ALMA MATER STUDIORUM
UNIVERSITA' DI BOLOGNA
SEDE DI CESENA
SECONDA FACOLTA' DI INGEGNERIA CON SEDE A CESENA
CORSO DI LAUREA SPECIALISTICA IN ICT

Titolo:

Studio del comportamento di agenti mobili attraverso Jirr e Jade

Elaborato in:
Sistemi Intelligenti Distribuiti

Studente:
Stefano Bromuri
Matricola:
0000232683

Requisiti:

Si richiede di modellare un software di simulazione di mondi virtuali in tre dimensioni utilizzando la metodologia AOP (agent oriented programming) in cui si mettano in evidenza gli aspetti riguardanti la mobilità degli agenti su più piattaforme distribuite. In particolare si vuole focalizzare l'attenzione sull'utilizzo di agenti mobili nel contesto della simulazione virtuale utilizzando come motore grafico Jirr (Java bindings for Irrlicht). Infine
si vogliono mettere in evidenza le peculirità dell'infrastruttura JADE per l'AOP sfruttando questa infrastruttura per creare gli agenti che agiranno nel sistema distribuito.

Analisi dei Requisiti:

I requisiti richiedono di sviluppare attraverso i tool di sviluppo JADE e Jirr, una infrastruttura ad agenti per mondi virtuali distribuiti. Si presuppone quindi che gli agenti abbiano una rappresentazione grafica, e siano in grado di spostarsi in mondi che hanno anche essi una rappresentazione grafica. Nei requisiti non è definito il livello di dinamicità che ci si aspetta di incontrare in questi mondi, ed è lasciata al progettista la decisione riguardo alle capacità di apprendimento degli agenti. Le richieste principale da parte del committente riguardano l'utilizzo di JADE e lo studio della mobilità degli agenti in ambienti distribuiti. L'analisi e il progetto dovranno necessariamente focalizzarsi su questi due aspetti e, nel qual caso rimanesse sufficiente tempo, un aspetto secondario sicuramente rilevante sarà l'intelligenza degli agenti e la loro capacità di interagire con questi mondi virtuali.
Nel progetto non è richiesta quindi solo una implementazione, ma è richiesto anche un lavoro di apprendimento di due nuove tecnologie, che sono Jirr e JADE.
Un requisito non funzionale potrebbe riguardare l'implicazione legale riguardo all'avere agenti mobili che si spostano da una piattaforma a un'altra: per garantire la buona fede di queste entità, potrebbe essere necessario, in fase di sviluppo, includere un sistema di firme digitali che assicuri l'identità dell'agente che si sta trasferendo.

Analisi del Problema:

Dalla analisi dei Requisiti, si evince che il problema principale riguarda la mobilità degli agenti da piattaforma a piattaforma. Questo problema implica sicuramente altri problemi, tipici dei sistemi distribuiti, quali: clonazione non voluta di entità software, serializzazione, perdita di consistenza del o dei sistemi implicati, perdita di informazione.
Volendo avere una rappresentazione virtuale del mondo e degli agenti, un problema potrebbe riguardare i tool con cui si creano i modelli di mondo e le mesh di animazione, in quanto potrebbero non essere facilmente reperibili o non economici. Sicuramente un problema è derivato, riguardo alla grafica, dal motore grafico stesso: questo infatti vincolerà necessariamente il nostro progetto all'utilizzo dei modelli 3D che sono dichiaratamente **supportati da questo.**
Problema di non scarsa rilevanza risulta essere la modalità di comunicazione tra gli agenti in gioco, in particolare il problema riguarda lo scegliere un metodo di comunicazione diretto, in cui è

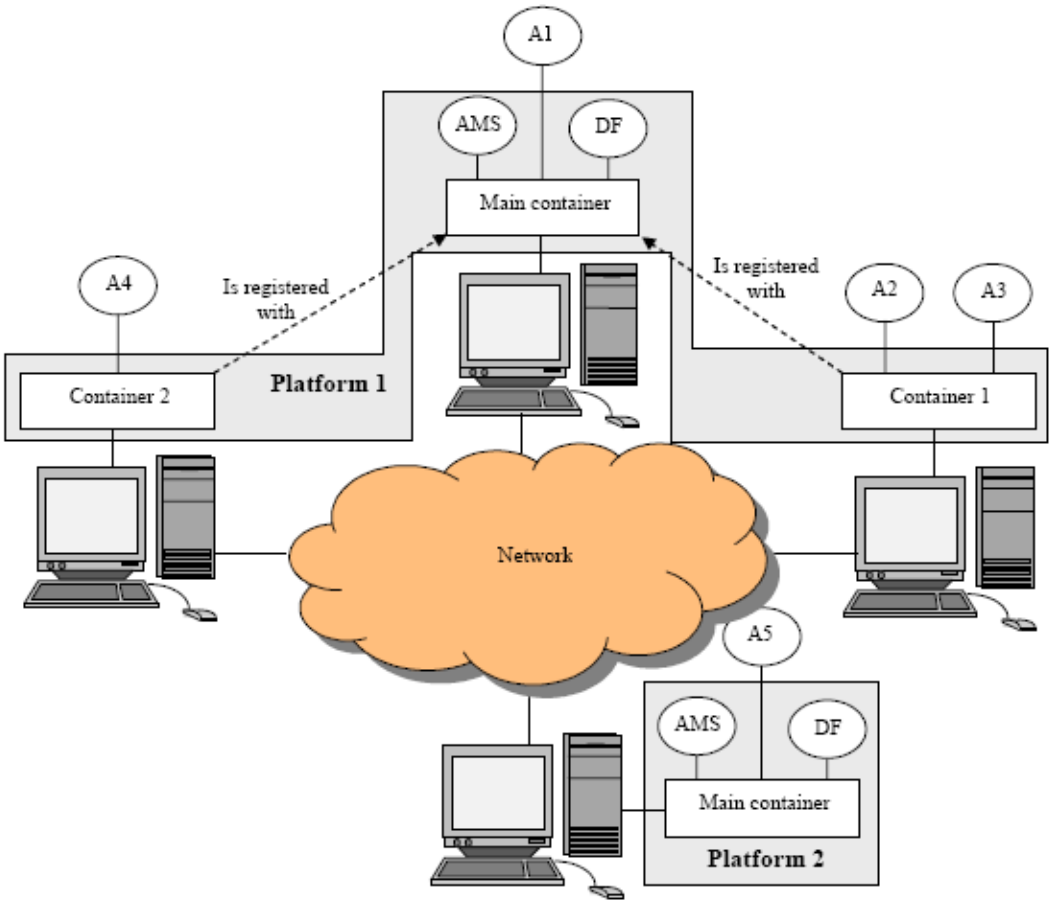
scaricata sugli agenti la responsabilità riguardo alla comunicazione, o un metodo di comunicazione indiretto in cui la responsabilità di mettere in collegamento gli agenti è invece scaricata su un mezzo/artefatto di comunicazione.
Una tematica fondamentale da affrontare riguarderà il comportamento degli agenti: il problema risulta essere, in questo caso, l'intelligenza con cui vengono costruiti, la capacità di adattamento all'ambiente virtuale della simulazione e la creazione di abilità di coordinazione significative alla simulazione.
Infine l'ultimo problema riguarda l'accessibilità del mondo: o meglio il grado in cui è possibile esplorare l'ambiente rispetto alla velocità con cui questo varia nel tempo.

Analisi delle tecnologie:

Jade:

Jade è un middleware che facilita lo sviluppo di sistemi multi-agente. Include:

- Un runtime environment dove gli agenti di Jade possono "vivere" e che deve essere attivo su uno o più host prima che un agente possa essere eseguito sull'host specifico.
- Una libreria di classi che i programmatori possono usare per sviluppare i loro agenti
- Una serie di tool grafici che consentono di monitorare e amministrare le attività degli agenti in esecuzione



Container:
Ogni istanza in esecuzione del Jade runtime environment è chiamata Container dato che può

contenere svariati agenti. Il set di container attivi è chiamato Platform. In una piattaforma deve essere sempre attivo un singolo Main container e tutti gli altri container si registrano presso questo. Conseguentemente il primo container che viene eseguito su un Platform deve essere un Main Container, mentre gli altri container saranno normali e dovranno sapere (conoscendo l'host e la porta) dove si trova il Main Container a cui devono registrarsi.

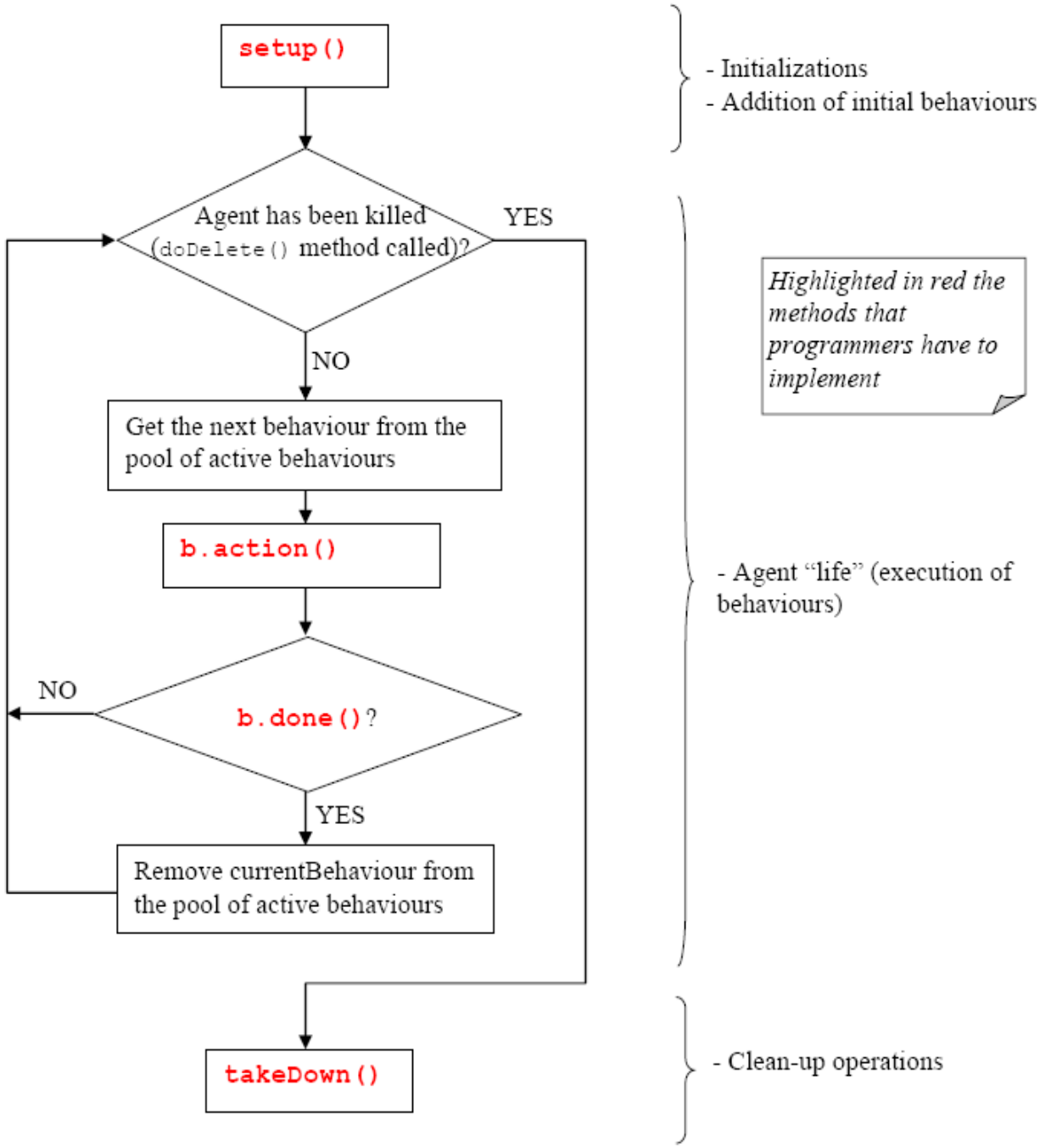
Oltre alla capacità di accetare la registrazione da altri container, un main container differisce dagli altri normal container perchè contiene altri due agenti speciali:

- AMS (Agent Management System) che controlla l'unicità dei nomi degli agenti e rappresenta l'autorità nella platform per quanto riguarda la creazione/cancellazione di agenti nei vari container.
- Il DF (Directory Facilitator) che fornisce un servizio di pagine gialle, o meglio attraverso questo un agente può trovare un altro agente specificando il servizio di cui necessita.

Agenti:

Le azioni che un agente vuole eseguire sono rappresentate come “behaviour”. Un behaviour è un task che un agente può portare a termine.
Un agente può eseguire diversi behaviour concorrentemente. E' importante notare che lo scheduling di un behaviour di un agente non è pre-emptive, ma cooperative. Questo significa che quando un behaviour viene messo in esecuzione, questo ci rimane fino a che non ritorna il controllo al chiamante. Anche se comporta del lavoro addizionale, per certi versi è un vantaggio:

- Permette di avere un singolo Java Thread per agente (questo è importante in ambienti con limitate risorse come i cellulari.
- Fornisce migliori performance dal momento che i behaviour si alternano estremamente più velocemente che gli Thread java.
- Quando un behaviour cambia non implica nessuna informazione salvata sullo stack, questo permette dei servizi avanzati come prendere uno “snapshot” dell'agente e salvarlo in memoria (agent persistency) oppure permette di trasferire l'agente ad un altro container per l'esecuzione remota (agent mobility).

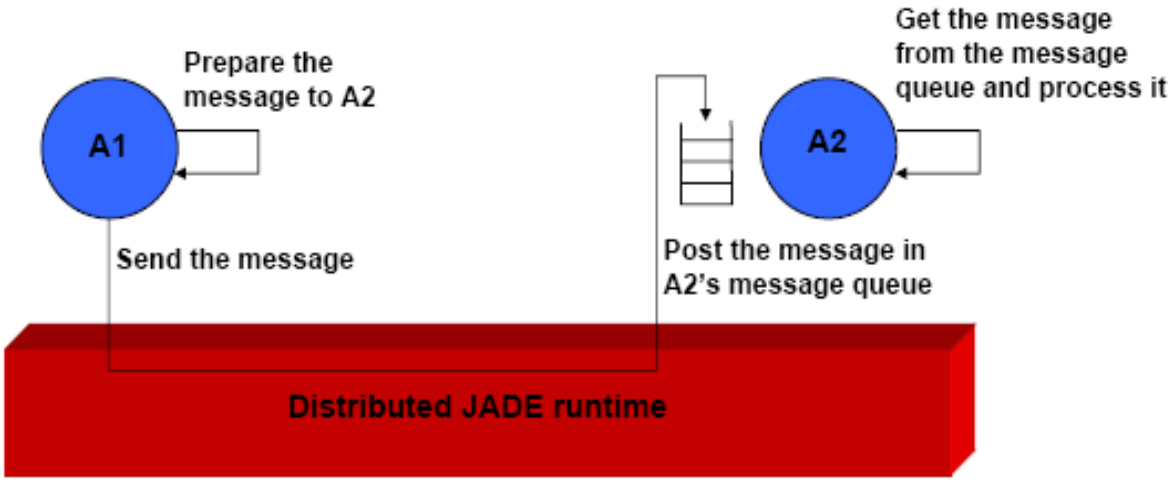


Comunicazione:
Meccanismo: passaggio asincrono di messaggi. Ogni agente ha una coda di messaggi entranti
L' effettiva lettura dei messaggi è ad arbitrio dell'agente. Un agente può:

- leggere il primo messaggio in coda
- leggere il primo messaggio che soddisfa certi requisiti

Un behaviour può essere bloccato in attesa della ricezione di un messaggio: sincronizzazione.

L'invio di messaggi gode della trasparenza della collocazione fisica di mittente e destinatario (garantita dall'ambiente run time distribuito).



Campi di un messaggio FIPA:

- sender
- receiver
- performative
    - request
    - inform
    - query_if
    - cfp
    - propose
    - accept_proposal
    - reject_proposal
- content
- language
- ontology
- conversation_id
- protocol

Mobilità:

Utilizzando JADE, si possono sviluppare agenti mobili, che sono in grado di migrare o copiare se stessi in diversi host nella rete. In questa versione di JADE, solo la mobilità intra-piattaforma è supportata, cioè un agente mobile può navigare attraverso differenti container, ma è confinato a una singola JADE platform.

Spostarsi o clonarsi è considerata una transizione di stato nel normale ciclo di vita di un agente, come tutte le altre operazioni relative al ciclo di vita. La migrazione dell'agente può essere iniziata o dall'agente stesso o dall'AMS.

JADE supporta la "hard mobility" i.e. mobilità di stato e codice.

- Stato: un agente può

i. mettere in stop la propria esecuzione nel container locale
ii. muoversi a un container remoto e
iii. ricominciare la propria escuzione dal punto esatto in cui era stato interrotto.

- Codice: se il codice dell'agente migrante non è disponibile al container di destinazione, è

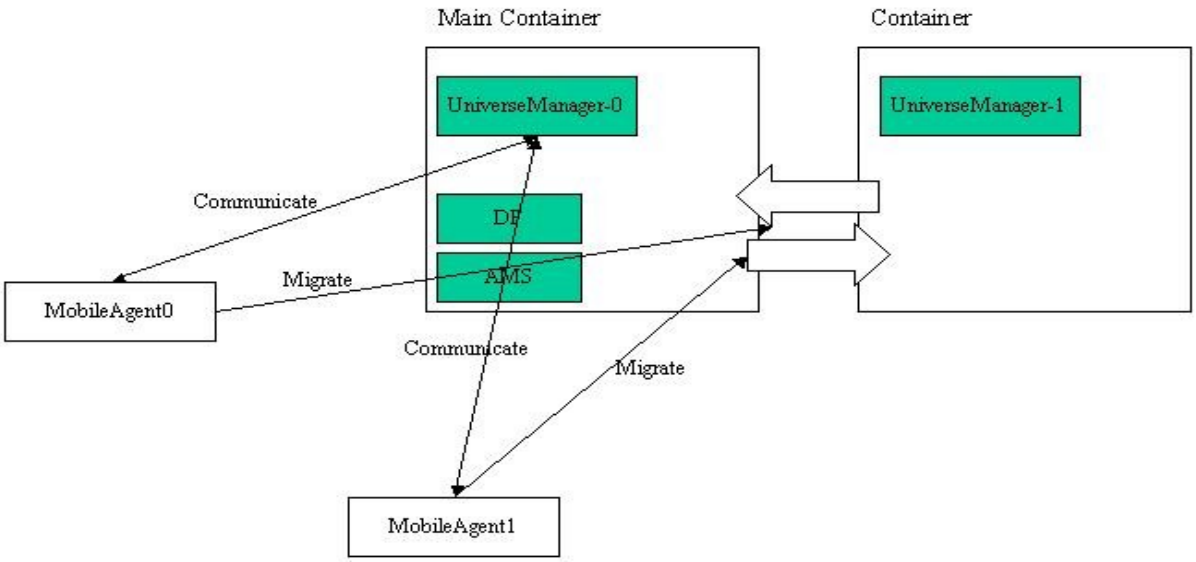
automaticamente fornito.

- Al fine di potersi muovere un agente deve poter essere serializzabile
- La mobilità può essere
  i. iniziata dal metodo doMove() della classe Agent
  ii. forzata dall'AMS

Jirr:

Jirr (Java Irrlicht) è un motore grafico basato su Irrlicht, un motore scritto in c++ che offre varie Features che permettono di astrarre dalle OpenGL e dalle DirectX, librerie che definiscono le primitive base per la rappresentazione grafica di un mondo virtuale. In realtà si era pensato di utilizzare Java3D, purtroppo però non essendo un motore grafico maturo, si è preferito utilizzare un supporto che consentisse una maggior astrazione da aspetti base, come il caricamento di modelli 3D, parte lacunosa del Java3D.

Architettura Logica:



Come si vede dalla figura, si pensa di organizzare il sistema sfruttando la decomposizione in container della piattaforma di Jade. Quindi per ogni container si pensa di inserire un UniverseManager che avrà poi il compito di comunicare con 1-N MobileAgent che agiscono sul mondo. L'interazione è a scambio di messaggi in quanto, per rispettare la stessa natura degli agenti è impensabile di procedere con una interazione a chiamata di procedura anche perchè, in una architettura in cui ci sono componenti mobili attraverso la rete, sarebbe una complicazione notevole pensare le interazioni in tale maniera.
Da notare che nella architettura logica sono presenti anche AMS e DF, agenti forniti già nel Main Container di Jade, che offrono l'importante servizio di ricerca di cui già si è discusso in fase di analisi delle tecnologie. Tramite questo servizio, un agente che migra attraverso la rete, può, una volta raggiunto il Container a cui l'utente ha deciso di spostarlo, o a cui lo stesso agente ha deciso di andare (a seconda di come l'agente è stato definito), conoscere lo UniverseManager operante su quel Container e conseguentemente registrarsi presso esso per l'esplorazione del mondo che lo UniverseManager sta gestendo.

Diagramma delle attività dello UniverseManager:

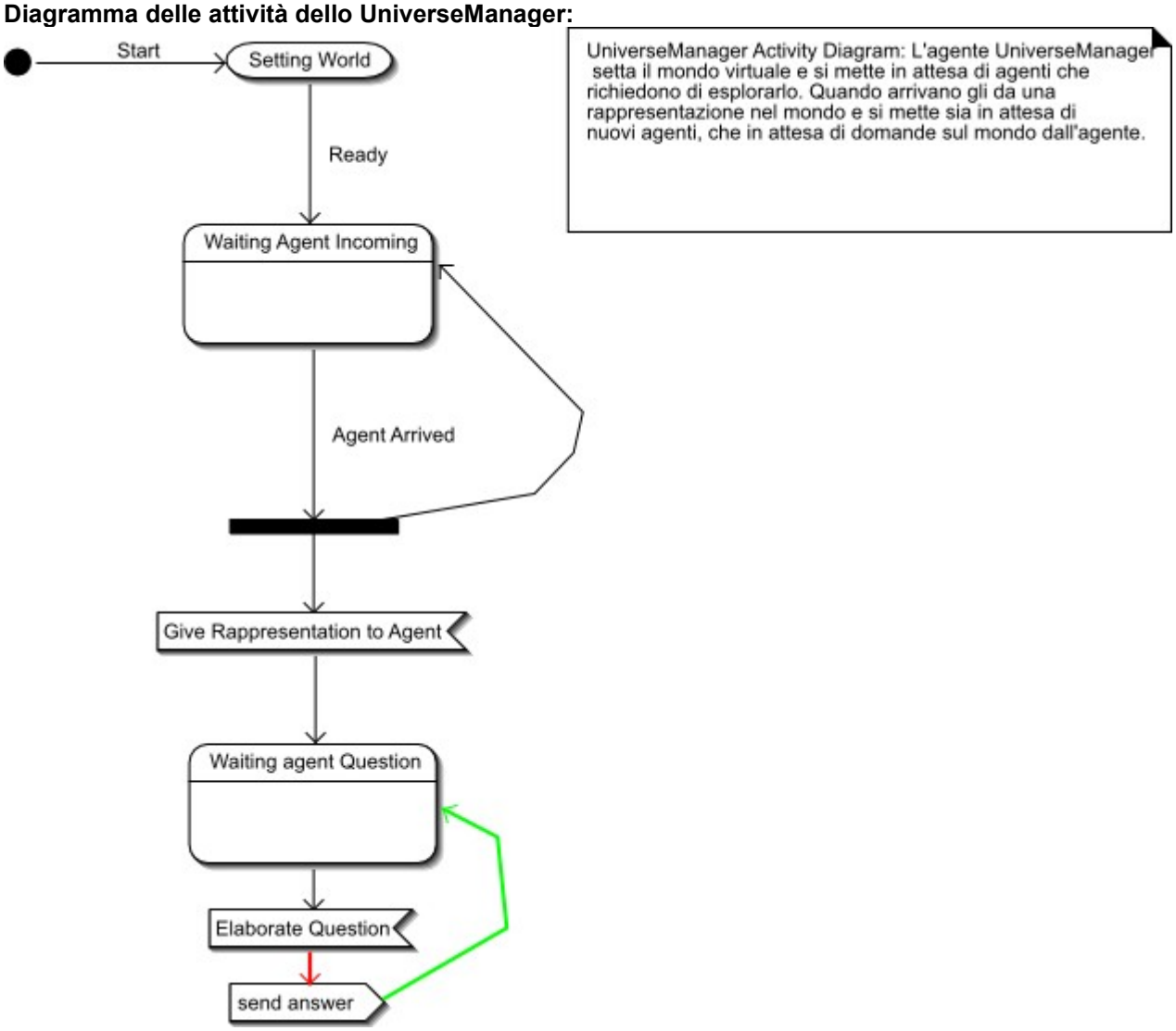
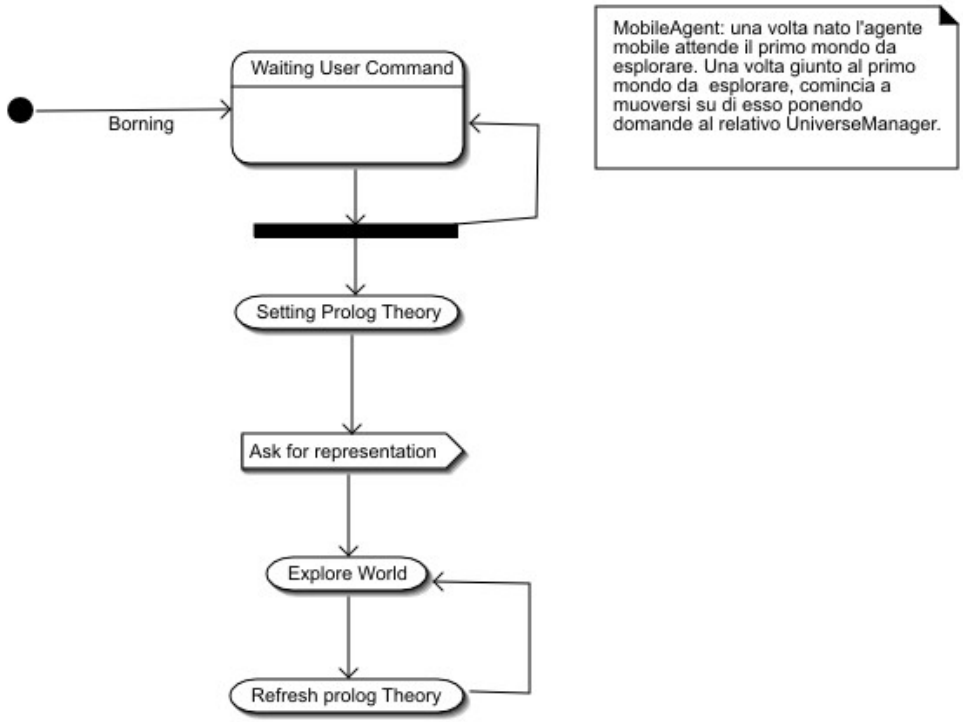


Diagramma delle attività del MobileAgent:



Aspetti non considerati:
Quello che manca in questa prima bozza di architettura logica e che in futuro sarà necessario considerare, nel caso si voglia andare oltre alla semplice sperimentazione, è l'interazione dell'utente con gli UniverseManager, considerare la possibilità di avere più UniverseManager per piattaforma e l'interazione tra i MobileAgent. Questi aspetti sono volutamente trascurati in quanto il fine di questo progetto è dimostrare la possibilità di superare il limite dell'attuale simulazione di mondi virtuali nel Gaming Online attraverso una architettura ad agenti intelligenti e mobili.
C'è un ultimo aspetto che potrebbe, in futuro, valer la pena considerare: normalmente nel Gaming Online i mondi virtuali sono staticamente assegnati a una macchina. Chiaramente allo stato dell'arte per rendere meno pesante la gestione si cerca di dividere in parti il mondo virtuale. Se però 500 persone si trovano sulla stessa parte di mondo, c'è poco da fare la macchina crolla sotto il peso delle richieste degli utenti. Una soluzione a questa situazione potrebbe essere quella di suddividere ulteriormente il mondo in parti, e lasciare la possibilità a queste parti di migrare nel cluster di computer che compone il servizio al fine di distribuire equamente le richieste su tutte le macchine presenti.

Progetto:

Believes Desires Intentions: MobileAgent

Dovendo porre una scelta sul come progettare l'agente che si preoccuperà di esplorare l'ambiente, si è deciso di utilizzare come struttura di base la logica BDI. Dove per "Believes" si intende una teoria logica prolog in cui l'agente immagazzina informazione sotto forma di fatti su cui è in grado di fare valutazioni introspettive. Per "Desires" si intende l'insieme di quei Behaviour di Jade appartenenti all'agente relativi alla esplorazione del mondo e all'interazione diretta con esso. Per "Intentions" si intende il Behavior/Desire attivo in un particolare momento del tempo.
Essendo questa una simulazione e non essendo specificato niente nei requisiti, si è deciso fare la scelta riguardante l'intelligenza artificiale in questo frangente. Lo scopo sarà quindi progettare e implementare l'agente in maniera che abbia tre comportamenti principali:

1)Al comando dell'utente l'agente deve spostarsi da un container di JADE a un altro. Questo punto non riguarda l'architettura BDI in se, ma piuttosto riguarda la possibilità di sfruttare il supporto di JADE alla mobilità.
2)L'agente deve ricercare risorse (rappresentate come alberi dal motore grafico)
3)L'agente deve, una volta raggiunto un numero prefissato di risorse, analizzare e trovare tra le sue conoscenze un'area libera n cui costruire una abitazione (rappresentata dal modello di una casa)

Come già detto il framework JADE offre la possibilità di compiere azioni prima dello spostamento dell'agente da un container a un altro e appena l'agente è giunto in un container. Intanto per implementare lo spostamento si vedrà di riutilizzare il modello già fornito nei codici di esempio di agente mobile con GUI. Nell'esempio l'agente possiede una lista di locazioni disponibili, una lista di locazioni visitate, la possibilità di spostarsi tra queste locazioni e una GUI che comunica con l'agente a scambio di messaggi. Modificando opportunamente questo codice, si può risparmiare tempo per raggiungere prima l'obiettivo.
Se si accede per la prima volta a un container in cui è presente uno UniverseManager che ha pubblicato dei servizi compatibili all'agente mobile tramite un directory facilitator, è opportuno per l'esplorazione del mondo gestito dallo UniverseManager, utilizzare una BaseTheory prolog di questo tipo:

```
in(Msg):- retract(Msg) .
out(Msg):-assert(Msg) .
rd(Msg):-Msg .

c(tile(X,Y,Z),tile(T,J,Z),tile(F,G,Z),tile(N,O,Z),risorse(P)) :- Z is free,  T is X + 1,  J is Y,  F is X,
G is Y + 1, N is X + 1,  O is Y + 1.
risorse(X) :- X > 3.
```

Questa BaseTheory verrà arricchita attraverso l'ExplorerBehaviour e il BuilderBehaviour. E' importante notare che deve esistere una teoria differente per ogni mondo visitato per poter sfruttare l'esperienza su più mondi e non su uno soltanto. Da notare il fatto c(tile(),tile(),tile(),tile()):-…. E il fatto risorse(X):-…. Questi due fatti sono inseriti nella teoria prolog per abilitare ragionamenti semplici sulle risorse possedute dall'agente e sulla conformazione del territorio nel mondo visitato.

TileOntology:

Per seguire le specifiche FIPA è conveniente definire una ben determinata ontologia su cui gli agenti possono comunicare per aggiornare la loro vista/rappresentazione in modo coerente.
Jade offre la possibilità di definire una ontologia per comunicare non solo via stringhe, ma attraverso concetti strutturati.
Definiamo quindi un fatto, Tile(X,Y), che ha lo scopo di rappresentare una casella del mondo virtuale, l'atto di comunicazione di un messaggi Tile è di fatto una "QUERY_IF", cioè l'agente mobile chiede allo UniverseManager lo stato della casella X,Y .
Viceversa lo UniverseManager risponde al MobileAgent attraverso il predicato Explorable(Type,Status,X,Y), dove per Type si intende il tipo di casella (notexplorable, bound,house,tree,nothing) per status si intende un booleano che rappresenta la possibilità di spostarsi o meno su questa casella, e X,Y identificano ovviamente la posizione nello spazio. Quindi è un atto di comunicazione di tipo "INFORM" .
Una volta conosciuto lo stato di una casella, un agente può decidere che azioni compiere: se la casella è di tipo tree il MobileAgent può decidere di raccogliere la risorsa e lo comunica attraverso una azione Action("Take",X,Y) . Oppure se non è presente nessun oggetto nella posizione di cui si chiede informazione, l'agente comunica allo UniverseManager la sua volontà di spostarsi nella casella X,Y attraverso una azione Action("GoTo",X,Y).
Di fatto queste informazioni strutturate altro non sono che classi JAVA: la possibilità di definire informazione strutturata è un passo in avanti rispetto all'utilizzare stringhe su cui dover operare una tokenizzazione. Attraverso questa possibilità con JADE si può ottenere un livello di interazione molto complessa che magari il solo ACL definito da FIPA non forniva, allo stato puramente naturale. Per quanto riguarda la codifica del messaggio in contenuto, pare conveniente sfruttare il codec base: questo codec altro non fa che mappare la serializzazione di un oggetto in una stringa, chi riceve il messaggio sarà poi in grado di riconoscere l'informazione strutturata conoscendone l'ontologia. In realtà un messaggio FIPA compliant richiede anche un linguaggio di riferimento tramite cui interpretare il messaggio. Non volendo considerare questo aspetto, ci si è limitati a utilizzare il codec base definito da JADE nel caso si comunichino oggetti, piuttosto che semplici stringhe. L'aspetto positivo di procedere in questa maniera è l'assoluta immediatezza, l'aspetto negativo si vede nel debugging: utilizzando questo approccio infatti la stampa a video dei messaggi ricevuti da un agente è inutile in quanto appare fondamentalmente del byte code. Utilizzando un codec in codifica e decodifica si potrebbe visualizzare un messaggio strutturato secondo un certo criterio, comprensibile a occhio umano. Questo è sicuramente un aspetto che in successive/possibili revisioni del progetto sarà necessario considerare.

ExplorerBehaviour:

Questo Behaviour riguarda l'esplorazione del mondo offerto dallo UniverseManager nel container in cui l'agente è entrato. Non volendo avventurarsi nella ricerca di percorsi, che aumenterebbero la complessità del progetto, si è deciso di utilizzare una strategia di esplorazione che affronta un mondo diviso in caselle, piuttosto che uno spazio continuo. Questo crea una dipendenza tra le capacità dell'agente e il tipo di mondo che può esplorare, ma affrontare uno spazio continuo crea problemi a livello di esplorabilità e rappresentabilità in concetti dello spazio stesso, sia esso virtuale, sia esso reale.
Viceversa, seppur non si voglia trattare con un ambiente continuo, si vuole che l'agente sia però in grado di affrontare un ambiente non deterministico: ad ogni passo, l'agente aggiorna la propria conoscenza del mondo per adattarsi ai cambiamenti di questo. Affermando questo fatto, si pretende quindi che l'agente, una volta presa una scelta, si assicuri che quella scelta sia realmente attuabile.
Per evitare che lo spostamento dell'agente nel mondo virtuale sia eccessivamente casuale, conviene adottare una politica di esplorazione che dia priorità a caselle del mondo virtuale che non siano mai

state esplorate, mentre le caselle già esplorate vengono ripercorse solo se è una scelta obbligata, cioè tutte le caselle intorno sono già state esplorate. La memorizzazione dell'esperienza relativa alla esplorazione di ogni mondo, viene effettuata tramite teorie prolog in cui le varie caselle sono descritte da fatti prolog:

tile(1,2,free).
tile(3,2,free).
tile(4,5,free).
tile(4,10,bound).

Un'altra dipendenza logica che è opportuno far notare riguarda l'ontologia/e con cui si rappresenta il mondo e l'ontologia/e che l'agente mobile è in grado di comprendere. Come detto nella sezione TileOntology, qualora l'ontologia dell'agente mobile sia meno ricca di quella dello UniverseManager, l'agente mobile avrà capacità limitate di esplorazione sul mondo. Il problema si può risolvere parzialmente tramite il DF: è possibile pubblicare tra i servizi del DF, oltre al tipo di servizio, anche le ontologie su cui si basa un certo servizio, quindi un agente mobile che non è in grado di operare su tali ontologie potrebbe evitare di entrare in quel mondo. Ai fini di questa trattazione, essendo una simulazione che vuole dimostrare la superabilità di un certo limite, si è dato per scontato che l'ambiente sia completamente osservabile dall'agente, anche se non deterministico e dinamico.

BuilderBehavior:

Questo Behaviour si attiva nel caso in cui l'agente sia riuscito a raccogliere un certo numero di risorse, come definito nel fatto presente nelle teoria prolog base, già citata in precedenza, e nel caso in cui sia presente una ben precisa conformazione delle caselle esplorate, questo ragionamento viene eseguito invocando il fatto c(tile(X,Y,Z),tile(T,J,Z),tile(F,G,Z),tile(N,O,Z),risorse(P)) :- Z is free, T is X + 1, J is Y, F is X, G is Y + 1, N is X + 1, O is Y + 1.
Qualora questa invocazione abbia successo, sequenzialmente si vuole che:
L'agente si porti nella casella identificata dalle coordinate X,Y presenti nella stessa invocazione del fatto, che controlli se la conoscenza relativa a tale casella e a quelle considerate nel fatto prolog è corretta e nel caso sia tale proceda a modificare l'ambiente inserendo una costruzione utilizzando le risorse, o altrimenti, nel caso in cui la conoscenza di tali caselle si fosse presentata errata/obsoleta, torni alla modalità di esplorazione dopo aver aggiornato coerentemente la propria conoscenza del mondo.

MobileAgentGui:

La GUI fornita all'utente per spostare l'agente da un container a un altro della piattaforma JADE, è codice già fornito tra i demo di JADE. Non viene di fatto compiuto nessun riadattamento/progetto riguardo a questo pezzo di codice, infatti per i nostri scopi è più che sufficiente.

ServeIncomingMessages:

Anche questo è un Behaviour il cui codice è già stato definito per il MobileAgent originale, quindi di fatto il progetto si limita a modificarlo allo scopo. Quello che si è fatto è stato adattarlo affinché

fosse in grado di ricevere messaggi con un certo ben definito template in relazione alla TileOntology precedentemente definita e causare quindi la reazione dell'agente in base a questi messaggi, come per esempio l'aggiornamento della teoria Prolog, o l'invio dei messaggi Action("GoTo",X,Y), Action("Take",X,Y), Action("Build",X,Y) nel caso si sia ricevuta risposta positiva su una casella di cui l'agente ha richiesto lo stato/tipo di esplorabilità.

GetAvaibleLocationsBehaviour:

Questo Behaviour si preoccupa di fornire la lista di container disponibili per lo spostamento dell'agente. Anche questa classe è di fatto già fornita e non viene riadattata in nessun modo.

UniverseManager:

UniverseManager è l'agente che ha come compito quello di gestire il mondo vituale a cui gli agenti mobili accedono. Il mondo è rappresentato in caselle tramite un file XML ed ha una vista tridimensionale, sempre gestita da UniverseManager. Lo UniverseManager ha il compito di fornire una rappresentazione tridimensionale agli agenti che vogliono accedere al mondo che questo incapsula, e di fornire il supporto all'interazione a questo mondo, scrivendo sui file XML lo stato del mondo, in modo da simulare la persistenza ai cambiamenti, e, scambiando messaggi con i MobileAgent, si preoccupa di tenere aggiornata la vista in modo coerente. La vista del mondo viene modellata come un Behaviour dello UniverseManager, questo perché, seppur possa sembrare un errore di progetto, la sincronizzazione della Vista, che per le particolarità del motore grafico dovrebbe essere modellata come uno Thread con i Behaviour dell'agente stesso è un problema che esula dalla simulazione in se, è un problema che in realtà si può affrontare in un secondo momento. Inoltre, se fosse un sistema commerciale, la stessa vista dovrebbe essere remota (sulla macchina di un presunto utente che cerca di accedere a un mondo per visitarlo) e scambiare messaggi con lo UniverseManager, che localizzato su uno dei server in cui si trovano i mondi del servizio si preoccupa solo con dell'interazione col mondo in termini di foglio XML. La vista localizzata sul server ha comunque senso per mantenere vive le astrazioni: osservando le operazioni in termini di rappresentazioni tridimensionali, il debugging del comportento degli agenti interagenti col mondo è molto più semplice che nei termini tradizionali, sia per quanto riguarda il testing dell'intelligenza artificiale, che per quanto riguarda le variazioni del mondo stesso.

Inoltre, la prima motivazione che mi ha spinto a modellare il mondo come un agente risiede nella spiegazione che da McCarthy quando parla di Intentional System: ascrivere a un sistema complesso intenzionalità è utile quando ci da informazioni sullo stato del sistema. Più specificamente, più un sistema è complesso tanto più è difficile dare una spiegazione meccanicistica. Conseguentemente, spiegazioni di basso livello diventano non praticabili.
Essendo il mondo che consideriamo non episodico, discreto, (potenzialmente solo)parzialmente accessibile e non deterministico, dargli la connotazione di agente, rende più realistica la simulazione stessa.
La seconda motivazione è di carattere tecnologico: l'interazione avviene logicamente a scambio di messaggi, quindi è una comunicazione asincrona agevolata dallo stesso framework JADE, mentre la modellazione a oggetti, a parte la complicazione derivata dalla stesso paradigma della chiamata a procedura , non consente questo grado di libertà oltre a comportare evidenti problemi di accesso alle risorse e sicurezza. E' impensabile consentire a un agente proviene dall'esterno, l'accesso al flusso di controllo di un sistema a cui potenzialmente potrebbero accedere un certo numero di utenti.
In secondo luogo, per definizione l'oggetto stesso è una entità passiva che non varia nel tempo, nel

nostro caso invece abbiamo una entità persistente che accetta potenziali utenti ed agenti mobili ad interagire con il mondo. La scelta quindi non è azzardata, certo è opinabile in quanto il mondo in se non inferisce informazione, non apprende, non ha intenzioni o desideri.
Più che un agente, si potrebbe parlare di un ActiveObject, cioè un ibrido tra i due paradigmi.

MessageReading:

Questo Behaviour si occupa dello scambio di messaggi con gli agenti che visitano il mondo virtuale, come nel caso dell'agente mobile il dominio del discorso è la TileOntology, quindi uno UniverseManager è in grado di comunicare informazioni relative alle caselle che compongono il mondo virtuale ed è in grado di apportare cambiamenti al mondo in base alle azioni comunicati dai MobileAgent. Per esempio Action("Take", X, Y) causa la modifica del file XML che descrive lo stato del mondo, modifica che si propaga immediatamente alla vista sul mondo. Questa modifica avviene in quanto MessageReading mantiene un riferimento a XMLReadingBehaviour e un riferimento XMLWritingBehaviour i quali si occupano fisicamente di cambiare la vista(oggetti caricati col motore grafico) e il file XML .

RenderingBehaviour: è una estensione di ConcurrentBehaviour, definito in JADE, ha come scopo quello di mantenere concorrenti XmlReaingBehaviour e XmlWritingBehaviour .

XmlReadingBehaviour: Legge il file XML rappresentante lo stato del mondo e gestisce gli oggetti grafici presenti nella vista modificandoli in base alle modifiche apportate dinamicamente. E' fisicamente l'entità software che si preoccupa della gestione del motore grafico stesso.

XmlWritingBehaviour: è il Behaviour che si preoccupa di scrivere le modifiche apportate dagli agenti mobili al mondo, sul file XML, ogni volta che viene fatta una modifica che può influenzare la vista stessa.

Questioni Legali:

Jade:

JADE è sotto licenza LGPL:
la Licenza Pubblica Generica Attenuata (LGPL), si applica a specifici pacchetti software, tipicamente librerie, della Free Software Foundation e di altri autori che decidono di usare questa Licenza. Chiunque può usare questa licenza, ma suggeriamo prima di valutare attentamente se questa licenza, piuttosto che la normale Licenza Pubblica Generica, sia la migliore strategia da usare per ogni specifico caso, sulla base delle seguenti spiegazioni.
Quando si parla di software libero (free software), ci si riferisce alla libertà, non al prezzo. Le nostre Licenze Pubbliche Generiche sono progettate per assicurarsi che ciascuno abbia la libertà di distribuire copie del software libero (e farsi pagare per questo, se lo si vuole); che ciascuno riceva il codice sorgente o che, se vuole, possa ottenerlo; che ciascuno possa modificare il programma o usarne delle parti in nuovi programmi liberi; e che ciascuno sappia di poter fare queste cose.
Per proteggere i diritti dell'utente, abbiamo bisogno di imporre restrizioni che vietino ai distributori di negare tali diritti o di chiedere agli utenti di rinunciarvi. Queste restrizioni si traducono in determinate responsabilità a carico di chi distribuisce copie del software o di chi lo modifica.
Ad esempio, chi distribuisce copie di una libreria LGPL, sia gratis sia in cambio di un compenso, deve concedere ai destinatari tutti i diritti che ha ricevuto. Deve anche assicurarsi che i destinatari ricevano o possano ottenere il codice sorgente. Se è stato collegato altro codice alla libreria, deve fornire tutti questi codici ai destinatari, in modo che essi possano ricollegarli alla libreria dopo

averla modificata e ricompilata. E deve mostrar loro queste condizioni della licenza, in modo che essi conoscano i propri diritti.LGPL fa meno per proteggere la libertà dell'utente rispetto alla normale Licenza Pubblica Generica. Essa fornisce inoltre minori vantaggi agli sviluppatori di software libero nella competizione con programmi non liberi. Questi svantaggi sono la ragione per cui usiamo la Licenza Pubblica Generica per molte librerie. Tuttavia, la Licenza Pubblica Generica Attenuata fornisce dei vantaggi per certe circostanze speciali.
Ad esempio, in rare occasioni, può presentarsi la necessità particolare di incoraggiare l'uso più ampio possibile di una determinata libreria, in modo che divenga uno standard de facto. Onde raggiungere quest'obiettivo, i programmi non liberi devono essere in grado di utilizzare la libreria. Un caso più frequente è che la libreria libera svolga lo stesso compito di librerie non libere molto usate.
In questa situazione, ha poco senso limitare la libreria libera al solo software libero, quindi utilizziamo la Licenza Pubblica Generica Attenuata.
In altri casi, il permesso di usare una specifica libreria in programmi non liberi consente a un maggior numero di persone l'uso di un'ampia quantità di programmi liberi. Per esempio, il permesso di utilizzare la libreria C del Progetto GNU in programmi non liberi consente a molte più persone di usare l'intero sistema operativo GNU, come pure della sua variante più comune, il sistema operativo GNU/Linux.
Sebbene la Licenza Pubblica Generica Attenuata tuteli la libertà degli utenti in misura minore, garantisce all'utente di un programma collegato alla Libreria la libertà e i mezzi per eseguire tale programma usando una versione modificata della Libreria.
Seguono i termini e le condizioni precise per la copia, la distribuzione e la modifica. Si faccia molta attenzione alla differenza tra "opera basata sulla libreria" e "opera che usa la libreria". La prima contiene codice derivato dalla libreria, mentre la seconda deve essere combinata con la libreria per poter funzionare.

Jirr/Irrlicht:
Jirr si basa sulla licenza zlib/libpng, è una licenza open source ed è compatibile con la licenza GPL e LGPL. I punti degni di nota sono:

- Non devi dichiarare di aver scritto il software
- Alterazioni del software non devono essere presentate come l'orginali e
- Le restrizioni della licenza non devono essere nascoste.

Conclusioni:

Data la natura stessa del progetto, le conoscenze conseguite portandolo a termine sono poliedriche:

1) Ho appreso aspetti anche approfonditi del frameword JADE .
2) Ho approfondito la conoscenza delle meccaniche riguardanti i motori grafici .
3) Ho appreso come fare un parsing di un file XML attraverso le librerie DOM.
4) Ho appreso l'utilizzo del motore inferenziale TuProlog
5) Ho approfondito la conoscenza dello standard FIPA per gli ACL
6) Ho imparato a integrare più tecnologie
7) Ho sperimentato un semplice esempio di Agente BDI .
8) Ho imparato la differenza tra GPL ed LGPL .

L'esperimento considerato in questo progetto era probabilmente fine a se stesso, ma nonostante questo ritengo di aver arricchito notevolmente le mie conoscenze.