A.G.A.T.A.

Automatic **G**roup-buy **A**nd **T**rade **A**gents

Relazione sull'analisi, progettazione ed ingegnerizzazione di AGATA nell'ambito del corso di Sistemi Multiagenti

Realizzazione: **Domeniconi Luca**

Relatori: **Prof. Ricci Alessandro**
Santi Andrea

Indice generale

1. Introduzione

In questa trattazione prenderemo in considerazione il problema di costruire, partendo dalle sole specifiche, un sistema complesso come A.G.A.T.A che presenta alcuni comportamenti “intelligenti” e caratteristiche come proattività, reattività, abilità sociali, autonomia, adattamento, decentralizzazione del controllo e concorrenza, tipiche dei sistemi multi agenti (MAS).

Vedremo anche come sia possibile passare in maniera rapida da un modello del dominio, alla progettazione, fino ad arrivare all'implementazione di un prototipo funzionante in relativamente breve tempo, utilizzando l'astrazione ad Agenti ed Artefatti e mettendo in gioco tecnologie come Jason e Cartago (JaCa).

Per quanto riguarda le fasi di analisi e progettazione, vista la non eccessiva complessità (relativamente alla media dei MAS in generale) del prototipo di A.G.A.T.A preferiremo per quanto possibile non utilizzare metodologie sicuramente interessanti come SODA e Prometheus, ma ci concentreremo più su un approccio AI-based (Artificial Intelligence).

Verrà fatto anche un breve excursus, nel successivo capitolo, per permettere di comprendere al meglio l'argomento trattato ed il motivo per cui si sia scelto di affrontare il problema della costruzione di questo sistema complesso attraverso l'astrazione ad agenti ed artefatti e l'uso di tecnologie relativamente nuove come Jason e Cartago.

2. I Sistemi multi agente (MAS)

2.1 Introduzione

Oggigiorno gli scenari e le complessità introdotte dalle nuove applicazioni vanno ben al di là delle capacità dei modelli di programmazione utilizzati come riferimento per lo sviluppo di applicazioni (paridgmi object-oriented e component-based). Infatti troviamo molto spesso sitemi:

- Complessi ed organizzati sulla base di una struttura gerarchica che può cambiare in maniera dinamica e imprevedibile
- Contenuti praticamente in ogni oggetto che utilizziamo nella vita di tutti i giorni: nei nostri cellulari, nell'ambiente che ci circonda, nei nostri vestiti e addirittura anche all'interno del nostro corpo
- Sempre collegati alla rete internet, o alle reti applicative di interesse, grazie alla tecnologia wireless che renderà le interconnessioni totalmente pervasive.
- Sempre attivi e pronti a svolgere task e computazioni inerenti ai nostri interessi e alle nostre attività.

In particolare è possibile identificare quattro principali caratteristiche che differenziano i computing system di nuova generazione da quelli tradizionali:

- **Situatedness**: i nuovi computing systems sono situati all'interno di un dato ambiente (environment). Tali sistemi possiedo no cioè una esplicita nozione dell'ambiente in cui sono colloca- ti. I computing systems e l'ambiente sono fortemente legati e in grado di

influenzarsi reciprocamente: eventuali cambiamenti nel computing system sono in grado di modificare l'ambiente e viceversa.

- **Openness**: i nuovi computing systems sono sistemi aperti al cambiamento. Cambiamento inteso sia in termini di modifiche relative alla struttura e all'organizzazione interna del sistema, sia in termini del numero di entità computazionali presenti in quest'ultimo.

- **Località del controllo (locality in control)**: le entità software che caratterizzano il computing system sono entità autonome, proattive e che incapsulano il proprio flusso di controllo.

- **Località nelle interazioni (locality in interactions)**: le interazioni tra le varie entità del computing system sono prettamente, o comunque nella maggior parte, locali.

2.2 Gli agenti

I ricercatori che si occupano di sistemi multi-agente (MAS, Multi-Agent System) sono convinti che quest'ultimi possano rappresentare la risposta alle problematiche introdotte dalla corrente crisi del software. Infatti un sistema multi-agente è caratterizzato da entità la cui definizione si fonda sulle quattro proprietà alla base dei nuovi computing system introdotte nel paragrafo precedente. Queste entità prendono il nome di agneti e sono caratterizzati dalle seguenti proprietà:

- **Situatedness**: gli agenti sono entità situate all'interno di un certo environment. Un agente, per mezzo di opportuni sensori, è in grado di osservare e percepire informazioni dall'ambiente in cui è situato, ed è anche in grado di modificare tale ambiente eseguendo azioni per mezzo di una serie di opportuni attuatori.

- **Autonomia**: gli agenti sono entità autonome che incapsulano il proprio flusso di controllo. Ciò significa che l'agente è in grado di compiere decisioni in maniera autonoma e indipendente, quindi senza l'ausilio di alcun intervento esterno.

- **Pro-attività**: gli agenti sono entità pro-attive, ovvero entità che si adoperano in maniera opportuna, scegliendo le adeguate azioni da eseguire e/o particolari percorsi da intraprendere, al fine di raggiungere un dato obiettivo. È importante notare che grazie alla loro natura proattiva gli agenti sono in grado di adattarsi e di modificare il proprio comportamento a fronte di eventuali cambiamenti dello scenario applicativo e/o dell'ambiente.

- **Reattività**: gli agenti sono entità in grado di rispondere in maniera rapida (reattiva) ai cambiamenti dell'ambiente in cui si trovano situati.

- **Capacità sociali (social ability)**: gli agenti sono entità in grado di cooperare e di coordinarsi opportunamente al fine di portare a termine obiettivi e goals sia personali che sociali.

Un sistema multi-agente non è altro che un sistema composto da uno o più agenti situati in un dato ambiente. Solitamente son ben rari i MAS con un solo agente: nei casi più comuni infatti il MAS è caratterizzato da una moltitudine di agenti che operano comunicando e cooperando al fine di portare

a termine un insieme di obiettivi/goals che possono anche essere contrastanti.
Il primo importante beneficio derivante dall'introduzione di tale paradigma riguarda il fatto di fornire agli sviluppatori un potente mezzo con cui andare a modellare e strutturare la business logic di un sistema complesso. Infatti, si è soliti organizzare sistemi complessi sulla base di opportuni sottosistemi, ognuno con un proprio scopo/obiettivo, che lavorano in cooperazione al fine di portare a termine le funzionalità richieste dal sistema complessivo. Ciascun sottosistema può essere realizzato come una data organizzazione di agenti che vengono programmati al fine di portare a termine lo scopo/l'obiettivo del sottosistema.

Grazie alla duplice natura degli agenti (reattiva e pro-attiva) gli sviluppatori hanno la facoltà di costruire applicazioni che si adoperano al fine di portare a termine gli obiettivi per cui sono state realizzate (componente pro-attiva) reagendo e tenendo conto di eventuali cambiamenti e modifiche dell'ambiente (componente reattiva).

L'ultimo beneficio che si intende mettere in evidenza riguarda l'importanza dell'introduzione dell'astrazione di agente: tale astrazione è in grado di minimizzare il gap concettuale tra le entità che vengono utilizzate in fase di analisi e di design di un sistema, e le entità di prima classe e i costrutti che vengono realmente utilizzati per implementarlo.

2.3 Jason

In questo capitolo verrà presentato Jason, cioè la piattaforma che sarà utilizzata come riferimento per la programmazione degli agenti, all'interno dell'approccio agent-based per lo sviluppo di applicazioni complesse oggetto di questa trattazione. Jason è un interprete di una versione estesa di AgentSpeak, un linguaggio di programmazione ad agenti astratto, basato sul modello **BDI (Belief-Desire-Intention** ,vedi paragrafo successivo), che può essere considerato come una estensione della programmazione logica. Jason inoltre è sviluppato in Java e distribuito con licenza open-source. È importante far notare che oltre a Jason esistono diverse altre piattaforme e linguaggi basati sul modello di agente BDI: Jadex, 2APL, ecc. ecc.

2.4 Il modello di un agente BDI

Il modello BDI (Belief-Desire-Intention) trae ispirazione da studi relativi al comportamento umano compiuti da un team di psicologi. L'applicazione di tali studi alla ricerca relativa agli agenti si deve allo Stanford Research Institute che, nella seconda metà degli anni ottanta, ha dato vita al modello architetturale di agente BDI. L'architettura BDI è ad oggi l'architettura di riferimento per lo sviluppo di agenti razionali/intelligenti.
Secondo il modello BDI, un agente possiede quello che può essere definito come una sorta di stato mentale che può essere modellato in termini di credenze (beliefs), desideri (desires) e intenzioni (intentions). Tali astrazioni possono essere definite come segue:

- I **Beliefs** rappresentano le informazioni e le conoscenze che l'agente possiede sull'ambiente e sugli altri agenti appartenenti al MAS.

- I **Desires** rappresentano, traducendo letteralmente dall'inglese, i desideri dell'agente, ovvero quell'insieme di stati che l'agente vorrebbe raggiungere. Il fatto che un agente possegga un dato desiderio non implica necessariamente che quest'ultimo si debba adoperare per realizzarlo. Un desiderio infatti ha solo una potenziale influenza sul comportamento dell'agente BDI. In termini informali è possibile considerare i desideri come una sorta di possibili opzioni a disposizione di un agente: è perfettamente lecito quindi che un agente possegga due desideri mutuamente incompatibili tra loro.

- Le **Intentions** rappresentano quell'insieme di state of affairs che l'agente ha deciso di realizzare o di portare a termine e a cui sono associati un insieme di opportuni piani che contengono la lista di azioni astratte (course of actions) che porteranno l'agente al raggiungimento di tali intenzioni. Diversamente dai desires le intentions hanno una influenza certa sul comportamento dell'a- gente che è obbligato ad adoperarsi per il loro raggiungimento. Le intenzioni possono essere o goal che l'agente è chiamato a perseguire o perché è stato creato proprio a tale fine o perché tali goal sono emersi durante l'esecuzione delle sue attività.

In psicologia si usa una teoria, detta *intentional stance*, per definire un livello di astrazione con cui descrivere un sistema il cui comportamento può essere espresso e predetto in termini di beliefs, desires e intentions. Tale approccio trova una diretta applicazione nella vita di tutti i giorni dove utilizziamo senza accorgercene la la `psicologia del buon senso' (folk psychology) per descrivere il comportamento degli esseri umani.
Utilizzare l'intentional stance come uno strumento di astrazione con cui modellare un sistema sulla base di beliefs, desires e intentions, dà la possibilità di descrivere e predirre il comportamento di quest'ultimo senza la necessità di doversi focalizzare sul suo effettivo funzionamento.
Nel campo dell'IT da sempre si è alla ricerca delle giuste astrazioni che consentano agli sviluppatori di gestire e modellare le complessità dei sistemi da realizzare con maggior facilità (si pensi all'introduzione delle procedure, degli abstract data type, degli oggetti ecc. ecc.). È stata proprio la ricerca di nuove astrazioni a portare alla definizione e allo sviluppo del modello di agente BDI. I ricercatori infatti, costruendo questo modello di agente, hanno cercato di fornire agli sviluppatori un nuovo insieme di astrazioni, molto più vicine al modo di pensare degli esseri umani, su cui basare l'ingegnerizzazione delle proprie applicazioni software.

4. Introduzione ad AGATA

4.1 Cenni sui Group Buy

Un group buy non è altro che un gruppo di persone interessate all'acquisto dello stesso oggetto, che si uniscono in modo tale da ricevere uno sconto sul prezzo finale. Esso nasce dalla possibilità di produrre in serie un oggetto, tagliandone i costi di produzione su grandi quantitativi.

4.2 Un piccolo esempio

Se volessi acquistare uno scarico in titanio per la mia nuova moto da corsa, potrei contattare una ditta specializzata, che per un lavoro singolo e personalizzato potrebbe richiedermi la cifra di 300€.
Se invece ne volessi acquistare 10, a causa della possibilità di risparmiare su produzioni in serie (spedizione di materiali, lavoro in catena di montaggio,...) la stessa ditta potrebbe propormi un prezzo di 250€ (cadauno), quindi con uno sconto davvero interessante.
A meno che io non sia un rivenditore però, non sarei assolutamente interessato a ricevere 10

scarichi, per questo motivo, potrei avere la necessità di trovare altre 9 persone che ricercano lo stesso oggetto. Trovati questi soggetti, non mi resterebbe altro che far presente alla ditta, la nostra volontà di aderire alla loro offerta. Risparmiando così 50€ sul prezzo finale.

4.3 Come nascono i group buy?

I GB possono nascere in due modi differenti:

- a fronte di una richesta da parte dei consumatori ("*siamo interessati all'oggetto x, che sconto potete proporci se lo acquistiamo in y persone?*").
- a fronte di un offerta della ditta produttrice ("*possiamo offrire l'oggetto x a questo prezzo – scontato – se siete in y persone che vogliono acquistarlo*").

4.4 I group buy in rete

In rete è possibile trovare dei group buy su forum specializzati (specifici di un certo settore) e la maggior parte nasce a fronte di un offerta di una ditta produttrice.

La ditta propone l'offerta sul forum specializzato, specificando il numero di utenti che devono partecipare per poter usufruire dell'oggetto con quel determinato sconto. Gli utenti interessati si "inseriscono" in lista e quando la lista è piena (il numero di utenti è pari a quello specificato inizialmente dalla ditta), l'offerta si può realizzare. Gli utenti iscritti pagano il dovuto e la ditta spedisce gli oggetti a tutti.
Fino a che la lista non è piena, è possibile per qualsiasi utente, cancellarsi dalla lista stessa e non essere più vincolati all'offerta.

5. Descrizione del problema

AGATA vuole far si che ogni utente possa acquistare l'oggetto che desidera, spendendo la cifra che ha a disposizione (o anche meno) e impiegando il tempo che si era prefissato per fare l'acquisto (o addirittura meno).

AGATA utilizza liste di GB che saranno aperte dalle aziende stesse, per permettere agli utenti di usufruire di prezzi molto vantaggiosi.
Le liste sono chiaramente inserite all'interno di AGATA in maniera dinamica. Questo fa si che la lista migliore per comprare un oggetto possa non essere sempre la stessa per un certo utente.

Questo significa che AGATA deve stabilire in maniera continuata, fino all' acquisto dell'oggetto in questione, se l'utente è iscritto nella miglior lista (secondo i parametri forniti dall'utente stesso).

L'utente può in qualsiasi momento ritirare la volontà di acquistare un oggetto, a meno che non sia iscritto ad una lista che nel frattempo si è chiusa (e quindi è vincolato all'acquisto).

6. Analisi

6.1 Analisi del testo

Dopo la classica fase di iscrizione, un utente può voler:

1) acquistare un oggetto
2) non acquistare più un oggetto che voleva acquistare in precedenza

Esistono inoltre tante altre azioni che un utente potrebbe voler eseguire ma che non verranno considerate in questa trattazione.Come ad esempio :

1) iscriversi ad una lista particolare
2) cancellarsi da una lista in cui era iscritto
3) accettare l'iscrizione automatica ad una lista
4) non accettare l'iscrizione automatica ad una lista
5) …

6.2 Analisi delle azioni

Ecco una breve analisi delle azioni possibili:

1) acquistare un oggetto:
   Quando un utente esprime la volontà di acquistare un oggetto, il sistema deve controllare la presenza di quel determinato oggetto nelle liste aperte.
   - Se nessuna lista di quel particolare oggetto è presente, non eseguire azioni.
   - Se una o più liste sono presenti, il sistema iscriverà l'utente nella migliore (secondo i parametri forniti dall'utente) di queste.

   Ad ogni nuovo inserimento di una lista di un oggetto con lo stesso nome di uno degli oggetti ricercati, il sistema deve :
   - Se l'utente non è iscritto a nessuna lista per quell'oggetto, verificare se la nuova lista arrivata soddisfa le richieste dell'utente. Se le verifica, iscrivilo (se ancora possibile).
   - Se l'utente è iscritto ad una lista per quell'oggetto, verifica se la nuova lista è migliore della vecchia. Se ciò è vero, verifica che si possa togliere l'utente dalla vecchia lista.Se ciò è possibile eliminalo e inseriscilo nella nuova (se ancora possibile).

Come si può ben notare dall'analisi del testo, il problema presenta concetti di **proattività** e **reattività**:

- il sistema deve far si che l'utente possa comprare l'oggetto/i desiderato/i (**goal: acquisto**)

- il sistema deve garantire che l'utente sia sempre iscritto nella miglior lista possibile (secondo le sue esigenze) (**goal: iscrizione ottima, reattività alla creazione di nuove liste**)

6.2 Modello del dominio

Viste le caratteristiche del nostro problema emerse durante l'analisi del testo, possiamo in un' ottica AI-Based, descrivere il cuore intelligente del nostro sistema come un sistema esperto, che dalla letteratura sappiamo avere come caratteristiche:

- **Monitoring**: i dati si interpretano continuamente per la generazione di allarmi in situazioni critiche. Al sistema e` richiesta una risposta in tempo reale soddisfacente.
- **Planning**: determinare una sequenza intelligente di azioni per raggiungere un determinato obiettivo

Entrambe adatte quindi allo scopo di costruire un sistema proattivo (planning) e reattivo (monitoring) come il nostro.

6.2.1 Il Sistema esperto

Dobbiamo innanzi tutto considerare che il nostro sistema esperto agisce su un dominio dinamico (liste nuove che vengono inserite nel sistema oppure liste che si chiudono), pertanto non si deve limitare semplicemente ad applicare le regole di produzione per raggingere il goal ma esso deve anche considerare uno ad uno tutti i nuovi fatti che vengono inseriti all'interno del KB (monitoring).

1) una la lista a cui ero iscritto si chiude e l'oggetto voluto viene acquistato

Nuovo fatto: lista(idLista,nomeOggetto,costo,tempo,chiusa)

Regola: iscritto(idLista,nomeOggetto) → oggetto_acquistato(nomeOggetto,costo) & rimuovi il fatto voglio_oggetto(nomeOggetto,_,_) & rimuovi fatto iscritto(idLista,nomeOggetto)

2) se vuoi un oggetto, cerca tra le liste aperte già inserite se è qual'è la migliore (se esiste) e nel caso iscriviti

Nuovo fatto: voglio_oggetto(nomeOggetto,costo,tempo)

Regola: → migliorLista(nomeOggetto,costo,tempo)

3) se sei iscritto ad una lista e ne arriva una nuova, iscriviti alla nuova se è migliore della precedente (e se è possibile)

Nuovo fatto: lista(idLista,nomeOggetto,C1,T1,aperta)

Regola: voglio_oggetto(nomeOggetto,costoR,TempoR) & iscritto(y,nomeOggetto) & lista(y,nomeOggetto,C2,T2,aperta) & C2>C1 & T2<=TempoR → disiscrivitiIscriviti(idUtente,y,idLista,nomeOggetto)

4) se arriva una nuova lista e non sei iscritto a nessuna lista, iscriviti (sempre che sia accettabile rispetto alle tue richieste).

Nuovo fatto: **lista(idLista,nomeOggetto,C1,T1,aperta)**

Regola: voglio_oggetto(nomeOggetto,costoR,TempoR) & not iscritto(_,nomeOggetto) & costoR>C1 & T1<=TempoR → iscriviti(idUtente,idLista,nomeOggetto)

5**) la nuova volontà cancella la precedente**

Nuovo fatto: **voglio_oggetto(nomeOggetto,costoR,TempoR)**
Regola:
voglio_oggetto(nomeOggetto,C,T) → rimuovi il fatto voglio_oggetto(nomeOggetto,C,T)

Il **goal** risulta essere **oggetto_acquistato(nomeOggetto,costo),** anche se in verità il nostro sistema esperto dovrebbe continuare a lavorare in quanto non è detto che non vi siano altri oggetti che l'utente vuol comprare in quel momento.

Vengono utilizzate diverse azioni che possono essere compiute dal sistema esperto:

- **migliorLista(nomeOggetto,costo,tempo)** funzionalità che permette di stabilire qual'è la miglior lista secondo i parametri di costo e tempo richiesti, se l'operazione ha successo nel KB sarà inserito un fatto **lista(idLista,nomeOggetto,costo,tempo,aperta)** come risposta a questa interrogazione.

- **iscriviti(idUtente,idLista,nomeOggetto)** funzionalità che permette di inserire un utente in una certa lista, se l'operazione ha successo nel KB sarà inserito un fatto **iscritto(idLista,nomeOggetto)** che informerà dell' avvenuto compimento della funzionalità

- **disiscriviti(idUtente,idLista,nomeOggetto)** funzionalità che permette di eliminare un utente in una certa lista, se l'operazione ha successo nel KB sarà eliminato un fatto **iscritto(idLista,nomeOggetto)** in modo tale che la situazione sia coerente con la realtà

- **disicrivitiIscriviti(idUtente,idListaOld,idListaNew,nomeOggetto)** funzionalità che permette di inserire un utente in una certa lista se esso può viene eliminato con successo da un'altra lista. Se l'inserimento non avviene, allora anche l'eliminazione è cancellata e si ritorna allo stato iniziale.

Si può anche notare che il sistema esperto è **fortemente reattivo**, infatti esso cerca di raggiungere i suoi goal attraverso la messa in atto di piani scatenati tutti da eventi (o immissioni di fatti nel KB).

6.2.2 Il dominio del sistema esperto

Come si può notare il nostro sistema esperto utilizza dei fatti all'interno del suo KB per riuscire a portare a termine il suo goal, come:

1) iscritto(idLista,nomeOggetto)
2) lista(idLista,nomeOggetto,costo,tempo,chiusa)
3) oggetto_acquistato(nomeOggetto,costo)
4) voglio_oggetto(nomeOggetto,_,_)

Il dominio del nostro sistema esperto (Knowledge Base) risulterà essere formato da:

- informazioni specifiche sull'utente gestito dal sistema esperto considerato (n.4 cosa vuole acquistare, n.3 quali oggetti ha acquistato,n.1 a quale lista è iscritto)

- informazioni sulle liste (n.2 informazioni generali e se è chiusa o aperta).

Le liste (una lista) sono caratterizzate da :

- id lista
- nome oggetto
- ditta produttrice
- prezzo
- tempo stimato di chiusura della lista (vedi appendice)
- data di inserimento nel sistema
- numero di persone totali (per definire la lista come chiusa)
- stato della lista (chiusa/aperta)
- numero di persone inserite attualmente

6.2.3 Considerazioni sul modello del dominio

Si è potuto notare come il dominio su cui lavora il sistema esperto sia formato da due differenti tipi di informazioni. Il primo tipo è specifico dell'utente e del lavoro svolto dal suo sistema esperto (quindi potenzialmente diverso per ognuno di essi), mentre il secondo tipo è generale e legato essenzialmente alle liste e a tutti gli utenti del nostro sistema (quindi uguale per tutti gli utenti ed i loro sistemi esperti).

Ad esempio:

Utente 1 / Sistema esperto 1:

1) oggetto_acquistato(“tv Sony 3d”,300)

2) voglio_oggetto(“tv Sony 3d”,300,20)

3) lista(10,“tv Sony 3d”,300,20,chiusa)

Utente 2 / Sistema esperto 2:

4) voglio_oggetto(“tv Sony 3d”,300,10)

5) lista(10,“tv Sony 3d”,300,20,chiusa)

Si può vedere come i fatti specifici (1,2,4) siano diversi mentre quelli riguardanti le liste (3,5) siano uguali (perchè riferiti alla stessa lista).

Per questo motivo dovrebbe essere conveniente mantenere queste informazioni, cioè quelle che riguardano le liste, in un'entità differente dal sistema esperto stesso, in modo tale garantire consistenza e correttezza negli accessi concorrenti a questi dati. Per semplicità, chiameremo questa entità “Liste”.

Inoltre il sistema esperto compie delle azioni per raggiungere i suoi goal, come **miglior_lista**, **iscriviti** e **disiscriviti**. Ovviamente queste operazioni possono essere portate a termine da questa nuova entità dato che tutte le informazioni sulle liste sono contenute al suo interno.

Quindi emergono chiaramente nel nostro sistema due diversi tipi di entità, da una parte proattive e reattive, come il sistema esperto e dall'altra invece quelle “usabili” dalle prime per raggiungere i loro goal, come l'entità “Liste”. Per questo motivo, si può procedere alla progettazione utilizzando un metamodello A&A che presenta questi concetti mappati rispettivamente su Agenti ed Artefatti, in modo tale da garantire un percorso logico semplice e di facile ed immediata realizzazione.

7. Progettazione

Passando alla fase di progettazione, possiamo inizialmente discutere gli eventuali vincoli progettuali, che - visto il problema - possono essere :

- rendere eventualmente possibile “spostare” parte della computazione lato utente (vedi web 2.0) in modo da rendere più snello il sistema lato server.

- parallelizzare il più possibile la computazione, in modo da sfruttare le nuove architetture e velocizzare quindi il sistema stesso.

Entrambi i vincoli possono essere soddisfatti utilizzando un architettura modulare come quella realizzabile tramite il metamodello A&A. E' quindi per questo motivo e per quanto detto alla fine del capitolo precedente che possiamo procedere alla fase di progettazione utilizzando gli Agenti per realizzare la parte proattiva e reattiva del nostro sistema e gli Artefatti per sviluppare la parte “passiva” che garantisce gli strumenti necessari agli agenti per portare a termine le loro attività.

7.1 Struttura

Dal punto di vista strutturale, riferendoci a quanto detto nelle considerazioni sul modello del dominio, mapperemo in maniera univoca con un artefatto l'entità Liste:

- *Liste*: artefatto che si occupa di permettere l'accesso a strutture dati che contengono le liste (nel caso del prototipo anche tenere memorizzate le liste con le loro informazioni generiche e gli utenti iscritti alle liste in tempo reale)

Inoltre occorrerà per garantire che ogni utente sia loggato correttamente all'interno del sistema un artefatto Utenti:

- *Utenti*: artefatto che si occupa di permettere l'accesso (iscrizione, cancellazione) a strutture dati che contengono gli utenti (nel caso del prototipo anche tenere memorizzati gli utenti di AGATA e le informazioni riguardanti la loro iscrizione al sistema)

Per quanto riguarda invece il sistema esperto è chiaramente mappabile viste le caratteristiche con un agente:

- *Osservatore*: agente che si preoccupa realizzare i goal degli utenti, rimanendo reattivo all'arrivo di nuove liste

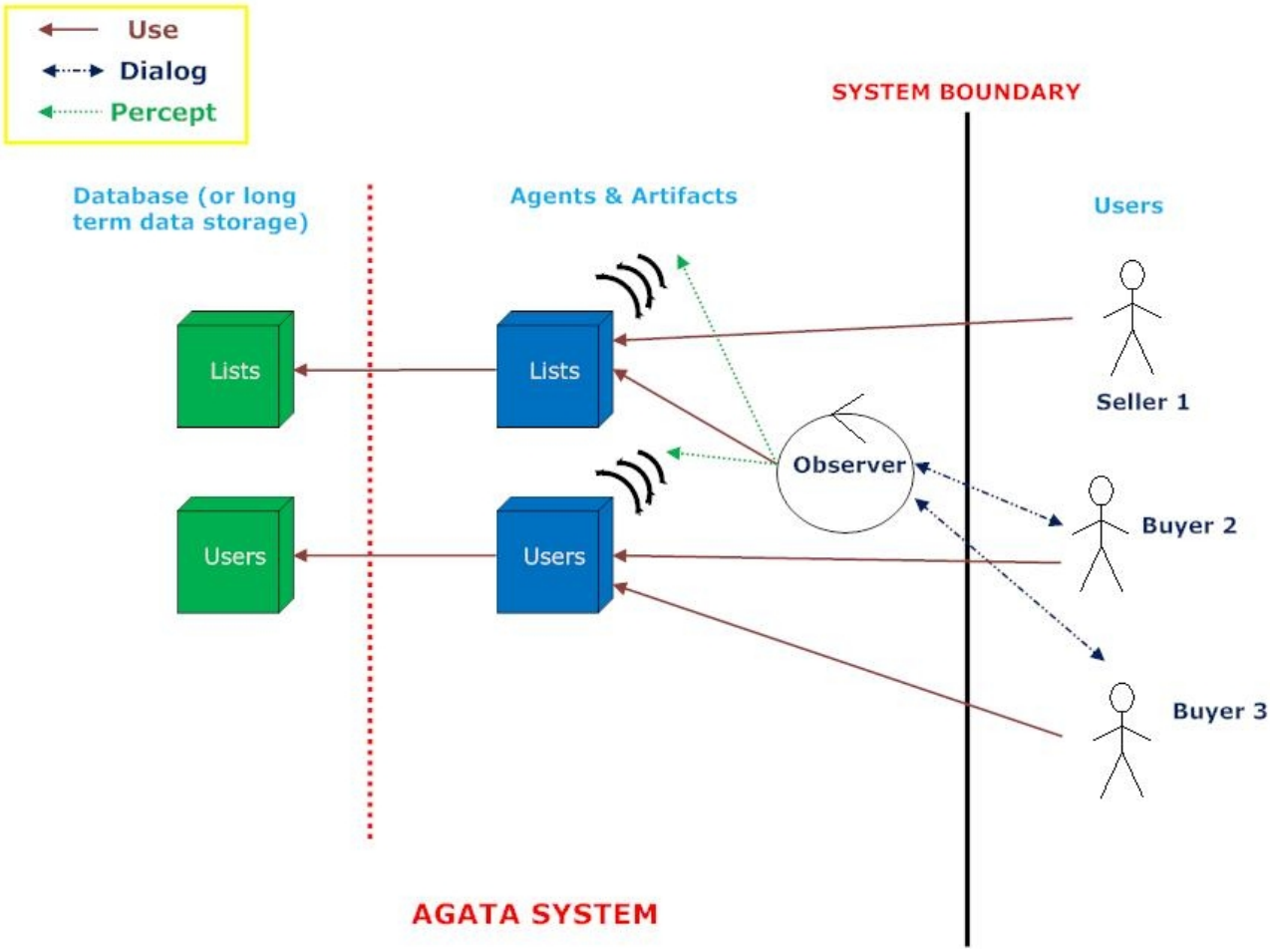


Illustrazione 1: Progettazione del sistema con un solo agente-osservatore. Nessuna possibilità di "distribuire" il sistema.

Per adempiere ai vincoli di progettazione si può fare in modo che esista un osservatore per ogni utente oppure anche un pool di agenti per ogni utente o anche uno per ogni richiesta di acquisto dell'utente (vedi illustrazione 2).

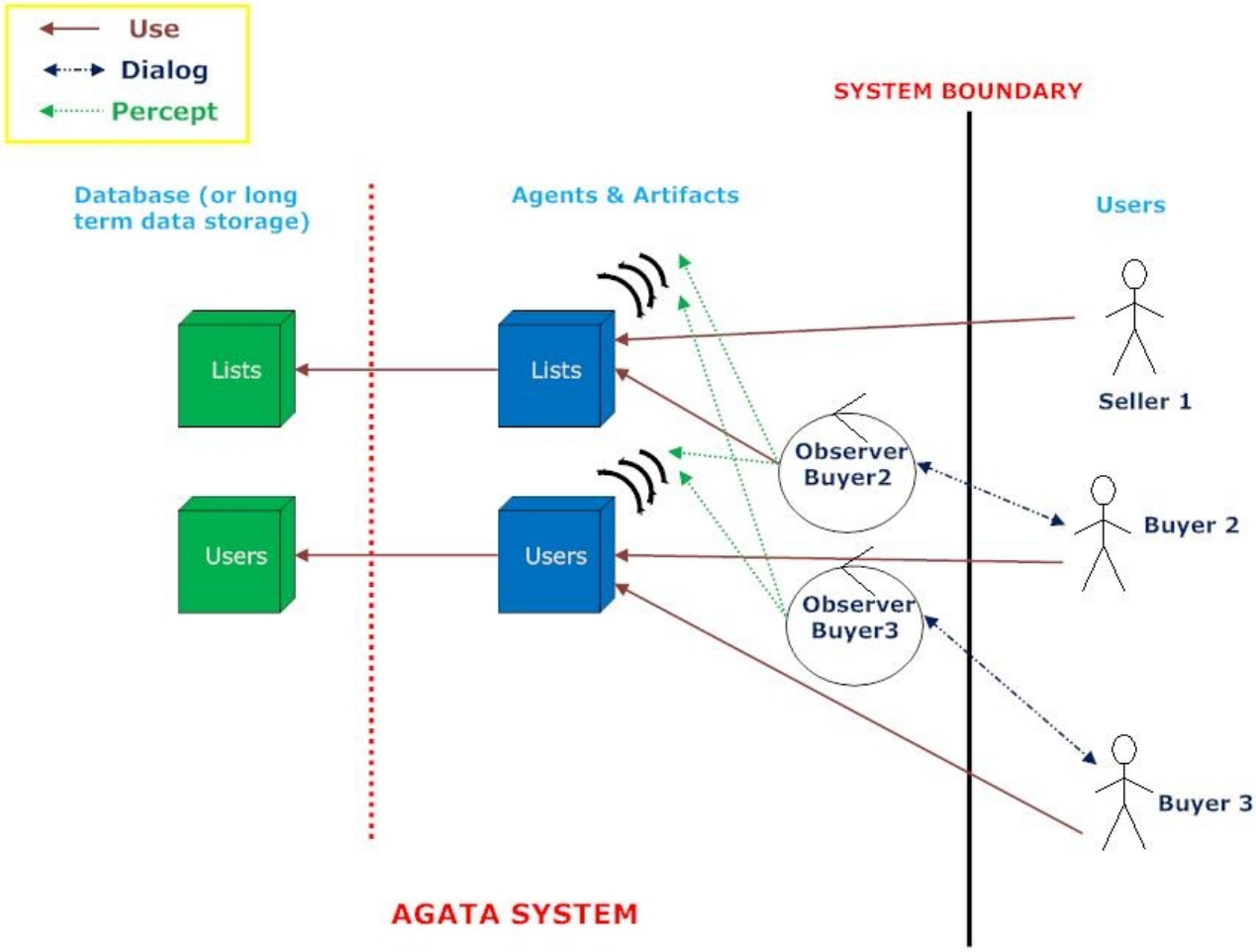


Illustrazione 2: Progettazione del sistema con un agente-osservatore per ogni utente. Possibilità di "distribuire" il sistema, spostando gli agenti sui diversi client.

Così facendo l'agente (o gli agenti) Osservatore per un certo utente, potrebbe risiedere anche lato client (vedi illustrazione 3).

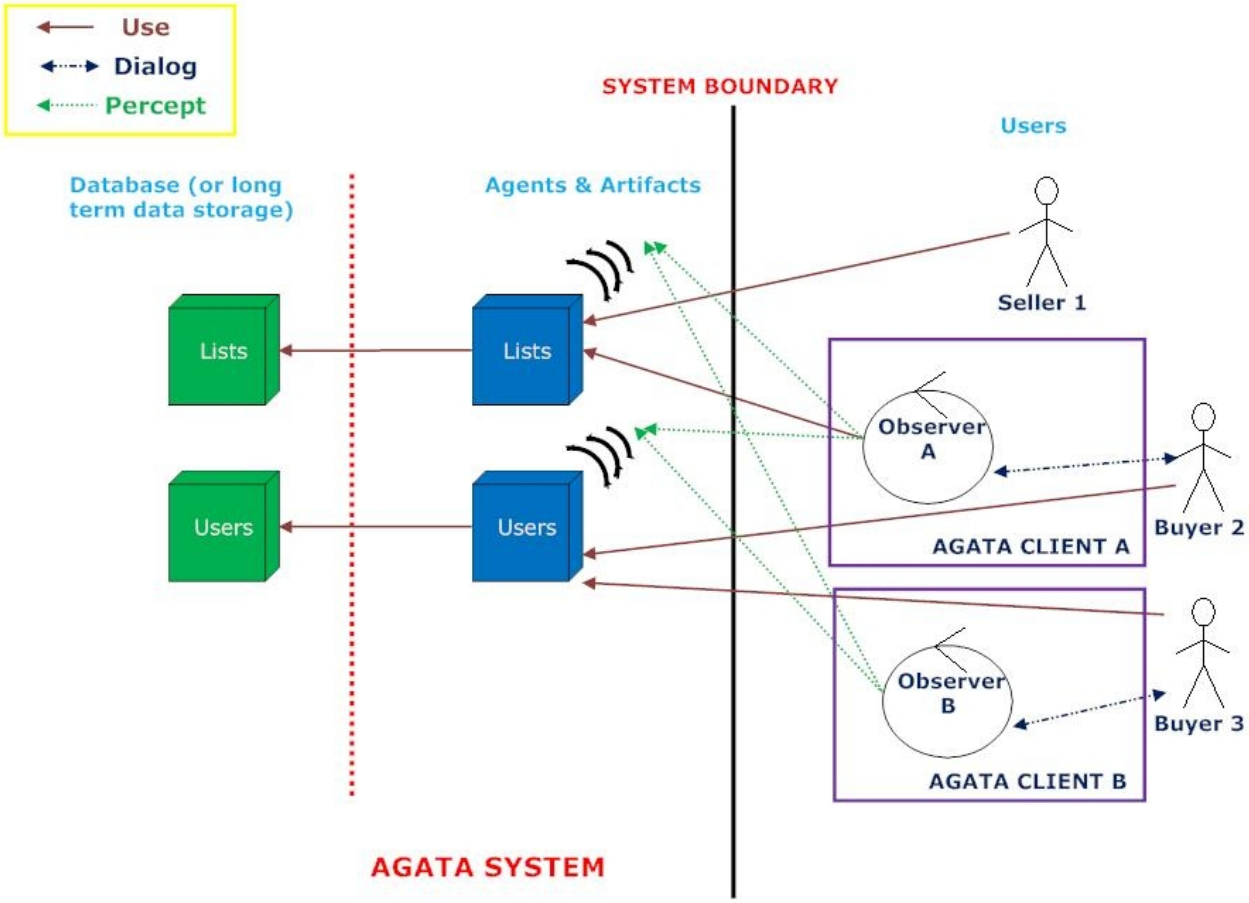


Illustrazione 3: Progettazione del sistema con un agente-osservatore per ogni utente spostato direttamente lato client.

7.1.1 Artefatti: operazioni

Nel modello del dominio abbiamo visto come l'entità “Liste” che noi abbiamo mappato con un artefatto, necessiti delle seguenti funzionalità:

1) migliorLista(nomeOggetto,costo,tempo)
funzionalità che permette di stabilire qual'è la miglior lista secondo i parametri di costo e tempo richiesti, se l'operazione ha successo nel KB sarà inserito un fatto **lista(idLista,nomeOggetto,costo,tempo,aperta)** come risposta a questa interrogazione.

2) iscriviti(idUtente,idLista,nomeOggetto)
funzionalità che permette di inserire un utente in una certa lista, se l'operazione ha successo nel KB sarà inserito un fatto **iscritto(idLista,nomeOggetto)** che informerà dell' avvenuto compimento della funzionalità

3) disiscriviti(idUtente,idLista,nomeOggetto)
funzionalità che permette di eliminare un utente in una certa lista, se l'operazione ha successo nel KB sarà eliminato un fatto **iscritto(idLista,nomeOggetto)** in modo tale che la situazione sia coerente con la realtà

4) disicrivitiIscriviti(idUtente,idListaOld,idListaNew,nomeOggetto) funzionalità che permette di inserire un utente in una certa lista se esso può viene eliminato con successo da un'altra lista. Se l'inserimento non avviene, allora anche l'eliminazione è cancellata e si ritorna allo stato iniziale.

per cui intendiamo fare in modo che questo artefatto realizzi le seguenti operazioni:

1) **give_me_best_old_list(NomeOggetto, Costo, Tempo)**
   Segnale inviato in caso di successo:
   new_list_created_signal(idLista, nomeOggetto, costo, tempo)

2) **add_user_to_list(idUser, idList, response)**
   response – parametro di ritorrno se l'operazione è andata a buon fine

3) **remove_user_from_list(idUser, idList, response)**
   response – parametro di ritorrno se l'operazione è andata a buon fine

4) **remove_and_add_user(idUser, idListOld, idListNew, response)**

Inoltre dato che in questo artefatto sono presenti tutte le informazioni sulle liste, potrà essere utile:

1) **creare una nuova lista**

2) **se una lista si chiude, segnalare questo evento**

per cui saranno realizzate dall'artefatto le seguenti operazioni/funzionalità :

1) create_new_list(name, cost, time, nMaxUsers)
   Segnale inviato in caso di successo:
   signal("new_list_created_signal", name, cost, time)

2) **signal("list_closed_signal", idList)**
   segnale inviato quando una lista si chiude (raggiungendo il suo numero massimo di untenti)

per l'artefatto utenti:

1) verifica se un utente è iscritto al sistema sulla base di username e password

7.1.2 Agenti: piani e trigger event

Per quanto riguarda gli agenti osservatori, andiamo ad analizzare, tenendo conto di quanto detto nei capitoli precedenti, quali devono essere i vari piani e i trigger-event.

1) una la lista a cui ero iscritto si chiude e l'oggetto voluto viene acquistato

Nuovo fatto: lista(idLista,nomeOggetto,costo,tempo,chiusa)

Regola: iscritto(idLista,nomeOggetto) → oggetto_acquistato(nomeOggetto,costo) & rimuovi il fatto voglio_oggetto(nomeOggetto,_,_) & rimuovi iscritto(idLista,nomeOggetto)

si trasforma nel TI, scatenato dall'arrivo di un segnale **list_closed_signal(LID)** (la lista si è chiusa) inviato dall'artefatto liste.

```
[atomic]
+list_closed_signal(LID) : registered_into_list(LID,N,C,T,true) & want_buy(N,_,_)
←       -want_buy(N,_,_);
        -registered_into_list(LID,N,C,T,true);
        +item_bought(N,C);
```

2) se vuoi un oggetto, cerca tra le liste aperte già inserite se è qual'è la migliore (se esiste) e nel caso iscriviti

Nuovo fatto: voglio_oggetto(nomeOggetto,costo,tempo)

Regola: → migliorLista(nomeOggetto,costo,tempo)

si trasforma nel TI, scatenato dall'arrivo di un segnale **want_buy(NAME,MYCOST,MYTIME)** generato da un utente o da chi ne fa le veci:

```
[atomic]
+want_buy(NAME,MYCOST,MYTIME) : myUserId(UID)
←       give_me_best_old_list(NAME, MYCOST, MYTIME).
```

3) se sei iscritto ad una lista e ne arriva una nuova, iscriviti alla nuova se è migliore della precedente (e se è possibile)

Nuovo fatto: lista(idLista,nomeOggetto,C1,T1,aperta)

Regola: voglio_oggetto(nomeOggetto,costoR,TempoR) & iscritto(y,nomeOggetto) & lista(y,nomeOggetto,C2,T2,aperta) & C2>C1 & T2<=TempoR → disiscrivitiIscriviti(idUtente,y,idLista,nomeOggetto)

si trasforma nel TI, scatenato dall'arrivo di un segnale **new_list_created_signal(LID,N,C,T)** generato dall'artefatto “Liste”:

[atomic]
+new_list_created_signal(LID,N,C,T) :
myUserId(UID) & want_buy(N,CS1,TM1) & registered_into_list(OLDLID,N,CS2,TM2,true) & T<=TM1 & C<CS2
← remove_and_add_user(UID,OLDLID,LID,RESPONSE,RESPONSENOT);
-registered_into_list(OLDLID,N,CS2,TM2,true); *
+registered_into_list(OLDLID,N,CS2,TM2,RESPONSENOT); *
+registered_into_list(LID,N,C,T,RESPONSE); *

* Si è verificato in fase di test un problema riguardante il fallimento di un operazione all'interno di un piano che deve essere atomico (in pratica si blocca l'esecuzione del piano e non riesce a fallire). Per questo motivo, come si può vedere si utitilizza una modalità particolare di passaggio di parametri per le chiamate alle funzionalità degli artefatti. Ad esempio **RESPONSE** non è un input e viene utilizzato come parametro di ritorno della funzionalità **remove_and_add_user**. Questo perchè il piano deve essere atomico, quindi l'informazione non può essere ritornata con un segnale (che permetterebbe l'interleaving nel momento in cui il piano in questione termina) e inoltre non può essere trasmesso tramite fallimento dell'operazione, come detto all'inizio.
Tutto ciò per fare in modo che l'immissione e la cancellazione dei fatti sia effettuata solo se l'operazione è andata a buon fine (come da specifiche).

4) se arriva una nuova lista e non sei iscritto a nessuna lista, iscriviti (sempre che sia accettabile rispetto alle tue richieste).

Nuovo fatto: lista(idLista,nomeOggetto,C1,T1,aperta)

Regola: voglio_oggetto(nomeOggetto,costoR,TempoR) & not iscritto(_,nomeOggetto) & costoR>C1 & T1<=TempoR → iscriviti(idUtente,idLista,nomeOggetto)

si trasforma nel TI, scatenato dall'arrivo di un segnale **new_list_created_signal(LID,N,C,T)** generato dall'artefatto “Liste”:

[atomic]
+new_list_created_signal(LID,N,C,T) : myUserId(UID) & want_buy(N,CS1,TM1) & not registered_into_list(_,N,_,_,true) & T<=TM1 & C<CS1
← add_user_to_list(UID,LID,RESPONSE);
+registered_into_list(LID,N,C,T,RESPONSE);

7.2 Progettazioni alternative (scartate)

Prendendo in considerazione quanto detto nel paragrafo precedente, viene molto naturale progettare il sistema nella maniera vista in precedenza, dato che le funzionalità di proattività e reattività sono modellate sulla base delle richieste e delle volontà di ogni singolo utente e non di una lista. E' l'utente infatti che deve raggiungere il goal di acquisto o di iscrizione nella miglior lista e quindi è conseguenza logica implementare un agente (o un pool) che "segua" ogni utente.

Esistono però delle progettazioni alternative che sono:

- realizzare ogni lista come agente
- realizzare come agente solo quel componente che si occupa di gestire le liste (contenitore/gestore delle liste)

7.2.1 Realizzare ogni lista come agente

In questo caso l'agente-lista avrebbe il goal di mantenere inserito ogni utente al suo interno, se e solo se quella lista è la migliore nel sistema. Va da sé che questo significherebbe:

- scambio di comunicazioni con gli altri agenti-liste per verificare che l'utente preso in considerazione sia iscritto nella miglior lista del sistema (questo per ogni utente iscritto!)
- ad ogni nuovo agente-lista iscritto, scambio di comunicazioni con tutti gli altri agenti-lista per verificare quanto detto sopra

Tutto ciò porterebbe ad un esagerato aumento dell'overhead rispetto a quanto abbiamo visto nella progettazione principale dove, per controllare se un utente è iscritto nella miglior lista, bastarebbe solamente un operazione su un artefatto, mentre in quella che stiamo considerando ora, occorrerebbero tante "comunicazioni" quanti sono il numero degli agenti-lista presenti (-1; vedi illustrazione 4).

In questo caso inoltre non occorrerebbe più avere uno stoccaggio dei dati su database, se non per sicurezza, dato che le caratteristiche di ogni lista sarebbero memorizzate all'interno dell'agente stesso (anche per velocizzare le operazioni).

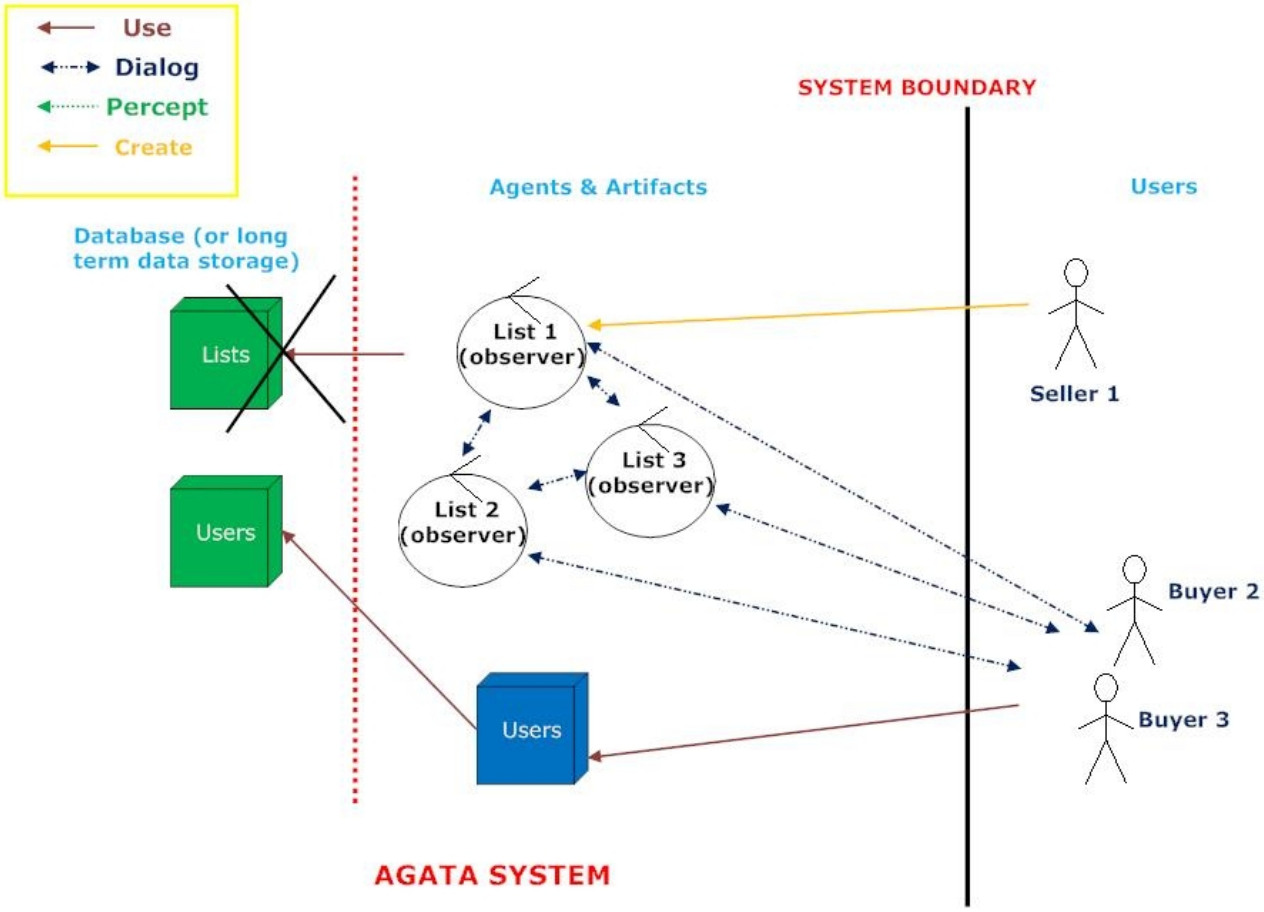


Illustrazione 4: Progettazione del sistema con agenti-lista. E' possibile notare la presenza di una numerosa interazione tra i vari agenti osservatori e anche tra gli osservatori e l'agente compratore.

7.2.2 Realizzare come agente solo il gestore le liste

Potremmo anche progettare il contenitore delle liste come agente ma sarebbe molto difficile implementare il sistema in maniera distribuita, dato che occorrerebbe replicare ognuno dei contenitori su ogni client (quindi una parte o tutto l'insieme delle liste su ogni client) . Questo porterebbe ad un ancor più esagerato overhead per mantenere la sincronizzazione tra le liste (basti pensare se un utente viene inserito in una particolare lista, in tutti i client dove risiede quella lista, andrà eseguito l'aggiornamento).

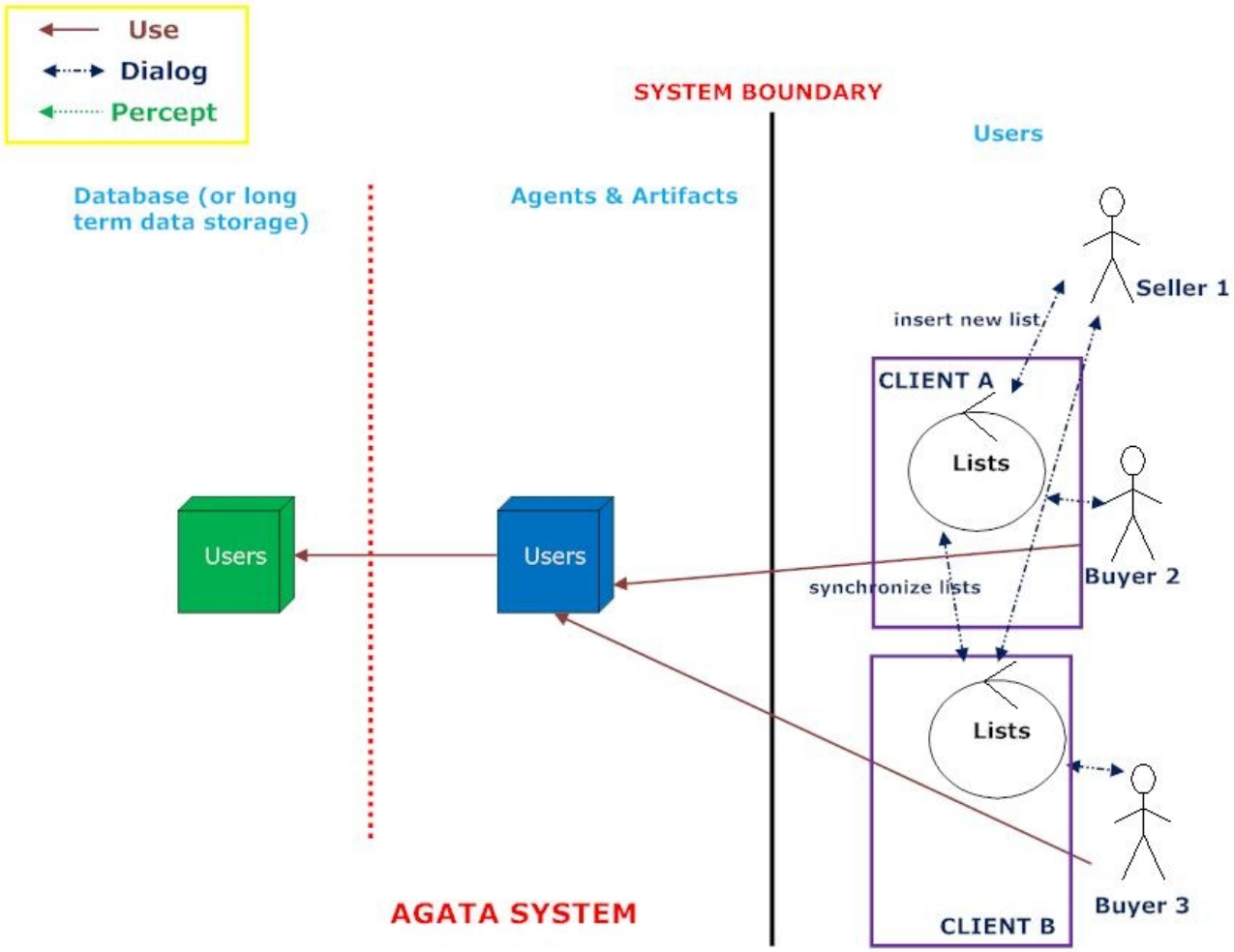


Illustrazione 5: Progettazione del sistema con agenti-liste lato client. E' possibile notare la presenza di una numerosa interazione tra i vari agenti osservatori e anche tra gli osservatori e l'agente compratore.In questo caso inoltre non viene visualizzato uno storage di sicurezza centralizzato che servirebbe sicuramente.

Ad ogni modo non è comunque preferibile inserire la parte intelligente all'interno di entità atte a incapsulare i dati del sistema (Liste o lista) visto che secondo il metamodello A&A esse si mappano con degli artefatti.

7.3 Progettazione con agenti Osservatori lato server

7.3.1 Introduzione:

Questa progettazione, anche se viola i nostri vincoli progettuali, sarà presa in considerazione, in quanto per lo meno inizialmente, l'applicazione potrebbe essere raggiungibile anche da un semplice browser e non per forza da un'applicazione lato client che contiene degli agenti.

In questo caso occorrerà che ad un utente siano associati uno o più agenti osservatori (*), se e solo se egli esprime la volontà di realizzare dei goal (ad esempio un acquisto). Tutto ciò per mantenere l'efficienza computazionale, in quanto sarebbe inutile associare un utente ad un agente se questo non vuole compiere nessuna azione rilevante.

Le associazioni sono svolte in questa semplice maniera:

- Volontà di acquistare uno o più oggetti & utente non associato → associati con un agente libero
- Volontà di acquistare un oggetto & utente associato → mantieni l'associazione
- Nessuna volontà di acquistare un oggetto (perchè sono stati acquistati oppure sono state eliminate le volontà) → termina l'associazione (puoi associare quell'agente con un'altro utente).

- si considera da ora in avanti un solo agente osservatore per utente.

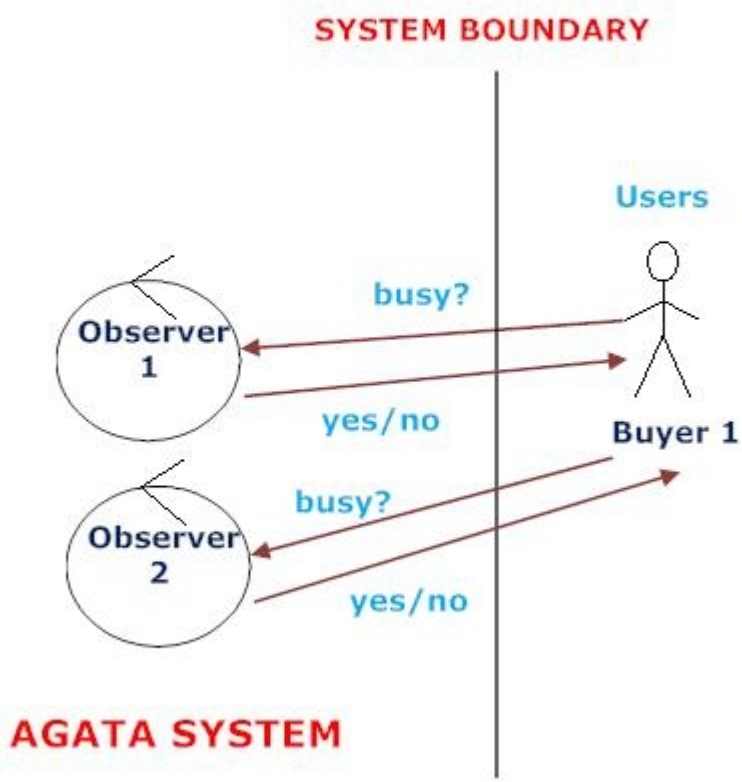
7.3.2 Struttura del sistema

Come abbiamo visto, per poter associare un utente ad un agente osservatore, occorre che l'utente o chi per lui, esprima la volontà di acquistare uno o più oggetti ed esso non sia ancora associato a nessun agente.

Se l'utente vuole richiedere l'associazione ha bisogno di conoscere :

- gli agenti osservatori del sistema a cui effettuare le richieste (vedi illustrazione 6) oppure:
- conoscere un entità che tratta direttamente con gli osservatori e si preoccupi di portare a compimento queste azioni per lui (illustrazione 7)

Illustrazione 6: Particolare di progettazione del sistema con interazione diretta tra agenti osservatore ed utente



La prima opzione non può essere realizzata in quanto il sistema deve essere trasparente e un utente non può conoscere gli osservatori ne quanti ne sono presenti all'interno del sistema stesso (informazione utile per gestire la fase di contatto).

La seconda invece necessita di costituire una sorta di interfaccia facente parte del nostro sistema che si preoccupi di memorizzare tutte le richieste degli utenti che non sono ancora associati e che si occupi di gestire la comunicazione per le associazioni con gli osservatori. E' importante mantenere tutte le richieste memorizzate poiché nel momento stesso in cui avviene l'associazione esse dovranno essere portate a compimento dall'agente.

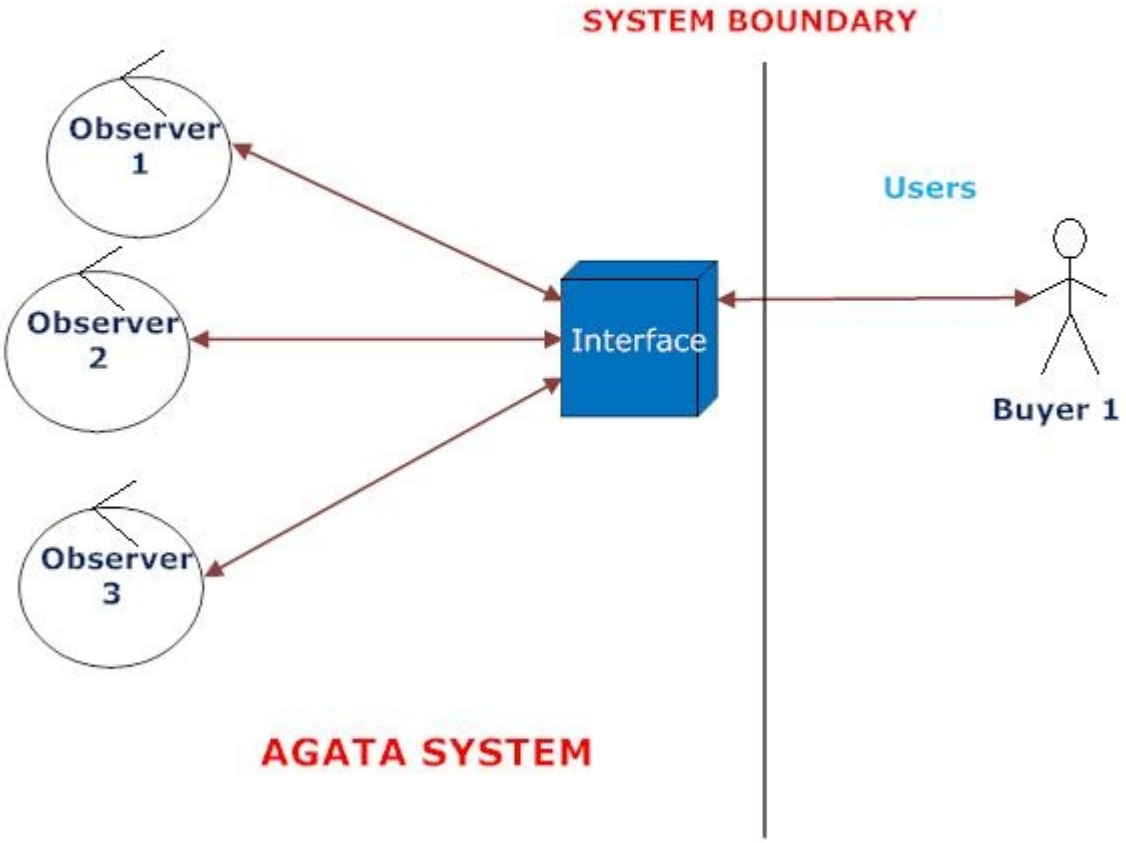


Illustrazione 7: Particolare della progettazione del sistema utilizzando un'interfaccia che serve per permettere un dialogo efficiente e trasparente tra l'utente e gli agenti osservatori

Nota: Si preferisce non utilizzare un diagramma delle interazioni poiché in questa fase l'obbiettivo è solo quello di far capire meglio il procedimento che ha portato a effettuare una scelta strutturale piuttosto che un'altra.

In questa maniera quando un agente osservatore si libera, può:

- richiedere a questa entità se sono presenti degli utenti in cerca di associazione ed in caso affermativo, associarsi con il primo di questa lista (vedi illustrazione 8).

  oppure:
- può manifestare il proprio stato di “libertà” (quando non è associato a nessuno) in modo tale che l'interfaccia, ricevendo questo cambiamento di stato, possa richiedere all'agente in questione l'associazione con uno degli utenti che aspettano di essere associati (vedi illustrazione 9).

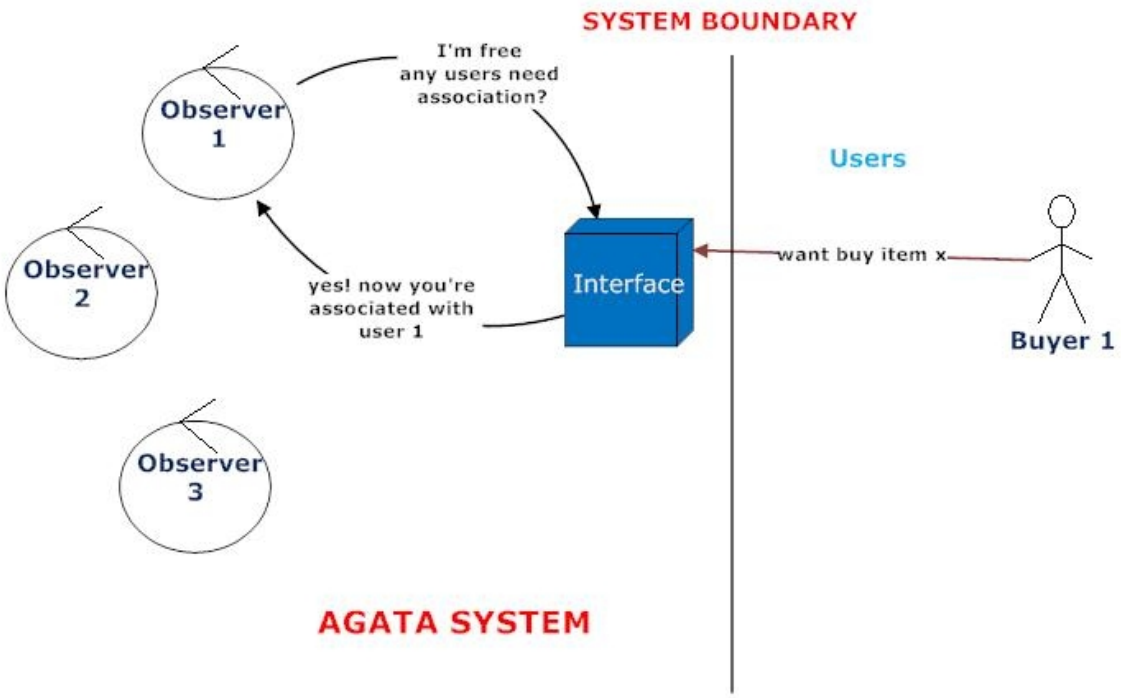


Illustrazione 8: Particolare della progettazione con interfaccia, dove si può notare che l'osservatore richiede all'interfaccia se sono presenti utenti in attesa di associazione. Questa richiesta è effettuata in polling fino a quando non arriva una risposta affermativa

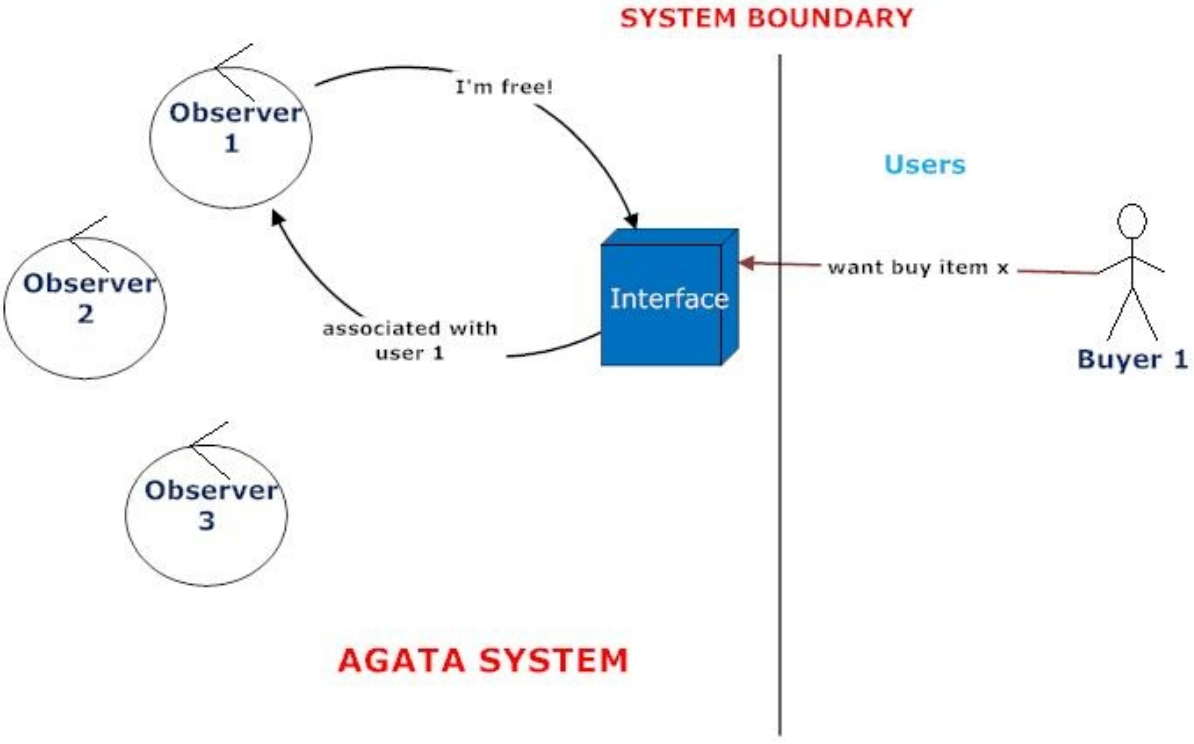


Illustrazione 9: Particolare della progettazione con interfaccia, dove si può notare che l'osservatore notifica solo il suo stato di momentanea libertà. Questa azione è effettuata una volta sola, sarà l'interfaccia a contattare l'osservatore nel caso ci siano presenti utenti in attesa di associazione.

Abbiamo scelto la seconda ipotesi di progettazione poiché nel primo caso le richieste dovrebbero essere effettuate a polling e quindi preferiamo solo far sapere all'interfaccia che l'osservatore è libero, in modo tale che essa possa contattarlo quando preferisce.

L'interfaccia però si preoccupa già di memorizzare tutte le richieste provenienti dagli utenti che ancora non si sono associati (e quindi mantenere anche tutte le associazioni in maniera dinamica). In questo tipo di progettazione dovrebbe anche farsi carico di contattare gli osservatori per le associazioni. In un ottica di suddivisione delle attività, preferiamo utilizzare un ulteriore agente che si occupi di quest'ultima mansione. Esso deve:

- conoscere tutti gli osservatori (per poterli contattare)

  ed inoltre:

- data la sua natura può reagire in maniera reattiva alla presenza di un agente osservatore libero e di un utente che necessita un associazione

Dato che avremmo la necessità di creare tutte le componenti del sistema in fase di start up, possiamo prendere la palla al balzo, facendo realizzare questa attività ad un agente "creatore" che per questo motivo può conoscere anche tutti gli agenti ed artefatti del sistema, e quindi può anche realizzare quato riportato nel paragrafo precedente. Inoltre essendo un agente può realizzare in maniera ottimale le funzioni reattive di cui necessitiamo.

7.3.3 Interazione

Ipoteticamente un utente che vuole interagire con AGATA, può essere già associato ad un agente o non esserlo. A seconda del caso avremmo due comportamenti completamente differenti (vedi rispettivamente l'illustrazione 10 e l'illustrazione 11).

7.3.3.1 Caso 1. Utente non associato

Analizziamo la prima situazione dove l'utente non risulta associato a nessun osservatore.
Qualsiasi richiesta che effettua al sistema attraverso l'interfaccia, fino al momento dell'associazione vera e propria (7) deve essere memorizzata all'interno dell'interfaccia stessa.
Quando arriva la prima (1) richiesta (ma potrebbero arrivarne altre dallo stesso utente ed è per questo motivo che si memorizzano), l'artefatto interfaccia, esterna tramite un segnale, questo evento (3).
L'agente Creator (creatore) è adibito all'ascolto di questo tipo di segnali ed inoltre può ricevere comunicazioni da parte degli agenti osservatori per quanto riguarda il loro stato di occupazione (-).
Quando il creatore ha ricevuto un segnale di richiesta di associazione (4) ed una comunicazione di nullafacenza ("I'm free") da parte di un osservatore può decidere di associare quell'utente con quell'agente libero (5).
Così facendo, l'agente osservatore incriminato può comunicare (6) all'interfaccia che egli risulta associato all'utente 1, in questo modo l'artefatto può mantenere aggiornate le liste di associazione e decidere se continuare a memorizzare richieste pendenti, oppure no.
Come risposta, l'interfaccia comunica (7) all'osservatore tutte le richieste fatte dall'utente dal momento che non era associato in modo che l'agente possa così soddisfarle.

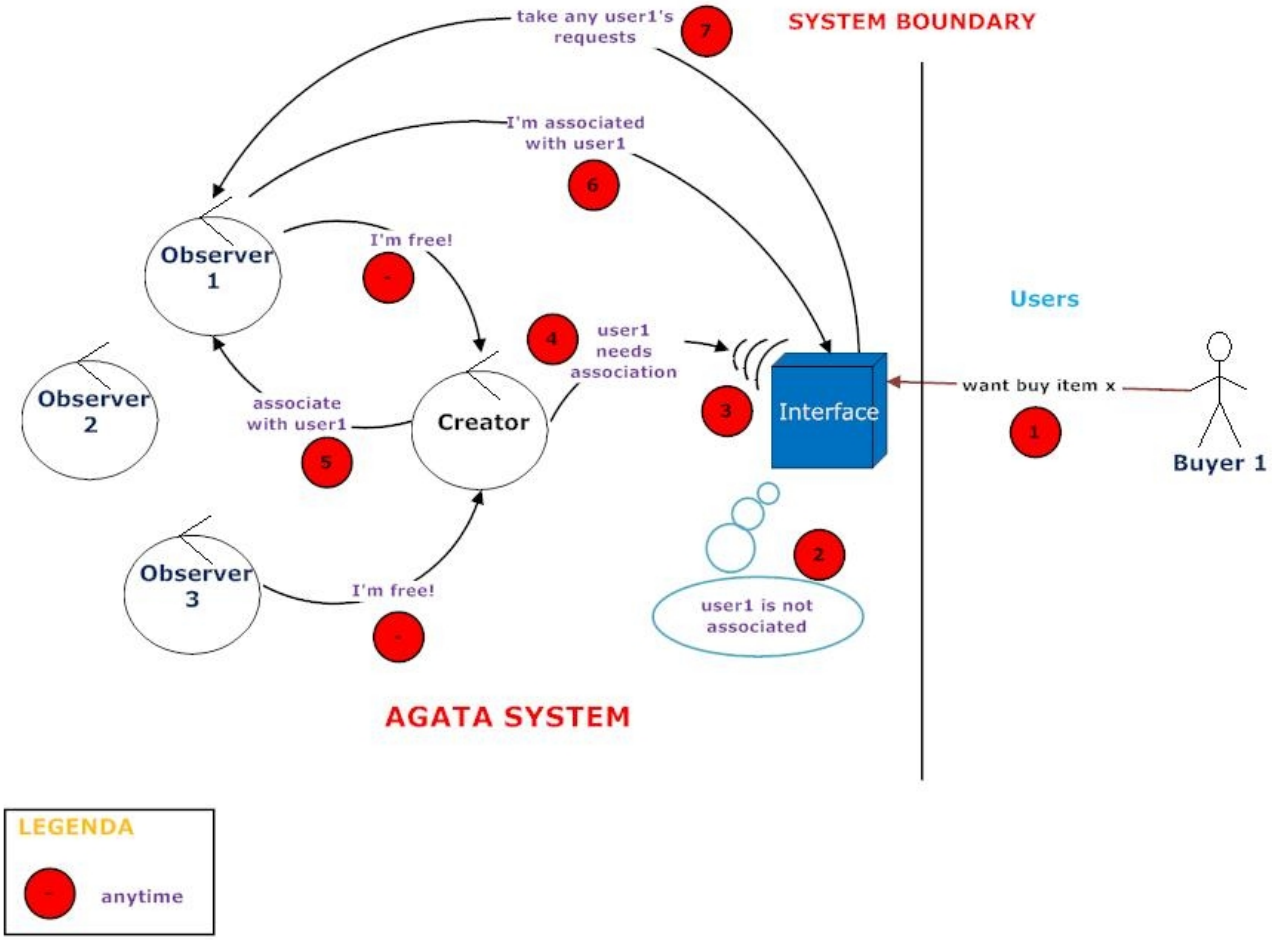


Illustrazione 10: Particolare dell'interazione tra le varie componenti del sistema, nel caso che avvengano delle richieste da parte di un utente che non è ancora associato.

7.3.3.2 Caso 2. Utente già associato

Analizziamo la prima situazione dove l'utente risulta già associato ad un osservatore. Viene analizzato il mittente di ogni richiesta (1) che giunge all'interfaccia.
Questo artefatto controlla se l'utente che ha effettuato quella richiesta specifica è associato o meno (2).
Se è già associato, esterna un segnale con il nome dell'azione da eseguire ed il destinatario (3).
Gli agenti osservatori ascoltano, mentre stanno eseguendo il loro lavoro, questi segnali (4) e se uno di essi è il vero destinatario del segnale, lo processa, mettendo in coda la richiesta del suo utente associato.

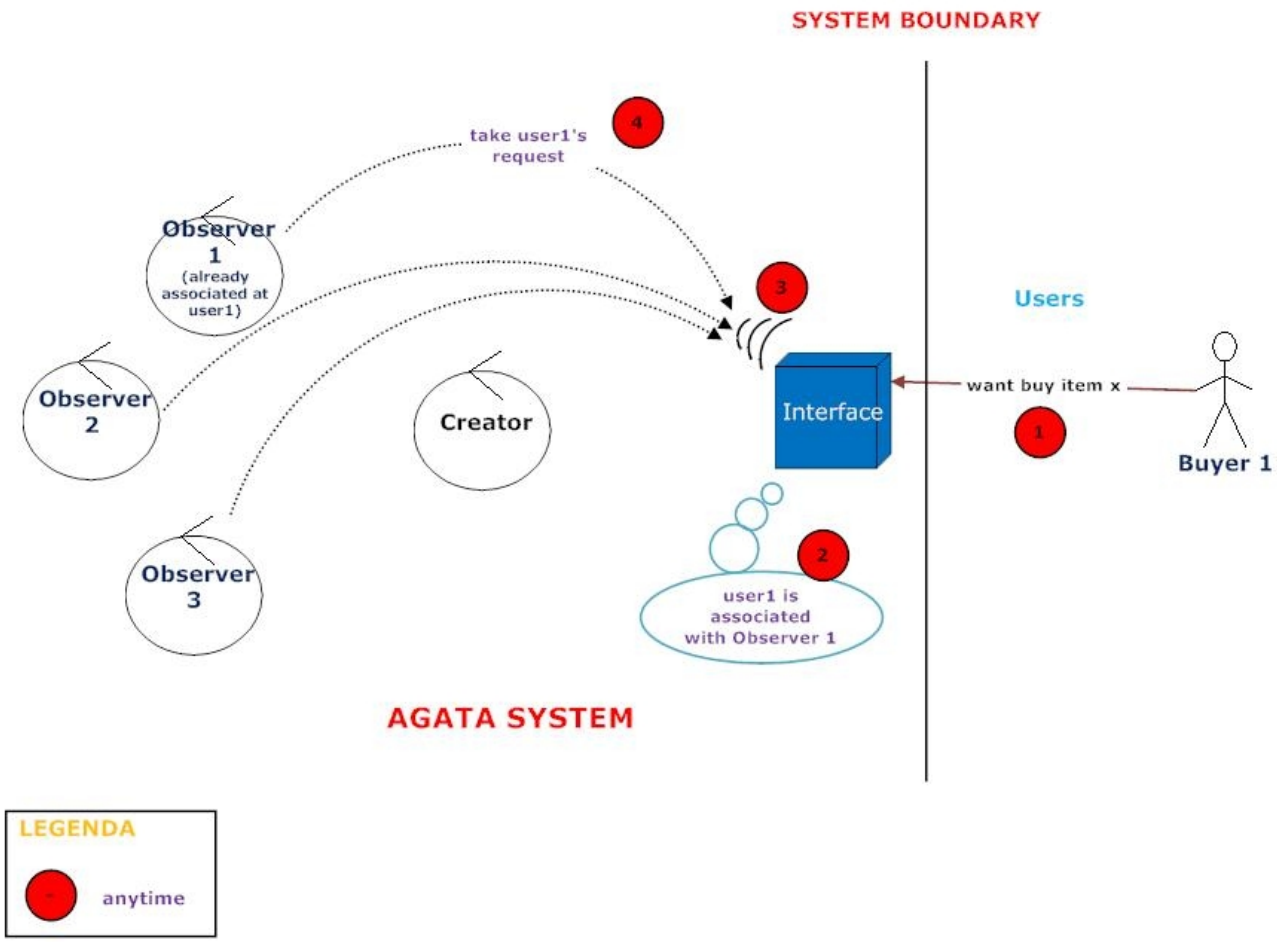


Illustrazione 11: Particolare dell'interazione tra le varie componenti del sistema nel caso l'utente che effettua una richiesta sia già associato ad un agente osservatore

7.4 Progettazione con agenti Osservatori lato client:

In questo caso, gli osservatori risiedono già sul client e sono già "associati" a quel particolare utente. Quindi tutta la fase di "associazione" non deve essere implementata.
Per portare a termine questa progettazione, basterà solamente spostare gli agenti osservatori sui client, in modo tale che la maggior parte della computazione avvenga lato utente.

8. Implementazione

Per quanto riguarda l'implementazione si è scelto di utilizzare la tecnologià di JaCa (Jason + Cartago) in quanto ci offre una mappatura perfetta tra le entità utilizzate nella fase di progettazione.

Inoltre si è dovuto tener conto che - essendo una fase prototipale - devono essere simulate:

- le richieste degli utenti
- le creazioni delle liste

Per questo motivo si è utilizzato un agente che simulava sia le richieste degli utenti che le creazioni delle liste (agente Creator) ed un artefatto apposito (AgentInterface) che incapsulasse le funzionalità offerte da Agata, come interfaccia verso il mondo esterno (vedi progettazione).

8.1 Implementazione in JaCa

In seguito sono riportate le realizzazioni degli agenti di AGATA in JaCa.

8.1.1 Agenti Osservatori

```
associated_at_user(false).
counter(0).
creator(c).

!startWorking.

+!startWorking : true
  <- ?myTool(C);  // discover the tool
        ?myUserInterface(D); // discover the agent-user interface
    focus(C);
        focus(D);
        !find_my_name;
        !tell_creator_im_free.
```

```
+my_name(Me)
        <- println("Nome agente ",Me).

+!find_my_name
 <-  .my_name(Me);
          +my_name(Me).

@tell_creator_im_free [atomic]
+!tell_creator_im_free:  creator(In) & my_name(Me)
        <-      .send(In,tell,agent_signal(Me)).

// ricezione di una richiesta dal creatore, per essere associato ad un utente
@new_user_for_u [atomic]
+new_user_for_u(UID) : associated_at_user(false) & creator(IN) & my_name(ME)
<-  .send(IN,untell,agent_signal(ME));
        +myUserId(UID);
        -+associated_at_user(true);
        println("Mi sono associato ad un nuovo utente, uid: ",UID);
        // fai presente che l'utente è ora associato e preleva tutte le richieste fatte fino a questo
momento dall'utente (a cui ti sei associato ora)
        get_all_requests(UID).



@want_buy_signal [atomic]
+want_buy_signal(UID,NAME,MYCOST,MYTIME) : myUserId(UID)
        <-      println("Utente ",UID," vuole comprare: ",NAME," con costo massimo=",MYCOST,
" entro tempo=", MYTIME);
                +want_buy(NAME,MYCOST,MYTIME).



@want_buy [atomic]
+want_buy(NAME,MYCOST,MYTIME) : myUserId(UID)
        <-      // se presente restituisci (tramite segnale, come se fosse una nuova lista creata) la
miglior lista tra quelle che sono già presenti nel sistema
                give_me_best_old_list(NAME, MYCOST, MYTIME).



@new_list_created_signal_already_registered [atomic]
```

```
+new_list_created_signal(LID,N,C,T) : myUserId(UID) & want_buy(N,CS1,TM1) &
registered_into_list(OLDLID,N,CS2,TM2,true) & T<=TM1 & C<CS2
        <-
        // toglilo dalla precedente lista
        remove_user_from_list(UID,OLDLID,RESPONSEREMOVE);
        // se la lista precedente non era chiusa posso rimuoverlo correttamente e aggiungerlo nella
nuova

add_user_to_list(UID,LID,RESPONSEREMOVE,RESPONSEADD,RESPONSEADDNOT);
   // se la nuova lista era aperta posso continuare
        -registered_into_list(OLDLID,N,CS2,TM2,true);
        +registered_into_list(OLDLID,N,CS2,TM2,RESPONSEADDNOT);
        +registered_into_list(LID,N,C,T,RESPONSEADD);
        println("Nuova lista creata, migliore della lista in cui sono registrato. Lista vecchia aperta:
",RESPONSEADD, " e lista nuova aperta:", RESPONSEREMOVE).


@new_list_created_signal_not_yet_registered [atomic]
+new_list_created_signal(LID,N,C,T) : myUserId(UID) & want_buy(N,CS1,TM1) & not
registered_into_list(_,N,_,_,true) & T<=TM1 & C<CS1
        <- // inserisci l'utente nella nuova lista
        // faccio una richiesta di inserimento in lista
        add_user_to_list(UID,LID,RESPONSE);
        +registered_into_list(LID,N,C,T,RESPONSE);
        println("Registrato nella nuova lista ",LID," Nome=",N," Costo=",C, " Tempo=", T, " : ",
RESPONSE).



// serve solo per eliminare inutili belief
+registered_into_list(LID,N,C,T,false)
        <- -registered_into_list(LID,N,C,T,false).


// una lista alla quale ero iscritto è stata chiusa con successo
@list_closed_signal [atomic]
+list_closed_signal(LID) : registered_into_list(LID,N,C,T,true) & want_buy(N,_,_)
        <- println("Lista ",LID, " chiusa, Oggetto :",N, " comprato al costo: ",C);
          -want_buy(N,_,_);
          -registered_into_list(LID,N,C,T,true);
          !check_other_actions_required.
```

```
// se non sono presenti altre azioni (want_buy), togli l'associazione tra l'utente e l'agente, per poter
// associare l'agente con un'altro utente (se richiesto)
@check_other_actions_required [atomic]
+!check_other_actions_required : not want_buy(_,_,_) & myUserId(UID) & creator(In) &
my_name(Me)
        <-      -myUserId(UID);
                -associated_at_user(true);
                +associated_at_user(false);
                remove_association(UID);
                .send(In,tell,agent_signal(Me)).



+!check_other_actions_required : want_buy(_,_,_)
        <-      println("sono presenti altre azioni...").



+?myTool(ListsId): true
  <- lookupArtifact("l0",ListsId);
     +myTool(ListsId).

-?myTool(ListsId): true
  <- .wait(10);
     ?myTool(ListsId).



+?myUserInterface(UserInterfaceId): true
  <- lookupArtifact("ui0",UserInterfaceId);
     +myUserInterface(UserInterfaceId).

-?myUserInterface(UserInterfaceId): true
  <- .wait(10);
     ?myUserInterface(UserInterfaceId).
```

8.1.2 Agente Creatore

```
Counter(0).

!create_and_use.

+!create_and_use : true
  <- !setupLists(Id);
        !setupUserInterface(IdUi);

        .my_name(Me);
        +my_name(Me);



        create_new_user("user1", "mypass1");
        create_new_user("user2", "mypass2");
        create_new_user("user3", "mypass3");

        create_new_list("tv sony",60, 10,20);
        create_new_list("tv sony",20, 10,9);
        create_new_list("tv sony bravia",20, 10,2);

    // use
        buy_new_item(1,"tv sony bravia",30,11);
        buy_new_item(1,"ipad apple",30,11);
        buy_new_item(2,"tv sony",70,11);
        buy_new_item(3,"tv sony bravia",40,11);
        buy_new_item(1,"tv sony",60,11);
        buy_new_item(3,"ipad apple",40,11);
        buy_new_item(2,"ipad apple",60,11);

    create_new_list("tv sony",6, 8,10);
        create_new_list("tv sony",7, 10,10);
        create_new_list("tv sony",5, 10,2);
```

```
        create_new_list("tv philips",500, 10,2);
        create_new_list("tv xxx",50, 10,3);
        create_new_list("ipad apple",2, 10,2);

        .wait(2000);
        create_new_list("ipad apple",2, 10,1);
        create_new_list("tv sony bravia",59, 9,6);
        create_new_list("tv sony bravia",10, 9,7);

        buy_new_item(1,"tv sony bravia 3d",30,11);
        buy_new_item(1,"ipad apple 3",30,11).

// ogni qual volta un agente si rende disponibile, poichè ha terminato il suo lavoro
// può essere associato ad un nuovo utente
@agent_1 [atomic]
+agent_signal(AID) : want_an_agent(UID)
 <- .send(AID,untell,new_user_for_u(_));
        println("agente ",AID," libero e c'è un utente ",UID," che aspetta");
        -want_an_agent(UID);
        .send(AID,tell,new_user_for_u(UID)).


@agent_2 [atomic]
+agent_signal(AID) : not want_an_agent(_)
 <- .send(AID,untell,new_user_for_u(_));
        println("Agente ",AID," libero ma nessuna richiesta da parte di un utente");
        +agent(AID).

// un utente ha bisogno di un agente, e c'è un agente libero -> associali
@agent_3 [atomic]
+want_an_agent_signal(UID) : agent(AID)
        <-      // ho trovato un agente per le mie operazioni
                println("Utente ",UID," ha bisogno di un agente, che è presente : agente ",AID);
                -agent(AID);
           // invia la richiesta di associarsi all'osservatore
                .send(AID,tell,new_user_for_u(UID));
                println("Inviata richiesta di associazione a agente ",AID).
```

```
// un utente ha bisogno di un agente, e non c'è un agente libero
@agent_5 [atomic]
+want_an_agent_signal(UID) : not agent(_)
        <- println("Utente ", UID," ha bisogno di un agente, ma non c'è ne uno disponibile ora...");
          +want_an_agent(UID).

// create the tool
+!setupLists(C): true
  <- makeArtifact("l0","c4jexamples.AgataLists",[],C);
  focus(C).



// create the tool
+!setupUserInterface(D): true
  <- makeArtifact("ui0","c4jexamples.AgentUserInterface",[],D);
  focus(D).
```

8.1.2 Artefatti

Si rimanda l'implementazione degli artefatti all'allegato di questa trattazione.

9. Screenshot del funzionamento

Nell'immagine seguente si può verificare il funzionamento generico di AGATA tramite l'utilizzo di JaCa.

Come si può vedere possiamo trovare tutte quelle operazioni viste in precedenza come le creazioni degli agenti, le creazioni degli utenti, le associazioni sulla base di richieste di utenti, l'inserimento in liste, la cancellazione da una lista per inserire l'utente in una lista migliore, la chiusura delle liste, etc etc..

Illustrazione 12: Particolare del funzionamento di AGATA tramite l'utilizzo di Jason + Cartago

10.Conclusioni e lavori futuri

Abbiamo visto come sia stato possibile costruire in maniera relativamente semplice e veloce, partendo dalle sole specifiche, il prototipo di sistema complesso come A.G.A.T.A che presentava caratteristiche tipiche dei sistemi multi agenti (MAS).

E' stato utilizzato un' approccio AI-based (Artificial Intelligence) per quanto riguarda la definizione del modello del dominio, visto che il problema presentava alcune peculiarità tipiche dei sistemi esperti (planning e monitoring). Da questa fase è emersa la necessità di definire due tipi di entità differenti, una proattiva e reattiva che mappasse le caratteristiche del nostro sistema esperto e l'altra che potesse realizzare alcune funzioni richeste dal sistema esperto stesso, oltre che a garantire l'integrità dei dati e la loro correttezza a fronte di accessi concorrenti. E' per questo motivo che abbiamo messo in gioco il metamodello ad Agenti ed Artefatti che si basa su delle entità (Agenti ed Artefatti appunto) che presentano le caratteristiche di cui abbiamo sentito la necessità nella costruzione del modello del dominio. Grazie a questa astrazione possiamo passare in maniera molto semplice e diretta alla fase di progettazione. Per quanto riguarda l'implementazione si sono utilizzate le tecnologie di Jason e Cartago (JaCa) poiché esse ci forniscono come entità di prima classe a livello di linguaggio, proprio gli Agenti ed Artefatti con cui abbiamo eseguito la progettazione. Per qesto motivo si è potuto procedere in maniera immediata e senza particolari problematiche alla realizzazione del nostro sistema.

Ovviamente dato che queste tecnologie sono relativamente nuove, si sono presentati alcuni problemi relativi ad aspetti implementativi che però non hanno minato la bontà del nostro percorso, visto che è stato sempre possibile utilizzare soluzioni alternative per far fronte a questi intoppi.

In futuro potrà essere interessante costruire la parte web per interfacciarsi con l'utente finale e anche una serie di web services che garantiscano a quest'ultimo di utilizzare il nostro sistema.