Relazione sul
Progetto di Programmazione In Rete
JAM: Java Agent Middleware
a.a. 2010/2011

Team 28:
Scotto Alberto

1. Introduzione

In questo progetto sono stati messi in pratica i seguenti argomenti/tecnologie di Java:

- RMI
- Thread
- Sincronizzazione mediante semaforo binario (dichiarando il metodo **synchronized**), e mediante semaforo n-ario solo per operazioni di lettura (usando il **ReadWriteLock**)
- Ereditarieta' e binding dinamico
- Gestione delle eccezioni
- GUI (Java Swing)
- Gestione eventi
- Applicazione del pattern Observer

2. I componenti del middleware

2.1 L'ADSL

Componente fondamentale per l'interazione tra gli agenti è l'ADSL, Agent Directory Service Layer, che realizza un livello di indirezione per contattare i vari agenti presenti nell'ambiente.
L'ADSL è implementato attraverso un'interfaccia remota, la classe **ADSL**, e una classe concreta, **ADSLImpl**.
Per ogni ambiente in cui "vive" un gruppo di agenti esiste uno e un solo oggetto ADSL.
Questo oggetto remoto è l'unico ad essere registrato esplicitamente nel registro RMI. Tutti gli altri (gli agenti) sono contattabili tramite l'ADSL.
Il trucco è che non appena viene avviato un agente, questo contatta l'ADSL facendo la lookup presso il registro RMI, e vi si "registra", passandogli lo stub della sua message box (di tipo RemoteMessageBox: è un oggetto remoto!).
In questo modo, quando un agente a1 deve comunicare con un altro agente a2, prima di tutto chiede all'ADSL il riferimento alla message box di a2. L'ADSL, quindi, passa ad a1 lo stub della message box di a2. Il risultato finale è che a1 può comunicare con a2 senza passare per il registro RMI di a2, e piu' in generale senza passare per il registro RMI di ogni agente che vorrà contattare durante la sua vita.

2.2 Gli agenti

Un agente è definito fondamentalmente per mezzo di tre classi: AgentID, JAMAgent e JAMBehaviour.
AgentID serve ad identificare gli agenti. Ogni agente è caratterizzato da un nome e da una categoria.
JAMAgent è la classe che descrive lo stato interno di un agente.
JAMBehaviour, infine, descrive i comportamenti che un agente può avere.
Un agente, quindi, è costituito da più oggetti software:

- un oggetto di tipo AgentID, per identificare univocamente l'agente con un nome e una categoria
- un oggetto di tipo JAMAgent, per descrivere lo stato dell'agente
- uno o più oggetti di tipo JAMBehaviour, tanti quanti sono i comportamenti che lo caratterizzano, che definiscono le azioni che può intraprendere

Inoltre, ogni agente possiede una **MessageBox** (una coda di messaggi), che serve per comunicare con gli altri agenti. Se un agente a1 vuole comunicare con un agente a2, a1 scrive un messaggio nella messagge box di a2; a2, se e quando vuole, può ricevere il messaggio di a1 leggendo dalla sua message box.

3. Le classi, nel dettaglio

Ora vediamo nel dettaglio come sono state implementate le classi più importanti.

3.1 ADSLImpl

Questa implementazione dell'ADSL presenta le seguenti features:

- è possibile eseguire `getRemoteMessageBox` da più agenti contemporaneamente. Infatti, come meccanismo di sincronizzazione, si è scelto di usare il `ReadWriteLock`, che permette a più lettori di leggere in parallelo;
- è in grado di cancellare automaticamente, nel regolare svolgimento delle sue operazioni, le RemoteMessageBox registrate che non sono più attive. Infatti, ogni volta che nell'eseguire un metodo scorre la lista delle RemoteMessageBox registrate (che sono oggetti remoti), verifica che quelle che incontra siano ancora attive.

Per memorizzare le message box remote degli agenti che si registrano, si è scelto di utilizzare una LinkedList di coppie <RemoteMessageBox, PersonalAgentID>.

- LinkedList perché è un insieme non ordinato di elementi, e perché ci si aspetta che le operazioni più frequenti siano le operazioni di accesso *sequenziale* e, contestualmente, di rimozione di elementi "interni" (i.e. non il primo ne' l'ultimo).
- Coppie di <RemoteMessageBox, PersonalAgentID> perché, anche se apparentemente ridondante, il PersonalAgentID è necessario quando l'oggetto remoto RemoteMessageBox memorizzato nella lista non è più attivo, e quindi non e' possibile chiedergli chi e' il relativo agente proprietario mediante `getOwner()`

3.2 ADSLImplSingleton

Questa classe estende le funzionalità di `ADSLImpl`, ponendo l'accento sul fatto che, in ogni istante, può esistere uno e un solo oggetto ADSL. Questo vincolo è implementato applicando il pattern Singleton.
A livello pratico non e' molto utile, per la verità, dato che mi assicura solo che su ogni singola JVM non ci sarà più di un oggetto ADSLImpl, mentre noi vorremmo assicurarci che ci sia un unico oggetto ADSL tra TUTTE le JVM interagenti!

3.3 AgentID

Nell'implementare questa gerarchia di classi, è stato fatto buon uso del binding dinamico.
Ci sono due aspetti di cui discutere.

La prima cosa da far notare è che si vuole confrontare un oggetto di tipo AgentID solamente con un altro oggetto di tipo AgentID.
Quindi si potrebbe semplicemente definire il metodo `public boolean equals(AgentID)`
Questo metodo però non ridefinisce il metodo equals di Object, perciò verrebbe meno il binding dinamico.
Allora sono stati implementati due metodi equals:

- public boolean equals(AgentID): confronta due oggetti di tipo AgentID
- public boolean equals(Object): ridefinisce il metodo equals di Object (ai fini del binding dinamico) ma richiama al suo interno il primo equals

Un altro aspetto, sempre relativo al metodo equals, è che l'uguaglianza tra due AgentID non è commutativa. Questo fatto è dovuto al meccanismo del binding dinamico implementato in Java, che realizza un *single dispatch*. Ovvero, il binding dei metodi è dinamico solo rispetto al parametro implicito this, e non anche rispetto a tutti gli altri parametri. In altre parole, il codice del metodo da

eseguire viene scelto a runtime solo in base al tipo effettivo dell'oggetto su cui viene invocato il metodo, mentre per quanto riguarda i tipi effettivi dei parametri del metodo, essi non vengono considerati nella scelta.
Dunque, il risultato è che, dato id1 di tipo CategoryAgentID e id2 di tipo PersonalAgentID:
id1.equals(id2) ≠ id2.equals(id1)
Infatti:
id1.equals(id2) a runtime verrà legato a CategoryAgentID.equals(AgentID), e quindi l'uguaglianza si baserà solo sulla categoria dei due ID.
id2.equals(id1) a runtime verrà legato a PersonalAgentID.equals(AgentID), e quindi l'uguaglianza si baserà non solo sulla categoria ma anche sul nome dei due ID.
In altre parole, nel primo caso i due ID saranno considerati uguali se e solo se hanno la stessa categoria; nel secondo caso i due ID saranno considerati uguali se e solo se hanno la stessa categoria *e* lo stesso nome.

3.4 JAMAgent

JAMAgent è la classe che permette di gestire l'agente, dall'inizializzazione alla terminazione.

1. con `init()` si inizializza l'agente, registrandolo presso l'ADSL
2. con `start()` si avvia l'esecuzione di tutti i comportamenti in quel momento associati all'agente e mai avviati dall'ultima inizializzazione
3. con `destroy()` si cancella l'agente dall'ADSL e si termina l'esecuzione di tutti i comportamenti associati all'agente

Si è scelto di vincolare la sequenzializzazione in cui possono essere invocati i metodi sopra, imponendo l'ordine mostrato. Se quest'ordine non viene rispettato, viene lanciata una IllegalStateException.
Eventualmente, dopo avere terminato un agente, è possibile riavviarlo, ripetendo la sequenza di operazioni mostrata sopra.

Un oggetto JAMAgent ha una lista di comportamenti associati che possono essere fatti eseguire con `start()`, ognuno su un thread differente.
Una volta inizializzato un agente, è possibile invocare `start()` un numero arbitrario di volte, con l'effetto che ad ogni chiamata verranno avviati solo i comportamenti che sono stati aggiunti dopo l'ultima chiamata a `start()`.
Non è possibile riavviare i comportamenti già avviati, a meno che non si termini l'agente con `destroy()`, e poi lo si riavvii con `init()` e con `start()`.

JAMAgent definisce anche i metodi per gestire la message box dell'agente:
- send: scrive sulla message box dell'agente destinatario
- receive: legge e cancella dalla propria message box il più vecchio messaggio proveniente da un certo agente
- isThereMessage: verifica se è presente un messaggio proveniente da un certo agente

3.5 JAMBehaviour

Un comportamento ha tre stati: avviabile, avviato (aka *done*), in terminazione/terminato.

- è *avviabile* non appena creato, oppure diventa *avviabile* con `reset()`
- da *avviabile* passa ad *avviato* con `JAMAgent.start()`
- da *avviato* passa a *in terminazione* con `done()`
- da *in terminazione* passa a *terminato* quando il thread termina effettivamente l'esecuzione di `action()`

Un comportamento viene effettivamente eseguito solo quando viene avviato da `JAMAgent.start()`. Può essere terminato con `done()`. Può anche essere resettato con `reset()`: in tal modo può essere rieseguito da `JAMAgent.start()`.

Ogni comportamento concreto deve definire i metodi `setup()`, `action()` e `dispose()` nel seguente modo: ogni InterruptedException deve essere rilanciata in forma di JAMBehaviourInterruptedException.

3.6 MessageBoxNoSync

Definisce una message box non sincronizzata.
Poiche' siamo in un contesto di programmazione concorrente, i metodi writeMessage, readMessage e isThereMessage sono stati dichiarati *protected*. Infatti, non essendo sincronizzati, non dovrebbero mai essere richiamati dall'esterno, ma solo dalla sottoclasse MessageBox, che definisce una message box *sincronizzata*, e che quindi è la vera "interfaccia pubblica" di una message box.

Per la coda di messaggi, come struttura dati si e' scelto di usare una LinkedList, in quanto le operazioni che si prevede siano eseguite piu' frequentemente sono operazioni di accesso sequenziale e, contestualmente, di rimozione di elementi "interni" (i.e. non il primo ne' l'ultimo).
Come suggerito in [1]:
"If you frequently add elements to the beginning of the `List` or iterate over the `List` to delete elements from its interior, you should consider using `LinkedList`."

3.7 MessageBox

Definisce una message box *sincronizzata*.
Come meccanismo di sincronizzazione si è scelto quello di default di Java. Per applicarlo è bastato dichiarare i metodi *synchronized*.
Questa scelta, diversa da quella presa in ADSLImpl, è motivata dall'osservazione che gli unici metodi di sola lettura sono i metodi isThereMessage, e si prevede che siano poco usati; i readMessage, piu' usati, non sono di sola lettura perché oltre a leggere, cancellano il messaggio appena letto.

3.8 RemoteMessageBox

Nell'interfaccia remota per le message box si sono specificati solo i seguenti metodi:

- `writeMessage`, con cui un agente puo' mandare un messaggio all'agente proprietario della message box remota
- `getOwner`, con cui un agente puo' conoscere il proprietario della message box remota
- print, che stampa a video l'oggetto message box in forma testuale

I metodi di lettura/cancellatura `readMessage` e `isThereMessage` non sono stati inseriti perche' gli agenti non proprietari della message box che vi accedono da remoto non hanno permessi di lettura/cancellatura, ma solo di scrittura.

4. La gestione delle eccezioni

La parte più difficile di tutto il progetto, forse, è stata quella in cui si sono dovute prendere delle decisioni su come gestire le eccezioni: quali lanciare, quali catturare e gestire *hic et nunc*, quali invece rilanciare in altra forma.

L'idea generale che si è cercato di perseguire è la seguente.
Le classi interne al middleware come JAMAgent, sicuramente non hanno idea di quale sia l'interfaccia del sistema (grafica? a linea di comando?), quindi non possono sapere quale sia il modo migliore per riportare gli errori, e a chi (all'utente? oppure allo sviluppatore tramite email?).

Perciò, nelle classi “interne” le eccezioni vengono semplicemente rilanciate e propagate, lasciando che della gestione se ne occupi chi ha a disposizione maggiori elementi per trattarle, ovvero chi estende le classi JAMAgent e JAMWhileBehaviour / JAMSimpleBehaviour.

Un altro principio ispiratore è quello di astrazione delle eccezioni.
In generale diciamo che:
se abbiamo un sistema (es: il middleware JAM) composto di vari sottosistemi (es: ADSL, JAMAgent, MessageBox, ecc.), e associamo ad ogni sottosistema una specifica eccezione creata ad-hoc (es: JAMADSLException), e, quando si verifica un errore in un certo sottosistema, lanciamo l'eccezione associata a quel sottosistema:
allora, chi a valle vede quell'eccezione, può facilmente dedurre quale sottosistema è stato coinvolto dall’errore.

Infine, si è cercato di prestare attenzione alle precondizioni di ogni metodo, lanciando una opportuna eccezione se le precondizioni non sono soddisfatte (es: IllegalArgumentException, IllegalStateException, ecc.).
Ad esempio, una tipica precondizione vuole che i parametri attuali di tipo riferimento siano diversi da null. Di conseguenza, in generale, se in un metodo almeno uno dei parametri attuali di tipo riferimento è uguale a null, viene lanciata una IllegalArgumentException.

Un’ultima osservazione da fare e’ che laddove si e’ dovuto rilanciare un’eccezione, l’eccezione originaria e’ stata comunque propagata insieme alla nuova, passandola come argomento al costruttore della nuova eccezione.

Ora vediamo nello specifico alcuni esempi significativi di come sono state gestite le eccezioni nelle varie classi del middleware.

4.1 JAMAgent

4.1.1 InterruptedException

A differenza di quanto riportato nelle specifiche del progetto, si è scelto di far lanciare a `send` e a `receive` la InterruptedException invece della JAMBehaviourInterruptedException. Questo perché i metodi `send` e `receive` non devono per forza essere invocati dall'interno di un JAMBehaviour (anche se l'uso comune dovrebbe essere quello), e quindi JAMBehaviourInterruptedException non è adatta *in generale*.
Si è perciò lasciato agli implementatori delle sottoclassi di JAMBehaviour, che definiranno i metodi `action() setup() e dispose()`, il compito di catturare la InterruptedException proveniente da `send` e da `receive`, e di rilanciarla in forma di JAMBehaviourInterruptedException.

4.1.2 JAMADSLException

4.1.2.1 RemoteException → JAMADSLException

Come suggerito dal codice del TestFinaleI, le RemoteException derivanti dalle chiamate remote all’oggetto ADSL, sono state rilanciate in forma di JAMADSLException (principio di **astrazione** delle eccezioni). Questo fa sì che per gli sviluppatori che utilizzeranno il middleware non sia necessario sapere che cosa è una RemoteException, nè che JAM usa RMI come meccanismo di programmazione distribuita.

4.1.2.2 Nella init()

Come suggerito dal TestFinaleI, le eccezioni derivanti dalla lookup del registro RMI (NotBoundException, MalformedURLException, RemoteException) vengono rilanciate come JAMADSLException, in quanto hanno propriamente a che fare con l'ADSL (anche se in verità, il fatto che l'URL del registro RMI dell'ADSL non sia sintatticamente corretto non implica che il "sottosistema ADSL abbia fallito"...).

4.1.3 JAMRemoteMessageBoxException

E' stata aggiunta l'eccezione JAMRemoteMessageBoxException, che indica un problema relativo ad una RemoteMessageBox.

4.1.3.1 Nel costruttore di JAMAgent

Il costruttore di JAMAgent, invece di rilanciare la RemoteException proveniente dalla creazione della MessageBox in forma di JAMADSLException, come suggerito dal TestFinaleI, la rilancia in forma di JAMRemoteMessageBoxException. Infatti la creazione della message box dell'agente non ha niente a che fare con l'ADSL.

4.1.3.2 Nella send

La send lancia una JAMRemoteMessageBoxException se si verifica un errore nello scrivere un messaggio su una message box remota. Il che può essere dovuto o al fatto che nessuno degli agenti specificati come destinatari è attivo (e quindi ADSL.getRemoteMessageBox ha restituito una lista vuota), oppure al fatto che è fallita la chiamata remota `rmb.writeMessage(msg)`.

4.2 JAMXXXBehaviour e sue sottoclassi concrete

4.2.1 JAMBehaviourInterruptedException

4.2.1.1 run()

Nel metodo run(), una JAMBehaviourInterruptedException non la considero un errore tale da dover terminare l'esecuzione dell'agente solo se l'utente ha *voluto* terminare l'agente, ovvero solo se e' lanciata da action() e se isDone() restituisce true.
In tutti gli altri casi, termino l'esecuzione del metodo.

4.2.1.2 action(), dispose(), setup()

Avendo modificato la gestione delle InterruptedException in JAMAgent, è stato necessario aggiungere nei metodi action() dispose() setup() la catch per la InterruptedException proveniente da JAMAgent.send, JAMAgent.receive e JAMAgent.sleep, e rilanciarla in forma di JAMBehaviourInterruptedException.

4.2.2 JAMRemoteMessageBoxException

Avendo aggiunto in JAMAgent.send la throw di JAMRemoteMessageBoxException, e' stato necessario aggiungere in action() la catch corrispondente.

4.3 MessageBox

4.3.1 JAMMessageBoxException

Nella classe MessageBox è stato fatto un uso intelligente di cattura delle eccezioni.
Infatti, nei metodi readMessage e writeMessage si sono usate come condizioni di wait le JAMMessageBoxException lanciate dalle readMessage e writeMessage di MessageBoxNoSync.

Questo ha permesso di risparmiare la chiamata a `isThereMessage` ogni volta che si deve testare la condizione di wait, il che avrebbe comportato uno scorrimento extra di parte (o tutta, nel peggiore dei casi) della coda di messaggi.

4.4 ADSLImpl

4.4.1 IllegalArgumentException

Nel metodo `insertRemoteMessageBox` e' definita come precondizione il fatto che l'oggetto remoto messageBox, argomento del metodo, che si vuole registrare presso l'ADSL, deve essere attivo.
Percio', se la precondizione non e' soddisfatta, viene lanciata una IllegalArgumentException e l'esecuzione del metodo viene cosi' terminata immediatamente.

5. Uso di assert

Oltre alle eccezioni, per gestire le condizioni di errore e' stato anche usato il costrutto **assert**, dove opportuno.
Assert permette di specificare una condizione necessaria che lo sviluppatore ritiene si debba verificare in un determinato momento. Qualora tale condizione non è verificata, il programma viene terminato.
Ad esempio, in ADSL.getRemoteMsgBox controllo con una assert che, nel caso in cui sto cercando la casella di un PersonalAgentID, la lista da restituire contenga 0 o 1 elemento

6. L'esempio delle aste

In questo esempio, vi sono due tipi di agenti:

- Gli **agenti banditori**, che gestiscono le aste
- Gli **agenti clienti**, che partecipano alle aste

Ogni cliente può partecipare ad un numero di aste arbitrario.
Ogni banditore può bandire una sola asta (vedi vincolo 6.2.1).
Ogni asta ha una certa durata, che è stabilita quando viene creato il comportamento del banditore.

6.1 Le classi

Sono state definite 8 classi, quattro per ogni tipo di agente (più due classi con il main in cui vengono create le GUI, una del banditore e una del cliente):

- Per l'agente banditore:
  - AgenteBanditore
  - BanditoreAgentID
  - BanditoreBehaviour
  - GestisciAstaBehaviour
- Per l'agente cliente:
  - AgenteCliente
  - ClienteAgentID
  - ClienteBehaviour
  - FaiOffertaBehaviour

AgenteBanditore e AgenteCliente sono le classi che definiscono lo stato dell'agente (estendono JAMAgent).

BanditoreAgentID e ClienteAgentID specializzano PersonalAgentID, specificando come categoria "Asta Banditore" e "AstaCliente", rispettivamente.

BanditoreBehaviour e ClienteBehaviour sono le classi che permettono di astrarre le funzionalità, rispettivamente, delle classi che definiscono i comportamenti del banditore e delle classi che definiscono i comportamenti del cliente.

GestisciAstaBehaviour e FaiOffertaBehaviour sono le classi che definiscono concretamente i comportamenti del banditore e del cliente, rispettivamente.

6.2 Vincoli

6.2.1 Vincoli sul banditore

Ogni agente banditore puo' bandire una sola asta; detto in altri termini, ogni agente banditore puo' avere al massimo un solo comportamento.
Questo vincolo è dettato dal fatto che l'agente banditore ha un'unica coda di messaggi e non ha una modalità di sola lettura dei messaggi che gli consenta di non cancellare i messaggi che legge. Di conseguenza, non è in grado di discriminare tra le offerte che arrivano per un'asta piuttosto che per un'altra.
Grazie alle "interfacce" (in realtà sono classi astratte) BanditoreBehaviour e ClienteBehaviour e' possibile implementare il vincolo ridefinendo il metodo addBehaviour di JAMAgent, inserendo un controllo sul numero di comportamenti associati all'agente, e impedendo l'esecuzione del metodo se esiste già un comportamento associato.

6.2.2 Vincoli di implementazione sui comportamenti associabili agli agenti

Ogni agente banditore ed ogni agente cliente deve essere associato a comportamenti specifici per ognuno di essi.
Ancora grazie alle "interfacce" e' possibile implementare il vincolo ridefinendo il metodo addBehaviour di JAMAgent, e inserendo il controllo a run-time sul tipo del comportamento che si cerca di aggiungere, assicurandosi che sia di tipo BanditoreBehaviour per il banditore, e ClienteBehaviour per il cliente.

6.2.3 Vincoli "di contratto"

Il banditore deve essere attivo e raggiungibile durante l'intera durata dell'asta, per accettare tutte le offerte che vengono effettuate dai clienti. Questo per far si' che nessuno dei clienti che partecipano ad un'asta venga penalizzato, ma abbiano tutti le stesse possibilita' di rilanciare l'offerta e concorrere cosi' all'aggiudicazione dell'oggetto messo all'asta.
Per implementare questo vincolo, e' stato ridefinita la destroy di JAMAgent in modo che abbia effetto se e solo se l'asta è terminata; altrimenti viene lanciata un'eccezione.

6.3 Problemi aperti

In genere, la comunicazione tra un cliente (C) e un banditore (B) avviene così:
C->B (domanda)
B->C (risposta)
C->B (domanda)
B->C (risposta)
[...]
C->B (domanda)
B->C (risposta)
B->C (inform esito)
Ora, se l'ultima risposta del banditore non viene ricevuta dal cliente, mentre l'invio della inform con l'esito dell'asta va a buon fine, si crea una situazione problematica. Infatti, il cliente leggerebbe l'esito quando è ancora in action(), e non sarebbe in grado di interpretare correttamente il

messaggio. Non solo: quando poi l'agente cliente entrasse nella dispose, attenderebbe invano la ricezione del messaggio con l'esito dell'asta, e bisognerebbe forzare la terminazione inviando un interrupt al thread che esegue il comportamento.
La soluzione potrebbe essere quella di aggiungere un metodo di sola lettura in MessageBox, alternativo a readMessage, che consenta di leggere il contenuto di un messaggio senza per forza cancellarlo dalla coda.

7. Possibili estensioni

7.1 Permettere ad ogni agente di avere più di una message box

Applicandolo all'esempio delle aste, questo permetterebbe di rimuovere il vincolo sul numero di aste che un agente banditore è in grado di gestire.
Infatti, disponendo di una coda di messaggi per ogni GestisciAstaBehaviour, basterebbe assegnare una coda diversa ad ogni comportamento affinché le offerte fatte ad uno stesso banditore vengano consegnate al comportamento giusto.

7.2 Aggiungere in MessageBox la possibilità di leggere un messaggio senza cancellarlo dalla coda di messaggi

Applicandolo all'esempio delle aste, questo permetterebbe di risolvere il problema individuato al punto 6.3, legato al fallimento di una receive.

8. Bibliografia

[1] http://download.oracle.com/javase/tutorial/collections/implementations/list.html