

Il problema generale della coordinazione è quello di governare l'interazione basata sia sul tempo, sia sulla natura delle azioni, sia mirata al completamento dello scopo globale del sistema.

Alcuni esempi di meccanismi di coordinazione possono essere:

- *Il Message passing* dove non ci sono particolari astrazioni a parte il messaggio e ogni pattern di coordinazione è aggiunto per esempio sotto forma di protocollo.
- *L'agent communication language (ACL)* che rispetta lo standard FIPA ACL e dove la semantica è gestita tramite ontologie. Neanche qui è presente un vero e proprio modello di coordinazione senza i protocolli.
- ...e altri come *Middleware* e *Web Server*.

Tuttavia, questi meccanismi si limitano ad abilitare la coordinazione senza governarla, cioè senza dire cosa un componente può o non può fare.

Per governare la coordinazione, nei sistemi distribuiti, si può ricorrere ad un *coordination model*.

Un particolare caso è quello che prevede la presenza di coordination media, entities, o coordinabili, e laws; dove il coordination medium abilita e governa l'interazione tra i coordinabili osservando le laws.

E' possibile utilizzare coordination model di due tipi:

- control-oriented, cioè basati sull'atto di comunicazione
- data-oriented, cioè basati sull'informazione scambiata durante la comunicazione. Un particolare esempio è la coordinazione *Tuple-based* (LINDA / TuCSoN).

Approccio oggettivo e soggettivo alla coordinazione:

L'approccio oggettivo si ha quando i concetti relativi alla coordinazione vengono estratti dagli agenti e definiti in astrazioni esterne (es. coordination artifacts).

In quello soggettivo, duale al precedente, i problemi di coordinazione sono direttamente affrontati dagli agenti al fine di raggiungere nel modo migliore il proprio obbiettivo.

Quest'ultimo può non bastare a definire modelli di coordinazione complessi o a gestire le interazioni agente-ambiente. È perciò buona pratica, in sistemi più estesi, unire i due approcci facendo attenzione a non minare l'autonomia degli agenti.

IN MERITO AL PROGETTO: per la coordinazione degli agenti in JADE è stato utilizzato un approccio prettamente logico basato, quindi, sull'ordinamento degli eventi (es. alternanza delle luci di semafori).

Nel nostro caso il problema di coordinazione era di medio-bassa difficoltà, per cui abbiamo optato per un approccio soggettivo con scambio di messaggi utilizzando i protocolli FIPA, come il Subscribe Pattern.

Tuttavia il codice può risultare poco elegante, perciò, in futuro, potrebbe essere valutata un'integrazione con TuCSoN4JADE in modo da arricchire il progetto dei benefici di un approccio oggettivo più moderno basato sulla coordinazione tuple-based.

Code mobility

A volte è necessario dover spostare in una nuova collocazione il codice da eseguire, per migliorare la scalabilità, il load balancing e anche la fault tolerance.

La mobilità può essere di tipo *strong* quando il segmento di esecuzione, con *stack* e *program counter*, viene spostato insieme al codice stesso; oppure *weak* nel caso venga spostato solo il codice. Nel primo caso, i processi possono essere bloccati, spostati e fatti ripartire dallo stesso punto su un'altra macchina, ciò, però, richiede molte risorse. La weak mobility è molto semplice ma in questo caso il codice viene eseguito ogni volta ex-novo (es. Javascript).

IN MERITO AL PROGETTO: JADE è una piattaforma distribuita per agenti, nel senso che il sistema può essere distribuito su diversi device che compongono la rete, inoltre, supporta una forma di intra-platform agent strong mobility.

Le nozioni di luogo e topologia sono fornite dai container e dalle piattaforme, con i container mappati sulla distribuzione fisica degli host.

3. Analisi

3.1 Requisiti

Qui di seguito si indicano le caratteristiche del problema.

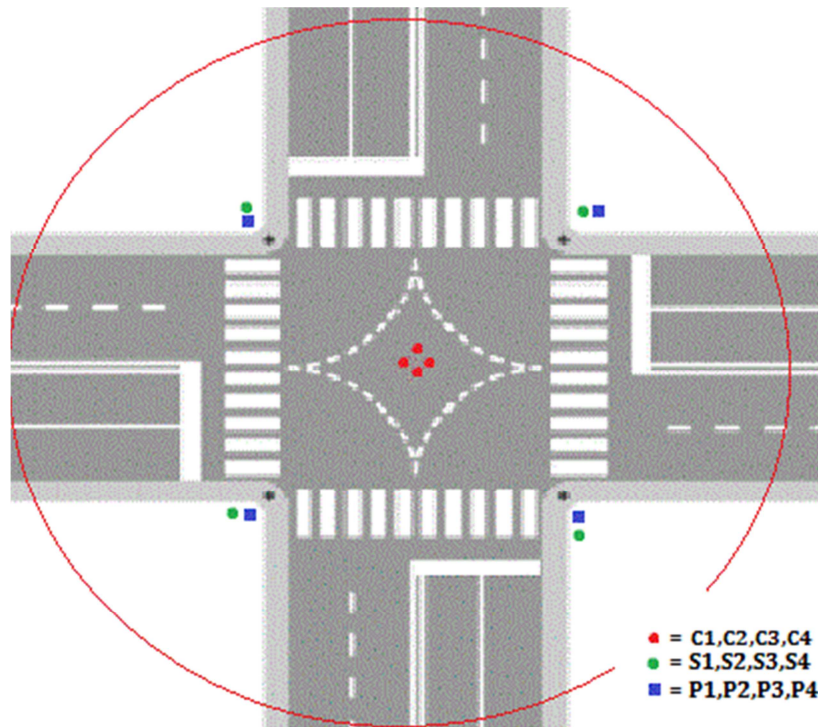


Figura 1 - Schema incrocio

S: agente con due scopi: la gestione della luce dei pedoni e la gestione della luce delle vetture. I semafori opposti sono accoppiati e assumono perciò lo stesso agente.

P: pulsante della prenotazione pedonale.

C: telecamera che riconosce i veicoli, le infrazioni (tramite comunicazione con S) e gli incidenti bloccando il traffico e chiamando i soccorsi.

Comportamento di C_i :

- Rileva i veicoli in un raggio di 100m.
- Ad ogni attraversamento della linea dell'incrocio comunica con il semaforo (colore) ed eventualmente invia alle forze dell'ordine gli estremi per la stesura della contravvenzione.
- Rileva sinistri, comunica ai semafori l'evento e chiama i soccorsi.

Comportamento di S_i e P_i :

Caso (0)

SITUAZIONE (INIZIALE): non ci sono veicoli.

Caso (0a): ci sono pedoni.

COMPORTAMENTO: Tutti gli S rossi. Tutti i P sono verdi.

Caso (0b): non ci sono pedoni.

COMPORTAMENTO: Tutti gli S rossi. Tutti i P rossi.

Caso (1)

SITUAZIONE: arrivano veicoli da entrambe le direzioni.

Caso (1a): ci sono pedoni.

COMPORTAMENTO: il semaforo assume il behaviour classico.

Caso (1b): non ci sono pedoni.

COMPORTAMENTO: il semaforo assume il behaviour classico ma non ci sono P attivi.

Caso (2)

SITUAZIONE: arrivano veicoli da una sola direzione

Caso (2a): ci sono pedoni.

COMPORTAMENTO: S diventa verde per quella direzione e diventa rosso se ci sono pedoni.

Caso (2b): non ci sono pedoni.

COMPORTAMENTO: S diventa verde per quella direzione e non cambia fino all'arrivo di un evento.

Caso (3)

SITUAZIONE: la telecamera X rileva un sinistro.

COMPORTAMENTO: tutti i semafori lampeggiano giallo.

N.B. Sequenza classica colori semaforo: VERDE -> GIALLO -> ROSSO

3.2 Parti critiche

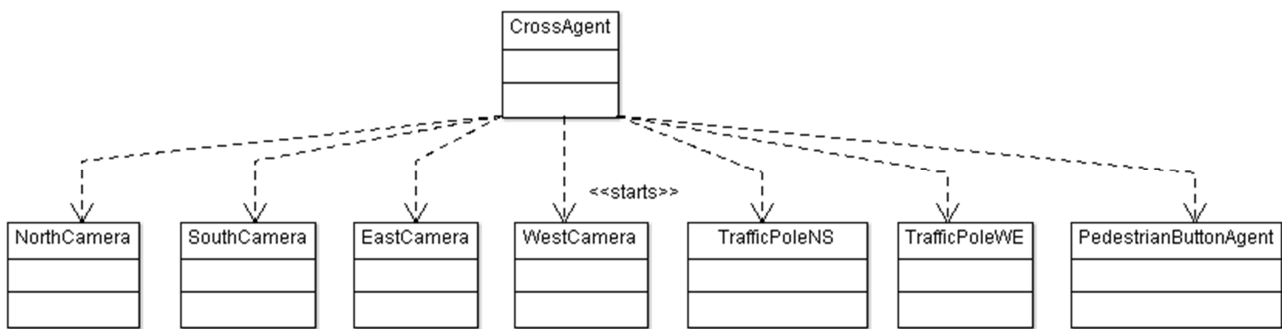
Durante la realizzazione del progetto è stata posta particolare attenzione su ciò che interessa la reattività del sistema e la gestione delle chiamate pedonali.

Nel primo caso, il comportamento dei vari agenti è stato modellato così da non impedire la ricezione di nuovi eventi che avrebbero potuto cambiarne lo stato.

Nel secondo, invece, l'esigenza proveniva dal dominio applicativo; è stato necessario fare in modo che il comportamento del pulsante pedonale fosse gestito evitando chiamate consecutive che avrebbero reso il semaforo pedonale sempre verde.

4. Architettura software

4.1 Architettura generale



4.2 Design dettagliato

4.2.1 CameraBehaviour

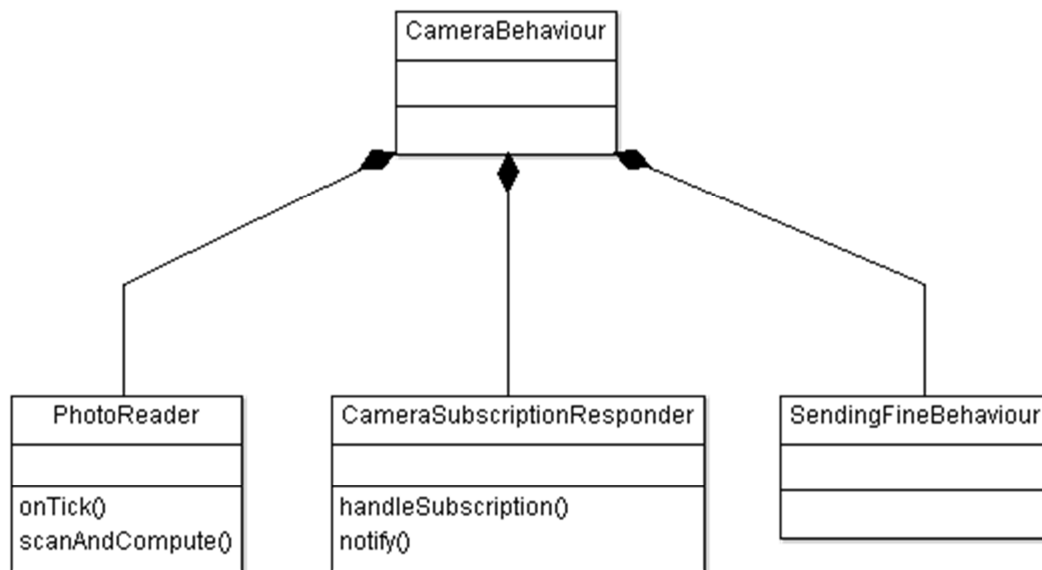


Figura 3 - Struttura *CameraBehaviour*

Il Behaviour utilizzato dalle 4 camere è un *ParallelBehaviour* composto da 3 subBehaviour definiti come inner class.

Il *PhotoReader* si occupa di gestire le informazioni che arrivano dalla camera e computarle tramite il metodo *scanAndCompute* in modo da identificare il tipo di evento accaduto che poi verrà notificato ai semafori. E' stato scelto un *TickerBehaviour* in modo da simulare la velocità di lettura dei frame della camera (in base agli fps).

Il *CameraSubscriptionResponder* viene utilizzato per implementare il protocollo d'interazione "Subscription", in sostanza i semafori si registrano alle notifiche inviate dalla camera in modo da reagire agli eventi.

Il *SendingFineBehaviour* ha lo scopo di mandare una multa quando la camera notifica un attraversamento della linea semaforica mentre il semaforo è rosso.

4.2.2 TrafficPoleBehaviour

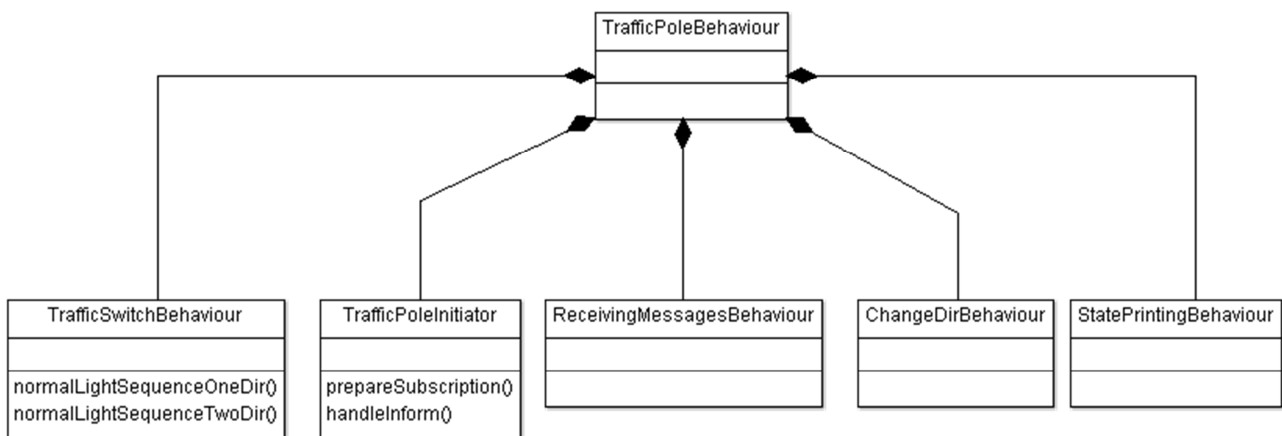


Figura 4 - Struttura *TrafficPoleBehaviour*

Il *TrafficPoleBehaviour* racchiude tutte le funzionalità delle lanterne semaforiche veicolari e pedonali. E' descritto tramite un *ParallelBehaviour* composto dai cinque rispettivi subBehaviour, anche qui espressi sotto forma di inner class, e copre l'intero set di funzionalità che il Behaviour padre comprende.

In primis, il *TrafficSwitchBehaviour* implementa le varie casistiche a cui l'incrocio è soggetto (sez. Analisi/Requisiti). Tramite i metodi *normalLightSequenceOneDir* e *normalLightSequenceTwoDir*, difatti, viene simulato il comportamento di alternanza cromatica di un tradizionale semaforo stradale. Il Behaviour gestisce anche l'avvenuto incidente e le varie modellazioni della prenotazione pedonale nei diversi casi. Il *TrafficPoleInitiator*, tramite il protocollo di comunicazione "Subscription", comunica con la rispettiva telecamera e gestisce gli eventi comunicati rendendo possibile il cambiamento di stato del sistema in modo dinamico (es. da caso 1(b) a caso 2(a)).

In aggiunta è presente la gestione di una coda di veicoli e l'interfacciamento dei semafori con gli altri elementi dell'incrocio (e degli altri incroci del circondario in caso di scale-up).

Per gestire la sincronizzazione e la coordinazione di due semafori di direzione diversa e opposta (i.e NS e WE) i Behaviour *ReceivingMessageBehaviour* e *ChangeDirBehavior* comunicano con il Behaviour principale attraverso un protocollo "topic-based".

4.3 Comunicazione e protocolli

4.3.1 Subscribe Interaction Protocol

Per la comunicazione tra i semafori, le camere e il pulsante pedonale si è deciso di utilizzare uno dei protocolli di interazione messi a disposizione da JADE, il Subscribe protocol. Tramite questo uno o più Behaviour si sottoscrivono alla notifica di alcuni tipi di eventi di un altro comportamento. Per gli eventi della camera questo tipo di protocollo si rivela molto adatto, in quanto permette di notificare l'ascoltatore ogniqualvolta la telecamera rilevi qualcosa.

Il Subscribe protocol è un'implementazione del protocollo FIPA corrispondente definito nell'immagine sottostante.

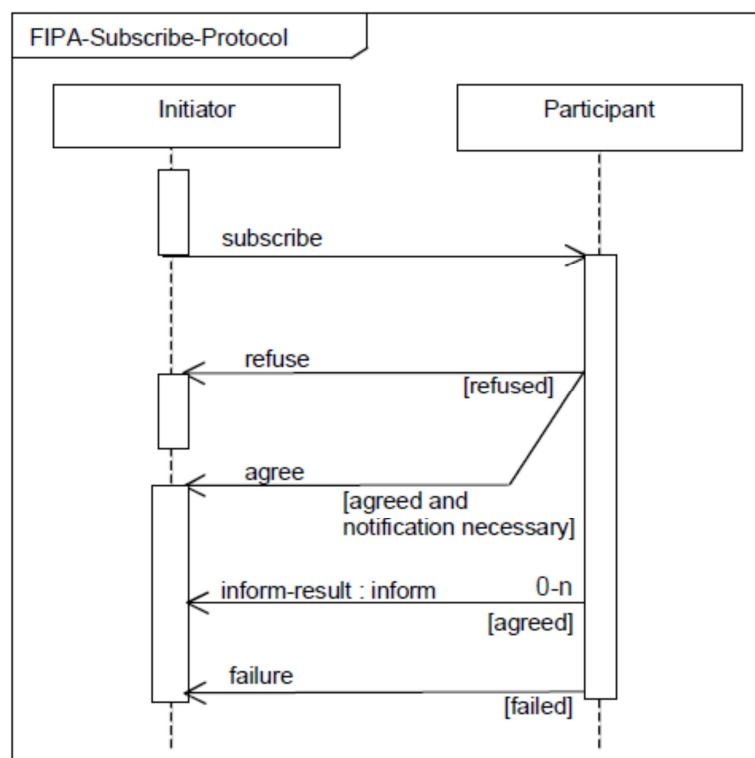


Figura 5 - Subscribe protocol (dalla JADE Programmer's guide)

Come si evince dalla figura 5, l'*Initiator*, cioè colui che si vuole sottoscrivere, manda un messaggio di tipo *subscribe* ad un altro Behaviour, che può rispondere accettando o rifiutando. In caso positivo il *Participant* o *SubscriptionResponder* manda un messaggio di tipo *inform* a tutti i Behaviour sottoscritti ogni volta che viene chiamato il metodo *send*.

Alla ricezione del messaggio sull'*Initiator* viene chiamata la callback *handleInform* che permette di gestire il messaggio.

4.3.2 Topic-based communication

Per il dialogo tra i semafori abbiamo deciso di implementare una comunicazione topic-based, in questo modo possiamo evitare di cercare ogni volta i semafori nell'AMS o nel DF, infatti i Behaviour si possono sottoscrivere a certi topic in modo da ricevere in modo automatico tutti i messaggi che riguardano un determinato argomento.

Per poter attivare questo tipo di comunicazione è stato necessario abilitare in tutti i container il `TopicManagementService`, non attivo di default.

Nel nostro caso specifico i semafori utilizzano due Behaviour con il pattern "receive-block" che attendono su un *MessageTemplate* che specifica il topic.

5. Hardware

Il deploy del progetto è stato fatto su tre RaspberryPi connessi tramite Ethernet ad un router domestico configurato in modo da renderli accessibili dall'esterno.

Uno dei dispositivi funge da master e contiene il *Main-Container*, mentre negli altri due sono presenti i container con all'interno gli agenti dei semafori.

6. Sviluppo

6.1 Testing

Durante l'analisi ci siamo posti immediatamente il problema della gestione dei test, per simulare le immagini raccolte dalle 4 telecamere, abbiamo generato dei file che rappresentavano la "realtà" dell'incrocio dividendo tra i vari casi descritti nell'analisi, in modo da identificare in modo semplice gli errori.

6.2 Divisione dei compiti

Addimando:

Gestione semafori e comunicazione

Antonini:

Gestione telecamere e generazione file di testing

Parte comune:

Analisi, progettazione del sistema, interfacciamento delle parti, testing, refactoring, sincronizzazione.

6.3 Note di sviluppo

Di seguito si riportano alcuni dettagli implementativi considerati rilevanti:

- Durante lo svolgimento del progetto abbiamo avuto a che fare con la gestione di tempistiche particolari, come quelle dei semafori e quelle di deattivazione del bottone pedonale. Alcune di queste sono state gestite costruendo dei *WakerBehaviour* appositi come anonymous inner class, ma nel caso dei semafori, per esempio, siamo stati obbligati ad utilizzare delle *Thread.sleep* poiché in JADE non esistono delle block temporizzate che non si sbloccino alla ricezione dei messaggi. Come si sa, però, gli agenti JADE sono eseguiti all'interno di un unico thread, quindi la *Thread.sleep* avrebbe bloccato tutto l'agente e non solo il Behaviour interessato, per ovviare a questo problema abbiamo utilizzato la classe *ThreadedBehaviourFactory* che permette di wrappare un determinato Behaviour all'interno di un thread dedicato. Questa soluzione, seppur non elegante, è risultata l'unica funzionale, poiché la nostra necessità era proprio quella di bloccare il Behaviour.
- Per ogni agente è presente solo un file che ne rappresenta il comportamento, questo poi contiene la definizione di tutti i sub-behaviour come inner class. Per pulizia ed organizzazione del codice sarebbe stato meglio avere un file per ogni Behaviour e sub-behaviour, questo, però, può portare a problemi di visibilità e alla necessità di utilizzare metodi pubblici sui Behaviour, cosa che avrebbe potuto minare l'autonomia caratteristica degli agenti provocando "l'uscita" dal paradigma agent-oriented.
- Lo *StatePrintingBehaviour* contenuto all'interno del *TrafficPoleBehaviour* è semplicemente un comportamento che permette di visualizzare lo stato del semaforo e rendere possibile la simulazione del sistema.

Ovviamente il progetto è da considerarsi in fase beta e quindi necessita ancora di una fase di refactoring e testing più approfondita.

7. Commenti finali

7.1 Conclusioni e lavori futuri

Lo sviluppo di questo progetto ci ha permesso di approfondire le conoscenze acquisite durante le lezioni del corso riguardo JADE e la programmazione di sistemi multi-agente.

Nel nostro caso il problema di coordinazione era di medio-bassa difficoltà, per cui abbiamo optato per un approccio soggettivo con scambio di messaggi. Tuttavia il codice può risultare poco elegante, perciò, in futuro, potrebbe essere valutata un'integrazione con TuCSoN4JADE in modo da arricchire il progetto dei benefici di un approccio oggettivo.

Rispetto a questa prima versione il team si ritiene abbastanza soddisfatto, nonostante i problemi incontrati e non ancora completamente risolti.

Appendice A - Guida utente

Per avviare la simulazione del progetto bisogna posizionarsi nella cartella del progetto e avviare il *Main-Container* con i servizi di gestione topic e mobilità agenti, come descritto di seguito (sintassi Linux/Mac).

```
java -cp libs/jade.jar:bin/ jade.Boot -gui -services  
"jade.core.messaging.TopicManagementService;jade.core.mobility.AgentMobilityService"
```

Successivamente, in due shell separate, vanno avviati i due container, sempre facendo attenzione ad attivare il servizio di gestione topic e quello di mobilità agenti.

Bisogna specificare, inoltre, che i container possano accettare agenti provenienti dall'esterno.

```
java -cp libs/jade.jar:bin/ jade.Boot -container -host IPHOST -services  
"jade.core.messaging.TopicManagementService;jade.core.mobility.AgentMobilityService" -accept-foreign-  
agents true
```

IPHOST definisce l'indirizzo IP del main container, in caso di prova in locale specificare *localhost*.

Dalla GUI, poi, va avviato il CrossAgent specificando come argomento l'id numerico dell'incrocio.

Nel repository sono già disponibili dei file che permettono l'esecuzione del progetto, in particolare simulano la situazione in cui:

1. Non ci sono veicoli
2. Non ci sono veicoli ma ci sono chiamate pedonali
3. Ci sono veicoli in entrambe le direzioni (con e senza chiamate pedonali)
4. Ci sono veicoli in una sola direzione

Bibliografia

[1] *Slide del corso Sistemi Distribuiti 2016/2017.*

[2] Omicini, Andrea, et al. "Multi-agent infrastructures for objective and subjective coordination." *Applied Artificial Intelligence* (2010).

[3] Mariani, Stefano, Andrea Omicini, and Luca Sangiorgi. "Models of Autonomy and Coordination: Integrating Subjective and Objective Approaches in Agent Development Frameworks." *Intelligent Distributed Computing VIII. Springer International Publishing, 2015.* 69-79.

[4] Bellifemine, F., Caire, G., Trucco, T., & Rimassa, G. *Jade programmer's guide.*

[5] Bellifemine, F., Caire, G., Trucco, T., Rimassa, G., & Mungenast, R. *JADE ADMINISTRATOR'S GUIDE.*