

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SECONDA FACOLTÀ DI INGEGNERIA
Corso di Laurea Specialistica in Ingegneria Informatica

UN'INFRASTRUTTURA A SPAZI DI TUPLE PER ECOSISTEMI DI SERVIZI PERVASIVI

Linguaggi e modelli computazionali L-S

Relatore:
Mirko Viroli

Presentata da:
Andrea Tosi

Correlatore:
Matteo Casadei

Sessione II
Anno Accademico 2009/2010

Alla mia famiglia...

Indice

Introduzione	vii
0.1 Scenario	vii
0.2 Pervasive Ecosystems	ix
0.2.1 Metafore fisiche	xi
0.2.2 Metafore chimiche	xv
0.2.3 Metafore biochimiche	xvii
0.2.4 Metafore sociali	xvii
0.3 SAPERE	xviii
0.4 Obiettivi	xx
 1 Requisiti dell'infrastruttura	 1
1.1 Spazi di tuple	3
1.1.1 Spazi di tuple biochimici	5
1.2 Primitive	7
 2 Piattaforme	 9
2.1 LIME	9
2.2 TOTA	13
2.3 EgoSpaces	14
2.4 TuCSoN	17
2.5 CarTAgO	19
 3 Design	 23
3.1 Funzionalità	25
3.1.1 Reaction Manager	28
3.2 Architettura di sistema	31
3.2.1 Interazione tra componenti	32
 4 Implementazione	 35
4.1 Progettazione UML	35
4.2 Prototipazione	38

4.2.1	Packages	38
4.2.2	Classi	39
4.3	Esempio applicativo	42
Conclusioni		51

Introduzione

0.1 Scenario

Le tecnologie wireless e mobile hanno cambiato profondamente le modalità di comunicazione all'interno della società. In molti luoghi pubblici ed in molte delle abitazioni è disponibile una connessione wireless e i nostri smart-phone sono sempre più prestanti, più simili a pc tascabili che a telefoni cellulari. Questo ricco ambiente tecnologico ha portato le persone a richiedere un grado di utilizzo della rete, e dei suoi servizi, sempre maggiore. Il Web risponde a questi bisogni introducendo social network, contenuti multimediali ed in generale il Web 2.0.

Ogni utente inietta (a volte senza rendersene conto) un alto numero di informazioni nella rete es. SMS, telefonate, ricerche online, social network, email, operazioni bancarie, pagamenti con carta di credito, acquisti online ecc. . . : tutte queste informazioni contribuiscono a formare una sorta di diario digitale che corrisponde sempre più alla cronaca della vita quotidiana. La crescente connettività apre inoltre grandi sfide tecnologiche soprattutto informatiche nell'ambito del pervasive computing, letteralmente "computazione pervasiva" è un modello di interazione uomo-macchina che prevede che la computazione, in un sistema informatico, si sposti all'interno degli oggetti che ci circondano: immaginiamo per esempio i servizi (bluetooth, wifi, gps, multimedia, . . .) che vengono integrati all'interno delle autovetture di ultima generazione.

Se portiamo questo modello all'estremo, non è difficile pensare ad uno scenario in cui il sistema è visto come un insieme altamente dinamico e non definito di servizi che possono essere utilizzati dall'utente dipendentemente dalla sua posizione nell'ambiente, dalle sue preferenze e dai suoi bisogni. Questo sistema dal punto di vista tecnologico può essere concepito come un'integrazione di più software diversi (servizi) che comunicano e si coordinano sfruttando un'infrastruttura comune, la quale è estremamente densa di informazioni, popolata di servizi general-purpose pervasivi, in grado di connettere dinamicamente i suoi componenti in base ai contesti informativi e fisici.

Questo scenario porta ad avere nuove sfide all'interno della ricerca ed a ridefinire i modelli e i framework esistenti.

Case Study: Slideshow service Oggigiorno siamo sempre più circondati di display e schermi sia indossabili sia installati negli ambienti urbani e di lavoro. Attualmente, tali periferiche vengono utilizzate per visualizzare informazioni statiche e sono controllate in maniera centralizzata indipendentemente dal contesto nel quale il display è inserito. Questo scenario può essere migliorato sia per gli utilizzatori sia per i fornitori rendendolo aperto e adattativo.

La visualizzazione delle informazioni dovrebbe essere dipendente dallo stato dell'ambiente, fisico e sociale, che circonda lo schermo (per esempio: non è il caso di proiettare pubblicità di alcolici nel caso in cui la maggior parte delle persone vicine allo schermo siano minorenni), inoltre le periferiche si dovrebbero coordinare tra di loro per offrire una comunicazione più personalizzata al fine di risolvere al meglio i bisogni degli utenti (es.: visualizzare su più schermi adiacenti una stessa informazione che interessa più persone).

Poichè lo scenario in questione è distribuito, si vuole inoltre che le informazioni sul funzionamento di tali schermi si diffondano autonomamente in rete, rendendo possibile una autoconfigurazione ed un'adattatività continua alle contingenze. In particolare sono gli utenti ad iniettare nel sistema le informazioni che vengono percepite ed interpretate dagli schermi che forniscono un servizio personalizzato.

Analizziamo ora in un caso concreto che vede l'utilizzo di una rete di display.

Immaginiamo di trovarci in una stazione ferroviaria e di percorrere il marciapiede tra un binario e l'altro. Di solito nelle grandi stazioni troviamo decine di megaschermi e display che proiettano partenze, arrivi e pubblicità di ogni genere. Supponiamo di essere in possesso di uno smartphone e che al suo interno siano memorizzate le nostre preferenze (appassionato di sport, film gialli, musica classica, ecc. . .).

Il dispositivo mobile al nostro passaggio davanti a uno di questi schermi (per esempio via bluetooth), comunica al sistema le nostre preferenze che vengono associate al servizio più idoneo il quale visualizza, sullo schermo, le informazioni più interessanti per noi.

Per questo caso specifico supponiamo che il servizio che meglio corrisponde alle nostre preferenze (in quel momento) sia uno slideshow viewer. Se le slides proiettate sono di nostro interesse ci fermeremo davanti ad uno degli schermi (o rallenteremo in modo da poter dare un'occhiata) per poi continuare nel nostro tragitto incrociandone altri. Lo schermo successivo sul nostro cammino dovrebbe coordinarsi con il primo in modo da non riprodurre sempre la stessa diapositiva, bensì aggiornarsi alla prossima slide.

Senza considerare l'apparato tecnologico per gestire un sistema di questo tipo (sistema gps, display intelligenti, smartphone, data-center per il contenimento dei dati, ecc. . .), cerchiamo di definire le funzionalità dell'infrastruttura software supponendo di utilizzare un pervasive ecosystem.

0.2 Pervasive Ecosystems

Per questa sezione si è fatto riferimento agli articoli [16] e [18].

Le informazioni che circolano all'interno del sistema descritto sopra sono molteplici ed eterogenee. L'infrastruttura dovrà dominare questa eterogeneità incapsulando meccanismi che permettano di indirizzare l'utente verso il servizio più idoneo ai suoi bisogni. Inutile sottolineare la necessità di apertura, interoperabilità e scalabilità di tale sistema in quanto, dovendo essere pervasivo e a continua disposizione degli utilizzatori, occorre immaginarlo dal funzionamento "eterno" quindi in continua evoluzione e disponibile 24-7.

Oggi le architetture SOA costituiscono un punto cardine per la realizzazione di sistemi di servizi.

Tali sistemi puntano a sfruttare un middleware che incapsuli diversi servizi utili ad associare alla richiesta l'attività desiderata: servizi di discovery per fornire una sorta di "elenco telefonico" con cui i diversi componenti ne possano rintracciare altri; servizi di contesto con cui i componenti possono reperire informazioni ambientali e quindi essere situati; servizi di orchestrazione che coordinano più attività al fine di realizzare una logica applicativa complessa.

Se si pensa di realizzare il caso di studio dei display utilizzando una SOA centralizzata occorre realizzare un unico middleware server che contenga tutte le informazioni e i servizi (informazioni sui display e pubblicità, servizi utente, sensori, ecc. . .). Ogni componente utente dovrà poi rivolgersi a questo server al fine di ottenere informazioni sui vicini (discovery) e informazioni ambientali (context). I dispositivi utente potranno essere indossabili e quindi mobili: questo aumenta largamente la necessità di mantenere in costante aggiornamento le informazioni di contesto e sui vicini e, quindi, il numero delle comunicazioni di rete tra i dispositivi. È chiaro come un approccio centralizzato non sia la miglior soluzione al problema.

Valutiamo ora la scelta di una SOA decentralizzata. Adottiamo dunque la stessa architettura vista precedentemente e distribuiamola su più nodi di rete (per esempio una per ogni display) in modo da distribuire il controllo dei servizi.

È chiaro che in questa maniera sfuma la possibilità di attribuire al singolo nodo politiche di coordinazione e comportamenti che riflettano le condizioni locali sia di contesto sia di vicinato. Utilizzando tale architettura è infatti impossibile ottenere una soluzione che si adatti alle caratteristiche spaziali del sistema. Il prodotto non è quindi adattativo in quanto non è altro che una riproduzione statica distribuita di un'architettura centralizzata.

Portando all'estremo la distribuzione otteniamo un insieme dinamico di componenti situati che si coordinano e agiscono su una rete di dispositivi mobili ed eterogenei. I servizi di discovery, context ed orchestration, sono a quel punto, all'interno di un unico servizio infrastrutturale che realizza poche semplici regole cablate che fungono da assiomi del sistema.

Tale sistema assomiglia molto ad un ecosistema in cui le entità sono situate, distribuite, adattative ed auto-organizzanti per definizione e nel quale esistono regole che influenzano il comportamento degli individui.

L'architettura di riferimento è quella di figura 1.

Al livello più alto troviamo i prosumers che accedono all'infrastruttura sia per usare e consumare informazioni e servizi sia per introdurre nuove funzionalità o dispositivi. Si promuove, quindi, l'apertura verso nuovi utenti e dispositivi che possono inserirsi o abbandonare il sistema in qualsiasi momento. Segue la descrizione dei vari livelli.

Species: tutti i componenti e le informazioni del sistema (informazioni contestuali, risorse di rete, dati persistenti e temporanei, utenti, servizi, dispositivi, ecc. . .) sono rappresentati uniformemente come specie. Tali entità rappresentano gli individui della popolazione dell'ecosistema (fisicamente situati) in grado di lasciare ed unirsi ad esso, vivere e morire.

Space: questo livello si occupa della gestione del ciclo di vita degli individui e della loro locazione spaziale in una precisa porzione del sistema. In questa maniera le azioni sono contestualizzate e dipendono dalle informazioni spaziali locali. Sostanzialmente questo livello costituisce un piccolo substrato software che funge da middleware che abilita la comunicazione tra i diversi dispositivi del sistema e tra individui. Questi ultimi, potranno sfruttare delle API che consentiranno di iniettare informazioni, o percepire eventi, con le quali essi possano coordinarsi al fine di realizzare una particolare business logic. Dal punto di vista del middleware occorre che si forniscano meccanismi di trasparenza che assorbano eventuali cambiamenti riguardanti arrivo o partenza di individui.

Il middleware dovrà essere fortemente event e data oriented, distribuito sulla rete, accessibile con modalità location-dependent da parte degli individui.

Eco-Law: l'ecosistema è regolato da un insieme di eco-laws che permettono di esplicitare le modalità con cui gli individui interagiscono tra di loro e con lo spazio. Tali leggi, pur essendo le stesse per ogni specifica implementazione, hanno effetti diversi sulle diverse specie. Per realizzare tale comportamento occorre introdurre una qualche descrizione semantica degli individui e delle informazioni di contesto che ne permetta una rappresentazione uniforme. Mentre nelle SOA il servizio di discovery è affidato ad un servizio interno al middleware, il nostro obiettivo è di realizzare la stessa funzionalità utilizzando quelle "annotazioni semantiche". Esse infatti, oltre a fornire una rappresentazione uniforme della conoscenza, fanno sì che, attraverso delle regole astratte (eco-laws) e con qualche tipo di meccanismo di matching, si riesca a determinare dinamicamente quali individui siano coinvolti nella regola, e come questi debbano essere manipolati. Si riuscirà così a realizzare politiche context-aware adattative avendo un unico set di regole.

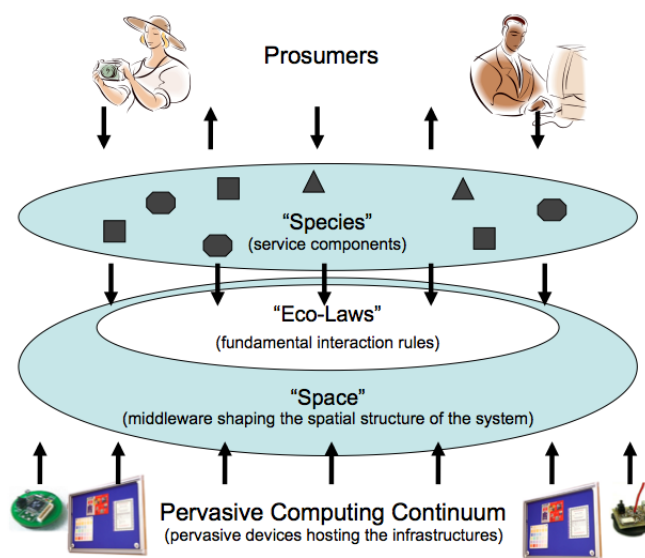


Figura 1: Pervasive Ecosystem

Data l'architettura appena introdotta è utile, a questo punto, definire quali tipi di metafore ecosistemiche è possibile rappresentare. Occorre precisare che determinare quali esse siano e, soprattutto, quali siano gli schemi di interazione che concretamente le rappresentino, è una sfida chiave del pervasive computing dei prossimi anni.

Tutte le metafore di seguito descritte sono per loro natura situate, adattative, evolutive ed auto-organizzanti. Considerando l'implementazione di tali modelli, occorre precisare che sarà necessario ridurre i dettagli in modo da diminuirne la complessità, esternando unicamente i pattern auto-organizzanti ed adattativi che si vogliono sfruttare.

0.2.1 Metafore fisiche

Queste considerano le specie come particelle computazionali, immerse in un ambiente insieme ad altre, mosse da un campo computazionale virtuale che coordina l'interazione. Le attività associate a tali particelle, possono essere viste come agenti reattivi e sono guidate da leggi che ne determinano i movimenti (come avviene la propagazione e la reazione ai cambiamenti nella topologia del campo o del sistema). Le particelle cambiano il loro stato a seconda della natura del campo che percepiscono: tale campo può essere semplice (euclideo) o virtuale/sociale: in entrambi i casi mappato con la topologia fisica della rete. Questa metafora dà luogo alla metafora di coordinazione detta *field based coordination* che può essere così riassunta:

- Le interazioni tra agenti sono abilitate e mediate da un campo virtuale propaga-

to nel sistema dagli agenti stessi o dall'ambiente in cui essi vivono (tipicamente con il supporto di un middleware adeguato, vedi TOTA [10]). Questi campi sono dei tipi di strutture dati distribuiti, contenenti informazioni riguardanti gli altri agenti o sul contesto in generale.

- Il valore locale del campo può essere acceduto localmente dagli agenti. Questo può essere usato come acquisizione di informazioni locali e permette che gli agenti siano coordinabili in maniera semplice (es. basta che seguano le linee di campo).
- Apertura e dinamismo dell'ambiente vengono prese in considerazione in ogni cambiamento del sistema, poiché ad esempio se un nuovo agente entra nel sistema, i valori del campo verranno opportunamente modificati e verranno create nuove linee.
- Questo feedback (lettura valori e modifica del campo) porta anche alla self-organization e self-adaptation in maniera globale.

Di seguito sono riportati alcuni esempi in cui un approccio Field-Based può essere vantaggioso.

Routing in Mobile Ad Hoc and Sensor Networks Le reti wireless ad hoc e le reti di sensori wireless sono degli esempi rappresentativi per sistemi di larga scala altamente dinamici. Queste reti stanno ricevendo sempre più attenzioni poiché forniscono un'infrastruttura per una comunicazione pervasiva a basso costo. In questo scenario il modo con cui si affronta il problema del routing e del controllo della rete è fondamentale: la configurazione manuale e precisa per ogni minimo cambiamento dell'ambiente non è possibile.

Da una parte la mancanza di un punto di controllo centralizzato impone ai vari nodi di cooperare e auto-organizzarsi per trovare un corretto algoritmo di routing, dall'altro gli stessi nodi potrebbero essere mobili (es. PDA): questo porterebbe a continui cambiamenti nella topologia di rete.

Un approccio field-based alla soluzione di questi problemi è quello del "Gradient Routing" [13] e "Direct Diffusion" [6].

Il Gradient Routing è un algoritmo di routing concepito per le reti mobili. In sintesi, quando A vuole mandare un messaggio a B inonda la rete di strutture dati field-like contenenti l'id sorgente (A), il messaggio e il numero di hop fatti per raggiungere quella destinazione. Quando B si vede recapitare il messaggio, non diffonde nuovamente il messaggio, ma risponde ad A seguendo il gradiente maggiore, ovvero ripercorrendo gli hop con tragitto minimo. Se però ci fossero dei cambiamenti nella topologia di rete o delle disconnessioni, la rete verrebbe nuovamente inondata da strutture dati field-like in modo tale da ricostruire la topologia corretta.

La Direct Diffusion è un algoritmo di routing proposto nelle reti di sensori. Il problema è come raccogliere le informazioni percepite da sensori in una vasta rete di sensori sparsi nell'ambiente.

Una workstation ai bordi della rete inserisce una struttura dati field-like che esprime l'interesse per un particolare set di eventi (query field). I sensori sono in grado di far pervenire alla workstation le informazioni tramite un meccanismo simile a quello descritto prima per il Gradient Routing. Inoltre la workstation può rinforzare alcune parti del campo applicando una sorta di Reinforcement Learning Algorithm. Questo fa sì che i sensori che percepiscono informazioni rilevanti continuino a farlo, mentre gli altri possono dimenticarsi del query field (risparmiando batteria, ad esempio).

Artificial World Altro caso in cui si utilizza la field-based coordination, è quello della simulazione di mondi virtuali.

Il classico esempio in cui si evidenzia un tale approccio è quello di The Sims, un videogame. In questo gioco lo scopo principale è simulare una vita reale creando il proprio personaggio, la propria famiglia, la propria casa ecc. . . L'utente guida il proprio avatar con le interazioni classiche dei videogames. Se si clicca su un oggetto del mondo (es. TV) questo propone un insieme di possibili azioni che il giocatore può compiere su tale oggetto (es. spegni/accendi TV, cambia canale). Contemporaneamente il giocatore deve soddisfare le necessità dei personaggi virtuali (es. bisogno di dormire, mangiare, ecc. . .).

Mentre le attività del personaggio sono condotte dal giocatore, vi sono un alto numero di altri personaggi (guidati dal computer) che coesistono nella realtà virtuale e svolgono una loro vita autonoma. Per le attività di questi personaggi computer-driven, the Sims adotta un approccio basato su campi computazionali chiamati "happiness landscapes" che, sparsi nell'ambiente virtuale, guidano i loro movimenti [7].

In particolare, i personaggi non-player si muovono autonomamente nella città con lo scopo di aumentare il loro grado di benessere seguendo il gradiente di specifici campi computazionali. Se ad esempio un personaggio ha fame, egli seguirà l'happiness landscapes che lo porta in prossimità di cibo (es frigorifero, ristorante). Dopo aver mangiato il personaggio seguirà un altro field che lo condurrà in altri luoghi e gli farà compiere altre azioni.

I campi sono propagati su aree diverse a seconda delle funzionalità che si intendono realizzare. Ad esempio, il campo del "letto" che soddisfa il bisogno "dormire" si propagherà anche al di fuori della casa in cui il personaggio vive, mentre il campo "ping pong" si propagherà solo nelle vicinanze del gioco. Ecco che questo simula un comportamento abbastanza realistico anche dei personaggi non-player, e consente di modellare i loro movimenti in maniera semplice ma efficace.

Prey and Predator Per catturare una preda un gruppo di lupi tende ad agire in maniera coordinata per circondarla impedendole di scappare. I lupi cacciano in branco riproducendo comportamenti di *swarm intelligence* [1]. Semplicemente quello che avviene è che, nel tentativo di catturare la preda, i lupi tendono a stare lontani gli uni dagli altri: questo fa sì che la pecora venga circondata.

È possibile modellare questo comportamento mediante un approccio field-based, assumendo che i lupi e le pecore generino un campo come mostrato in figura 2.

La preda tende a seguire le linee di campo generate dai lupi in maniera decrescente,

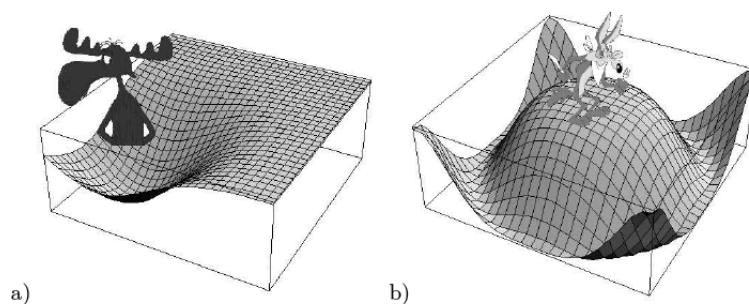


Figura 2: a) Campo preda b) Campo predatore

tentando così di scappare dai suoi predatori; i lupi tendono, allo stesso modo, a seguire le linee di campo discendenti sia del campo degli altri lupi sia del campo della preda. In questo modo quello che si ottiene è l'accerchiamento della pecora e la sua cattura.

Bird Flocking Gli uccelli in uno stormo virano in maniera coordinata e non collidono l'un l'altro seguendo un semplice algoritmo swarm.

Questo comportamento può essere spiegato assumendo che ogni uccello tenti di mantenere una certa distanza dai vicini ed a viaggiare alla stessa velocità di questi. Lo stormo è una struttura nella quale ogni uccello risponde costantemente e simultaneamente ad ogni modifica.

Sebbene ogni uccello percepisca solo informazioni sui vicini, esso può rispondere a cambiamenti generati da altri in maniera transitiva: lo stormo così esibisce un comportamento coordinato globale.

Per modellare questa strategia di coordinazione con un approccio field-based basta considerare che ogni uccello generi un campo come descritto in figura 3 e che questo campo sia aggiornato costantemente a seconda dei cambiamenti generati dallo stormo. Per mantenere la stessa distanza gli uni dagli altri, ogni uccello segue le linee discendenti del campo generato dagli altri uccelli: la forma del campo totale costringe gli animali a stare vicini, quasi fossero in una griglia regolare. Il movi-

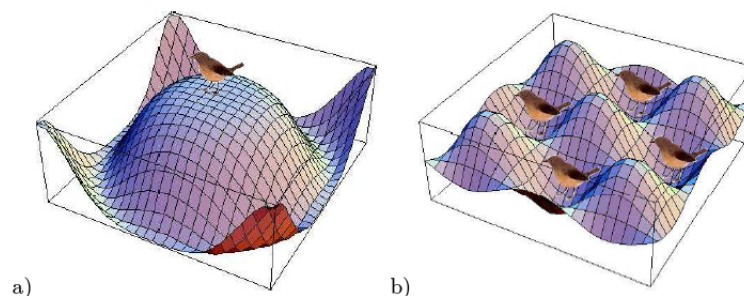


Figura 3: a) Campo di un singolo b) Campo stormo

mento di un volatile e il conseguente cambio del campo forzerà gli altri a muoversi opportunamente.

Per concludere, le metafore fisiche, non solo quella field based, sono state ampiamente studiate per le loro caratteristiche di auto-organizzazione spaziale e di grande scalabilità. Un difetto da evidenziare è la difficoltà con cui gestiscono la diversità delle specie e gli aspetti evolutivi del sistema. Le eco-laws infatti dovrebbero essere numerose al fine di gestire la diversità dei campi che si vogliono mettere in gioco e le conseguenti regole di propagazione: questo aumenta la complessità del modello.

0.2.2 Metafore chimiche

Queste considerano le specie come atomi/molecole computazionali (che ancora una volta possono essere viste come agenti reattivi) arricchite con descrizioni semantiche utili per guidare le reazioni chimiche. Esse sono descritte dalle eco-laws che sfrutteranno qualche sorta di matching tra individui al fine di determinare le dinamiche di esecuzione della reazione.

Utilizzando queste metafore il sistema è visto come un insieme di spazi confinati all'interno dei quali è presente una "soluzione" in cui le sostanze possono combinarsi e diffondersi.

Dal punto di vista concreto ad ogni individuo è associata una concentrazione e la dinamica di una reazione si basa sul lavoro di Gillespie [5] (vedi [16]). In questo caso una reazione chimica può essere vista come una formula $X + Y \xrightarrow{r} Z$ a cui è associato un rate markoviano r . La situazione rappresentata è quella in cui: quando una molecola X si combina con una molecola Y si genera una molecola Z (che sostituisce le due originali). La frequenza reale con cui tale reazione scatta è proporzionale alla concentrazione di X e di Y ed a r .

Con questo modello si possono sfruttare meccanismi di feedback negativo e positivo tipici dei sistemi chimici per realizzare dinamiche di funzionamento complesse. In

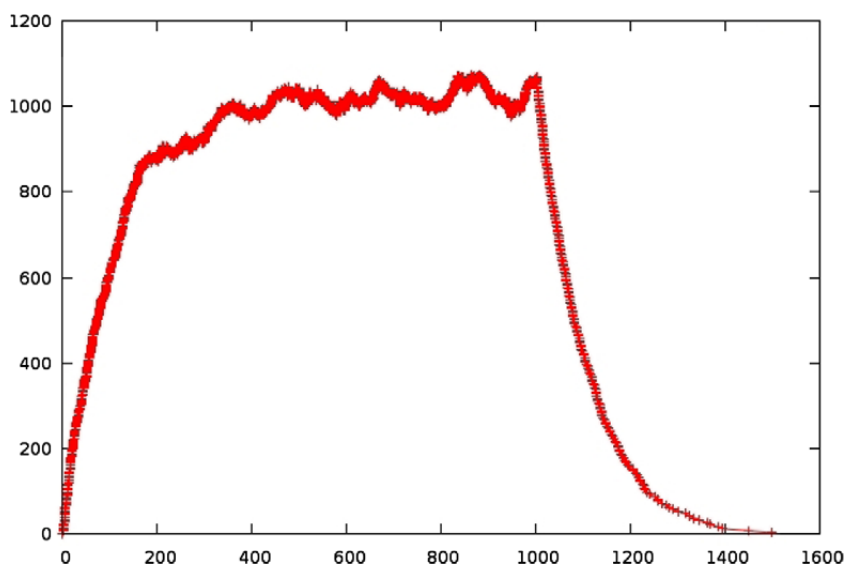


Figura 4: Fading-feedback X:time Y:service level

particolare la concentrazione delle sostanze chimiche può fungere da livello di attività per gli individui. Di seguito si riportano esempi di reazioni e di comportamento ottenuto.

Consideriamo due reazioni:

- $Service \xrightarrow{r1} 0$
- $ServiceFeedback \xrightarrow{r2} Service + ServiceFeedBack$

La prima reazione è la regola che rappresenta il decadimento radioattivo: ogni unità di sostanza *Service* svanisce con un certo rate. La seconda reazione rappresenta un feedback positivo (per esempio un utilizzatore entusiasta del servizio) che funge da catalizzatore chimico (ovviamente tale feedback può essere prodotto solo se la sostanza *Service* è presente nel sistema quindi se il servizio è disponibile all'utilizzo). In figura 4 si può vedere l'andamento della concentrazione di *Service* nel caso in cui $r1 = 0.01$ e $r2 = 10$. Al tempo 0 la sostanza *ServiceFeedback* ha concentrazione 1, mentre al tempo 10 essa viene rimossa ed il servizio decade esponenzialmente.

Esistono molti altri esempi di possibile comportamento: competizione tra servizi, composizione, diffusione, etc. . .

Concludendo, le metafore chimiche possono riprodurre comportamenti auto-organizzanti quali l'aggregazione dinamica di servizi e informazioni. Come nella chimica reale, anche in questo caso è possibile esprimere un insieme molto vasto di comportamenti utilizzando un set minimo di regole. Concretamente questo significa che si possono generare o modificare un numero crescente di annotazioni semantiche senza modificare il set base di eco-laws; inoltre, è possibile sfruttare i meccanismi di catalizzazione

di una reazione al fine di ingegnerizzare la dinamica di funzionamento del sistema. Una limitazione di tali metafore è l'obbligo di esprimere attività e comportamenti unicamente a livello della singola soluzione (quindi localmente) o tipicamente all'interno del vicinato: questo rende difficoltoso realizzare comportamenti di auto-organizzazione spaziale complessi.

0.2.3 Metafore biochimiche

In questo caso ci si focalizza sui sistemi biologici ed in particolare sui singoli organismi o colonie di organismi (come i formicai). Le specie o gli individui sono entità con uno scopo estremamente semplice che produce però effetti emergenti complessi. La comunicazione tra i componenti della colonia avviene tramite informazioni depositate dagli individui nell'ambiente che tipicamente evaporano con il passare del tempo (es. ferormone). Anche in questo caso il mondo in cui vivono questi organismi è un ambiente computazionale mappato sulla topologia fisica del sistema.

Contrariamente alle metafore fisiche, in questo caso gli individui non reagiscono passivamente alla presenza o meno di ferormone, ma sono in grado di modificarne la concentrazione a seconda del proprio stato interno: questo porta a dover definire delle eco-laws che esprimano questa dinamica.

Concretamente la metafora biologica può essere ottenuta modificando quella fisica ed in particolare la natura del campo di forza. È infatti sufficiente introdurre il concetto di evaporazione in modo da rendere variabile la forza del campo con il passare del tempo.

Le metafore biologiche, se analizzate in base alle tipologie di pattern auto-organizzanti forniti, possono essere considerate come l'evoluzione di quelle fisiche. Vi è infatti la possibilità di poter specificare vari tipi di ferormone e quindi vari tipi di comportamenti globali pur mantenendo limitato il numero di regole. Tuttavia esistono ancora molti pattern auto-organizzanti non pienamente riproducibili (es. morfogenesi); inoltre, non è facile interagire in maniera controllata (es. catalizzatore per la metafora chimica) con questi tipi di sistemi.

0.2.4 Metafore sociali

Queste considerano gli individui come animali che interagiscono all'interno di un ecosistema. Ogni animale può essere visto come un'entità complessa goal-oriented, appartenente ad una specie, in cerca di cibo per vivere e prosperare. Ovviamente le eco-laws dovranno esprimere le regole con cui un animale tramuta in cibo un individuo, come avviene la riproduzione e come interagisce con altri individui. Come nella metafora chimica, anche in questo caso, il mondo è diviso in compartimenti all'interno dei quali gli individui vivono e attraverso i quali si spostano.

La metafora sociale è utile per riprodurre comportamenti spazialmente auto-organizzanti

e supportano naturalmente la biodiversità, in quanto riproducono esattamente meccanismi che sulla terra si verificano da milioni di anni.

Come nel caso biochimico anche in questi sistemi è difficile introdurre delle logiche di controllo che permettano dall'esterno di gestire esattamente la dinamica del sistema.

0.3 SAPERE

Alla luce di quanto finora esposto, questa tesi si inserisce nell'ambito degli ecosistemi di servizi ed in particolare si affianca al progetto europeo SAPERE a cui partecipa il team di ricerca della Seconda Facoltà di Ingegneria di Cesena.

SAPERE si ispira agli ecosistemi naturali, in quanto si è convinti che la dinamicità e la decentralizzazione, nelle reti pervasive future, costituiranno concetti fondamentali per modellare i servizi, dati e dispositivi. Si adottano quindi metafore naturali direttamente all'interno dell'infrastruttura al fine di riprodurre comportamenti intelligenti globali che regolino la dinamica del sistema.

SAPERE usa un'architettura non layered simile a quella indicata in figura 1 (vedi figura 5). Il substrato incapsula le regole naturali che controllano la comunicazione e la manipolazione delle informazioni: rappresenta il luogo in cui individui di specie differenti interagiscono e si aggregano in modo da soddisfare i loro bisogni individuali al fine di provvedere alla sostenibilità dell'intero sistema.

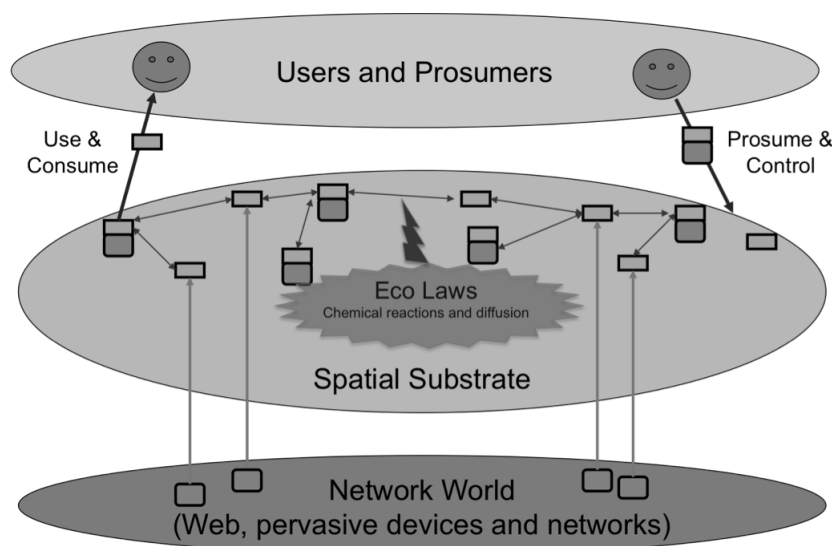


Figura 5: SAPERE High-level overview

SAPERE ha come scopo operativo quello di definire un framework altamente innovativo che prende ispirazione dagli ecosistemi naturali al fine di gestire un sistema

di servizi pervasivi: si opera una fondamentale ridefinizione dei sistemi odierni, delle architetture e degli algoritmi ad essi associati.

SAPERE si concentra su situadness, sel-awareness, adaptivity, comportamento autonomico, metafore ecosistemiche ed in generale vuole dare una risposta a questa domanda:

“Is it possible to conceive of a radically new way of modelling integrated pervasive computing services and their supporting environments, such that the apparently diverse issues of enabling situadness, divesity, evolvability, autonomicity, holistic adaptivity and advance forms of awariness, can all be uniformly addressed once and for all?”

Per le entità interne al substrato si adotta un modello di rappresentazione comune delle informazioni. Tutte le entità che vivono nell’ecosistema hanno associata una rappresentazione semantica. Questo è l’ingrediente fondamentale per ottenere interazioni dinamiche non supervisionate tra i componenti.

Le “annotazioni semantiche” sono definite *Live Semantic Annotations* (LSA). Ogni LSA descrive (sfruttando una certa rappresentazione semantica) il servizio e il dato a cui è associata, non rappresenta il corpo del servizio, bensì una sua descrizione sintetica. L’infrastruttura combina gli LSA al fine di realizzare un’interazione semantica tra componenti diversi e tra servizi e fruitori. In generale l’LSA è usata per (vedi figura 6):

- descrivere la struttura dell’individuo come informazione fissata a design time;
- gestire i continui cambiamenti in un contesto olistico di manipolazione delle informazioni associate ad un individuo;
- essere strettamente associata all’entità che descrivono ed in grado di riflettere le sue informazioni contestuali.

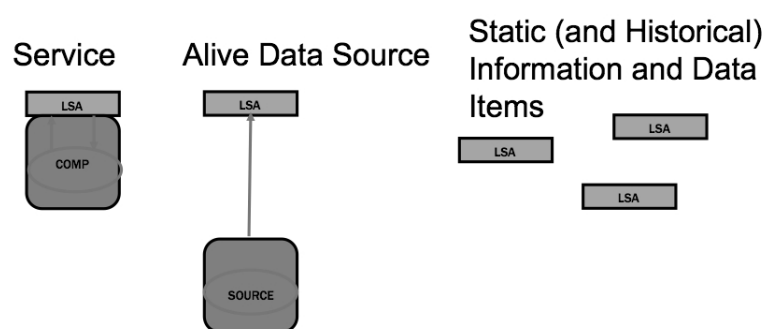


Figura 6: LSA

In SAPERE ogni servizio è in grado di iniettare LSA all’interno del sistema ed osservare come questi mutano o vengono aggregati. In figura 6 viene indicato grafi-

camente la funzione di un LSA. Questo può essere associato ad un componente (es. servizio), ad una sorgente dati (es. stream audio-video) oppure rappresentare dati statici utilizzati per la coordinazione delle entità software che utilizzano il sistema. Tali entità sono dette *prosumers* ossia contemporaneamente producers e consumers di informazioni (LSA).

In SAPERE vengono individuate le soluzioni ai principali problemi degli scenari pervasivi:

- il substrato (l'infrastruttura) del sistema deve riflettere naturalmente il modello spaziale ed astrarre dalla particolare tecnologia di rete in modo da essere altamente scalabile ed interoperabile;
- gestire in maniera distribuita i dati e i servizi è l'unico modo per ottenere un sistema adattativo ed auto-organizzante. Le nuove funzionalità dovranno essere "assorbite" dall'infrastruttura in maniera uniforme ed automatica come sostanze interessate in una reazione chimica. Questo porta ad avere auto-sostenibilità ed evoluzione;
- i dati e i componenti all'interno del sistema devono essere in grado di auto-aggregarsi al bisogno in maniera dinamica e flessibile, così da produrre conoscenza globale sul contesto e sulla particolare situazione;

Tali soluzioni sono concretizzate con l'utilizzo degli LSA e delle eco-laws. Queste ultime guidano le dinamiche interne all'ecosistema. In SAPERE sono considerate come una sorta di reazioni chimiche virtuali che interessano gli LSA.

L'idea è di rinforzare la spazialità introducendo meccanismi di diffusione e di rappresentare dinamiche goal-oriented per la composizione di dati e servizi.

Gli LSA interessati in una eco-laws potranno essere visti come reagenti o come prodotti che verranno selezionati e composti seguendo meccanismi di matching semantico.

0.4 Obiettivi

Obiettivo di questa tesi è quello di progettare e realizzare un prototipo di infrastruttura da collocarsi a livello "Spatial Substrate" dell'architettura di alto livello del framework SAPERE (figura 5). In generale le scelte progettuali ed implementative saranno fatte rispettando la visione di SAPERE e le sue linee guida.

La tesi si articolerà come segue: in principio si definiranno i requisiti di tale infrastruttura e i modelli che più corrispondono alle nostre esigenze (cap.1); in seguito si passerà alla descrizione di un insieme di infrastrutture pervasive a spazi di tuple in modo da analizzare lo stato dell'arte ed eventualmente adottarne una per l'implementazione (cap. 2); successivamente si illustrerà la progettazione definendo

l'architettura (cap. 3); per ultimo si descriverà l'implementazione ed il funzionamento del prototipo tramite un esempio applicativo (cap. 4).

Capitolo 1

Requisiti dell'infrastruttura

Il caso di studio appena descritto e lo scenario applicativo portano a dover ridefinire i modelli e i sistemi presenti oggi. Il progetto SAPERE (ed in piccola parte questa tesi) mira ad ottenere un sistema che possiede le seguenti caratteristiche.

Situadeness Essendo la computazione altamente distribuita ed il grado di interconnessione tra i dispositivi crescente, per poter definire e successivamente progettare e realizzare uno di questi sistemi occorre definire una nozione di *contesto*.

Come già detto l'user experience, ed in generale i servizi a disposizione, cambiano a seconda della posizione e delle esigenze del cliente. Il contesto rappresenta un confine virtuale che permette di definire un sottoinsieme di informazioni e dispositivi a disposizione dell'utilizzatore. I dati interni al sistema si distribuiranno quindi spazialmente all'interno della rete in modo da essere localmente associati ad un certo dispositivo o ad un certo servizio. Le informazioni sono a quel punto considerate *contestuali* associate quindi ad un certo contesto; tali informazioni possono anche essere riferite all'ambiente fisico che circonda l'utente rendendo quindi l'infrastruttura situata. Concludendo, occorre quindi introdurre la spazialità e la situadeness direttamente come requisiti infrastrutturali.

Lightweight Il middleware in questione non ha grandi requisiti in termini di potenza computazionale richiesta per l'esecuzione. L'infrastruttura deve poter estendersi a dispositivi quali: smartphone, netbook, notebook, dispositivi embedded, ecc... La mole ridotta consente una facile manutenzione e soprattutto permette di riservare le risorse dell'hardware ospitante alle esigenze della business logic (servizi).

L'infrastruttura è quindi considerata come un core che fornisce i meccanismi necessari per l'esecuzione delle eco-laws e la comunicazione tra servizi.

Prosumption e diversità Le reti, che diventeranno ben presto pervasive, non saranno ancora per molto utilizzate solo passivamente dalle persone. Gli utenti vor-

ranno partecipare attivamente alla costruzione e alla modifica della rete, a seconda dei loro bisogni. Questo porta a non avere più una definizione distinta tra utenti e servizi, bensì esisteranno peer (che definiamo prosumers) che sfrutteranno servizi e ne offriranno altri, in generale che consumeranno e produrranno informazioni.

Evolvability Il sistema che si sta definendo non è statico e dal comportamento definito a design-time, bensì muta ed evolve con il passare del tempo e modifica le sue funzionalità in base alle esigenze degli utenti ed ai servizi presenti. Non può essere inteso come un sistema software tradizionale in quanto è in grado di mantenere delle caratteristiche e delle proprietà globali pur avendo un insieme di funzionalità altamente variabile. Si parla infatti di sistemi “eternal beta”, ossia di cui non esiste una versione stabile consolidata.

Reactivity Considerando che all'interno dell'infrastruttura devono aver luogo reazioni che modificano le informazioni seguendo particolari eco-laws, il middleware dovrà essere *reattivo*, ossia in grado di percepire eventi di vario genere ed agire di conseguenza. Questa funzionalità è indispensabile se si vuole ottenere un'infrastruttura *adattativa* che muta e si evolve in base alle contingenze.

In sintesi la reattività è una caratteristica che, inserita a livello infrastrutturale, permette di ottenere molti vantaggi non funzionali che potranno essere i punti di forza per la diffusione di tale sistema.

Uniform representation: LSA I dati interni al sistema sono rappresentati in maniera uniforme in modo da sfruttare meccanismi per manipolarli, che siano generali a livello di infrastruttura.

L'infrastruttura deve astrarre dalla particolare rappresentazione semantica di un LSA e mettere in campo comportamenti astratti che dovranno essere poi specializzati a seconda delle particolari scelte tecnologiche.

Disponibilità 24-7 Un sistema di questo tipo deve essere pronto in ogni momento ad accogliere le richieste degli utenti e l'inserimento di nuove informazioni e servizi. I sistemi odierni che supportano questo carico non hanno comportamenti auto-organizzanti che permettono un'auto-analisi dei guasti o un'auto-organizzazione che permetta di isolare una falla, per cui la disponibilità del sistema varia in base alla reattività del team di manutenzione.

Il sistema che si vuole realizzare dovrà riprodurre comportamenti auto-organizzanti che individuino il guasto e lo aggirino senza minare le funzionalità dei servizi. È impensabile, ovviamente, eliminare l'intervento umano periodico ma si vuole ridurlo al minimo e renderlo meno costoso.

Self-* I servizi interni al sistema devono essere in grado, spontaneamente ed in modo opportunistico, di comunicare tra di loro e di utilizzare informazioni che vengano aggregate in maniera automatica in base ai bisogni. Tali comportamenti non possono essere raggiunti semplicemente dando la possibilità ai componenti di conoscere fatti isolati, bensì occorre gestire politiche che comprendano tutta la situazione del sistema.

Le metafore naturali aiutano in questo senso fornendoci funzionalità e proprietà globali ottenibili da semplici interazioni locali.

Esplicitando le eco-laws come “leggi fondanti dell’ecosistema” è possibile concretizzare le metafore fisiche, chimiche, ecc... in sistemi in grado di auto-organizzarsi e auto-sostenersi. I comportamenti self-* sono *emergenti* ossia rintracciabili osservando il sistema dall’esterno. Essi sono alla base in un sistema in continua evoluzione in cui variano i dati scambiati ed i servizi erogati, ma non i meccanismi e le leggi con cui l’infrastruttura manipola le informazioni.

Il middleware dovrà quindi fornire l’astrazione necessaria ad esplicitare le eco-laws come reazioni dichiarative associate ad un certo contesto ed implementare il giusto set di azioni elementari con cui manipolare le informazioni.

1.1 Spazi di tuple

Avendo a disposizione un case study che fornisce una visione piuttosto concreta del sistema, non è facile chiarire se il percorso di definizione dei requisiti è bottom-up o top-down: in generale sarà un compromesso tra i due (detto “a tenaglia”).

La topologia del sistema descritta nel case study lascia immaginare che esista una rete di dispositivi dotati di capacità computazionale (schermi LCD, smartphones, ecc...) interconnessi attraverso molteplici interfacce di comunicazione (bluetooth, wi-fi, ethernet, ecc...). È naturale pensare ad un’architettura logica che rispecchi, almeno topologicamente, quella fisica, quindi l’infrastruttura software sarà anch’essa una rete di nodi che contengono LSA.

I servizi che utilizzano questa rete sono entità attive che possono spostarsi da un nodo all’altro, in base alle loro esigenze, servendo lungo il loro cammino le richieste degli utenti. Ogni nodo può essere visto come uno spazio a cui è associata la logica di gestione delle eco-laws; questo fa sì che tutta la rete abbia un proprio comportamento self-* in quanto le eco-laws espresse a livello del singolo hanno effetti emergenti a livello di rete.

Le eco-laws sono reazioni che scattano autonomamente all’interno del nodo in base al contenuto informativo di esso (quindi in base alla presenza o assenza di dati). Le politiche di coordinazione sono interamente affidate all’infrastruttura, mentre ai servizi è delegata la business logic.

Poiché l’infrastruttura è light-weight, le operazioni servizi-middleware devono essere

minime di numero e ridotte nella loro complessità, pur offrendo la massima espressività.

Fin da ora, supponiamo l'LSA dotato di stato modificabile e associato ad un codice identificativo univoco. Definiamo inoltre il contesto di un servizio rappresentato dal singolo nodo. Questo porta a dover definire come rendere un nodo situated (non oggetto di questa tesi).

La context-awareness del servizio si ottiene semplicemente abilitandolo alla lettura, modifica, inserimento e rimozione di LSA da un nodo. È facile effettuare nello stesso modo anche context-recognition.

È già intuibile che uno dei fattori di forza del sistema è l'estrema semplicità di utilizzo. Sfruttando pochi pattern di interazione si vogliono ottenere comportamenti globali complessi.

Pensando alla metafora di comunicazione che meglio concretizza l'insieme dei requisiti si ricade sul modello a spazi.

Tale approccio è infatti pensato per la comunicazione di entità attive e offre flessibilità e semplicità di utilizzo.

Uno spazio di tuple¹ offre disaccoppiamento in tempo e in spazio ossia costituisce un luogo computazionale in cui i processi comunicano depositando e prelevando informazioni. Lo spazio fornisce la persistenza che abilita la comunicazione di processi anche se questi accedono allo spazio in momenti (disaccoppiamento in tempo) o in luoghi (disaccoppiamento in spazio) diversi. In questa materia è fondante il lavoro svolto da Galanter ed il suo linguaggio di comunicazione Linda [4]. Le sue caratteristiche principali sono:

Comunicazione generativa: fino a quando la tupla non viene esplicitamente rimossa dallo spazio, le tuple generate dai coordinabili (agenti) hanno un'esistenza propria e possono essere lette e modificate da tutti.

Accesso associativo: si accede alle tuple presenti nello spazio tramite il loro contenuto e la loro struttura, piuttosto che tramite nome, indirizzo o locazione.

Semantica sospensiva: le operazioni compiute sullo spazio potrebbero sospendersi per l'indisponibilità di una tupla oppure ripartire nel caso in cui la tupla cercata sia presente.

Indeterminazione: se più di una tupla unifica con il pattern specificato, se ne sceglie una non deterministicamente.

Tornando agli ecosistemi di servizi, l'approccio a spazi è da preferirsi in quanto è quello che più corrisponde alla topologia fisica del sistema: più TS interconnessi possono facilmente rappresentare la struttura a rete descritta in precedenza. Inoltre

¹D'ora in poi abbreviato con TS (tuple space(s))

un LSA può essere facilmente ricondotto ad una tupla il cui contenuto è rappresentato semanticamente, mentre un TS può essere la parte di un nodo che contiene gli LSA, in quanto definisce implicitamente un “compartimento” simile a quello che troviamo nelle metafore chimiche o sociali. Tuttavia occorre analizzare più profondamente quale preciso linguaggio di coordinazione adottare.

Linda fornisce un set minimo di primitive che facilita la sincronizzazione dei processi e che fornisce le funzionalità per leggere, rimuovere ed inserire informazioni nello spazio. Tuttavia un TS non fornisce nessun tipo di reattività interna in quanto è visto come un entità passiva che incapsula meccanismi bloccanti e non per l'accesso ai dati.

Un ecosistema pervasivo deve invece sfruttare, direttamente a livello architetturale, funzionalità reattive al fine di gestire le eco-laws. Non ultimo, nei requisiti è chiaro che un servizio deve poter osservare l'andamento delle informazioni di un nodo: questo comportamento può essere visto come una lettura non distruttiva di tuple da un TS. Ciò non è del tutto vero.

Usando le primitive Linda le uniche operazioni di lettura a disposizione sono la `rd` e la `rdp` (consideriamo anche le operazioni bulk associate) che leggono una tupla, rispettivamente con semantica bloccante e non. Entrambe le operazioni accettano come parametro un template per l'accesso associativo.

Supponiamo uno scenario in cui un servizio inietta un LSA e desidera monitorizzarne lo stato. L'approccio Linda in questo caso ha due grossi svantaggi:

1. il processo deve conoscere necessariamente il contenuto dell'LSA e, nel caso, in che modo si trasforma per fornire il giusto template all'operazione di lettura. Questo nel nostro caso è impossibile, in quanto un servizio non conosce il contenuto delle eco-law e quindi non ha modo di capire in che modo varierà il contenuto di un LSA;
2. dal punto di vista della sincronizzazione servizio-TS non viene fornita alcuna modalità per la notifica asincrona: o il processo rimane in attesa della comparsa della tupla specifica, o interroga in maniera non bloccante lo spazio finché la lettura va a buon fine.

Esistono diversi modelli che estendono Linda, uno dei quali è un interessante punto di partenza per la definizione di un nostro modello di coordinazione.

1.1.1 Spazi di tuple biochimici

Questa sezione vuole essere una sintesi del modello proposto in [15].

Qui una tupla è sempre associata ad un valore che ha la stessa valenza della concentrazione per una sostanza chimica. Questo valore viene utilizzato al fine di capire

quanto quella tupla influenza la politica di coordinazione globale. Come in una reazione chimica reale, tuple con concentrazione più bassa saranno meno interessate nella reazione, al contrario quelle con alta concentrazione saranno utilizzate più frequentemente.

Topologicamente il modello si articola in un insieme di TS collegati che rappresentano esattamente i compartimenti introdotti nelle metafore chimiche (sez. 0.2). Ad ogni TS sono associate delle reazioni che modificano la concentrazione delle tuple nel tempo *esattamente* come in una reazione chimica reale [5].

Oltre a tale meccanismo il modello è arricchito dell'operazione di diffusione: il processo tramite il quale una sostanza attraversa i confini di un compartimento chimico.

In questo modo si collega lo schema dei TS collegati in rete ad uno spazio biologico diviso in compartimenti comunicanti come tessuti, insieme di cellule, embrioni o altri organismi.

I TS biochimici basano il loro funzionamento interno su un particolare meta-modello stocastico che si basa sull'algoritmo di Gillespie[5] in cui si afferma che la dinamica di una reazione chimica può essere modellata e simulata attraverso una CTMC (Continuous-Time Markov Chain).

Una reazione è vista come composta da reagenti prodotti e rate (già descritto nella sez. 0.2). L'algoritmo, ad intervalli predefiniti di tempo, sceglie, stocasticamente ed in maniera proporzionale al prodotto delle concentrazioni dei reagenti ed al rate della reazione, una reazione da eseguire, la esegue e determina l'istante di risveglio successivo.

Questo sistema è tipicamente usato per effettuare esperimenti tramite la simulazione dell'andamento di un insieme di sostanze ed un insieme di reazioni: essendo un modello stocastico i risultati non sono sempre gli stessi.

La semantica operativa descritta in [15] estende la normale semantica Linda introducendo la gestione delle concentrazioni e della diffusione.

Concludendo possiamo prendere questo modello come quello che più si avvicina ai requisiti, tuttavia è fortemente legato alla metafora chimica ed in particolare sfrutta intensivamente l'algoritmo di Gillespie al fine di gestire le tuple interne al medium di coordinazione come sostanze chimiche interessate in una reazione dalle dinamiche reali. Il nostro approccio, e quello di SAPERE, è più generale in quanto mira a realizzare un'infrastruttura che possa rappresentare più tipi di meccanismi naturali non solo quelli chimici.

Prendiamo comunque atto che il modello che più si avvicina alla struttura fisica del sistema ed a quella logica di ecosistema è una rete di compartimenti che possono essere visti come TS.

È chiaro come Linda costituisca un vincolo in termini di espressività delle interazioni tra i servizi ed il medium definiamo perciò il nostro set di primitive.

1.2 Primitive

Segue il set di operazioni servizi-middleware:

- **inject**: inietta un nuovo LSA all'interno dell'infrastruttura.
- **observeById**: osserva un LSA. Questa primitiva non è unicamente una pura lettura ma la richiesta di notificare eventuali cambiamenti di un LSA. È auspicabile che questa operazione abbia diverse semantiche di esecuzione (bloccante o non bloccante), in modo da poter aumentare lo spettro dei pattern di comunicazione.
- **updateById**: modifica il contenuto di un particolare LSA.
- **removeById**: rimuove un particolare LSA.

Come è chiaro l'unità informativa che viene scambiata e manipolata nell'infrastruttura è l'LSA.

Il set di operazioni appena descritto deve fornire la massima espressività e costituisce una parte del sistema definita a design-time che non verrà più modificata. Ovviamente il processo che definisce tali primitive è stato top-down, ossia guidato da bisogni applicativi. Si è infatti pensato alle operazioni basilari da compiere sull'infrastruttura al fine di ottenere servizi dalle caratteristiche espresse precedentemente (situadness, ractivity, ecc. . .).

L'idea di fondo è affidare completamente la gestione della parte di comunicazione e coordinazione al middleware ed alle sue eco-laws. I servizi possono influenzare attivamente il comportamento del sistema modificandone il contenuto informativo e, nello stesso, tempo possono subirne i cambiamenti osservandone l'andamento.

È chiaro dalle primitive che questo modello di comunicazione è lontano da quello offerto da Linda [4]. La differenza fondamentale è la mancanza dello schema associativo, per il recupero delle informazioni.

All'interno del sistema che si sta formando, infatti, non si accede agli LSA in base al contenuto, bensì tramite un identificativo.

Considerando che ogni servizio, almeno all'atto della creazione, non ha una propria conoscenza interna pre-installata dello stato informativo del nodo, potenzialmente è in grado di conoscere e quindi modificare e rimuovere solo gli LSA che inietta. Questo porta ad uno schema di interazione più rigido e sicuro di quello offerto da Linda, in cui ogni processo può effettuare una **in** attraverso la quale distrugge deliberatamente informazioni depositate da altri processi.

Nel modello presentato si intende obbligare un servizio ad interagire sempre con il sistema previo inserimento di un LSA. Tale informazione può rappresentare una richiesta, l'identità del servizio, la descrizione del suo comportamento, di un dispositivo

e, più in generale, LSA di ogni genere. A quel punto l'individuo che ha effettuato l'**inject** dovrà necessariamente osservare (primitiva **observeById**) le informazioni appena inserite in modo da percepire come queste vengano modificate dall'infrastruttura. Tali modifiche sono la conseguenza dell'applicazione delle eco-laws e possono essere viste come la risposta ad un richiesta o l'associazione servizio-cliente.

Linda offre uno schema utile alla sincronizzazione dei processi sfruttando semantiche bloccanti (e non) delle primitive; tuttavia non costituisce un sistema che evolve autonomamente nel tempo ma varia unicamente a seconda delle modifiche apportate dai processi.

Concludendo l'insieme delle primitive descritte offre ai servizi dei meccanismi estremamente semplici, sia per interagire attivamente con lo spazio sia per osservarne il comportamento; inoltre, implicitamente si spinge l'utilizzatore del middleware a seguire uno schema di interazione che facilita la gestione delle politiche di coordinazione interne all'infrastruttura e, di conseguenza, l'applicazione estensiva delle eco-laws.

Capitolo 2

Piattaforme

Segue una descrizione delle principali piattaforme (Java-based) a spazi di tuple progettate per sistemi pervasivi.

Non ci si soffermerà sui dettagli sintattici, operazionali o implementativi, bensì si riporteranno i concetti fondamentali e le metafore fornite al fine di ottenere un insieme di funzionalità da confrontare con i requisiti del sistema.

2.1 LIME

È una piattaforma progettata per essere utilizzata in scenari in cui sia le entità computazionali (*agenti*) che i dispositivi fisici (*host*) sono mobili. All'interno di tali scenari l'interconnessione dei dispositivi crea reti ad hoc che rendono possibile l'interazione di più alto livello tra gli agenti. La mobilità di dispositivi e software fa sì che la comunicazione sia quindi *un fenomeno occasionale e temporaneo*.

LIME non fornisce trasparenza alla mobilità e alla distribuzione, bensì eleva tali concetti ad entità di prima classe del middleware.

Linda offre la possibilità di coordinare più processi che accedono ad uno stesso spazio condiviso persistente, globale e creato staticamente. Mantenere tali proprietà in presenza di mobilità fisica è complicato, in quanto la connessione non può sempre essere garantita; a volte è l'utente stesso a richiedere la disconnessione (per esempio per motivi di risparmio energetico dei dispositivi).

Per far fronte a questa problematica, *host* (e agenti) mobili sono considerati entità indipendenti che occasionalmente formano comunità transienti. La mobilità richiede vincoli più laschi sui TS e una riconfigurazione continua dei loro contenuti.

In LIME gli *host* sono considerati container mobili di agenti a cui forniscono la connettività.

La metafora fondamentale è il *transient tuple space* (TS transiente). Ogni agente è associato ai propri TS e vi accede tramite l'ITS (*interface tuple space*). Nel momento

in cui due o più agenti si trovano a condividere lo stesso host (o host connessi), i loro TS si fondono rendendo possibile la condivisione delle informazioni al loro interno e la conseguente comunicazione. Gli agenti interagiscono con lo spazio mediante le usuali operazioni Linda.

Il transient tuple space si crea a livello di host (*host-level tuple space*) o a livello di hosts interconnessi (*federated tuple space*) (vedi figura 2.1).

Quando un agente (mobile) lascia il transient tuple space, porta con sé i TS che vengono eliminati dallo spazio condiviso.

Analizzando lo spazio delle interazioni offerte dalla piattaforma, risulta impossibile riprodurre alcune dinamiche tipiche di un TS Linda[12] (vedi figura 2.2).

A tal proposito ad ogni tupla è associata una destinazione e sono state introdotte nuove primitive (estensioni di Linda). Così facendo nasce il concetto di tuple *misplaced* (malposizionate), ossia tuple in cui la destinazione e la locazione corrente non corrispondono.

In generale l'infrastruttura di LIME cerca di minimizzare il numero di tali tuple facendole migrare all'occorrenza da un TS all'altro.

Le nuove primitive inserite danno la possibilità di specificare la destinazione della tupla emessa nel caso di una *out*, o la locazione corrente e la destinazione nel caso di *in* o *read* (per maggiori informazioni sulla sintassi si veda [12]). Si noti che locazione corrente e destinazione, a seconda dei casi, possono rappresentare ID di host o TS e non sono campi obbligatori.

LIME non offre nessuna garanzia che una certa tupla giunga a destinazione, poichè ciò dipende dalla presenza simultanea sullo stesso host (o su host interconnessi) sia dell'agente che contiene la tupla misplaced sia dell'agente che contiene il TS destinazione: tale evento in un ambiente altamente dinamico potrebbe anche non accadere.

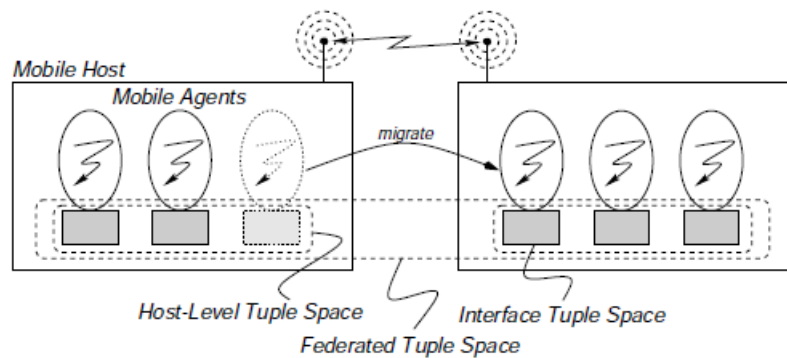


Figura 2.1: Host and Federated transient TS

LIME offre la possibilità di associare ad un TS delle *reactions* ossia di eseguire codice al verificarsi di particolari eventi (gestiti internamente come tuple in un par-

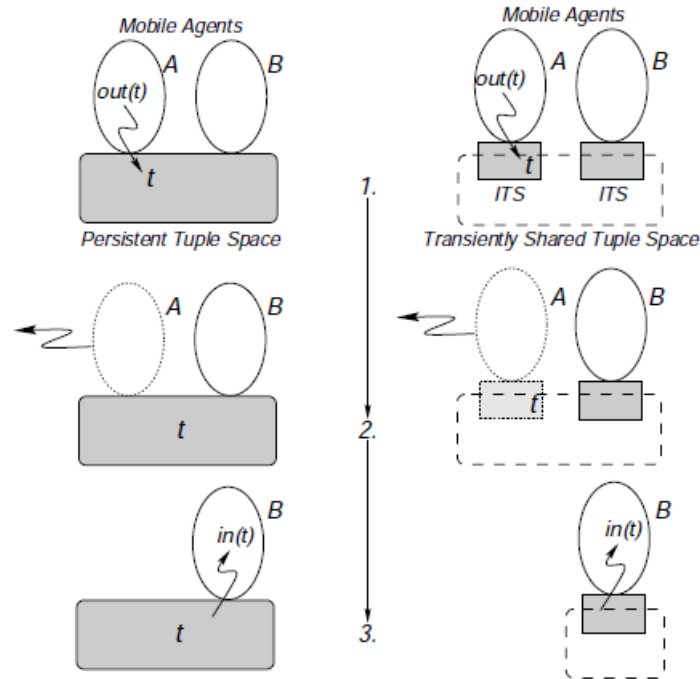


Figura 2.2: Persistent vs. Transient TS

icolare TS read-only a livello di agente chiamato *LimeSystem Tuple Space*).

Per catturare tali eventi LIME usa un agente stazionario che continuamente monitorizza il sistema. All'occorrenza, l'agente trasforma l'evento in una tupla che inserisce nel proprio LimeSystemITS. Tramite il Federated Tuple Space, generato dalla condivisione di ogni LimeSystemITS, ogni agente è in grado così di conoscere queste informazioni.

La reazione ad un evento, di fatto, può essere realizzata mediante la $in()$, poiché ha una semantica sospensiva. Questo introduce vari problemi:

Semanticamente: la tupla è scelta non deterministicamente e un altro agente potrebbe essere interessato allo stesso evento.

Performance: causa semantica sospensiva, ogni evento deve essere monitorato da un thread diverso: cosa spesso impraticabile.

Linda Problem: per come è definito il modello Linda, le tuple sono controllate attivamente dagli agenti. Per la reazione quello che si deve invece ottenere è una notifica, da parte dello Spazio di Tuple, dell'avvenuta occorrenza di un evento.

Encapsulating: l'incapsulamento nella reazione di un evento nella sua stessa definizione è ciò che si vuole ottenere.

Lime introduce perciò la nozione di *reactive statement* che ha una forma del tipo `T.reactTo(s,p)` dove `s` è un frammento di codice contenente non-reactive che viene eseguito in maniera *atomica* quando viene trovata una tupla che unifica con `p`. Dopo ogni non reactive statement: una reazione viene scelta in maniera non deterministica, la sua guardia viene valutata e si reagisce di conseguenza. Questo viene ripetuto finché ci sono delle reazioni abilitate.

Successivamente si torna alla normale esecuzione del programma. Gli statement reattivi sono eseguiti come se appartenessero ad un altro reactive program che viene eseguito in certi punti fissi, dopo ogni non reactive statement. Questo offre un adeguato livello di reattività, poiché tutte le reazioni registrate sono eseguite prima di ogni altro statement degli agenti co-locali, inclusi gli statement di engagement/disengagement[12].

Sebbene `p` ed `s` possano avere una forma arbitraria, di fatto la loro struttura è forzata da alcune considerazioni implementative: si dovrebbe realizzare un linguaggio di più alto livello o addentrarsi nella macchina virtuale java per verificare la veridicità dello statement `p`.

LIME permette di specificare in `p` solo il pattern che può unificare con una tupla presente nello spazio di tuple dove lo statement è eseguito. In questo modo l'unico statement che scatena un evento è l'inserimento di una tupla nello spazio.

Le medesime considerazioni sono da farsi riguardo a `s` che potrebbe contenere qualunque statement non reattivo, comprese operazioni sullo spazio di tuple.

In pratica, le reazioni sono eseguite in un thread di controllo separato appartenente al Lime System.

Le operazioni bloccanti sono di fatto vietate in quanto potrebbero bloccare l'esecuzione di tutte le altre. Inoltre, dal punto di vista delle performance, `s` dovrebbe essere mantenuto il più piccolo possibile per permettere un'elaborazione più veloce delle reazioni.

Ad esempio, permettendo di realizzare una `out(t)` si potrebbe causare un loop infinito (`t` potrebbe matchare con un `p` specificato nelle reazioni di altri agenti).

Sebbene vi siano notevoli problematiche, LIME permette l'esecuzione di operazioni sullo spazio di tuple in `s`, lasciando di fatto al programmatore il compito di non creare situazioni come quelle sopra citate.

La `reactTo()` (*strong reaction*, sincrona) comprende anche la locazione corrente e finale. Ha una forma del tipo `T.reactTo[w,l](s,p)` dove `w` (locazione corrente) deve sempre essere specificato, sia esso un identificativo di un agente mobile, o di un host.

Non sono permessi statement reattivi nel Federated TS (questo porterebbe all'esecuzione di transazioni distribuite su tutti gli host, cosa impraticabile). Perciò LIME

offre la possibilità di reagire in maniera asincrona alla disponibilità di una tupla in un ITS remoto fornendo la $T.\text{upon}(s, p)$ (*weak reaction*, asincrona).

Viene qui di seguito spiegato il funzionamento della upon ; la presenza di un Event Tuple Space(ETS) è solo chiarificatrice.

1. $\text{upon}(s, p)$ viene invocata sullo spazio di λ , la $\text{reactsTo}(s', p)$ è registrata su tutti gli ITS condivisi con λ e anche una $\text{reactsTo}(s, p')$ è registrata sull' ETS_λ .
2. quando una $\text{out}(t)$, con t che si unifica con p , viene eseguita da μ , la $\text{reactsTo}(s', p)$ viene scatenata e s' eseguito. s' esegue una $\text{out}(t')$ su ETS_μ con $t' = (t + \text{extra system info che permette di bindare una event tupla con l'agente che sta eseguendo la upon})$.
3. Una azione asincrona muove t' dall' ETS_μ all' ETS_λ . (senza garanzia che ciò venga eseguito subito t' sia presente su ETS_μ).
4. quando t' è presente sull' ETS_λ allora la $\text{reactsTo}(s, t')$ viene scatenata.

La trattazione sulle reazioni è stata fatta in riguardo ad agenti che risiedono sullo stesso host, ma un agente può registrarsi anche per reagire ad alcuni eventi di un host diverso rispetto a quello in cui risiede. Ciò, però, non ha effetto fino a che quell'agente non migri sull'host specificato. Questo è dovuto all'assioma di atomicità della reazione, non possibile per agenti risiedenti su host diversi.

2.2 TOTA

Tuple On Air è un'infrastruttura di coordinazione Java-based basata sul concetto di *field based coordination*[10]. Questo tipo di coordinazione prende spunto dalla metafora fisica di un campo di forza che attira a sé o respinge particelle cariche.

La coordinazione field based può potenzialmente essere implementata come un overlay-network su di un middleware che fornisca il supporto alla memorizzazione dei dati, comunicazione e notifica degli eventi.

TOTA rappresenta il campo di forza con un insieme di tuple distribuite sulla rete. Ogni tupla non è associata ad un nodo specifico, ma viene iniettata ed in maniera automatica si propaga e si diffonde seguendo un particolare pattern.

Il campo può quindi essere visto come una struttura dati distribuita, composta dall'insieme delle varie tuple.

Per realizzare questo schema TOTA si compone di una serie di peer interconnessi ognuno dei quali gestisce una propria versione locale del middleware. Sostanzialmente si forma una rete di TS attraverso i quali le tuple si diffondono seguendo particolari regole.

Una tupla è vista come coppia (C, P, M) dove:

C: è la regola di propagazione della tupla nello spazio e

P: è il contenuto della tupla (valore del campo in quel punto).

M: sono le regole di mantenimento, determinano come la tupla debba reagire in seguito ad eventi che accadono nell'ambiente. Questi eventi possono essere di varia natura: semplici time-alarm, come nel caso dell'evaporazione del ferormone, oppure possono essere relativi a cambiamenti della topologia di rete, il che è di fondamentale importanza per una consistenza e coerenza delle informazioni ambientali.

La tupla si propaga da un nodo all'altro lasciando traccia del suo passaggio. Tale propagazione è effettuata dall'infrastruttura in maniera automatica.

Gli agenti sfruttano l'insieme delle tuple come valori del campo in un certo spazio, realizzando pattern di movimento auto-organizzanti simili a quelli sperimentati da un insieme di particelle cariche immerse in un campo di forza.

Il middleware fornisce, lato agente, le seguenti primitive:

inject: inietta una nuova tupla nel sistema;

read, readOneHop, keyrd, keyrdOneHop: operazioni per la lettura (sia per contenuto sia per chiave) locali e remote (a distanza uno dal nodo locale);

subscribe, unsubscribe: consente di registrarsi o cancellarsi dall'interfaccia di gestione eventi del middleware;

delete: rimuove una tupla dal nodo;

store, move: permettono di memorizzare la tupla nel TS locale o di propagarla ai vicini.

Per garantire l'adattatività TOTA (vedi figura 2.3) consente di gestire gli eventi ed in particolare quelli relativi alla modifica della topologia della rete. Ogni tupla nel sistema, infatti, non è realizzata come thread java bensì come oggetto passivo che reagisce, innescando la diffusione, a tali eventi.

2.3 EgoSpaces

Le difficoltà che si incontrano nella realizzazione di applicazioni mobile può essere generalizzata nel bisogno di gestire una grande quantità di dati distribuiti riguardanti il contesto. Questo ha portato allo sviluppo di EgoSpaces [8]: un middleware che fornisce la trasparenza alla mobilità, alla distribuzione ed alla connettività estemporanea. Tecnicamente sviluppato su LIME eredita da esso le metafore principali

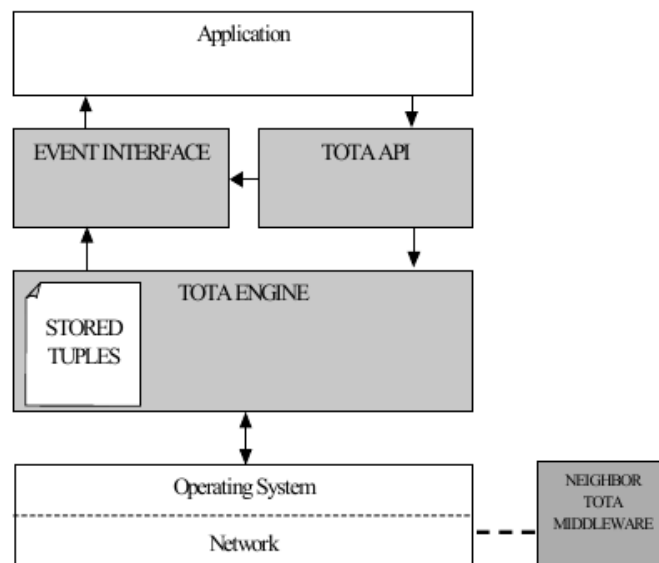


Figura 2.3: TOTA architecture

e le estende con nuovi concetti.

Il middleware è costruito su una nuova concezione di contesto che permette alle singole applicazioni di limitare la quantità di informazioni con cui hanno a che fare. Ogni applicazione può definire più contesti che riflettono le diverse esigenze; ogni contesto può interessare molteplici sorgenti dati. Il middleware sgrava lo sviluppatore da dover gestire connessioni e disconnessioni di rete all'ordine del giorno in scenari mobile.

I concetti fondamentali su cui si basa la sono:

- il contesto dovrebbe essere generalizzato in modo da consentire alle diverse applicazioni di accedere utilizzando modalità comuni ad informazioni di tipo totalmente diverso (posizione spaziale, banda di trasmissione disponibile, ecc...);
- applicazioni diverse devono sfruttare un contesto personalizzato sui loro bisogni;
- nessuna applicazione deve avere una conoscenza intrinseca di default, bensì ottenerla al bisogno dalla rete;
- se l'ambiente applicativo è vasto occorre introdurre logiche di interazione distribuite con il contesto;

- tecniche di programmazione di alto livello permettono allo sviluppatore di non concentrarsi su problematiche troppo specifiche e di non perdere mai di vista la business logic.

In EgoSpaces ogni host ha una locazione fisica ed un profilo che descrive le sue proprietà. Un agente ha anch'esso un profilo e una locazione logica (che indica l'host su cui l'agente sta venendo eseguito). Ogni agente può richiedere al middleware di *costruire il proprio contesto informativo* personale chiamata *view*[8]. Ogni view è utilizzata solo dall'agente che ne ha richiesto la creazione ed è il frutto della fusione dei TS degli altri agenti (come in LIME).

L'interfaccia d'uso di una view contiene le classiche primitive Linda. Inoltre ogni host/agente ha associato un TS read-only chiamato *profile* contenente le informazioni specifiche di quel host/agente.

I vincoli associati ad una view sono relativi a:

network: EgoSpaces si basa su SICC[9] che fornisce l'astrazione di rete vista come albero contenente i soli hosts che rispettano vincoli sulla distanza (la quale può essere calcolata con logiche application-specific). Sfruttando tali vincoli è possibile estendere il contesto oltre l'immediato confine del host (nodi vicini a distanza uno).

Per utilizzare SICC l'agente deve implementare: i meccanismi per calcolare il peso di un collegamento, la funzione di costo usata per determinare la lunghezza di un path, un limite superiore a tale funzione.

host: riguardano proprietà logiche del host. Esempi: id host, identità dell'utente che sta utilizzando l'host, proprietà sul servizio che l'host ospita. Concretamente visto che il profilo di un host è memorizzato come tupla, per calcolare questi vincoli si effettua un pattern matching sulle tuple di profilo.

agent: in questo caso i vincoli riguardano le informazioni contenute nel profilo dell'agente che partecipa alla view. Tali informazioni possono riguardare le funzioni dell'agente o le caratteristiche dell'entità che quell'agente rappresenta.

data: espressi come template sul contenuto dei vari TS impegnati nella view. Tali template possono essere anche utilizzati in associazione con meta-dati usati all'atto del matching, in modo da aumentare l'espressività del filtro con cui vengono selezionate le informazioni.

In figura 2.4 si ha una visione grafica della costruzione di una view da parte di un agente posizionato in un'autovettura X in movimento nel traffico.

Ad ogni view possono essere associate delle *transactions*, un insieme di azioni eseguite come unica operazione atomica, oltre ai seguenti tipi di *reactions*:

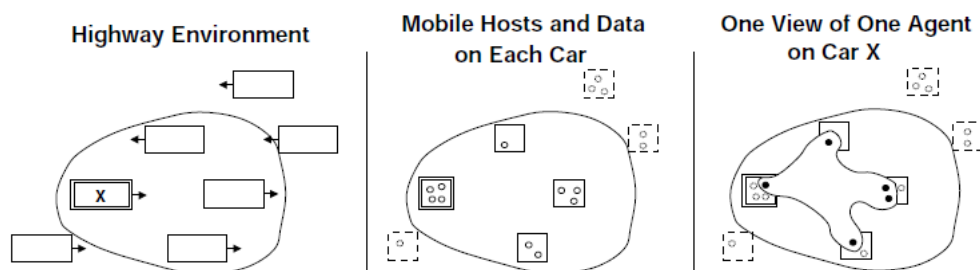


Figura 2.4: View construction

basic: si esegue codice se esiste una tupla che corrisponde ad un certo template (come in LIME);

extended: permette di associare alla presenza di una tupla una transazione in modo che l'intera reazione sia eseguita in un unico blocco atomico. La tupla scatenante deve essere all'interno del TS locale;

followed: simile alla *upon* di LIME coinvolge più agenti remoti. La cattura dell'evento (visto come tupla) e l'esecuzione del codice della reazione sono gestiti separatamente come blocchi atomici.

2.4 TuCSoN

Letteralmente Tuple Centres Over Network: rappresenta un'infrastruttura per la comunicazione di processi/agenti tramite *centri di tuple* distribuiti in rete [3].

Un centro di tuple (TC) può essere considerato un TS programmabile in grado di reagire ad eventi. Un esempio di "linguaggio di programmazione per TC" (utilizzato tutt'ora all'interno di TuCSoN) è ReSpecT[11].

TuCSoN gestisce tuple logiche viste come termini della logica del prim'ordine. Tuple e reazioni sono fatti e predicati di una teoria prolog: questo permette di sfruttare il pattern matching e l'unificazione del linguaggio.

L'architettura interna del sistema è a livelli (vedi figura 2.5):

Agent Coordination Context Layer: regola le interazioni tra agenti e middleware e incapsula politiche di sicurezza;

TC Layer: si occupa della gestione interna dei TC;

Topology/Fabric Layer: incapsula le funzionalità di comunicazione tra TC. La rete di TC è vista come composta da *place* raggruppati in *domain*, per ogni domain vi è un TC *gateway* che regola le comunicazioni inter-domain.

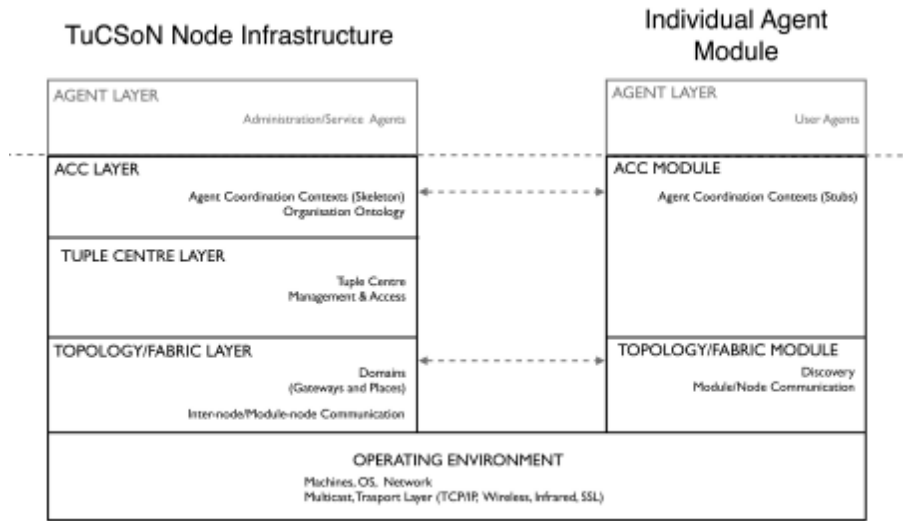


Figura 2.5: TuCSoN architecture

Come indicato in figura 2.5 il middleware prescinde dalla particolare tecnologia utilizzata per la costruzione degli agenti (agent-layer) disaccoppiando, attraverso l'ACC, l'agente dal medium di coordinazione.

Ogni agente interagisce con l'infrastruttura tramite un ACC Stub che fa riferimento ad uno skeleton lato middleware. Lo stub è un'interfaccia d'uso linda-like, le cui operazioni sono scelte in base all'identità dell'agente applicando politiche di sicurezza.

Come già detto l'infrastruttura è distribuita in rete, ogni TC è associato ad un nodo ed ha un id univoco, in particolare un TC può essere denotato in maniera assoluta usando la sintassi *tc@node* dove *node* rappresenta l'indirizzo internet del nodo, o in maniera relativa unicamente con l'id del TC: in questo si specifica che si vuole accedere ad un TC interno al contesto locale del nodo.

Si può specificare il comando *tc@node?op(tuple)* indicando che si vuole eseguire l'operazione *op* che interessa la tupla *tuple* sul TC *tc* interno al nodo *node*.

TuCSoN gestisce i nodi dell'infrastruttura basandosi sull'attuale struttura di internet.

Si definiscono *gateway* quei nodi capaci di autenticare e di autorizzare agenti all'accesso ad altri nodi ed ai loro TC. Ogni *gateway* costituisce la radice di un albero di nodi che eredita una certa politica di sicurezza per l'accesso.

Considerando il modello di coordinazione offerto da TuCSoN [3], ogni gateway ed ogni nodo contiene il proprio set di TC e le modalità che ne regolano l'accesso da parte degli agenti, ogni gateway inoltre contiene un TC di default che funge da media di comunicazione standard per l'accesso alle informazioni riguardanti altri TC (nodi del sotto-albero) ed altri gateway.

TupleCentres Un TC può essere visto come un TS a cui viene associato un comportamento reattivo programmabile. Come in altre architetture TS reattive (es. MARS [2]) viene definito un modello computazionale in cui il TC è in grado di eseguire operazioni sul TS interno in base agli eventi che percepisce. Tali eventi sono le operazioni (Linda) che i processi, o agenti, effettuano sull'infrastruttura. In particolare ReSpecT [11] definisce un proprio modello computazionale per TC in cui una reazione associata ad un TC è composta da una tripletta:

testa: evento scatenante. I possibili eventi esprimibili in questo campo sono definiti nella tassonomia presente in [11];

guardia: un'espressione booleana che attiva o meno l'esecuzione del corpo;

corpo: insieme di operazioni Linda che l'infrastruttura esegue sul TS interno (vedi modello computazionale).

TuCSon eredita lo stesso modello computazionale di ReSpecT per i TC ed è progettato per astrarre dallo specifico linguaggio delle reazioni.

2.5 CarTAgO

Concepito come meta-modello per la costruzione di MAS (Multi-Agent Systems) è tutt'ora un framework (java-based) per la costruzione e gestione di artefatti seguendo la visione A&A[14].

L'astrazione degli agenti e artefatti, alla base di questo framework, promuove una nuova metodologia, per modellare ed ingegnerizzare i sistemi, che introduce un nuovo elemento che influenza la fase di design, di sviluppo e di mantenimento.

Un artefatto può essere visto come un entità computazionale passiva, function-oriented, esplicitamente concepita per fornire qualche tipo di servizio che possa essere usato dagli agenti per supportare i loro bisogni individuali o collettivi.

Il framework essenzialmente fornisce:

- le API con cui definire tipi utili di artefatti;
- le API con cui gli agenti possono interagire con l'ambiente di lavoro (composto di artefatti);
- un sistema eseguibile che supporta la creazione e la gestione dinamica di più ambienti di lavoro.

Non si introduce nessun modello o architettura specifica per gli agenti o per la definizione delle loro società: il framework è inteso per poter essere integrato con le

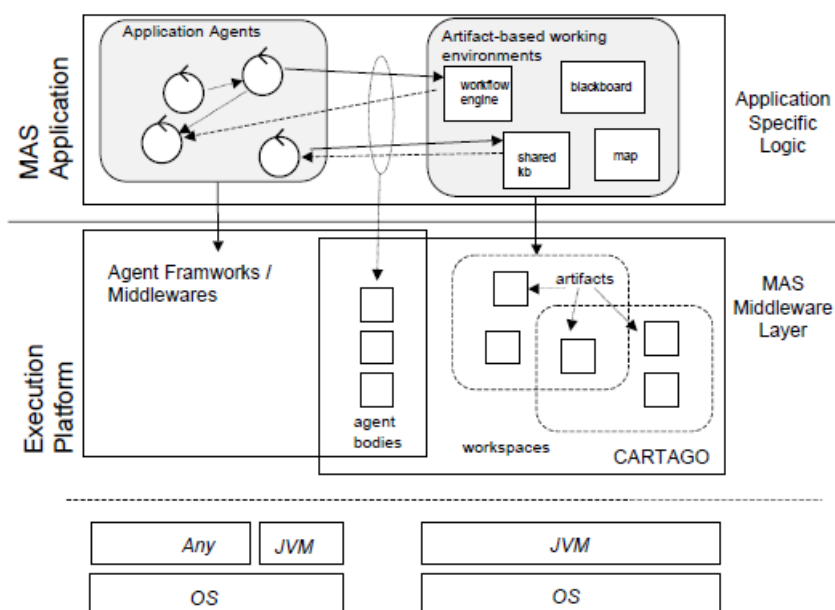


Figura 2.6: MAS adopting CarTAgO

tecnologie esistenti, possibilmente eterogenee. La figura 2.6 mostra l'architettura di un MAS adottando CarTAgO.

Un MAS può essere visto come un insieme di agenti che comunicano tra loro immersi in un ambiente informativo dinamico e modificabile dagli agenti stessi. CarTAgO considera l'ambiente di un MAS come un insieme di *artefatti*.

Lo spazio applicativo è modellato in termini di un *working environment* popolato di artefatti di cui il middleware gestisce il ciclo di vita organizzandoli in *workspace*. Gli agenti sono entità autonome situate che eseguono azioni ed effettuano percezioni sugli artefatti mediante *sensori* ed *attuatori*. Questi ultimi permettono di eseguire operazioni mentre i sensori osservano eventi generati dall'artefatto.

Ogni artefatto produce, durante il suo ciclo di vita, tutta una serie di stimoli che possono essere percepiti dai sensori di un agente. Esso può intenzionalmente collegarsi (attraverso l'operazione *sensing*) o scollegarsi a molti tipi di sensori realizzando così politiche di buffering, ordinamento prioritario, ecc. . .

Ogni artefatto è caratterizzato da un'interfaccia d'uso: ogni operazione è descritta da un nome e una lista di parametri tipati. L'esecuzione di un'operazione causa un cambiamento nello stato interno dell'artefatto e la produzione di eventi osservabili. L'interfaccia d'uso di un artefatto può cambiare in corrispondenza del suo stato osservabile e può quindi variare a run-time.

In modo da supportare l'utilizzo di agenti intelligenti, quindi in grado di effettuare deduzioni, ogni artefatto è dotato di una descrizione delle proprie funzionalità: una

sorta di “manuale di istruzioni”.

Gli artefatti sono logicamente inseriti all'interno di workspace che possono essere usati per definire la topologia logica del sistema.

Capitolo 3

Design

Dal capitolo precedente è evidente come ognuna delle piattaforme presentate sia specializzata per un particolare bisogno applicativo.

LIME concentra gli sforzi al fine di individuare l'architettura più idonea alla mobilità; per EgoSpaces la priorità è la contex-awareness; TOTA vuole essere un'infrastruttura per la coordinazione field-based. Le uniche due piattaforme/framework che prescindono da un particolare contesto applicativo sono CarTAgO e TuCSoN.

Quasi tutte le tecnologie presentate (tranne CarTAgO e TOTA) sono Linda-based ossia incapsulano la semantica operativa di un TS Linda. Nel nostro caso le primitive che sono state introdotte nella sezione 1.2 non corrispondono in tutto e per tutto ad operazioni Linda quindi è prevedibile che, nel caso si adotti una delle piattaforme sopra, occorrerà uno sforzo per ridefinire le nuove primitive in termini di operazioni Linda o sarà necessario mettere mano all'implementazione interna della piattaforma aggiungendole da zero.

Analizzando le varie tecnologie, CarTAgO offre una solida base concettuale su cui modellare il sistema come MAS ed utilizza metafore astratte e generali che prescindono da un particolare scenario applicativo. Prendendo come modello di ecosistema di servizi quello descritto nei primi capitoli (figura 1), mi risulta tuttavia difficile adattarlo in un'ottica A&A. Usare tale meta-modello (ed il framework associato), infatti, porta con sé specifiche progettuali e costrutti ben precisi che in questa fase risultano, a parer mio, più vincolanti che abilitanti.

TOTA adotta una buona tecnica per la distribuzione dei dati come tuple in TS sparsi sulla rete, ma non rappresenta le reazioni come entità di prima classe, mentre dai requisiti emerge che le eco-laws devono essere rappresentate direttamente a livello di infrastruttura. Nonostante ciò le tuple TOTA contengono fisicamente le leggi che sfruttano per la loro propagazione e queste possono essere viste come particolari eco-laws che incapsulano la metafora fisica field-based.

Concludendo, ritengo comunque che la memorizzazione e l'applicazione delle eco-laws non sia da associare fisicamente ai dati (tuple o LSA).

LIME è altamente flessibile e altrettanto predisposto ad ambienti mobile, ma non sfrutta appieno le potenzialità dei TS. Per far fronte all'estrema mobilità degli host, infatti, adotta un modello di comunicazione che difficilmente può essere utilizzato nel nostro sistema: i TS transienti. È lecito presupporre che non tutti gli host e gli agenti siano mobili, ma che esistano dei nodi fissi. In LIME ogni agente si porta con sé le proprie tuple e può comunicare con un altro agente solo se questo è temporalmente e spazialmente connesso con il primo.

Da una prima analisi questo schema fa perdere tutti i vantaggi della comunicazione a spazi: disaccoppiamento in tempo e in spazio. In realtà LIME introduce tecniche alternative e nuove primitive per reintrodurre tali caratteristiche. È tuttavia innegabile che queste primitive appesantiscano l'architettura e rendano complicata la progettazione del sistema. Inoltre i TS transienti costituiscono un concetto che riesco difficilmente ad utilizzare per l'architettura di un ecosistema di servizi come quello descritto nei primi capitoli.

Il primo modello che emerge da quella descrizione è una sorta di rete di nodi (schermi) che memorizzano localmente delle informazioni di contesto che vengono periodicamente modificate dalle eco-laws; i servizi (anche mobili) non fanno altro che utilizzare tali informazioni al bisogno, cercando di servire al meglio gli utenti.

Usando le reazioni LIME per rappresentare le eco-laws si sarebbe costretti ad adottare una soluzione poco efficiente: il set di tali regole deve essere unico a livello di sistema, occorrerebbe associare ad ogni TS-transiente di ogni agente lo stesso set di reazioni con ridondanza e spreco di risorse.

EgoSpaces è sviluppato su LIME e per gli stessi motivi non è considerato idoneo, tuttavia presenta buone fondamenta concettuali ed esprime in maniera efficace il contesto, introducendo modalità innovative per definirlo.

TuCSon è organizzato rispettando esattamente la topologia di sistema desiderata e definisce i nodi della rete come TC (TupleCentres), quindi TS arricchiti di comportamenti reattivi esprimibili tramite reazioni. Tuttavia l'architettura interna di un TC basa il concetto di reattività sul controllo e non sul dato. Più precisamente la reazione è innescata in base all'operazione che viene eseguita sul TC mentre dal nostro modello è chiaro che una reazione scatta autonomamente in base al contenuto del TS.

Concludendo, anche se alcune motivazioni appena espresse possono risultare a prima vista deboli o capziose è da sottolineare che non si sta mettendo in discussione che le piattaforme esaminate possano ricoprire i requisiti descritti, bensì la semplicità e l'efficienza con cui lo fanno. Ricordiamo che ottenere un middleware lightweight è estremamente difficile adattando un sistema già esistente troppo potente o specializzandone uno progettato per un diverso scenario applicativo. Inoltre il processo di mapping (citato ad inizio capitolo) tra Linda e le operazioni definite in 1.2 potrebbe risultare fallimentare o penalizzante in termini prestazionali.

Infine questa tesi è mirata allo sviluppo di un prototipo, per cui prima di adottare

una particolare piattaforma occorre analizzare se questa sia sufficientemente matura e stabile per sopportare agilmente la fase implementativa: non tutte quelle proposte lo sono.

Dovendo individuare la tecnologia che meglio soddisfa i requisiti, *la scelta non ricade su nessuna di quelle descritte.*

3.1 Funzionalità

Questa fase d'analisi punta a trovare una prima architettura (relativa al livello "Spatial Substrate" della figura 5) tenendo conto delle primitive (sez. 1.2) e dei requisiti (sez. 1).

Inizialmente occorre individuare le funzionalità di un nodo ed i vari componenti che implementino le primitive e abilitino l'interazione con i servizi. Durante tutto il processo si adotteranno soluzioni e scelte che promuoveranno l'estendibilità e la flessibilità. Si introdurranno, infatti, meccanismi astratti ed interfacce che prescindano da una particolare rappresentazione concreta o da un particolare algoritmo. Così facendo, oltre ad aumentare il riuso del codice, si velocizza l'implementazione di quelle parti legate all'interazione tra componenti, ossia le parti che realizzano il comportamento astratto dell'infrastruttura.

Diversi linguaggi per le eco-laws Un elemento che non è emerso dai requisiti è la modalità di rappresentazione di un'eco-law. A scanso di equivoci consideriamo che vi sia un certo linguaggio (rappresentazione astratta) con cui esprimere una legge, ma che a questo punto del progetto non sia ancora stato definito.

L'architettura dovrà considerare una rappresentazione concreta delle reazioni abbastanza generica da poter essere utilizzata dallo specifico linguaggio che verrà definito (tesi [17]). Tale linguaggio sfrutterà le primitive e i meccanismi offerti dall'infrastruttura e definirà una particolare tecnica di matching che permetta di selezionare a quali individui (LSA), presenti nello spazio, è applicabile una eco-laws.

Diverse strutture per gli LSA Così come per le eco-laws anche gli LSA avranno una particolare struttura e rappresentazione sintattica. In questa fase si prescinde da tale struttura e si rappresenta un LSA come coppia $\langle ID, Contentenuto \rangle$ dove il contenuto ha un'indicazione generica.

In linea teorica il contenuto di un LSA astrae dal particolare linguaggio usato per le eco-laws. Tuttavia, con l'attenzione rivolta allo sviluppo del sistema, è lecito supporre che la definizione di un particolare linguaggio porti con sé anche un interprete che attribuisce la semantica operativa. Sfruttando linguaggi diversi occorrerà introdurre più interpreti appesantendo così l'infrastruttura.

Concludendo già da questa fase si suggerisce di trovare un linguaggio unico per la rappresentazione sia del contenuto degli LSA sia per le eco-laws.

Componenti Seguendo i requisiti e la lista delle primitive emerge che la struttura del middleware più adatta è una rete di nodi.

Ogni nodo sarà dotato delle seguenti funzionalità:

1. memorizzare uno o più LSA (informazioni contestuali e non);
2. notificare eventuali cambiamenti di un LSA ad uno o più servizi (primitiva `observeById`). Tale funzionalità permette di introdurre la reattività all'interno del sistema;
3. eseguire operazioni che modifichino, rimuovano e aggiungano LSA in corrispondenza della presenza di uno o più LSA nel nodo (semantica operativa di una eco-law).
4. inviare un LSA in rete.

Definiamo un primo insieme di componenti in modo da distribuire più efficacemente ed efficientemente possibile le funzionalità descritte sopra. Tale insieme potrà, con il proseguimento del progetto, modificarsi e specializzarsi.

Si definisce il componente *space* (lo spazio) che funge da contenitore per gli LSA. A tale componente sono associate le stesse primitive di cui può usufruire una eco-law: queste operazioni sono indipendenti dal particolare scenario applicativo e sono il set di primitive minimo per un sistema di elaborazione delle informazioni.

- **inject**: inietta un nuovo LSA all'interno dello spazio. Quest'operazione torna l'Id che l'infrastruttura ha assegnato all'LSA.
- **updateById**: modifica il contenuto di un particolare LSA.
- **removeById**: rimuove un particolare LSA.
- **readById**: restituisce un particolare LSA.
- **readByContent**: restituisce una lista di LSA che corrispondono ad un particolare template. Si è deciso di introdurre questa primitiva in quanto si vuole disporre di una modalità efficiente per ottenere un LSA di cui non si conosce l'id. Non si vuole ancora definire né la natura del template né i meccanismi interni di selezione del risultato, in quanto non si è detto nulla sulla struttura interna degli LSA.

La parte dello spazio che realizzerà tale operazione sarà, fino all'ultimo, lasciata generica in modo da assicurare completa estendibilità dell'infrastruttura.

- **diffuseById**: consente di comunicare un particolare LSA ad altri nodi della rete.

Le operazioni descritte, possono essere viste come le primitive di una macchina astratta con cui esprimere la semantica operativa delle eco-laws. Notiamo che alcune operazioni hanno la stessa signature (ma semantica diversa) di quelle definite nei requisiti (sez. 1.2): questo è fisiologico in quanto, queste ultime, sono operazioni talmente basilari che si possono direttamente applicare ai componenti di più basso livello.

La seconda funzionalità riguarda la primitiva **observeById**. Utilizziamo il pattern publish-subscribe in cui un publisher pubblica informazioni che vengono notificate a diversi subscribers: nel caso in esame l'informazione pubblicata è un LSA inserito, modificato o rimosso. Il componente *notifier* incapsula tale pattern: concretamente è un'entità che contiene una lista di LSA ognuno, associato ad una lista di servizi interessati ad osservarlo. Questa lista viene aggiornata in fase di sottoscrizione. Se un LSA osservato viene modificato, il *notifier* viene avvisato, scorre la lista associata a quell'LSA e distribuisce la notifica (vedi comportamento astratto riportato in figura 3.1).

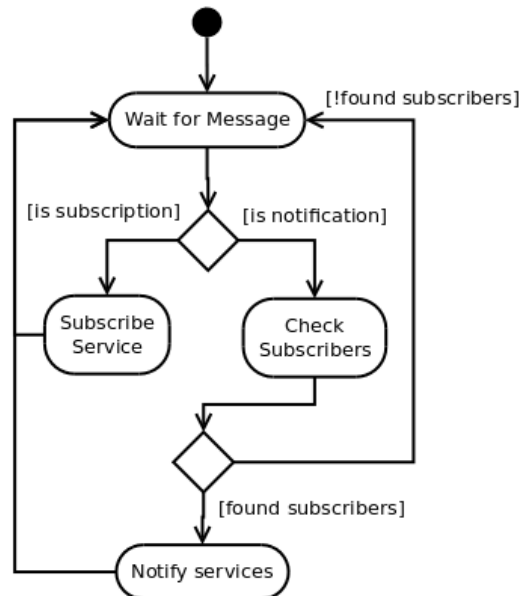


Figura 3.1: Notifier behavior

Il componente che gestisce la logica di esecuzione e la memorizzazione delle eco-law è chiamato *reaction manager*. Ci occuperemo più in dettaglio di tale componente in seguito; per il momento è in grado di interagire con lo spazio in quanto le eco-laws (chiamate anche reazioni) hanno effetto persistente sul nodo.

Ovviamente i servizi sono considerati anch'essi componenti (*service*) ed in particolare costituiscono la business logic del sistema. I *service* interagiscono con il notifier e con lo spazio poichè devono essere in grado di eseguire le primitive di *inject*, *removeById*, *updateById* (sfruttando lo spazio) e *observeById* (sfruttando il *notifier*). Possiamo pensare ai servizi come ad applicazioni autonome che hanno un proprio ciclo di vita.

Dal case study è chiaro che il dispositivo utente (lo smartphone) abbia installato un servizio in grado di comunicare con il nodo a lui più vicino (un display): occorre quindi pensare di inserire nell'architettura dei componenti che abilitino la comunicazione in rete. Per questo aspetto si è scelto uno schema *stub-skeleton* sia per la comunicazione nodo-nodo che per quella nodo-servizio. In quest'ultimo caso, lo *stub* risiede all'interno del servizio e può essere considerato come la parte dell'infrastruttura che abilita l'accesso al middleware. Lo *skeleton*, che risiede sul nodo, si occupa dell'esecuzione delle azioni richieste dallo *stub*¹.

Stub e *skeleton* consentono di disaccoppiare il servizio dal nodo, permettendo così di realizzare un deployment più flessibile. In altre parole possiamo pensare a host dalla scarsa capacità computazionale, quindi non in grado di ospitare un intero nodo, che gestiscono un unico servizio, più leggero e meno oneroso. Inoltre, è possibile integrare nel sistema agenti mobili in grado di spostarsi da un host all'altro: in questo caso un servizio potrà risalire la rete in modo da giungere, per esempio, più vicino possibile all'utente così da migliorare la qualità del servizio erogato.

Lo scenario che si prospetta vede quindi la distribuzione fisica del middleware che varia a seconda delle capacità dell'host: casi in cui host potenti ospitano sia servizi che un nodo, casi in cui un host più ridotto ospita unicamente un servizio (vedi figura 3.2).

3.1.1 Reaction Manager

Comportamento Astratto Il funzionamento del *reaction manager* (indicato in questa sezione come *manager*) è dipendente dalla semantica associata alle eco-laws e quindi dal linguaggio utilizzato per descriverle.

Cerchiamo ora di pensare ad un possibile comportamento astratto.

Prima di tutto occorre che il manager sia in grado di eseguire una transazione (una alla volta per evitare problemi di consistenza) sullo spazio e deve poter di accedere a quest'ultimo in lettura al fine di controllare se una determinata reazione è attivabile. Infine deve essere progettato tenendo conto che potrà gestire contemporaneamente più tipologie di reazioni diverse, ognuna con una sua particolare semantica d'esecuzione. Quest'ultima affermazione non è riferita al tipo di linguaggio utilizzato per le reazioni, bensì alla loro semantica operativa. Dovendo infatti rappresentare

¹TuCSon utilizza il medesimo schema di comunicazione tra agenti e TC e TC-TC.

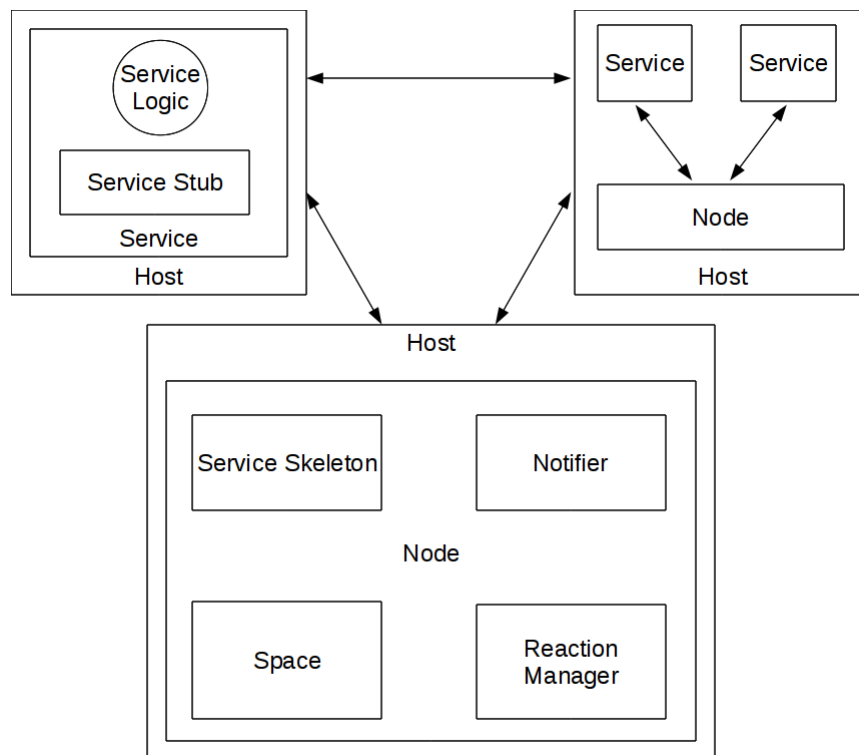


Figura 3.2: System Logic Architecture

diversi tipi di metafore naturali, è prevedibile che ognuna di queste sia realizzata con reazioni che utilizzano precisi algoritmi. È utile, perciò, abilitare il manager alla gestione di diversi algoritmi anche se fino ad ora non ne abbiamo esempi concreti. Il *reaction manager* come l'intero nodo deve essere situato ossia in grado di gestire eventi ambientali: uno per tutti il passare del tempo. Dai requisiti è chiaro come una delle prerogative del sistema sia quella di riprodurre dinamiche naturali per cui è obbligatoria la gestione di eventi temporali al fine di poter simulare comportamenti reali (vedi algoritmo di Gillespie [5]).

Segue la descrizione della figura 3.3.

Wait for Event: stato di quiete in cui il manager non effettua nessuna operazione in attesa di ricevere un evento (per ora solo temporale).

Select activable reaction: in seguito ad una notifica il manager seleziona un insieme di reazioni eseguibili in base al tipo di evento. Una reazione si definisce eseguibile se lo stato dello spazio abilita la sua esecuzione. L'output di questa fase è una lista di reazioni eseguibili.

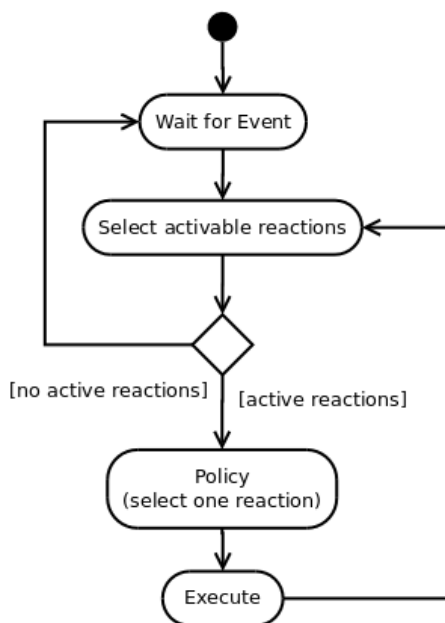


Figura 3.3: Reaction manager activity diagram

Policy: fase in cui dalla lista precedentemente creata si seleziona la reazione da eseguire seguendo una particolare politica (es. non deterministica, ecc...).

Execute: stato in cui la reazione viene resa persistente eseguendo sullo spazio (nella giusta sequenza) le operazioni associate ad essa.

Per ora questo stato si occupa anche di effettuare rollback in caso di errori.

Nel caso in cui il manager esegua una reazione (**Execute**) significa che lo stato dello spazio è cambiato (una reazione che ha solo operazioni di lettura è priva di significato); è quindi possibile che, in conseguenza di tale cambiamento, altre reazioni siano attivabili: occorre ricontrollare il set delle reazioni. Questo il motivo dell'indicazione di jump in corrispondenza dello stato **Execute**.

Semantica operativa di una eco-law: TIMED o ASAP Precisiamo ora due possibili semantiche associate ad una reazione. Per non rischiare di definire tipologie troppo stringenti che, in seguito, non corrispondano alle esigenze del linguaggio o che risultino poco espressive, cerchiamo di procedere in maniera generale.

Sfruttando la situadness del manager alcune eco-laws possono scattare ad intervalli temporali ben definiti soprattutto se modellano il comportamento di sistemi chimici o biologici: chiamiamo tali reazioni *timed*. Su questa linea possiamo pensare ad altre eco-laws che debbano, invece, essere attivate non appena si verifichino le

condizioni giuste: ossia il prima possibile, senza sottostare a nessun vincolo temporale. Chiamiamo queste reazioni *As Soon As Possible* o *ASAP*.

Queste due tipologie, assolutamente generiche, definiscono tuttavia grandi vincoli progettuali. In primo luogo gestire lo scorrere del tempo, agendo in corrispondenza dello scadere di un time-out, presuppone un comportamento attivo: questo porta a considerare il *reaction manager* come un'entità che, indipendentemente dai servizi, agisce autonomamente sullo spazio e che non è in comunicazione con nessun altro componente. In secondo luogo la semantica *ASAP* obbliga il manager ad essere sensibile alle operazioni eseguite all'interno del middleware, in modo che possa agire il più velocemente possibile.

Per integrare i due comportamenti diciamo che il *reaction manager* è sensibile ad eventi temporali e operazionali e che, ciclicamente a seconda del tipo di evento che riceve, innesca meccanismi di controllo e di selezione delle reazioni.

Gli eventi temporali saranno generati all'interno del componente, quelli operazionali comunicati dallo *skeleton*. Tale scelta non modifica il comportamento astratto di figura 3.3.

3.2 Architettura di sistema

Analizzando le reazioni *ASAP*, a fronte delle diverse entità attive presenti nell'architettura (*notifier* e *skeleton*), sorge il problema di quantificare il livello di reattività di una reazione. Avendo, infatti, il *reaction manager* un comportamento parallelo ai servizi (per servizi in questo caso si intendono le istanze dello *skeleton* che agiscono nel nodo per conto dei servizi veri e propri) è possibile che si verifichino corse critiche, quindi che si mandino in esecuzione reazioni considerate attivabili ma le cui precondizioni di attivazione vengono inficiate dalle operazioni di uno o più servizi. Tali reazioni dovranno fallire in sede di esecuzione ed avere la possibilità di effettuare il rollback.

Per evitare questa situazione occorre che si sincronizzino le operazioni sullo spazio in modo che, tra un'operazione e l'altra, il *reaction manager* compia un ciclo di funzionamento completo (dal sensing dell'evento all'esecuzione della reazione). Ossia *chel'esecuzione di una reazione ASAP agli occhi del servizio sia istantanea*.

Ancora una volta, non conoscendo la semantica finale delle reazioni o se la situazione di errore vista precedentemente è accettabile, cerchiamo una soluzione architetturale che permetta di scegliere se regolare o meno l'accesso allo spazio. A tale scopo si è introdotto il componente *space manager*.

Space Manager Questo nuovo componente incapsula i meccanismi di sincronizzazione idonei all'esecuzione di una transazione e all'accesso concorrente allo spazio. A seconda della sua configurazione può essere attivo o passivo. Nel caso sia configurato

per consentire l'accesso concorrente, costituisce un punto di sincronizzazione passivo che gestisce la risorsa condivisa (spazio) tra più attori (più *skeleton* e *reaction manager*).

Nel caso in cui sia configurato in modo da cedere la parola al *reaction manager* tra un'operazione e l'altra, costituisce un elemento attivo che accoda le operazioni provenienti dai servizi e le esegue nell'ordine di arrivo, intervallando di volta in volta il funzionamento del manager.

Diffuser Si è scelto di spostare la funzione di diffusione di un LSA dallo spazio ad un componente separato chiamato *diffuser*.

La diffusione non è una primitiva a cui possono accedere i servizi, essa può essere gestita solamente all'interno di una eco-laws.

La progettazione di questo componente per ora non viene considerata, poichè non è stato definito chiaramente in fase di analisi come verrà gestita la topologia logica della rete di nodi. Perciò il *diffuser* non verrà citato nei successivi schemi o descrizioni.

Rappresentazione interna di una eco-law Una eco-laws è vista come una reazione che scatta nel caso uno o più LSA corrispondano a determinati template. Questi ultimi possono esseri intesi come pre-condizioni che devono verificarsi per attivare la reazione o come reagenti chimici; mentre le operazioni associate possono essere intese come post-condizioni o come prodotti chimici. Concretamente sono *filtri* ossia LSA dal contenuto variabile.

Ad una reazione sono associate un certo numero di operazioni che devono essere eseguite in maniera transazionale sullo spazio. Queste operazioni possono spaziare tra tutte quelle relative al componente *space*. Tale rappresentazione concreta consente, al momento della definizione del linguaggio delle eco-laws e della definizione della struttura precisa degli LSA, di specializzare unicamente la parte del sistema che si occupa del matching LSA-filtro (parte che è utilizzata anche internamente allo spazio per realizzare l'operazione `readByContent`).

3.2.1 Interazione tra componenti

Analizziamo come i vari componenti interagiscono tra di loro.

Procedendo per gradi raffiniamo l'architettura logica di figura 3.2 introducendo lo *space manager* (vedi figura 3.4).

Ogni comunicazione tra componenti può essere intesa come scambio di messaggi dal diverso contenuto informativo: *notifiche*, *sottoscrizioni* e *operazioni*.

Le notifiche rappresentano un particolare evento accaduto all'interno del sistema (temporale o operativo), le sottoscrizioni sono messaggi gestiti dal notifier che contengono informazioni sul mittente e sull'LSA che si vuole osservare, le operazioni

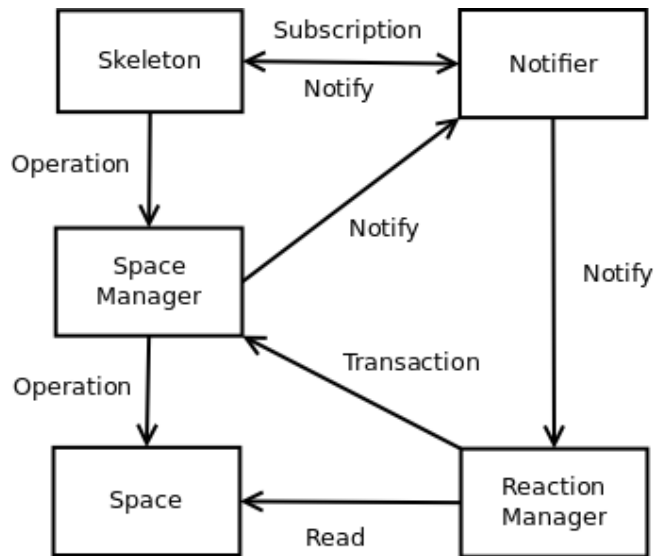


Figura 3.4: Component interactions

(o transazioni) sono messaggi gestiti dallo space manager al fine di operare sullo spazio.

Segue una sintesi delle funzionalità principali dei componenti a fronte delle modifiche apportate in fase di progetto.

Skeleton e Stub: lo skeleton si occupa di eseguire le operazioni comunicate dallo stub.

Si ricorda che le possibili operazioni lato stub sono la rimozione, modifica, inserimento e osservazione di un LSA: per le prime tre lo skeleton può direttamente rivolgersi allo **space manager**, per l'ultima occorre effettuare una sottoscrizione (*subscription*) presso il **notifier**.

In seguito alla sottoscrizione, il **notifier** può notificare lo skeleton relativamente a modifiche all'LSA osservato.

Si consideri il patter executors per gestire internamente la possibilità di avere contemporaneamente più comunicazioni con più servizi.

Per ora facciamo riferimento ad uno scenario in cui, dato un servizio, questo possa eseguire più operazioni in contemporanea sul middleware e che per ogni operazione vi sia una coppia stub-skeleton attiva. Pur non essendo la soluzione che minimizza le risorse occupate è quella che facilita l'implementazione.

La comunicazione stub-skeleton è bloccante ossia lo stub resta in attesa fin quando lo skeleton non ha effettuato l'operazione. Il servizio potrà sfruttare semantiche bloccanti e non bloccanti per tutte le primitive del middleware in quanto comunica con uno o più stub senza restare necessariamente in attesa della conclusione di un'operazione.

Notifier: incapsula il patter publish-subscribe per realizzare la primitiva `observeById`.

Riceve notifiche (*notify*) dallo **space manager**, le propaga allo **skeleton** (previa *subscription*) e al **reaction manager**.

Il comportamento interno è attivo e rispecchia quello di figura 3.1.

Space manager: gestisce l'accesso allo spazio e si occupa di propagare le notifiche, al **notifier**, relative alla modifica, rimozione, inserimento di LSA.

Comunica con il **reaction manager** al fine di sincronizzare (se richiesto) l'esecuzione delle operazioni sullo spazio con l'esecuzione delle reazioni.

Occorre realizzare tale componente con una duplice natura (configurabile) attiva o passiva a seconda che si vogliano gestire reazioni ASAP istantanee o concorrenti.

Reaction manager: accede direttamente allo spazio in lettura per determinare quali reazioni siano attivabili, viene notificato dal **notifier** al fine di gestire eventi operazionali e quindi reazioni ASAP e comunica con lo **space manager** per eseguire la transazione associata alla reazione.

Per ora consideriamo che tale componente si occupi anche della creazione e gestione degli eventi temporali.

Il comportamento è attivo: si consideri quello di figura 3.3 come funzionamento astratto che verrà specializzato nel prototipo.

Space: funge da contenitore di LSA.

È possibile accedervi tramite le primitive `readByContent`, `readById`, `inject` e `removeById` (vedi secondo elenco sez. 1.2). Internamente è gestito come componente passivo che incapsula logiche di accesso dati efficienti e tecniche di `matching template-lsa`.

Capitolo 4

Implementazione

In questa sezione si discuteranno le scelte implementative considerando il paradigma object-oriented e nello specifico JAVA per l'implementazione del sistema.

4.1 Progettazione UML

Si faccia riferimento al diagramma completo delle classi di figura 4.2. In tale diagramma sono riportate le classi e le interfacce CORE dell'infrastruttura. Per una visione più completa si veda la descrizione 4.2 e i sorgenti.

Come descritto nella sezione precedente la maggior parte dei componenti dell'infrastruttura è attiva ed ha un comportamento ciclico. Con queste basi si è scelto di utilizzare per l'implementazione uno schema che favorisca il riuso del codice in modo da poter dare ad ogni componente una struttura comune, che venga specializzata per realizzare il comportamento specifico.

Definiamo quindi una classe `InternalAgent` contenente una lista di messaggi ed il metodo `sendMessage(Message)` che consente di inserire un messaggio in coda. Tale classe viene specializzata da `Notifier`, `AResponseManager` e `SkelAgent`. Tutte gli oggetti che specializzano `InternalAgent` hanno il seguente comportamento:

1. attesa ricezione messaggio
2. analisi messaggio
3. esecuzione task relativo in base allo stato interno
4. torna ad 1

Il punto 3 viene implementato a seconda della funzione specifica della classe.

Per quanto riguarda le altre classi il loro funzionamento interno ricalca sostanzialmente quello definito in fase di progetto.

Occorre tuttavia descrivere più in dettaglio l'implementazione del `AReactionManager`, motivando la presenza dell'interfaccia `IAlgorithm`, ed il che modo in cui avviene la sincronizzazione con `ISpaceManager`.

AReactionManager, IAlgorithm, ISpaceManager `AReactionManager` realizza le funzioni del *reaction manager* e prescindere dal particolare linguaggio delle reazioni.

Facendo riferimento allo schema di figura 3.3, il metodo astratto `chooseActiveReactions` è corrispondente allo stato “select activable reactions” che costituisce lo stato il cui comportamento varia a seconda della semantica del linguaggio.

Internamente il manager, come già accennato, può gestire più tipologie di reazioni, ogni reazione ha una propria dinamica di attivazione ossia un proprio algoritmo a cui fare riferimento. Per chiarire il concetto, immaginiamo che si definiscano regole che rappresentino vere e proprie reazioni chimiche a cui sia associato un certo rate (frequenza) di attivazione (per esempio eseguibili usando l'algoritmo di Gillespie [5]). Estendendo l'interfaccia `IAlgorithm` si può specificare come le reazioni vengono scelte per l'attivazione in base al rate.

In particolare, invocando il metodo `getExecutableReactions` si ottengono le reazioni che sono considerate attivabili (quindi potenzialmente eseguibili) dall'algoritmo. La logica di esecuzione di una reazione non dipende dal linguaggio e dal tipo di reazione ma unicamente dal tipo di `IPolicy` (vedi stato “Policy” figura 3.3) per questo è direttamente cablata all'interno del reaction manager astratto.

La sincronizzazione tra *space manager* e *reaction manager* è realizzata come segue. Supponiamo che più skeleton agiscano contemporaneamente sullo space manager (oggetto che dovrà implementare `ISpaceManager`) e che si vogliano gestire le reazioni ASAP come istantanee; occorre quindi:

1. invocare il metodo `setConcurrency(false)` per configurare lo space manager al fine di disabilitare l'accesso concorrente allo spazio;
2. lo space manager accoda tutte le operazioni degli skeleton, esegue la prima arrivata ed attende (in maniera bloccante o meno) l'invocazione del metodo `step()`;
3. vengono generate tutte le notifiche relative all'operazione eseguita: tra i destinatari c'è anche il reaction manager;
4. il reaction manager alla ricezione della notifica compie un ciclo di lavoro completo che si può risolvere nell'esecuzione di più reazioni sullo spazio.
La transazione associata ad ogni reazione viene eseguita tramite lo space manager invocando il metodo `doTransaction(Transaction)`;

5. alla fine del ciclo il reaction manager invoca sullo space manager il metodo `step()`
6. torna ad 2 finchè non ci sono più operazioni da eseguire.

Nella sequence diagram di figura 4.1 è indicato lo schema di interazione per l'esecuzione della prima operazione arrivata allo space manager.

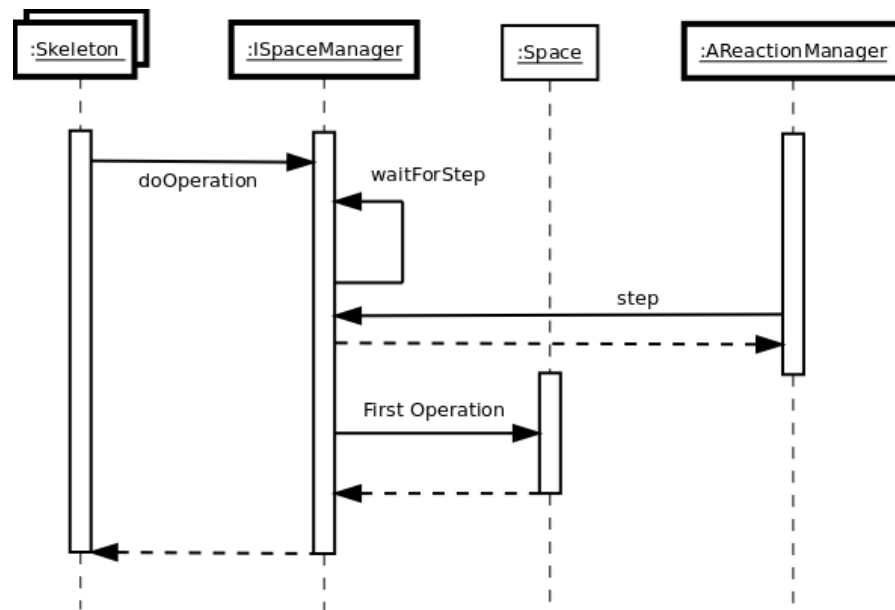


Figura 4.1: Skeleton - AReactionManager - ISpaceManager interaction

Notifier e IContentFilter Un servizio potrebbe essere installato su un dispositivo con scarse risorse computazionali (pensiamo a dispositivi embedded), per questo non in grado di memorizzare e gestire un intero LSA (in quanto non sono prevedibili le sue dimensioni).

In fase di implementazione si è pensato, quindi, di rendere più flessibile l'observing arricchendo l'operazione `Observe` di un filtro che indica quale parte specifica dell'LSA si intende osservare. Questo permette di gestire una quantità di informazioni molto più ridotta e di diminuire il traffico in rete.

Concretamente è stata modificata l'interfaccia d'uso del middleware dando la possibilità (lato stub) che il servizio potesse esplicitare un oggetto filtro sull'LSA da osservare (vedi `IContentFilter` e `Subscription`): l'operazione di observing va a buon fine se il contenuto dell'LSA corrisponde, seguendo un particolare algoritmo di matching, al filtro.

Service e middleware Non è oggetto di questa tesi definire una possibile architettura per un servizio. Tuttavia al fine di giungere ad un prototipo funzionante occorre definirne una modalità di comunicazione servizio-middleware.

L'architettura proposta prevede che sia possibile specializzare stub e skeleton utilizzando un qualsiasi protocollo di scambio dati: TCP, HTTP, SSL, SOAP e altri.

La scelta è stata di utilizzare un canale TCP su cui lo stub invia, serializzando, direttamente le operazioni che lo skeleton, lato nodo, deve eseguire. Lo skeleton risponde con un messaggio **Response** che contiene il risultato dell'operazione e se questa è andata o no a buon fine.

4.2 Prototipazione

Il codice è stato organizzato seguendo il pattern Model View Controller.

4.2.1 Packages

- `sapere.model`: contiene le classi e le interfacce indipendenti dal particolare caso di studio o scenario applicativo che fungono da fondamenta per l'infrastruttura.
- `sapere.model.communication`: contiene la gerarchia di messaggi scambiati tra i componenti di un nodo.
- `sapere.controller`: contiene le classi specifiche per il caso di studio, in particolare quelle che incapsulano comportamenti attivi.
- `sapere.controller.space`: contiene le classi associate alle funzionalità del componente *space*.
- `sapere.controller.reactionManager`: contiene le classi associate alle funzionalità del componente *reaction manager*.
- `sapere.controller.notifier`: contiene le classi associate alle funzionalità del componente *notifier*.
- `sapere.controller.n2n`: contiene le classi che realizzano la comunicazione node-to-node ossia il componente *diffuser*.
- `sapere.controller.stub`: contiene le classi che realizzano la comunicazione node-to-service lato servizio.
- `sapere.controller.skel`: contiene le classi che realizzano la comunicazione node-to-node lato nodo.

4.2.2 Classi

Questa descrizione non comprende le classi principali del sistema in quando sono già state descritte nelle sezioni precedenti. Molte delle classi indicate sotto non sono presenti nell'UML di figura 4.2 in quanto accessorie, specifiche dell'esempio applicativo o non considerate essenziali alla comprensione del progetto o del funzionamento del sistema.

- `sapere.model.IOperation`: interfaccia che rappresenta un'operazione astratta eseguita all'interno dell'infrastruttura.
- `sapere.model.LogicContentFilter`: specializzazione di `IContentFilter`. Tale classe implementa un particolare meccanismo di matching utilizzato all'interno dello spazio per la `readByContent`. In questo caso il matching è sintattico, si considera il contenuto dell'LSA di tipo `LogicLsaContent` e si sfrutta l'unificazione logica per effettuare l'associazione filtro-contenuto. Come motore logico viene utilizzato il Prolog in particolare un suo porting su JAVA (tuProlog). Il filtro è inteso come un termine logico non ground che viene unificato con il contenuto dell'LSA. Il metodo che realizza il matching è `filters` il quale ritorna un `LogicLsaContent` che rappresenta il valore della variabile X interna al filtro.
- `sapere.model.LogicLsaContent`: implementa `IContent`, rappresenta un tipo di contenuto per LSA: un termine della logica del primo ordine.
- `sapere.model.Token`: classe di utilità, rappresenta un oggetto utilizzato per la sincronizzazione all'interno della operazioni (vedi `SpaceOperation`).
- `sapere.model.SpaceOperation`: rappresenta un'operazione effettuata sullo spazio e implementa l'interfaccia `IOperation`. Oggetti di questa classe possono fungere da parametri al metodo `doOperation` dello *space manager*. Internamente contiene un oggetto `Token` che viene sfruttato per sincronizzare il comportamento dello `SkelAgent` con quello dello `SpaceManager`: ogni oggetto skeleton si occupa di eseguire un'unica operazione sullo spazio, per questo invocano sullo `SpaceManager` il metodo `doOperation`. Il corpo di tale metodo ha comportamenti diversi a seconda della configurazione dell'oggetto: concorrente o sequenziale. Il flusso di controllo dello skeleton, nel caso concorrente, procede eseguendo direttamente l'operazione mentre, nel caso sequenziale si mette in attesa sull'operazione (metodo `waitForExec`) finché questa non diventa eseguibile. Internamente l'implementazione concreta dello `SpaceManager` (`sapere.controller.space.SpaceManager`) ha una lista di operazioni che marca come eseguibili (sbloccando lo skeleton) non appena il *reaction manager* (il cui comportamento astratto è implementato dalla classe

`sapere.controller.reactionManager.AReactionManager`) notifica all'avvenuta conclusione di un ciclo di funzionamento (invocando il metodo `step` sullo `SpaceManager`).

- `sapere.controller.SpaceAgent`: rappresenta la generalizzazione di un thread che agisce internamente al nodo comunicando con lo `SpaceManager` al fine di eseguire una `SpaceOperation`. `SpaceAgent` è stata introdotta in modo da rendere la costruzione di eventuali altri attori che agiscano sullo spazio più agile ed efficace. La classe in esame può anche essere utile in fase di testing dello `SpaceManager` o dello `Space` per ottenere velocemente un surrogato di servizio che agisca sul nodo.
- `sapere.controller.Node`: rappresenta un nodo completo di tutti i componenti (`SpaceManager`, `SimpleReactionManager`, `Notifier`, `SkelManager`, `Space` e `NodeManager`).
- `sapere.controller.n2n.NodeManager`: insieme al `DiffuseWorker` costituisce l'implementazione del componente *diffuser*. Tale classe incapsula un server sequenziale che riceve operazioni di inject da nodi remoti e le esegue sul nodo locale. La porta di ascolto è definita a livello di nodo.
- `sapere.controller.n2n.DiffuseWorker`: insieme al `NodeManager` costituisce l'implementazione del componente *diffuser*. Tale worker entra in gioco nel caso in cui lo space manager si trovi a dover eseguire (all'interno di una transazione) un'operazione di diffusione (ossia un inject remota). Lo `SpaceManager` utilizza il pattern executor lanciando, per ogni operazione di diffusione, un worker a cui passa l'LSA da diffondere.
In questa versione il `DiffuseWorker` al suo interno controlla se l'LSA rispetta una precisa struttura (ossia se corrisponde ad un preciso filtro). Le informazioni sull'indirizzo IP a cui inviare l'LSA sono da ritrovarsi all'interno dell'LSA stesso; mentre la porta è assegnata a livello di nodo.
- `sapere.controller.reactionManager.ASAPAlgorithms`: è l'implementazione concreta dell'interfaccia `IAlgorithm` e rappresenta una tipologia di reazione. In questo caso ogni algoritmo ha la responsabilità di memorizzare ed attivare tutte le reazioni della propria tipologia. Nel caso l'algoritmo realizzi un particolare meccanismo che interessi anche eventi temporali sarà suo compito attivarsi e notificare il *reaction manager* affinché compia un ciclo di funzionamento. Invocando il metodo `getExecutableReactions` si ottiene una lista di reazioni candidate per l'esecuzione.
Questa implementazione concreta (come `TimeAlgorithm` citato sotto) sfrutta una inner class `ReactionMatcher` che costituisce un worker che ha come scopo valutare se una reazione è attivabile o meno. Tale processo prevede che

si sfruttino i meccanismi di matching (sintattici) interni allo spazio (classe `LogicContentFilter`) per selezionare gli LSA che costituiranno i reagenti (o pre-condizioni) della reazione. Inoltre, per realizzare l'associazione fra reagenti e tra reagenti ed operazioni associate ad una reazione, si è sfruttato il prolog: ogni operazione viene trasformata (all'interno del `ReactionMatcher` in una regola la cui testa è il contenuto dell'LSA soggetto dell'operazione (definito solo per operazioni di `inject`, `diffuse` in quanto per le altre si sfrutta l'ID) ed il corpo sono i filtri (visti come termini variabili) della reazione. Successivamente vengono asserite tali regole in una teoria logica insieme al contenuto degli LSA ottenuti dalla `readByContent` sullo `Space`. Per ultimo si effettua una query avente come termine la testa della regola precedentemente inserita: se il motore restituisce un esito negativo significa che non esiste un set di LSA all'interno dello spazio che abilita la reazione, se l'esito è positivo gli LSA scelti vengono cablati alla reazione che viene considerata eseguibile.

L'algoritmo ASAP non ha al suo interno nessuna gestione temporale per cui cerca di attivare (ogni volta che viene interrogato) tutte le reazioni contenute al suo interno.

- `sapere.controller.reactionManager.TimeAlgorithms`: implementa `IAlgorithm`, internamente sfrutta gli stessi meccanismi descritti sopra. Dovendo gestire il passare del tempo è un'entità attiva in grado di notificare autonomamente il *reaction manager*: in questo caso la notifica avviene a cadenze regolari di tot secondi.
- `sapere.controller.reactionManager.SimpleReactionManager`: estende la classe astratta `AResponseManager`, implementa il metodo `chooseActiveReactions` e determina la particolare policy di esecuzione. Internamente non fa altro che scorrere i vari algoritmi in modo da popolare la lista delle reazioni candidate all'esecuzione.
- `sapere.controller.reactionManager.SimpleReaction`: come già detto una reazione è vista come un insieme di filtri e una transazione (set di operazioni), tale classe concretizza questa struttura.
- `sapere.controller.space.Space`: implementazione concreta di `ISpace`. Internamente gli LSA sono memorizzati in un hashmap (chiave: ID Lsa, valore: LSA) in grado di gestire in maniera efficiente la concorrenza (`ConcurrentHashMap`).
- `sapere.controller.stub.SapereMiddleware`: classe di utilità introdotta al fine di facilitare l'interazione servizio-middleware. I metodi di questa classe sono statici ed internamente incapsulano il pattern executor che lancia uno `StubAgent` per ogni richiesta. Questo consente che un servizio possa gestire più operazioni in contemporanea sul middleware.

- `sapere.controller.stub.StubAgent`: si occupa della gestione della comunicazione servizio-nodo. Invia, al corrispondente skeleton, l'operazione da eseguire.
- `sapere.controller.stub.SapereOperation`: implementa l'interfaccia `IOperation` e rappresenta l'operazione che viene richiesta lato servizio.
- `sapere.controller.stub.SapereService`: costituisce un primo prototipo di servizio ed implementa `InternalAgent`. Anche se si presuppone che venga costruito con un'architettura indipendente da quella dei componenti del middleware si è scelto di dotarlo della stessa struttura di questi ultimi (questa scelta è dovuta alla generalità della classe `InternalAgent`). Il servizio in questione può accedere al middleware tramite la classe `SapereMiddleware` ed è dotato di GUI che offre la possibilità all'utente di eseguire le operazioni sull'infrastruttura (si veda la sezione 4.3 per ulteriori informazioni).
- `sapere.controller.skel.SkelManager`: implementa il pattern executor, costituisce un multi-server sulla porta adibita alla comunicazione con i servizi (definita a livello di nodo). Ad ogni connessione avvia uno `SkelAgent`.
- `sapere.controller.skel.SkelAgent`: implementa l'interfaccia `SpaceAgent`, si occupa della gestione della comunicazione nodo-servizio e quindi alla corrispondente esecuzione delle operazioni sullo spazio.
- `sapere.controller.ComponentManager`: classe di utilità che permette di accedere ai vari componenti in maniera uniforme all'interno del sistema; al suo interno definisce una serie di metodi statici che permettono l'inserimento dei vari componenti (spazio, notifier, reaction manager, ecc. . .) in una sorta di contenitore. Ogni oggetto concreto al momento della sua costruzione (all'interno del costruttore) si registra presso il `ComponentManager` e diventa a quel punto aperto alla comunicazione con tutti gli altri oggetti del sistema. Ricordiamo che la maggior parte degli oggetti interni ad un nodo estendono `InternalAgent` ossia una classe che abilita alla comunicazione di messaggi a prescindere dalla business logic dell'oggetto.

4.3 Esempio applicativo

Al fine di fornire un esempio di utilizzo concreto del prototipo, in questa sezione si presenterà una possibile applicazione, molto semplice ma che evidenzia un po' tutte le funzionalità del sistema.

Abbiamo a disposizione due nodi interconnessi su cui sono disposti tre servizi. Non si definiscono la natura degli host fisici (quindi dei dispositivi) che ospitano i

nodi in quanto esulano dagli obiettivi dell'esempio.
I servizi sono i seguenti:

- un fornitore generico che si rende disponibile ad eseguire un certo task;
- un cliente che possiamo considerare come il servizio installato su di uno smart-phone che inietta nel sistema le preferenze e le richieste dell'utente;
- un servizio di sistema che deve effettuare la context recognition quindi raccogliere le informazioni ambientali al fine di costruirsi internamente il proprio contesto.

Concretamente ognuno di questi servizi è un **SapereService**, datato di GUI attraverso la quale il tester simula il comportamento di un servizio. Topologicamente il fornitore ed il cliente sono associati ad uno stesso nodo, mentre il servizio di sistema è localizzato presso il vicino (vedi figura 4.3).

Su entrambi i nodi sono installate le stesse eco-laws espresse senza usare un particolare formalismo: sono direttamente cablate a livello di codice. Per motivi di tempo non è, infatti stato sviluppato alcun interprete ed alcun preciso linguaggio (si veda al riguardo la tesi [17]).

Qui di seguito si riportano i frammenti di codice relativi all'installazione di un nodo:

```
public Node(int diffportin,int diffportout,int servicesPort){

    DIFF_PORT_IN = diffportin;
    DIFF_PORT_OUT = diffportout;
    SERVICES_PORT = servicesPort;
    new Space();
    new SpaceManager(true);
    SimpleReactionManager manager = new SimpleReactionManager();
    new Notifier();
    new SkelManager();
    new NodeManager();

    manager.addAlgorithm(new TimeAlgorithm());
    manager.addAlgorithm(new ASAPAlgorithm());

    /*
        INSERIMENTO ECO-LAWS
    */

}
```

Per la costruzione di un nodo si devono esplicitare tre diverse porte di servizio:

diffportin: porta su cui il nodo accetta operazioni di diffusione;

diffportout: porta su cui il nodo invia operazioni di diffusione;

servicesPort: porta con cui il nodo comunica con il servizio.

In questa maniera è possibile, in fase di testing, creare una rete di nodi con collegamenti ad hoc.

Come è chiaro dal codice, all'interno del costruttore della classe `Node`, vengono costruiti tutti gli oggetti che incapsulano le funzionalità del nodo ed in particolare si comunica al `ReactionManager` che dovrà gestire due tipi di algoritmi per reazioni: Time e ASAP. Inoltre notiamo che, in questo esempio specifico, lo `SpaceManager` viene configurato in modalità “concorrente” quindi il `ReactionManager` avrà un comportamento completamente disaccoppiato da quello dei servizi.

Indichiamo ora quali reazioni vengono installate sul nodo. Per questo introduciamo una notazione simile a quella usata per le reazioni chimiche (senza rate) dove al posto delle sostanze prodotto si indicano direttamente le operazioni da eseguire sullo spazio (questa scelta è stata presa per migliorare la comprensione del lettore). In questo caso le operazioni sono indicate come termini logici di arità variabile a seconda del numero dei parametri della primitiva. In particolare:

- `update(p1,p2)`: `p1` rappresenta l'indice del reagente al quale ci si riferisce e `p2` il nuovo contenuto dell'LSA.
- `inject(lsa)`: `lsa` rappresenta il nuovo LSA che si introduce nello spazio.
- `diffuse(p1)`: `p1` rappresenta l'indice del reagente che si vuole diffondere.
- `remove(p1)`: `p1` rappresenta l'indice del reagente che si vuole rimuovere dallo spazio.

Il soggetto della primitiva può essere o un nuovo LSA o uno dei reagenti: in quest'ultimo caso si utilizza un riferimento numerico (partire da 0) con cui si indicano i reagenti, o meglio gli LSA che corrispondono al template del reagente, cominciando il conteggio da quello all'estrema sinistra. Il tipo di reazione è indicato tra parentesi quadre ed il nome prima dei due punti. Da notare che le lettere maiuscole interne ad una regola rappresentano variabili logiche che possono unificare, l'intera regola può essere vista come un insieme di termini prolog per cui l'uso della stessa variabile sia tra i reagenti che tra le operazioni significa in realtà che si vuole introdurre lo stesso valore.

```

CR: response(Q)
    +
    stat
    ->
    update(1,stat(Q))+remove(0) [TIME]

BIND: service(Q)
    +
    request(Q)
    ->
    update(0,served(Q)) [TIME]

RESP: served(Q,A) + request(Q)
    ->
    update(1,request(Q,A))
    +
    inject(go(response(A),'127.0.0.1'))
    +
    remove(1) [ASAP]

DIFF: go(,_ ,_)
    ->
    diffuse(0)
    +
    remove(0) [ASAP]

```

CR è la reazione che si occupa di abilitare la context recognition (CR). In questo caso l'esempio è triviale in quanto viene recuperato solo il corpo della risposta di ogni servizio che ha accettato una richiesta. Il servizio CR dovrà quindi emettere nello spazio un LSA dal contenuto **stat** per rendersi disponibile a raccogliere una delle risposte presenti nel contesto (nodo).

BIND è una regola che effettua il collegamento tra una richiesta, con un particolare soggetto, ed il suo servizio specifico (sfruttando l'unificazione ed il matching sintattico). Per comunicare al servizio che il binding è stato realizzato si modifica l'LSA **service** in **served**.

RESP è la reazione che permette di comunicare al cliente la risposta del servizio; questa in generale può contenere, in un caso concreto, le indicazioni su come accedere al servizio o sul particolare protocollo di comunicazione. Tale reazione introduce la risposta del servizio (campo **A** dell'LSA **served**) nell'LSA "richiesta" **request**. Inoltre la richiesta viene eliminata, in quanto è stata servita dall'infrastruttura, ed iniettata un LSA **go** contenente la risposta. **DIFF** è la reazione che si occupa della diffusione di

un LSA. L'LSA che deve viaggiare da un nodo all'altro deve rispettare una precisa struttura (in questo caso `go(LSA_da_diffondere, IP_host_destinazione)`).

Riportiamo, per completezza, il frammento di codice JAVA che installa la reazione BIND all'interno del nodo:

```
SimpleReaction bind = new SimpleReaction(IAAlgorithm.TIME);
LogicContentFilter bind_f1 = new LogicContentFilter("service(Q)");
LogicContentFilter bind_f2 = new LogicContentFilter("request(Q)");
bind.getReagents().add(bind_f1);
bind.getReagents().add(bind_f2);
bind.getTransaction().getOperations().add(
    new SpaceOperation(
        SpaceOperation.UPDATE, "0", null, new LogicLsaContent("served(Q)")));
manager.addReaction(bind);
```

Funzionamento applicazione Il servizio `SapereService` si presenta come in figura 4.4, all'atto della creazione occorre specificare la porta con cui esso interagisce con il middleware (che viene poi trascritta nel titolo del frame).

Dopo aver lanciato i due nodi ed i tre servizi, l'ordine di avvio non è importante ma occorre specificare correttamente le porte di comunicazione in modo da distribuire 2 servizi su un nodo ed uno sull'altro (figura 4.3).

Occorre ora chiarire la funzionalità associata al pulsante della gui “inject-observe”. Il servizio realizzato e tutte le classi ad esso associate (stub e skeleton) costituiscono un'implementazione per il testing quindi mirano a realizzare tutti gli schemi di interazioni utili per verificare le funzionalità del middleware. A questo proposito si è notato che disaccoppiare l'operazione `inject` dall'`observe` può causare, causa la concorrenza interna del sistema, un interleaving delle operazioni tale per cui all'atto dell'osservazione, l'LSA in questione, potrebbe non esistere più. Per aggirare questa situazione, che non è considerata comunque indesiderata ma fisiologica, si è introdotta una logica per la gestione atomica delle due operazioni (controllata dal pulsante “inject-observe”).

Ogni servizio che vuole interagire con il middleware deve effettuare la `inject` di un LSA ed in particolare:

- il servizio *service* dovrà effettuare la “inject-observe” di un LSA `service(example)` dove `example` rappresenta la descrizione del servizio offerto.
- il servizio *request* dovrà effettuare la “inject-observe” di un LSA `request(example)` dove `example` rappresenta la descrizione del servizio richiesto.
- il servizio *CR* dovrà effettuare la “inject-observe” di un LSA `stat`.

Appena lo spazio rispetta le precondizioni per l'attivazione di una reazione questa scatta generando la generazione delle relative notifiche che giungono ai servizi.

Nell'esempio per prima scatta la reazione **BIND**. Quando il *service* riceve la notifica che l'LSA **service** si è modificato in **served** può modificarlo a sua volta specificando nella GUI l'id dell'LSA ed il nuovo contenuto che sarà **served(service,ok)** dove **ok** è una qualsiasi risposta. A quel punto scatta la reazione **RESP** che diffonde la risposta **response(ok)** e modifica l'LSA **request(example)** in **request(example,ok)** dopodiché lo rimuove. Da notare che anche se l'operazione di remove a prima vista vanifica l'update in realtà non è così: in corrispondenza dell'update viene, infatti, generata la notifica che giunge al servizio *request*.

Lo spazio a questo punto contiene l'LSA **go** per cui scatta la reazione **DIFF**. Sul secondo nodo, se l'operazione di **inject(stat)** è stata eseguita scatta la reazione **CR** che modifica l'LSA **stat** in **stat(Q)** notificando il servizio *CR*.

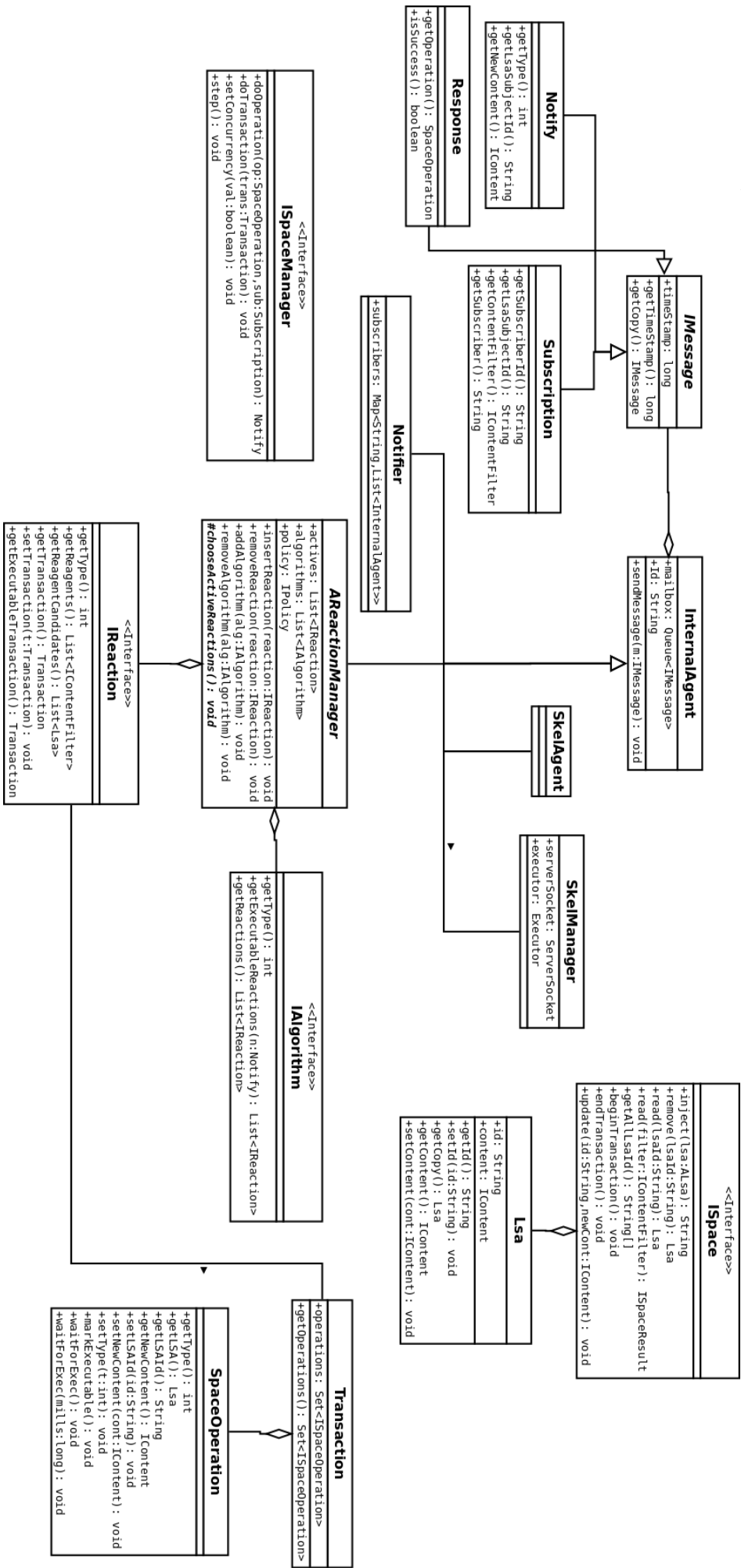


Figura 4.2: Class Diagram

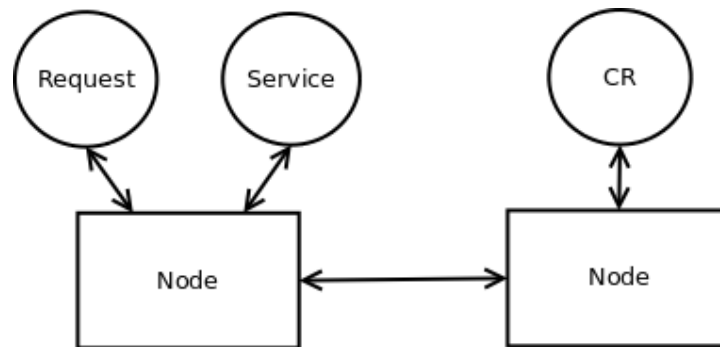


Figura 4.3: Esempio applicativo: CR = context recognition service, Request = request service, Service = server service

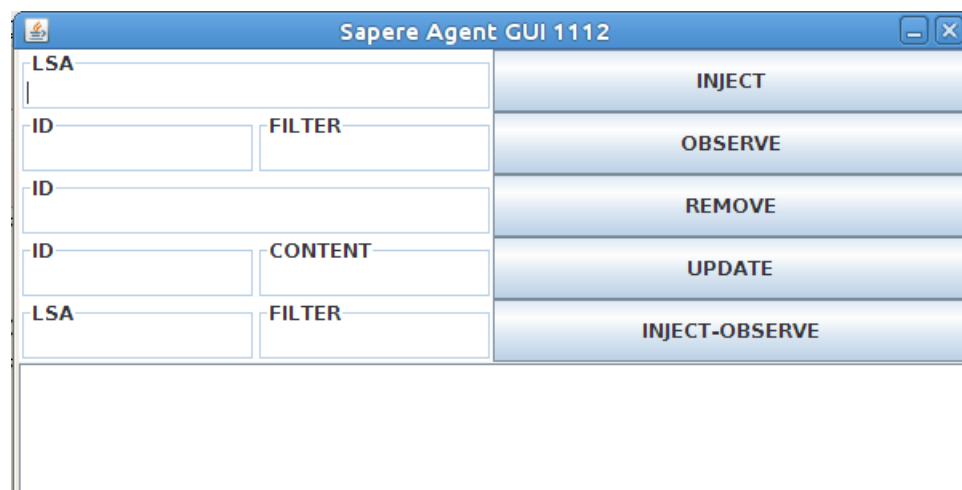


Figura 4.4: Sapere Service GUI

Conclusioni

Gli ecosistemi di servizi rappresentano un settore di ricerca in espansione e forniscono approcci e tecniche estremamente innovative per affrontare la complessità crescente dei sistemi software futuri. Tali sistemi dovranno essere personalizzabili in base ai bisogni degli utilizzatori riducendo al massimo l'intervento umano per la manutenzione ed il controllo.

Il progetto SAPERE ha gettato le basi per la definizione dei requisiti di una possibile infrastruttura per ecosistemi di servizi pervasivi: con questa tesi si è raffinata tale definizione e si è affrontato il progetto e la realizzazione di un prototipo che offre un'ottima base per gli sviluppi futuri.

Analizzando i requisiti si sono definite le funzionalità architetturali e successivamente si sono analizzate varie tecnologie utilizzabili come possibile piattaforma operativa per lo sviluppo del sistema. A seguito dell'analisi delle varie proposte, nessuna di queste è stata valutata idonea e, quindi, si è deciso di procedere con lo sviluppo tout court.

Di ogni piattaforma sono stati comunque analizzati pregi e difetti valutando i diversi approcci a problemi comuni fino ad ottenere una visione dello stato dell'arte in materia di sistemi pervasivi. Il sistema che si è progettato rispetta i requisiti di un ecosistema informatico ed integra nuove funzionalità a concetti utilizzati da altre piattaforme.

Si è scelto di non adottare alcun meta-modello di riferimento, ma si è preferito procedere seguendo le linee guida di SAPERE e della letteratura sugli ecosistemi a servizi, considerando sempre come requisito non funzionale principale la lightness.

Il prototipo sviluppato può essere facilmente esteso e specializzato. La maggior parte del codice prodotto realizza comportamenti che prescindono dai diversi meccanismi di matching o linguaggi per le eco-laws. Questo consente agli sviluppatori futuri di poter usufruire di un core funzionante, di estenderlo senza manipolarlo sostanzialmente.

Si è trascurato di definire aspetti legati alla sicurezza o particolari protocolli per le comunicazioni di rete, in quanto estranei allo scopo principale della tesi: fornire un'architettura di riferimento ed una solida base implementativa altamente estensibile.

L'esempio applicativo costituisce una concretizzazione che può essere utilizzata a prescindere dallo scenario, soprattutto in fase di testing di applicazioni future.

Bibliografia

- [1] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm intelligence: from natural to artificial systems*. Oxford University Press, Inc., New York, NY, USA, 1999.
- [2] Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, and Ingegneria Università. Not for Distribution or Attribution : for Review Purposes Only MARS : a Programmable Coordination Architecture for Mobile Agents. (i):1–20.
- [3] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. In *Coordination Languages and Models*.
- [4] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, January 1985.
- [5] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.
- [6] Ramesh Govindan and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. pages 56–67, 2000.
- [7] S. Johnson. Wild things. 2002.
- [8] Christine Julien and Gruia-catalin Roman. EgoSpaces : Facilitating Rapid Development of Context-Aware Mobile Applications TR-UTEDGE-2005-004 EgoSpaces : Facilitating Rapid Development of Context-Aware Mobile Applications. *Engineering*, 2006.
- [9] Christine Julien, Gruia-catalin Roman, and Qingfeng Huang. SICC : Source Initiated Context Construction in Mobile Ad Hoc Networks Qingfeng Huang SICC : Source-Initiated Context Construction in Mobile Ad Hoc Networks. 2006.
- [10] Marco Mamei and Franco Zambonelli. Theory and Practice of Field-Based Motion Coordination in Multiagent Systems. *Applied Artificial Intelligence*, 20(2-4):305–326, February 2006.

- [11] Andrea Omicini. Formal ReSpecT in the A&A perspective. *Electronic Notes in Theoretical Computer Science*, 175(2):97–117, June 2007. 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA’06), CONCUR’06, Bonn, Germany, 31 August 2006. Post-proceedings.
- [12] Gian Pietro Picco, Amy L Murphy, and Gruia-catalin Roman. Lime : Linda Meets Mobility. *Computer*.
- [13] R. Poor. Embedded networks: pervasive, low-power, wireless connectivity. 2001.
- [14] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. CArtAgO : A Framework for Prototyping Artifact-Based Environments in MAS. pages 67–86, 2007.
- [15] Mirko Viroli and Matteo Casadei. Biochemical tuple spaces for self-organising coordination. In John Field and Vasco T. Vasconcelos, editors, *Coordination Languages and Models*, LNCS, pages 143–162. Springer, Lisbon, Portugal, June. 11th International Conference (COORDINATION 2009), Lisbon, Portugal, June 2009. Proceedings.
- [16] Mirko Viroli and Franco Zambonelli. A biochemical approach to adaptive service ecosystems. *Information Sciences*, 180(10):1876–1892, 2010.
- [17] Sascia Virruso. Un approccio semantico per la coordinazione di sistemi pervasivi. 2010.
- [18] Franco Zambonelli and Mirko Viroli. From service-oriented architectures to nature-inspired pervasive service ecosystems. 2010.