

La coordinazione nei sistemi ad agenti mobili

*Davide Rizzo, Corso di Laurea in Ingegneria Informatica, A.A. 2001/2002
Prof. Alessandro Genco*

ABSTRACT

Parecchi sforzi da parte dei ricercatori nell'area degli agenti autonomi ed intelligenti si sono concentrati negli aspetti tipicamente *intra-agent*, come i linguaggi e le architetture strettamente legati alla struttura interna dell'agente. I sistemi multi agente invece, non potendo essere considerati semplicemente come una moltitudine di individui, hanno bisogno di definire le caratteristiche del mondo che li ospita, nonché le regole in vigore e gli individui che lo popolano: in una parola hanno bisogno di un modello di coordinazione. Nel seguito, dopo avere introdotto il concetto di coordinazione, viene adottata una tassonomia dei modelli attualmente disponibili che sono analizzati in dettaglio (anche per mezzo di un piccolo esempio applicativo di data retrieval); vengono poi studiate in dettaglio alcune tra le soluzioni più diffuse in ambito distribuito (IBM Aglets, JavaSpace, Mars ecc) per metterne in evidenza le caratteristiche ed gli eventuali difetti. Infine vengono messe a confronto le due soluzioni estreme nel caso pratico dell'implementazione di un applicativo per gestire le aste su Internet. Chiude il documento una piccola panoramica sui progetti futuri.

1. Introduzione

La coordinazione è un aspetto chiave del mobile computing, poiché, nella teoria come nella pratica, è necessario separare il trattamento dei singoli componenti dalla gestione delle interazioni tra i componenti stessi [1]. Implementare adeguati strumenti di coordinazione può comportare diversi vantaggi, soprattutto perché viene limitata la conoscenza che un componente deve avere degli altri per comunicare con essi.

Cominciamo col dire che un modello di programmazione (*programming model*) completo può essere costituito da due parti: un modello computazionale (*computation model*) ed un modello di coordinazione (*coordination model*) [2]. Il modello computazionale permette ad un programmatore di costruire una singola entità

computazionale: esso comporta dunque un tipo di computazione single-threaded e step-at-a-time. Il modello di coordinazione consente invece di unire entità separate in un insieme di entità asincrone che comunicano. Un'entità è un programma, un processo, un thread o, in generale, una qualsiasi entità capace di simulare una macchina di Turing. Per completezza, diciamo che i suddetti insiemi di entità vengono definiti da alcuni ricercatori *ensembles*.

Un modello computazionale ed un modello di coordinazione possono venire integrati in un unico linguaggio, oppure possono rimanere separati in due linguaggi distinti, nel qual caso i programmatori scelgono per ognuno uno specifico modello di riferimento. Questa seconda soluzione è probabilmente la migliore, dato che, se viene trattato in modo ortogonale rispetto alla computazione, il problema della coordinazione di insiemi asincroni (*asynchronous ensembles*) può essere risolto più facilmente.

La coordinazione, ad ogni modo, non è un banale scambio di informazioni, ma più propriamente uno scambio di informazioni tra agenti attivi [3]. Ogni linguaggio di coordinazione deve permettere agli agenti di comunicare con altri agenti il cui stato è in continua ed imprevedibile evoluzione [4]. Questo è il motivo per cui l'ortogonalità è più che auspicabile. Tuttavia, sono state spesso intraprese strade alternative all'ortogonalità, e quindi alla separazione tra coordinazione e computazione. Per esempio, la maggior parte dei sistemi distribuiti tradizionali si basa sulle RPCs (Remote Procedure Calls), che però risultano inapplicabili nel campo delle applicazioni parallele (*parallel applications*).

Ogni modello di coordinazione è costituito da tre elementi [24][26]:

1. *Coordinables*: sono le entità le cui interazioni vengono regolate dal modello (es: processi Unix, threads, ecc.);
2. *Coordination Media*: sono le astrazioni che consentono le interazioni (es: semafori, monitors, tuple spaces, ecc.);
3. *Coordination Laws*: possono essere definite in termini di *communication language* (linguaggio per lo scambio di informazioni e strutture dati) e *coordination language* (insieme di primitive di interazione).

Definiti questi elementi, aggiungiamo che la coordinazione consiste di meccanismi di identificazione delle entità più prossime, di scambio di informazioni, di sincronizzazione, ecc. che possono essere:

- *espliciti*: un'entità fa riferimento ad un'altra quando vuole inviargli un messaggio;
- *impliciti*: la coordinazione è totalmente trasparente alle entità.

Vedremo più avanti in cosa consista questa differenza, ad ogni modo è opportuno sottolineare che un aspetto importante della coordinazione è la capacità di determinare *who else is around* (quali altre entità sono raggiungibili). Una minima conoscenza degli altri partecipanti alla computazione è infatti richiesta alle unità mobili nella maggior parte dei modelli, malgrado tali modelli vengano spesso definiti *trasparenti*. Questa conoscenza, d'altro canto, potrebbe essere usata per ottimizzare le prestazioni, anche se questo aspetto esula dagli obiettivi di questo documento.

1.1 La coordinazione nei sistemi ad agenti mobili

Le tradizionali applicazioni distribuite sono sviluppate attorno ad un set di processi staticamente assegnati a determinati ambienti di esecuzione, cooperanti tra di loro senza un controllo diretto da parte della rete. Il paradigma ad agenti mobili, invece, definisce le applicazioni come un set di entità attive (agenti) capaci di cambiare il loro ambiente di esecuzione [13], trasferendosi in altri nodi durante le loro attività (mobilità). La tendenza a sviluppare software secondo tale paradigma è ampiamente giustificata dai vantaggi apportati rispetto agli approcci tradizionali [30] in quanto, 1) gli agenti mobili riducono pesantemente la banda passante richiesta per le loro attività, muovendosi localmente nelle risorse di cui hanno bisogno piuttosto che richiedendo il trasferimento di vaste quantità di dati; 2) possono migrare assieme al codice per pilotare risorse remote senza avere la necessità di un server specifico, portando così ad uno scenario più flessibile; 3) non richiedono una connessione di rete costante, poiché le entità possono muoversi non appena essa sia disponibile e rinviare le interazioni in caso di mancato collegamento e, come conseguenza 4) il paradigma ad agenti mobili intrinsecamente si adatta ai sistemi per il mobile computing.

Negli ultimi anni sono apparsi parecchi sistemi ed ambienti di programmazione per lo sviluppo di applicazioni distribuite basate su agenti mobili, nonostante ancora oggi vi siano ampie strade da percorrere nella ricerca per arrivare ad una tecnologia ampiamente accettata e riconosciuta come stabile in relazione agli appropriati linguaggi di programmazione, ai modelli di coordinazione, alla sicurezza, l'efficienza e la standardizzazione

Anche nelle applicazioni ad agenti mobili, una delle attività fondamentali è la *coordinazione* tra gli agenti e le entità che questi incontrano durante la loro esecuzione, siano essi altri agenti o semplici *risorse* (hardware e/o software) disponibili sull'ambiente di esecuzione. Ad ogni modo, la mobilità e la vastità dello scenario relativo a tale paradigma introducono problemi diversi rispetto a quelli tradizionali basati sul paradigma RPC (remote procedure call).

Lo scenario si complica ulteriormente se si considerano le applicazioni in cui il numero degli agenti coinvolti è molto alto: se da una parte infatti i linguaggi appositi come FIPA e KQML riescono in qualche maniera a risolvere il problema della coordinazione mediante la comunicazione, mentre per i problemi di interoperabilità soluzioni come quella proposta da CORBA hanno dalla propria un ampio livello di maturità, tali soluzioni si concentrano soprattutto sulla comunicazione peer-to-peer e mancano di una visione globale dell'insieme agenti/ambienti di esecuzione. Da queste considerazioni nasce il bisogno di un modello di coordinazione specifico nel caso di sistemi ad agenti mobili, inteso come "l'arte di gestire le interazioni e le dipendenze tra attività o – nel contesto degli agenti mobili – tra agenti". Per cominciare a fissare le idee, diciamo che un modello di coordinazione risulta necessario per:

- evitare l'anarchia ed il caos al fine di raggiungere un obiettivo comune (gli agenti potrebbe anche non essere al corrente della loro natura collaborativa)
- scambiare risorse ed informazioni distribuite
- gestire le dipendenze tra le azioni degli agenti
- incrementare l'efficienza complessiva

Nel seguito ci occuperemo soprattutto delle caratteristiche dei coordination media e delle coordination laws dei sistemi ad agenti mobili, tralasciando la descrizione degli agenti stessi e della loro morfologia.

2. I modelli di coordinazione

Come già anticipato, durante la propria vita da nomade, un agente ha bisogno di coordinare le proprie attività con altre entità che possono essere sia agenti che risorse. Più in dettaglio, un'applicazione può essere composta da parecchi agenti mobili che cooperano per portare a termine un compito assegnato, e per il quale dunque hanno bisogno di coordinare le proprie attività

In particolare, ci chiederemo a) come gli agenti comunicano tra di loro e sincronizzano le rispettive attività, e b) come gli agenti interagiscono con gli ambienti di esecuzione.

Il primo punto riguarda da vicino il primo tipo di coordinazione, al quale ci si riferisce col nome di *inter-agent coordination* (coordinazione inter-agente). Il problema di coordinare agenti nasce dal fatto che un'applicazione può essere costituita da parecchi agenti (eventualmente mobili) che cooperano al fine di perseguire un obiettivo e per il quale devono assolutamente sincronizzare il loro lavoro, nonché scambiarsi dati e conoscenza in generale.

Il secondo punto è particolarmente critico in quei casi in cui gli agenti mobili debbano muoversi attraverso Internet per accedere a risorse remote e servizi localizzati su nodi della rete. Infatti, quando gli agenti migrano su un nodo, hanno bisogno di accedere alle risorse e servizi disponibili nel nuovo ambiente che li ospita. A questo secondo tipo di coordinazione ci si riferisce con nome di *agent-to-hosting-environment coordination*.

Studi intensivi sulle caratteristiche della coordinazione [4] hanno portato alla definizione di diversi modelli che andremo ora ad analizzare.

2.1 Tassonomia dei modelli di coordinazione

In questa sezione viene definita una semplice ma esaustiva tassonomia dei possibili modelli di coordinazione per le applicazioni ad agenti mobili: per cominciare, diciamo che le differenze sostanziali tra le varie proposte ruotano attorno ai concetti di *accoppiamento spaziale e temporale* [5]. In particolare:

1. i modelli di coordinazione spazialmente accoppiati, d'ora in poi chiamati *spatially coupled*, richiedono che le entità coinvolte debbano condividere un *name space* comune, mentre i modelli *spatially uncoupled* rappresentano le interazioni di tipo anonimo
2. i modelli di coordinazione temporalmente accoppiati, d'ora in poi chiamati *temporally coupled*, inducono una forma di sincronizzazione tra le entità, mentre quelli *temporally uncoupled* portano ad interazioni asincrone.

Da quanto appena detto seguono direttamente le quattro grandi famiglie di modelli [6][7][8], in questo contesto chiamati *direct*, *blackboard-based*, *meeting-oriented* e *Linda-like* (Figura 1).

		Temporal	
		Coupled	Uncoupled
Spatial (Name)	Coupled	Direct <i>Odissey, Agent-TCL</i>	Blackboard-Based <i>Ambit, ffMain</i>
	Uncoupled	Meeting-Oriented <i>Ara, Mole</i>	Linda-like <i>Jada, MARS, TuCSoN</i>

Figura 1 Tassonomia dei modelli di coordinazione

Come esempio illustrativo, considereremo una semplice applicazione di data retrieval su Internet [7], in cui si suppone che un agente sia mandato su di un sito remoto per analizzare le pagine html e restituire gli URL che contengono una specifica parola chiave. L'agente clonerà se stesso per ogni link remoto trovato sulle pagine analizzate e manderà i cloni nei suddetti siti. Una coordinazione inter-agente sarà necessaria per evitare visite multiple di diversi agenti sullo stesso sito, mentre la coordinazione agent-to-hosting-environment si prenderà cura di stabilire un preciso protocollo per accedere alle informazioni sul sito.

Per completezza, prima di procedere con l'analisi dettagliata dei modelli appena introdotti, diciamo che esiste una seconda tassonomia delle architetture di coordinazione, proposta da Nwana [9] ed accettata in alcuni ambiti di ricerca: seconda tale schema, un modello di coordinazione può essere classificato, a seconda delle tecniche implementative utilizzate, in una delle quattro categorie seguenti:

1. *strutturazione organizzativa*: vengono definiti dei pattern organizzativi a priori allo scopo di stabilire implicitamente le responsabilità, le capacità e la connettività di tutti gli agenti coinvolti. Vengono dunque fissate relazioni a medio/lungo termine tra le entità che dovranno coordinare le loro attività per svolgere un dato compito – tipicamente vengono adoperati i paradigmi master/slave e client/server
2. *contrattazione*: basata sul cosiddetto Contract Net Protocol, basata sulla metafora di una struttura di tipo mercato decentralizzata
3. *multi-agent planning*: gli agenti definiscono un piano decisionale che descriva tutte le azioni e le interazioni necessarie al raggiungimento degli obiettivi prefissati. La pianificazione (planning) può essere centralizzata o decentralizzata
4. *negoziazione*: intesa come "...processo di comunicazione di un gruppo di agenti al fine di raggiungere un mutuo accordo di un qualche tipo [10]. In questo ambito si classificano i modelli di negoziazione *game theory-based*, *plan-based* e *human-inspired*.

2.1.1 Coordinazione Direct

Nei modelli di coordinazione di tipo direct (Figura 2), gli agenti iniziano a comunicare nominando esplicitamente i partner coinvolti (spatial coupling). Ciò implica intrinsecamente una forma di sincronizzazione temporale (temporal coupling). Per quanto riguarda la coordinazione inter-agente, i due agenti devono accordarsi su di un protocollo di comunicazione comune, tipicamente di tipo peer-to-peer. La coordinazione con le risorse

disponibili sull'host (coordinazione agent-to-hosting-environment) invece fa uso del classico paradigma client-server.

Pur presentando alcune caratteristiche interessanti e comode in particolari condizioni, l'adozione generale di tale modello di coordinazione non è consigliabile, innanzi tutto perché le continue interazioni che si richiedono hanno bisogno di connessioni di rete altamente stabili, rendendo così la comunicazione altamente dipendente dal fattore di congestionamento della rete. In aggiunta, la comunicazione tra entità che risiedono su ampie reti (come Internet) richiede protocolli di routing particolarmente complessi, che potrebbero influire in termini di prestazioni nell'esecuzione dei compiti dell'agente (l'agente dipenderebbe fortemente dalla latenza della rete).

Infine, poiché le applicazioni ad agenti mobili sono intrinsecamente dinamiche (attraverso la creazione dinamica di agenti), può essere difficile adottare un modello spatially coupled nel quale i partner coinvolti nella comunicazione debbano identificarsi in maniera precisa e puntuale; fra l'altro, gli agenti intesi come entità autonome non possono conoscere a priori da quanti componenti è composta l'applicazione alla quale appartengono (nel caso di sistemi multi-agente) e comunque, pur venendone a conoscenza, il fatto di coordinarsi tramite sincronizzazione va contro l'autonomia stessa degli agenti, che vengono così a dipendere dagli altri.

Ad ogni modo, in presenza di ampi scenari di utilizzo, la coordinazione direct può effettivamente rivelarsi vincente se applicata all'accesso a risorse locali (coordinazione agent-to-hosting-environment): un server locale rappresenta il manager di tali risorse e gli agenti interagiscono con esso tramite il paradigma client-server.

Nel nostro esempio applicativo, i server WWW locali possono fornire direttamente le pagine html agli agenti

I più noti tra i sistemi ad agenti mobili che usano il modello di coordinazione direct sono le *IBM Aglets* e *Agent Tcl*.

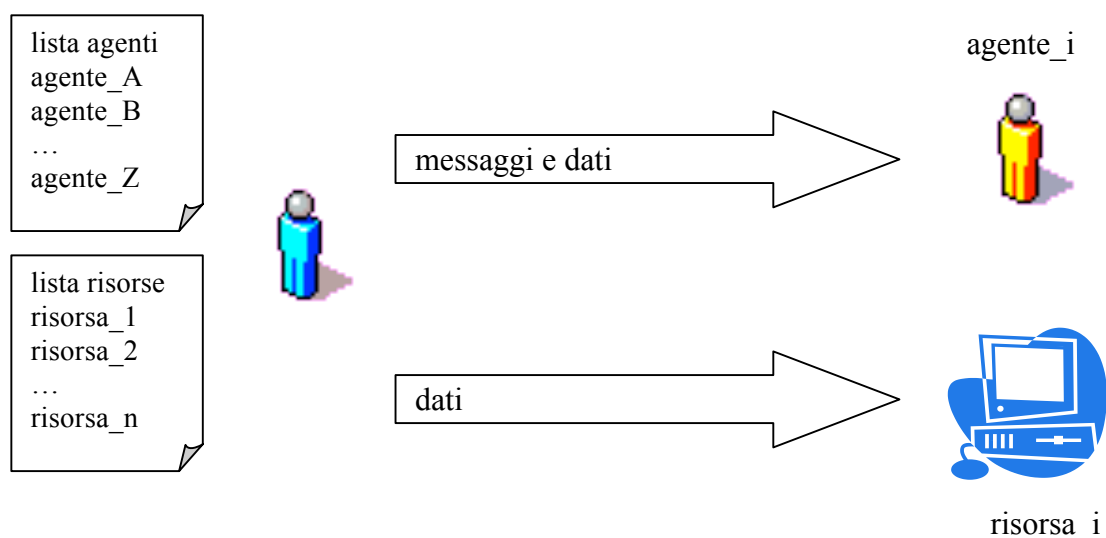


Figura 2 La coordinazione Direct: un agente comunica solo con agenti e risorse delle quali è e conoscenza mediante liste da tenere costantemente aggiornate

2.1.2 Coordinazione Meeting-Oriented

Nel modello *meeting-oriented*, gli agenti possono interagire senza la necessità di nominare esplicitamente i partner coinvolti: le interazioni piuttosto avvengono in un

determinato contesto che funge da rendez-vous per tutti gli agenti interessati. A parte i punti di incontro aperti a chiunque – che possono astrarre il ruolo di server in un ambiente di esecuzione – un'entità attiva deve necessariamente assumere la figura di supervisore (o *initiator*) per aprire materialmente lo spazio di meeting (Figura 3a). Spesso, gli incontri si rivelano vincolati ad avvenire localmente: per evitare il problema legato a comunicazioni non locali, un meeting ha inizio in un ben precisato ambiente di esecuzione, e soltanto gli agenti locali possono parteciparvi. Chiaramente, dal momento che gli agenti devono essere a conoscenza sia dei nomi usati nel meeting sia degli eventi che li forzano a unirsi al meeting stesso, non può realizzarsi un completo spatial uncoupling (Figura 3b).

Mentre il modello meeting-oriented parzialmente risolve il problema dell'identificazione esatta dei partner coinvolti, introduce comunque una stretta forma di sincronizzazione tra agenti; se pensiamo che in parecchie applicazioni la schedulazione e la posizione degli agenti è difficilmente prevedibile, il rischio di perdita di interazioni con tale modello è molto alto.

Nell'applicazione usata come test, per evitare che diversi agenti visitino lo stesso sito, si può pensare di introdurre un agente aggiuntivo per ogni sito visitato: quando un agente ha esplorato un sito, crea un agente che funga da meeting server, forzandolo a rimanere sul sito in attesa di eventuali altri agenti che potrebbero arrivare in un secondo momento. Così facendo, una volta arrivato in un sito, prima di procedere con l'esplorazione dei dati residenti, l'agente controlla se il sito è stato già visitato. Tale soluzione non è comunque particolarmente adeguata, soprattutto perché si presuppone che l'agente che funge da meeting server abbia la possibilità di rimanere attivo sul sito ospitante.

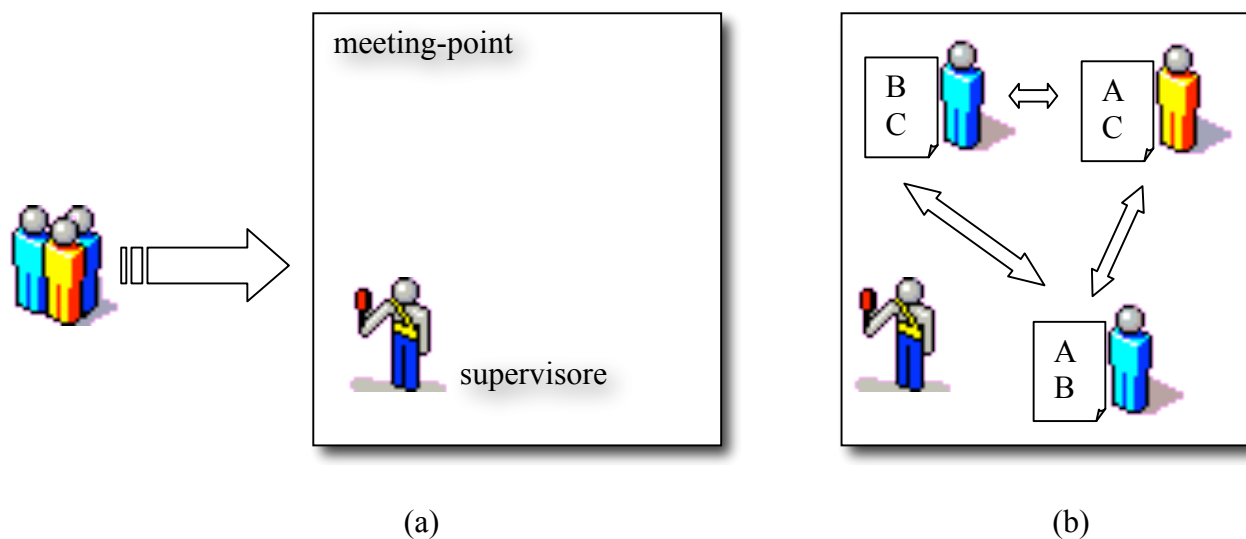


Figura 3 La coordinazione Meeting-oriented: un gruppo di agenti che debbano interagire si riuniscono in un meeting-point (a) e, dopo un fase di login gestita da un'entità supervisore, ottengono pathname espliciti dei partecipanti con i quali potere interagire (b).

La coordinazione meeting-oriented è implementata in *Ara* nel seguente modo: un agente assume il ruolo di meeting server che instaura un punto di incontro in un ambiente ospitante; gli agenti in arrivo possono parteciparvi per coordinarsi tra di loro. Il concetto di

comunicazione event-based e la sincronizzazione, definiti dal gruppo OMG ed analizzati per l'applicazione al paradigma agenti mobili, possono essere catalogati come modelli di coordinazione meeting-oriented. Associando i meeting agli eventi infatti si ottiene una forma di spatial uncoupling.

2.1.3 Coordinazione blackboard-based

Nel modello *blackboard-based*, le interazioni avvengono tramite data space condivisi, detti appunto blackboard (lavagna), ognuno locale al proprio ambiente ospitante, usati come depositi comuni per immagazzinare e recuperare messaggi. Finché gli agenti si attengono ad un comune identificatore per i messaggi e comunicano per mezzo di blackboard, essi risultano spatially coupled.

Il vantaggio sostanziale di questo modello [11] deriva dal completo asincronismo tra gli agenti: i messaggi vengono lasciati sulle blackboard senza alcun bisogno di sapere se e quando i destinatari li leggeranno (Figura 4). Si possono facilmente intuire i vantaggi che ne derivano nell'uso in ampi scenari dove la posizione e la schedulazione degli agenti non possa essere né monitorata né tanto meno garantita. In aggiunta, essendo ogni interazione tra agenti forzata a realizzarsi mediante blackboard, gli ambienti ospitanti possono facilmente controllare ciò che avviene, aumentando il grado di sicurezza che, come già visto, rappresenta uno dei punti deboli dei modelli di coordinazione tradizionali. Per quanto riguarda il nostro applicativo di riferimento, la coordinazione inter-agent si può agevolmente implementare tramite il modello blackboard: quando un agente arriva in un determinato sito, innanzi tutto controlla la blackboard locale in cerca di un messaggio che segnali l'eventuale visita di un precedente agente; se questo viene trovato, l'agente sa che il sito è stato già visitato e migra da un'altra parte, altrimenti, dopo averlo visitato, sarà lui stesso a inserire tale messaggio sulla blackboard. Riguardo la coordinazioni agent-to-hosting-environment, la blackboard può essere sfruttata per mandare informazioni agli agenti senza richiedere la presenza di un manager di risorse specializzato; inoltre l'ambiente ospitante può decidere di inserire nella blackboard le informazioni che desidera condividere, proteggendo così i dati privati da accessi incontrollati. Ad esempio l'ambiente potrebbe includere, sotto forma di messaggi, la lista di tutti e soli i files che rende pubblici.

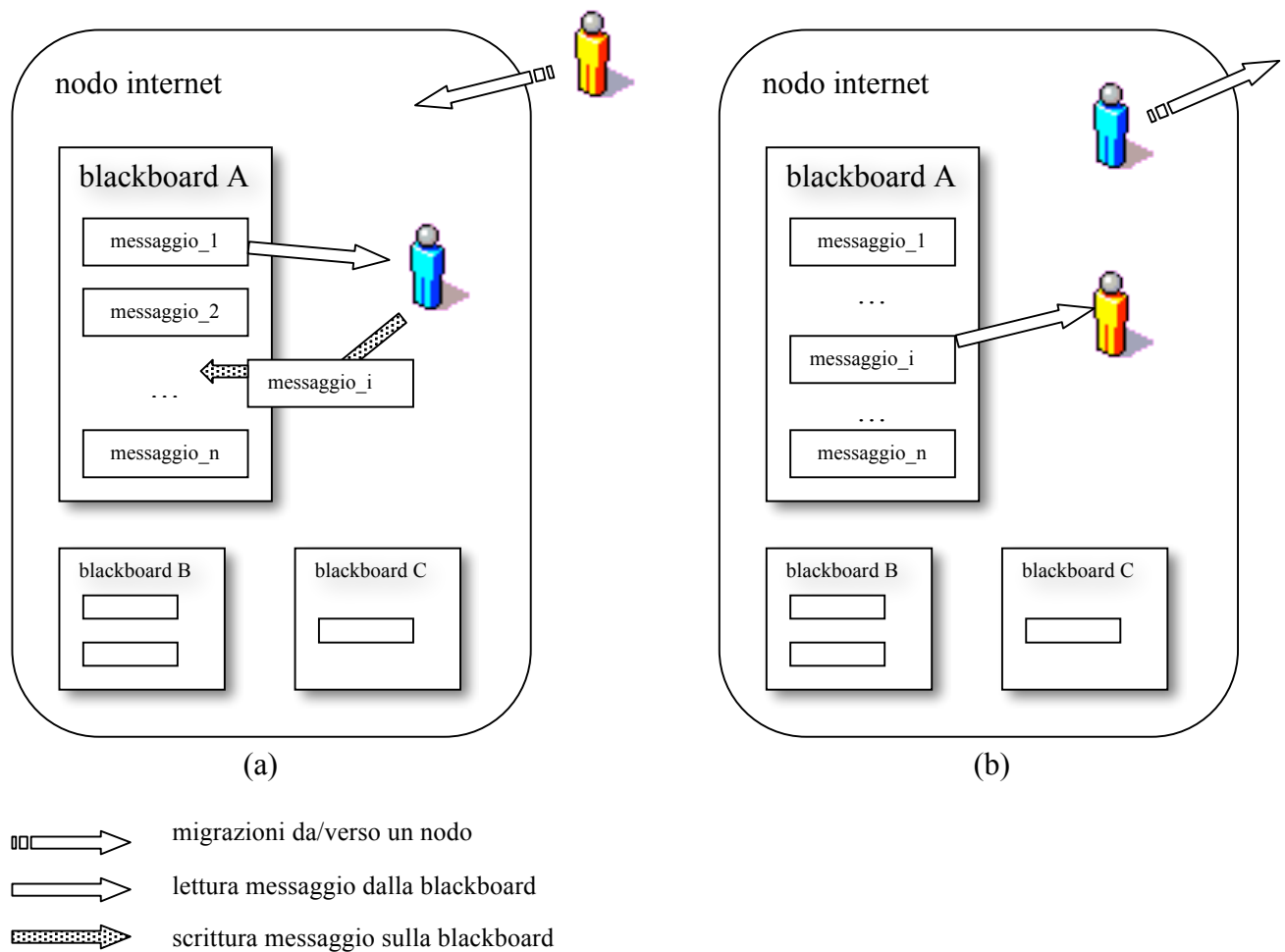


Figura 4 Coordinazione blackboard-based: un agente legge e scrive messaggi su una delle blackboard disponibili nel nodo (a); in maniera asincrona un secondo agente potrà leggere il messaggio lasciato precedentemente (b)

Sono parecchie le soluzioni disponibili che sfruttano appieno il modello di coordinazione blackboard-based: in *Ambit*, un recente modello per il mobile computing, gli agenti possono leggere e scrivere messaggi su blackboard locali dei siti che li ospitano; gli agenti di *ffMAIN*, invece, interagiscono – sia con altri agenti che con l’ambiente che li ospita – tramite un data space accessibile attraverso un protocollo HTTP, dotando così il modello di una debole forma di spatial uncoupling.

2.1.4 Coordinazione Linda-like

L’ultimo modello di coordinazione presentato, quello di tipo Linda-like, è anche quello che attualmente riscuote maggiore successo a livello di ricerca [8], pur non essendo molte le architetture mature che lo implementano. La coordinazione Linda-like si avvicina molto a quella blackboard-based, anche se viene aggiunto un meccanismo di tipo *associativo*: le blackboard associative, comunemente chiamate *tuple space*, raggiungono una forma di full uncoupling, non avendo bisogno né di una sincronizzazione temporale né di una mutua conoscenza delle entità da coordinare (Figura 5).

La coordinazione associativa si adatta perfettamente alle applicazioni ad agenti mobili: in un contesto ampio ed eterogeneo come Internet, infatti, una conoscenza completa ed aggiornata degli ambienti di esecuzione e delle altre architetture ad agenti può essere difficile, se non impossibile. D'altronde, visto che gli agenti richiedono in qualche modo un meccanismo di tipo pattern-matching per adattarsi alle situazioni incerte, dinamiche e non prevedibili, risulta comodo integrare tali meccanismi direttamente nel modello di coordinazione, semplificando la programmazione dell'agente stesso e riducendo la complessità dell'applicazione. Nel nostro esempio di riferimento il meccanismo associativo può non risultare necessario per la coordinazione inter-agente, poiché gli agenti sono a conoscenza del messaggio da recuperare dalla blackboard per evitare visite multiple; se però la nostra applicazione fosse composta da diversi tipi di agente, ognuno preposto alla ricerca secondo diverse keywords, l'associatività più che utile risulta necessaria: un agente dovrebbe controllare la blackboard alla ricerca di un messaggio di segnalazione che indichi la keyword che sia compatibile con quella di un altro tipo di agente, diversamente ignorerebbe i messaggi lasciati da agenti con keyword non compatibili con quella in suo possesso.

Riguardo le interazioni agent-to-hosting-environment, se i pathnames di tutti i files accessibili fossero disponibili sulla blackboard, gli agenti potrebbero semplicemente cercare tuple che includano il campo estensione html.

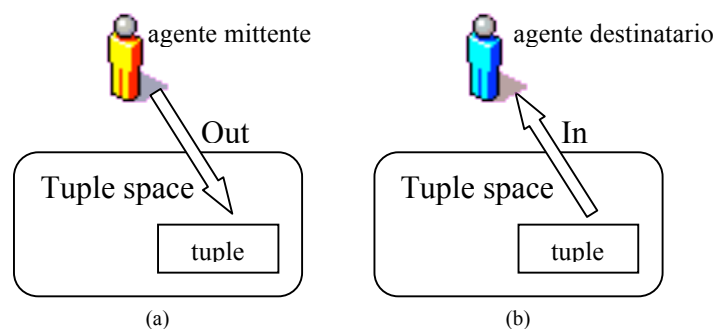


Figura 5 La coordinazione Linda-like mediante scrittura (a) e lettura (b) di tuple.

Il concetto di blackboard associative è stato implementato, oltre che dall'architettura *JavaSpace*, dal sistema *Jada*: il cosiddetto *ObjectSpace* rappresenta l'astrazione usata dagli agenti mobili per salvare e recuperare riferimenti agli oggetti in modo associativo. Inoltre gli agenti possono creare *ObjectSpaces* provati per interagire senza interferenza da parte dell'ambiente host.

Il modello Linda-like può venire esteso dotandolo della cosiddetta *reattività* [12]: un tuple space reattivo in particolare non è più solamente un semplice deposito con un meccanismo associativo neutro e senza storia bensì, dotandolo di reazioni programmabili che possano influenzare gli accessi, esso viene ad connotarsi come uno spazio con un suo stato peculiare, reagendo meglio ed in maniera più specifica alle richieste degli agenti. Le

reazioni possono fra l'altro accedere al tuple space, modificarne il contenuto ed influenzare la semantica usata dagli agenti per interagire con esso.

I vantaggi che derivano dall'adozione della reattività sono molteplici [13]: le reazioni possono essere usate per implementare specifiche politiche locali di interazione tra gli agenti e l'ambiente di esecuzione, al fine di raggiungere un miglior controllo e difendere l'integrità dell'ambiente stesso da agenti potenzialmente maliziosi; in aggiunta, le reazioni possono adattare dinamicamente la semantica delle interazioni alle specifiche caratteristiche dell'ambiente di esecuzione, semplificando quindi la programmazione dei compiti dell'agente. Ancora, un *reactive space* consente di specificare le regole della coordinazione inter-agent come se si trattasse di direttive di alto livello, separando così nettamente la fase algoritmica da quella puramente di coordinazione nella programmazione dell'agente.

D'altronde, la possibilità da parte di un'applicazione di definire il proprio tuple space contribuisce a separare chiaramente i concetti inerenti l'algoritmo da quelli puramente legati alla coordinazione: gli agenti sono incaricati di eseguire il loro compito secondo un certo algoritmo, le reazioni rappresentano le particolari regole di coordinazione specifiche per quella applicazione. Così facendo, un agente non ha bisogno di conoscere le caratteristiche peculiari ed i servizi disponibili nell'ambiente nel quale è ospitato: egli accede semplicemente al tuple space locale usando lo stesso stile di coordinazione sia per quella inter-agent che per il comportamento reattivo dell'ambiente. Inoltre il tuple space reattivo può aiutare sensibilmente ad aumentare il controllo e difendere l'integrità dell'ambiente da agenti potenzialmente dannosi.

Se da un lato parecchie proposte nel settore della coordinazione in generale puntano sulla necessità di aggiungere la reattività al modello piatto Linda, sono poche a tutt'oggi le proposte direttamente legate al mondo degli agenti mobili: il modello *TuCSon* definisce *tuple centre* programmabili per la coordinazione di agenti mobili di tipo *knowledge-oriented*; il tuple space definisce un'interfaccia di tipo Linda e le reazioni vengono programmate come tupla logiche di primo ordine. Altro esempio che può facilmente essere applicato alla tecnologia ad agenti mobili è rappresentato dal progetto *PageSpace*, il quale definisce un modello Linda-like arricchito per applicazioni Web distribuite.

2.2 Coordinazione context-dependent

Nella applicazioni Internet basate su agenti mobili, è conveniente astrarre la rete come una molteplicità di ambienti di esecuzione (es: nodi, domini ecc) e sviluppare le applicazioni in termini di agenti pienamente consapevoli della natura distribuita degli obiettivi ed in grado di muoversi da un ambiente all'altro per accedere alle risorse remote allocate. Gli agenti, durante la loro esecuzione, hanno bisogno di interagire, comunicare e sincronizzare le loro attività sia con altri agenti (coordinazione inter-agent) sia con le risorse disponibili nell'ambiente ospite (coordinazione agent-to-hosting-environment). Come già visto, entrambi le modalità di coordinazione possono agevolmente realizzarsi mediante l'uso di coordination media quali blackboard, tuple space ecc., ognuno associato ad un determinato ambiente. Tali infrastrutture ben si adattano alla natura mobile degli agenti rafforzando d'altronde il principio della località nelle interazioni. È pur vero però che quando gli agenti si inseriscono in vasti contesti, soprattutto se eterogenei e non facilmente

prevedibili, le interazioni puramente inter-agent mostrano alcuni limiti, non foss'altro per il fatto che gli agenti migrano in ambienti di esecuzione differenti durante la loro esecuzione, venendo a contatto con dati di natura diversa, organizzati in spazi diversi e gestiti da agenti diversi a seconda dei casi. Questa forma di dipendenza dal contesto è intrinseca nella mobilità degli agenti, indipendentemente dallo spazio d'interazione. Ecco perché nasce la necessità di forme di coordinazione *context-dependent* più sofisticate [14], che in qualche maniera si basino sul ruolo attivo del contesto. Tutto questo perché: 1) ogni ambiente di esecuzione ha le proprie caratteristiche e politiche di sicurezza, e potrebbe avere bisogno in qualche maniera di imporre specifiche leggi agli agenti che ospita, necessitando quindi una forma di coordinazione *environment-dependent*; 2) gli agenti di una determinata applicazione potrebbero avere bisogno di attenersi a particolari metodologie per raggiungere lo scopo per il quale sono stati programmati indipendentemente dalle caratteristiche dell'ambiente che li ospita, manifestando quindi il bisogno di una forma di coordinazione *application-dependent*.

Abilitando gli spazi di interazione a programmare dinamicamente le proprie reazioni in risposta agli eventi degli agenti tali spazi possono comportarsi in maniera differente a seconda dell'agente coinvolto nell'interazione specifica. Questa caratteristica può quindi essere usata come forma di coordinazione *context-dependent*, ed in particolare: 1) l'amministratore del sito può adattare il comportamento dello spazio di interazione per rafforzare – in maniera trasparente all'agente – specifiche politiche di sicurezza; 2) gli agenti possono dinamicamente adattare il loro comportamento su un sito in accordo alle relative leggi. Per analizzare in dettaglio questi concetti, consideriamo preliminarmente un architettura per le interazioni che sia locale e uncoupled: così facendo le interazioni vengono a essere limitate ad uno spazio preciso associato all'ambiente di esecuzione ed inoltre l'uso di uno spazio di interazione rende possibile le comunicazioni tra agenti senza la necessità di conoscerne il nome e la posizione precisa. Un modello locale e uncoupled inoltre suggerisce di pensare all'applicazione in termini di mobilità.

Quando un agente migra in un nuovo ambiente di esecuzione il suo spazio di interazione, e quindi la sua percezione del mondo circostante, cambia conseguentemente al suo movimento: ciò che otterrà da una data interazione nel nuovo sito sarà presumibilmente diverso da ciò a cui era abituato nel sito precedente. Questa forma di coordinazione *context-dependent* è intrinseca nella mobilità dell'agente e nell'adozione di spazi di interazione indipendenti e locali. Comunque, se lo spazio si riduce ad una semplice infrastruttura per immagazzinare e reperire dati, messaggi o tuple, la semplice forma di dipendenza dal contesto appena descritta diventa abbastanza complessa da gestire dall'agente. Innanzi tutto, coordinare le proprie azioni con altri agenti residenti in un ambiente estraneo richiede la soluzione ai tipici problemi che si incontrano nei cosiddetti *open systems*, cioè un comportamento non prevedibile, una certa molteplicità di linguaggi e protocolli usati, interfacce eterogenee e così via. In secondo luogo, ogni ambiente ha le sue specifiche caratteristiche e può talvolta forzare l'agente che ospita a coordinare le proprie attività in una ben determinata maniera per motivi di sicurezza e controllo delle risorse allocate. Nonostante tutte queste costrizioni, gli agenti di un'applicazione potrebbero ancora richiedere di agire in un bene determinata maniera in base a necessità di vario tipo, ma comunque legate alla propria natura e totalmente sconnesse a quella dell'ambiente ospite.

Nel caso di spazi di interazione statici e non programmabili, i problemi appena esposti porterebbero alla realizzazione di agenti molto pesanti e complessi, che richiederebbero costose attività di mantenimento per adattarsi ai continui cambiamenti degli ambienti ai

quali sarebbero destinati. In alternativa si potrebbe però pensare di avere a che fare direttamente con lo spazio di interazione, per poterlo gestire in maniera consona alle peculiarità dell'agente che dovrà sfruttarlo. Così facendo tale spazio abbandona la sua caratteristica di staticità per diventare completamente attivo e programmabile.

Ad esempio, usando un modello con tuple space programmabili, si potrebbe: 1) caratterizzare il tipo di accessi in base all'identità dell'agente; 2) esprimere un nuovo comportamento (reazione) che lo spazio dovrebbe assumere in risposta a tali accessi.

Ci si accorgerà come tale modello altro non sia che quello precedentemente presentato come Linda-like reattivo, pur se sotto il punto di vista della coordinazione context-dependent che non semplicemente come coordination media.

L'adozione dunque di uno spazio di interazione programmabile porta allo scenario descritto in Figura 6:

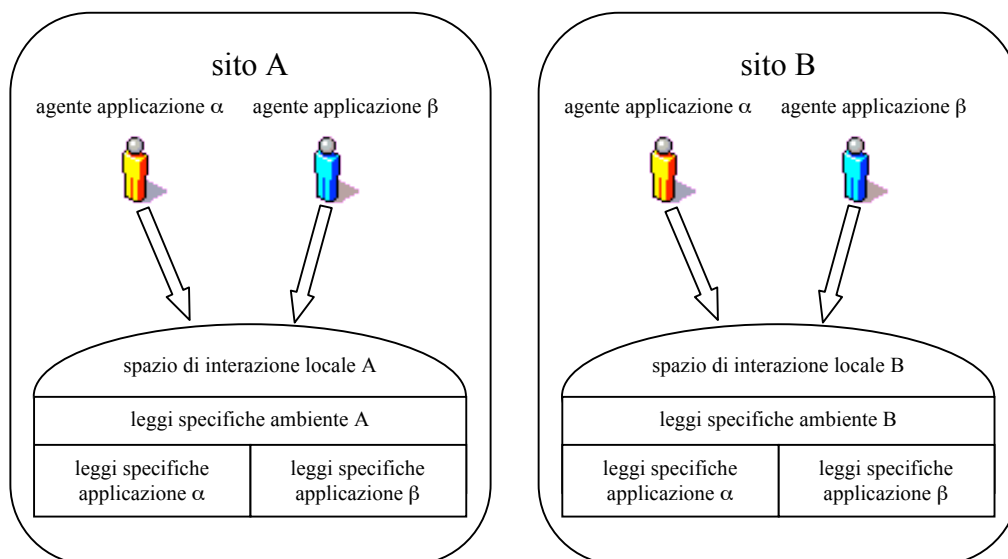


Figura 6 La coordinazione context-dependent

Gli agenti in esecuzione in un certo sito, qualunque sia l'applicazione alla quale appartengono, sono soggetti alle leggi locali dell'ambiente. Inoltre, gli agenti di una determinata applicazione possono sfruttare le proprie leggi di coordinazione sul sito che visitano al fine di influenzare le attività di coordinazione di tutti gli altri agenti della stessa applicazione.

2.2.1 Coordinazione environment-dependent

Gli agenti, nonostante accedano allo spazio reattivo, sia esso un tuple space, sempre con la stessa interfaccia, possono tuttavia presentare variazioni nella semantica delle proprie attività di coordinazione (così come nella percezione dell'ambiente che li circonda) dovute allo specifico comportamento programmato nel tuple space locale. In altre parole, le stesse

interazioni possono avere effetti differenti a seconda del sito nel quale vengono eseguite. Questa peculiarità può essere usata sia per incorporare politiche di sicurezza sia per aiutare gli agenti che entrano in un nuovo ambiente senza doversi preliminarmente ed esplicitamente accordare su tutte le caratteristiche dei mezzi che esso mette a disposizione.

Rafforzare la sicurezza.

Quando un sito si rende disponibile all'esecuzione di agenti mobili, deve essere cosciente del fatto che possibili agenti maliziosi potrebbero tentare di minare l'integrità dei propri dati e risorse, che devono dunque essere protetti da accessi inautorizzati. Gli agenti, d'altro canto, possono non essere al corrente di tali politiche di sicurezza e pertanto le loro interazioni verosimilmente possono scatenare un alto numero di security exceptions, che devono essere gestite da loro stessi. Senza entrare nel merito della sicurezza nei sistemi distribuiti, ed in particolare in quelli ad agenti mobili, diciamo comunque che se tutte le interazioni nell'ambiente sono mediate da un tuple space programmabile, i problemi appena menzionati possono trovare una potente ed elegante soluzione: l'amministratore infatti può programmare il comportamento del tuple space in modo tale da renderne l'accesso senza pericoli per l'integrità e senza delegare all'agente la risoluzione delle eccezioni che solleva. Nella solita applicazione d'esempio, se un agente alla ricerca di files html cerca di estrarre tupla che li rappresentano, mentre ha solo il diritto di leggerle – una specifica reazione programmata nel tuple space potrebbe consegnare la tupla desiderata all'agente senza però rimuoverla dal proprio spazio, e senza sollevare eccezioni di alcun tipo.

Maneggiare l'eterogeneità.

Abbiamo già avuto modo di discutere come gli agenti debbano forzatamente confrontarsi con un mondo eterogeneo. Nel caso di un tuple space reattivo bisogna tuttavia guardare il problema da una prospettiva completamente diversa dal solito: un tuple space infatti, potendosi programmare in modo tale da reagire agli accessi degli agenti, può rendere tali richieste omogenee, per lo meno alla vista degli agenti e limitatamente all'ambiente di esecuzione locale. D'altra parte gli agenti possono percepire il tuple space come se fosse conforme alle proprie aspettative, senza che per questo si debbano raddoppiare le rappresentazioni dello spazio o forzarle ad una rappresentazione meglio gestibile.

Supporto alle interazioni aperte.

L'uso della programmabilità dello spazio di interazione per adattarsi alle esigenze eterogenee si estende naturalmente a quei casi in cui un sito si suppone sia aperto all'esecuzione di agenti appartenenti ad applicazioni ed associazioni differenti, possibilmente eterogenei in termini di linguaggi e protocolli supportati ma non per questo senza il bisogno di coordinarsi gli uni con gli altri. Ancora una volta, un tuple space (o qualunque spazio di interazione che sia programmabile) può essere utile se usato come mediatore, in maniera tale da supportare le attività di coordinazione tra agenti di natura diversa. Ancora, si potrebbe volere usare un sito fidato come arbitro imparziale per le interazioni tra agenti appartenenti ad applicazioni diverse con un comportamento possibilmente troppo egoistico.

2.2.2 Coordinazione application-dependent

Gli sviluppatori possono sfruttare la programmabilità dello spazio reattivo in molti modi: possono usarlo infatti per facilitare l'accesso alle informazioni su un sito, per supportare lo scambio di grandi dati tra agenti e per implementare complessi protocolli di coordinazione. Più in generale, essi possono usare la programmabilità dello spazio reattivo in maniera tale da adattare il modello di coordinazione generale alle proprie specifiche ed ai propri bisogni. Ovviamente, poiché tali modifiche al comportamento dell'ambiente ospite sono operate per incontrare specifiche caratteristiche dell'applicazione, deve essere fornito un meccanismo di sconfinamento per assicurarsi che il comportamento relativo di un agente influenzi solo gli agenti della stessa applicazione, e non indistintamente tutti quelli che condividono lo stesso spazio di interazione ma che appartengono ad applicazioni differenti.

Includere i protocolli di coordinazione all'interno dello spazio di interazione piuttosto che nella logica dell'agente aiuta a snellire la fase di progettazione e di mantenimento dell'applicazione e dei relativi agenti, poiché sarà compito dell'amministratore dell'ambiente di esecuzione dettare delle regole per il comportamento in risposta agli accessi degli agenti, nonché aggiornarle e modificarle in risposta a precise esigenze, siano esse di sicurezza o più semplicemente di controllo.

Riguardando lo sviluppo di un'applicazione ad agenti mobili dal punto di vista della coordinazione context-dependent piuttosto che da quello tradizionale di un modello statico (sia esso di tipo direct, blackboard-based, meeting-oriented o Linda-like), ci si rende facilmente conto di come la disponibilità di una infrastruttura basata sulla programmabilità degli spazi d'interazione locali può avere un impatto positivo nell'ingegnerizzazione del software ad agenti. Infatti, tale modello invita naturalmente a sviluppare l'applicazione separando nettamente gli aspetti puramente *intra-agent* – legati ai specifici ruoli computazionali dell'agente – da quelli *inter-agent* – legati alle interazioni con altri agenti e con l'ambiente di esecuzione. Di fatto, lo sviluppo di un'applicazione può procedere parallelamente su due filoni principali:

1. *intra-agent engineering*: che tipo di informazioni devono recuperare i miei agenti dal sito che visitano? Come devono analizzare le informazioni ed estrarne dati utili? Come vengono restituiti tali dati all'utente?
2. *inter-agent engineering*: come si influenzano gli agenti tra di loro mentre visitano lo stesso sito? Che tipo di informazione possono scambiarsi attraverso lo spazio di interazione programmabile?

Riepilogando, attraverso l'uso di un modello di coordinazione context-dependent viene a configurarsi una divisione netta di ruoli tra gli sviluppatori e gli amministratori dei siti: quando un nuovo tipo di applicazione sta per essere immessa in Internet, gli amministratori di un sito possono sviluppare ed implementare tutte le leggi di coordinazione environment-dependent che ritengono necessarie per facilitare l'esecuzione della nuova tipologia di agenti e contemporaneamente proteggersi da usi impropri.

La separazione dei concetti in fase progettuale viene mantenuta anche nel codice finale: il codice che implementa le leggi di coordinazione (sia esso di tipo environment-dependent che application-dependent) è infatti separato da quello dell'agente, potendo inoltre essere aggiunto, modificato o esteso in maniera totalmente modulare ed indipendente.

Ad oggi comunque manca un'integrazione forte di tali infrastrutture di coordinazione con gli Agent Communication Languages (ACL's) per permettere agli agenti di interagire tra di loro per mezzo di espressioni di alto livello.

2.3 I linguaggi di coordinazione e Berlinda

Come già visto l'uso di un appropriato modello di coordinazione, da scegliere in base alle esigenze specifiche dell'applicazione, risulta indispensabile nello sviluppo di sistemi multi agente. L'impiego alternativo di un semplice linguaggio di coordinazione, piuttosto che di una architettura completa, può comunque risultare sufficiente se non addirittura preferibile in quei casi in cui il numero delle entità coinvolte nelle interazioni è ridotto o comunque facilmente gestibile. Secondo la definizione, un linguaggio di coordinazione per agenti definisce il tipo di elementi, le operazioni per costruire e distruggere elementi nonché le attività tipiche di coordinazione che servono a manipolare i coordination media. In tale ambito dunque, la coordinazione degli agenti viene espressa con mezzi linguistici d'alto livello, ed il comportamento risultante viene determinato dalla semantica del particolare linguaggio scelto ed implementato nell'architettura sottostante. Sono parecchie le soluzioni attualmente disponibili che fanno uso di un linguaggio di coordinazione: nell'Object Management Architecture, gli oggetti vengono manipolati attraverso Corba-API; negli approcci alla programmazione parallela come PVM si fa uso di scambio di messaggi attraverso una specifica API; KQML usa messaggi tipati definendone contestualmente la semantica. Piuttosto che presentare una panoramica delle soluzioni e delle tipologie di linguaggi che possono essere definiti linguaggi di coordinazione, fuori dagli obiettivi di questo documento, viene presentata brevemente l'architettura Berlinda [15], un modello che definisce un set di *foundation classes* comuni ai vari linguaggi di coordinazione ed in aggiunta la definizione del cosiddetto linguaggio di *meta coordinazione*, necessario a facilitare la comunicazione attraverso ambienti multipli ed eterogenei (Corba in questo senso può essere inteso sia come linguaggio di coordinazione che come linguaggio di meta coordinazione).

2.3.1 Berlinda

In sintesi, l'approccio di Berlinda consiste nella definizione di formati comuni e API specifiche per l'integrazione degli agenti, associati all'uso di un linguaggio di meta coordinazione per affrontare l'interoperabilità ed interfacciarsi con sistemi multipli: viene definito un modello altamente astratto di coordinazione in senso ampio, che viene in seguito istanziato in uno specifico linguaggio di coordinazione tra quelli supportati. La struttura comune dei vari linguaggi di coordinazione che possono essere espressi ed implementati tramite Berlinda non si ferma ai multiset usati come coordination media: vengono infatti usati i *shared medium*, astrazioni centrali per il concetto di locazione (un multiset, un deposito di interfacce o uno spazio virtuale di messaggi sono tutti esempi di shared medium). I concetti fondamentali di Berlinda, equivalenti in tutto e per tutto a quelli già introdotti in taluni casi con nomi diversi nelle architetture di coordinazione, sono:

- *coordination medium*, la struttura dati condivisa tra gli agenti. Fornisce le operazioni di manipolazione in forma di linguaggio di coordinazione. In particolare, si astrae dalla struttura dati concreta, fornendo però l'implementazione di multiset medium
- il medium è una collezione di *elements*, vettori i cui campi appartengono al set di tipi supportati dal sistema
- gli elements possono avere una *signature* (firma) che fornisca meta informazioni sull'elemento corrente
- gli elements forniscono una *matching function* che li relaziona in base alla semantica del linguaggio di coordinazione; può essere una qualche forma di meccanismo pattern matching di tipo Linda o degenerare nel semplice riconoscimento dell'identità dell'entità coinvolta
- gli *agents* sono thread di esecuzione che utilizzano le operazioni di manipolazione fornite sui coordination medium

Con questi cinque punti si coprono infatti tutti gli aspetti che si sono già studiati, con diverso approccio, nei modelli di coordinazione. Gli schemi che seguono (Tabella 1, Tabella 2) mostrano l'architettura generale di Berlinda ed il sistema Linda, visto come linguaggio di coordinazione all'interno di Berlinda.

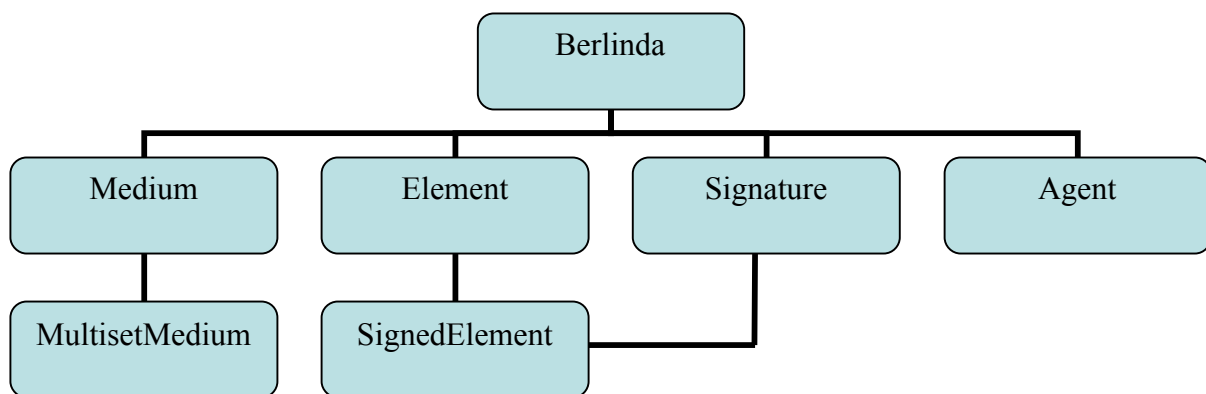


Tabella 1| modello Berlinda

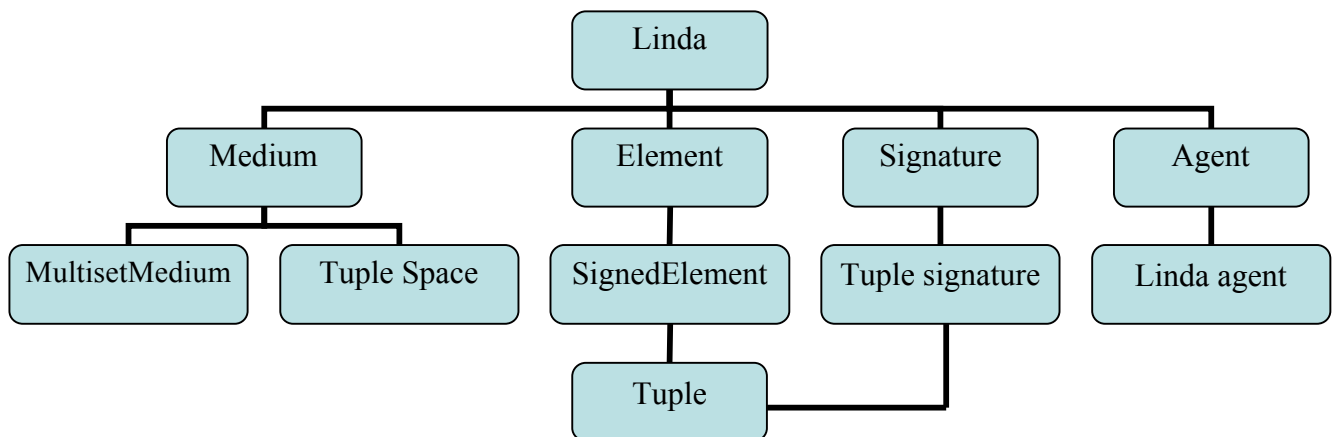


Tabella 2 Il linguaggio Linda nel modello Berlinda

3. Implementazione dei modelli di coordinazione.

Presentiamo ora, dopo avere già classificato i modelli di coordinazione, alcune tra le soluzioni più diffuse nel settore delle architetture ad agenti mobili, concentrandoci esclusivamente sulle peculiarità coordinative piuttosto che sull'intero framework.

3.1 IBM Aglets

Uno dei sistemi ad agenti mobili più diffuso è quello sviluppato dall'IBM Tokyo Research Laboratory, noto come Java Aglets [16]. Si tratta di un'estensione del linguaggio Java con supporto alla *weak mobility* (mobilità debole). Gli agenti, chiamati Aglets, altro non sono che threads Java.

Le Aglets comunicano scambiandosi messaggi nella forma di oggetti della classe *Message*, benché il sistema sia progettato in modo tale da non permettere loro di accedere direttamente ad un altro agente invocandone i metodi, anche se pubblici; al contrario, ad ogni Aglet viene associato un *proxy*, cioè una sorta di contenitore che nasconde il suo interno (l'Aglet, appunto) garantendo però che le operazioni possano effettuarsi tramite un'apposita interfaccia (Figura 7).

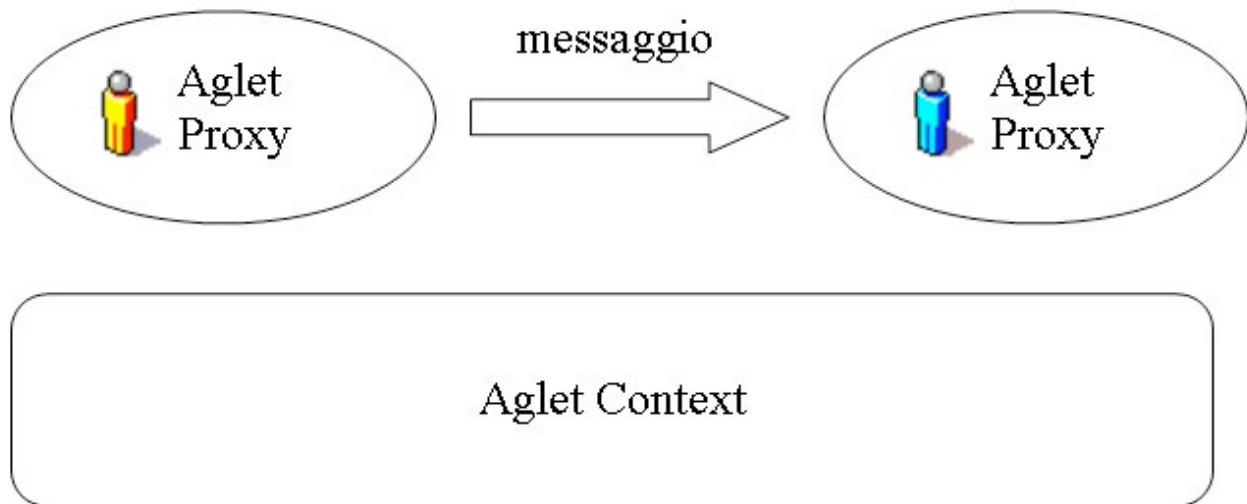


Figura 7 Coordinazione message-passing nel sistema IBM Aglets

Un messaggio viene inviato per mezzo di una chiamata all'opportuno metodo sul proxy dell'Aglet, ed ecco spiegato perché il mittente deve necessariamente tenere traccia dei movimenti dell'Aglet destinataria.

La presenza di proxy serve inoltre a ridurre, anche se di poco, l'accoppiamento spaziale e temporale del modello di coordinazione, poiché di fatto il primo viene diviso in due parti, una riguardante l'Aglet mittente e l'altra relativa al proxy dell'Aglet destinataria; in maniera simile il secondo viene alleggerito poiché limitato alla sola sincronizzazione dei proxy coinvolti nello scambio del messaggio.

Il sistema delle Aglets supporta inoltre diversi tipi di messaggio nativi per invocare specifici metodi dei proxy:

1. *sendMessage*, per mandare un messaggio e bloccare l'esecuzione corrente finché il destinatario ha completato la ricezione
2. *sendAsyncMessage*, per mandare un messaggio senza bloccare l'esecuzione corrente; il risultato verrà restituito in un secondo momento
3. *sendOnewayMessage*, per mandare un messaggio senza bloccare l'esecuzione corrente; differisce dal precedente in quanto non restituisce alcun valore

Le Aglets forniscono la nozione di *context* (contesto), che astrae l'ambiente di esecuzione e presenta un set di servizi di base, es: la creazione di un Aglet. Un servizio interessante è l'ottenimento della lista delle Aglet presenti in un determinato context. In genere, il proxy di un'Aglet viene recuperato specificando un identificatore unico per ognuna di esse, l'*AgentId*: tale operazione si può effettuare in modi diversi, come ad esempio chiedendo il context, che agisce come una sorta di name server locale, per mezzo del metodo *getAgletProxies()*; in alternativa si potrebbe usare il metodo *getProperty(String)*, assicurandoci che il sistema abbia ottenuto un riferimento attraverso il metodo *setProperty(String, Object)*.

In aggiunta, ogni context permette agli agenti di accedere alle proprie risorse locali abilitando la coordinazione di tipo agent-to-hosting.

In ultimo, essendo il sistema puramente scritto in Java, può usare i meccanismi RMI (Remote Method Invocation) del linguaggio, agendo ancora una volta sotto una forma di accoppiamento spaziale e temporale.

3.2 Ara

Sviluppato presso la University of Kaiserslautern, il sistema Ara (Agents for Remote Action) usa la stessa soluzione di base per la portabilità e la sicurezza: gli agenti vengono eseguiti su una macchina virtuale, tipicamente un interprete ed un sistema di run-time (Figura 8), che servono a nascondere i dettagli dell'architettura del sistema host ed a confinare le azioni degli agenti ad un ambiente ristretto. Concretamente, gli agenti Ara vengono programmati un qualche linguaggio interpretato [17] ed eseguito mediante un apposito interprete, supportato dal cosiddetto core di Ara. La relazione tra core e interprete è particolare: isolare le caratteristiche essenziali (come catturare le specifiche C dello stato di un agente dall'insieme del linguaggio C) nell'interprete e concentrare tutte le funzionalità indipendenti dal linguaggio adottato nel core. Per supportare la compatibilità con i modelli esistenti, Ara non prescrive un linguaggio di programmazione per i suoi agenti, bensì un'interfaccia alla quale agganciare i linguaggi desiderati: questa caratteristica rende possibile l'uso di diversi interpreti associati ad un unico core, il che rende le funzioni di quest'ultimo uniformemente disponibili a tutte le soluzioni scelte. A parte il supporto per la migrazione, le funzioni principali del core includono il management, la comunicazione, la persistenza ed i meccanismi di sicurezza.

Per quanto riguarda le scelte coordinative, Ara enfatizza le interazioni locali tra agenti per mezzo di due semplici meccanismi, il passaggio di messaggi sincro ed un tuple space. Nel primo caso, il core introduce il concetto di *service point*, simbolicamente chiamato punto di rendez-vous, dove gli agenti possono interagire sia da client che da server attraverso lo scambio di messaggi dal formato arbitrario nella forma richiesta/risposta. Ogni richiesta è marcata col nome dell'agente client che poi viene usata dall'agente server per rispondere.

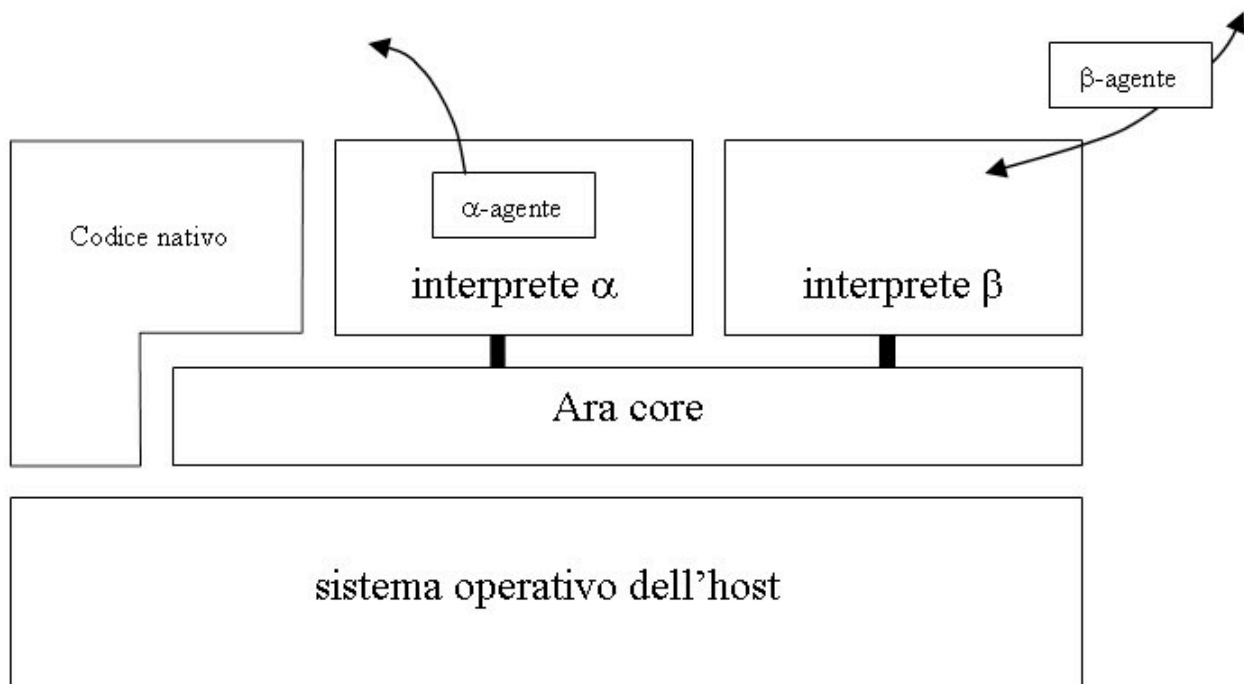


Figura 8 L'architettura del sistema Ara

L'altro meccanismo di comunicazione tra gli agenti è il tuple space implementato da un thread di sistema, accessibile come server e come service point. Gli agenti possono depositarvi dati strutturati da essere ritirati in maniera asincrona dai destinatari. Ad ogni modo, l'indipendenza dal linguaggio del sistema Ara costringe all'uso di un'interfaccia per usare tale spazio di interazione, tipicamente un sofisticato linguaggio di mapping come CORBA, rendendo ancora più complicato il meccanismo generale rispetto alla semplice soluzione del service point.

3.3 ffMAIN

Il progetto ffMAIN (Frankfurt Mobile Agents Infrastructure) fa parte di un più ampio contesto di ricerca nella mobile communication, ed è stato sviluppato con l'obiettivo di realizzare una piattaforma che consentisse la massima flessibilità sia nel tipo di applicazioni supportate che nelle metodologie di implementazione degli agenti. Inizialmente basato su http, la piattaforma base si è poi spostata verso il sistema TclHttpd e la tecnologia Perl e gli sforzi implementativi si sono concentrato su di una combinazione che fosse indipendente dal linguaggio adottato (Java, Tcl ecc) ma legata al protocollo standard http.

Alla base dell'architettura ffMAIN [18] vi è la nozione di *server*, cioè di un programma (come un mail server, un FTP server ecc) in esecuzione su ogni computer accessibile agli agenti ed incaricato dell'esecuzione degli stessi localmente. I compiti basilari prevedono quindi l'accettazione dell'agente, sia da altri server che da utenti, la creazione degli appropriati ambienti di esecuzione, la supervisione degli agenti e l'organizzazione del loro trasporto e della comunicazione.

Per ogni agente in esecuzione su di un server, viene dunque creato un ambiente di esecuzione specifico, e le risorse localmente disponibili vengono utilizzate mediante un'interfaccia apposita tra gli agenti ed il server stesso, lasciando agli agenti il ruolo di client.

Per quanto riguarda le interazioni, gli agenti comunicano tra di loro mediante quello che viene chiamato *information space*, cioè una blackboard locale ad ogni agent server contenente triple (tuple non associative con tre elementi) formate da un *item's key*, un *access control list* ed un *value*, memorizzati in forma complessa o semplicemente binaria, e non interpretate dall'agent server. Gli agenti possono leggere e scrivere nell'information space in maniera sia distruttiva che non distruttiva; le operazioni sono atomiche e serializzate per assicurare la mancanza di race condition e in generale l'inconsistenza dei dati. Per ogni elemento, le operazioni possono essere eseguite da un agente specifico, da un gruppo di agenti o da tutti gli agenti che lo desiderino, il tutto secondo delle liste di controllo d'accesso.

L'uso dell'information space ovviamente porta ad un disaccoppiamento temporale tra gli agenti e consente in maniera molto semplice l'implementazione di schemi multicast in ambito locale. A tali spazi si può infine accedere dall'esterno, ad esempio attraverso un browser WWW: è sufficiente associare agli oggetti un tipo MIME che venga restituito al browser una volta interpretato, aumentando ancora di più le possibilità di impiego di questa tecnologia nell'ambito della rete Internet.

3.4 JavaSpace

La tecnologia JavaSpace [19], sviluppata da Sun Microsystems, costituisce un semplice meccanismo unificato per la comunicazione e la coordinazione dinamica e per lo scambio di oggetti Java. JavaSpace fa uso di RMI e della serializzazione per consentire di avere la cosiddetta *distributed persistence* e permettere la realizzazione di algoritmi distribuiti. JavaSpace è dunque una nuova piattaforma per i sistemi distribuiti, e consente di semplificarne notevolmente il design e la realizzazione. Un secondo obiettivo conseguito da questa tecnologia è quello di far sì che il client-side delle applicazioni sia realizzabile interamente in Java, per di più con un numero limitato di classi. La tecnologia JavaSpace è fortemente influenzata da Linda. I *JavaSpace services* sono basati sul concetto di *entry*. Una *entry* è un gruppo *tipato (typed)* di oggetti. Tutte le *entries* sono istanze di classi Java che implementano una determinata Java interface (`net.jini.core.entry.Entry`). Il concetto di *entry* corrisponde perfettamente a quello di *tupla* tipico di Linda. I campi delle *entries* possono essere settati a *values (actuals)* di Linda) o rimanere *wildcards (formals)* di Linda). Sono consentite le operazioni di *write* (la *out* di Linda), *read* e *take* (le *rd* e *in* di Linda). Si può inoltre richiedere a JavaSpace di notificare (*notify*) quando viene scritta una *entry* che corrisponde (*match*) ad un determinato template. Ciò viene fatto con il modello ad eventi contenuto in `net.jini.core.event`.

Cerchiamo ora di chiarire il concetto di *match* (corrispondenza). Due *entries* t e u corrispondono se:

1. u è un'istanza della classe di t o di una delle sue superclassi; questo estende il modello Linda perché permette la corrispondenza anche tra tupla di tipo diverso appartenenti alla stessa gerarchia;
2. i campi di u che rappresentano tipi primitivi (es: `char`, `integer`) coincidono con quelli di t ;
3. i campi di u che non rappresentano tipi primitivi (es: oggetti) sono uguali, nella loro *serialized form*, a quelli di t ; due oggetti Java hanno la stessa *serialized form* solo se le loro *instance variables* hanno gli stessi valori (è ammessa la ricorsione attraverso l'inclusione di oggetti);
4. ad un valore nullo in t corrisponde un valore nullo in u .

Oltre a quelle di tipo semantico che abbiamo appena evidenziato, JavaSpace presenta importanti differenze da Linda: al contrario dei Linda systems, JavaSpace usa il *rich typing*. Ciò significa che le *entries* stesse, e non solo i loro campi, sono *tipate*. In altre parole, due *entries* aventi gli stessi field types ma data types diversi sono due *entries* diverse. Per esempio, un insieme di due campi numerici può rappresentare sia un punto che un vettore. Mentre in Linda due tuple rappresentanti un vettore ed un punto sarebbero formalmente uguali e quindi perfettamente indistinguibili, in JavaSpace le *entries* corrispondenti al punto ed al vettore apparterrebbero a due classi diverse; essendo oggetti, le *entries* JavaSpace hanno associato un insieme di metodi, ovvero un determinato comportamento (*behavior*); anche i campi delle *entries* sono oggetti Java. Di conseguenza, tutti i sistemi costruiti usando JavaSpace sono interamente object-oriented.

In JavaSpace, le operazioni sono invocate su un oggetto Java che implementa l'interfaccia *JavaSpace*. Tale interfaccia non è remota: ogni implementazione di un JavaSpace service esporta oggetti che implementano l'interfaccia *JavaSpace* localmente sul client, e che comunicano con l'effettivo JavaSpace service grazie ad un interfaccia

implementation-specific. Un'implementazione di un qualsiasi metodo *JavaSpace* può comunicare con un *JavaSpace* service remoto. Esaminiamo gli aspetti essenziali delle operazioni:

- *Write*: una *write* corrisponde alla *out* di Linda e inserisce una entry in un *JavaSpace* service, ovvero in uno spazio. La *write* viene invocata passando un oggetto *Entry* come parametro. L'operazione restituisce un oggetto *Lease*, che corrisponde a un *lease* espresso in millisecondi. Quando il *lease* scade, la entry viene rimossa dallo spazio;
- *Read*: possono essere effettuati due tipi di lettura, ovvero esistono due modi di cercare una entry che corrisponda ad un certo template *Entry*. In entrambi i casi, l'operazione restituisce una copia dell'entry trovata, o null se non esiste alcuna entry corrispondente al template. La *readIfExists* ritorna immediatamente una entry o null, e attende un certo timeout solo se ci sono conflitti con le possibili tuple corrispondenti al template. La *read* ordinaria, invece, attende comunque fino a che una entry corrispondente non è stata trovata, a meno che il timeout non scada;
- *Take*: questa operazione, che corrisponde alla *in* di Linda, funziona esattamente come la *read* (*take* vs *takeIfExists*), con la differenza che le entries vengono tolte dal *JavaSpace* service (spazio).

Le operazioni su uno spazio non prevedono nessun ordine di trattamento prestabilito. Un ordine di tipo inter-thread può essere realizzato solo attraverso un apposito meccanismo di coordinazione. Per esempio, se due threads T e U invocano rispettivamente una *write* ed una *read* su due entries tra loro corrispondenti, la *read* potrebbe non trovare nessuna entry anche se la *write* ritorna prima di essa. Solo se T ed U cooperano per assicurare che la *write* ritorni prima che la *read* cominci, U avrà la possibilità di leggere la tupla immessa da T (a meno delle *take* eseguita da un terzo thread).

Due aspetti innovativi di *JavaSpace* sono i seguenti:

- *Snapshot*: ogni volta che una stessa entry viene utilizzata per eseguire un'operazione, viene ripetuto un processo di serializzazione esattamente identico. Il metodo *snapshot* fornisce un mezzo per ridurre l'impatto dell'uso ripetuto di una certa entry.
- *Notify*: una richiesta *notify* invocata su un template "registra" l'interesse per le entries, tra quelle che verranno in seguito immesse nel *JavaSpace* service, che corrispondano al template stesso. Al momento dell'arrivo di ogni entry di questo tipo avrà luogo una notifica, sottoforma di eccezione.

Un concetto importante di *JavaSpace* è quello di transazione (*transaction*). La *JavaSpace* API utilizza il package `net.jini.core.transaction` per supportare la realizzazione di transazioni atomiche che raggruppino operazioni multiple in un bundle che agisca come un'operazione atomica. Le operazioni di *read*, *take* e *write* che hanno il parametro *transaction* settato a null agiscono come se si trovassero in una transazione che contiene solo l'operazione stessa.

3.5 Mars

Sviluppato all'università di Modena nel contesto del progetto di ricerca MOON [20], il sistema Mars (Mobile Agent Reactive Spaces) implementa un'architettura di coordinazione Linda-like per agenti mobili Java-based in esecuzione su reti eterogenee (Internet) [21][22]. Viene assunto che ogni nodo della rete ospita un mobile agent server in grado di accettare ed eseguire gli agenti in arrivo e che conservi permanentemente un riferimento al tuple space locale: quando un agente arriva, la prima operazione sarà sempre la richiesta di tale riferimento al server locale (Figura 9).

Mars è composto da parecchi tuple space indipendenti [23], disponibili sia per la coordinazione inter-agent che per quella agent-to-hosting-environment. Ogni tuple space è associato ad un nodo ed è accessibile da agenti che vengono eseguiti localmente. Per l'integrazione di Mars con qualunque sistema per agenti mobili è richiesto solamente che il server – che accetta ed esegue gli agenti che entrano in un nodo – fornisca agli agenti un riferimento al tuple space Mars locale.

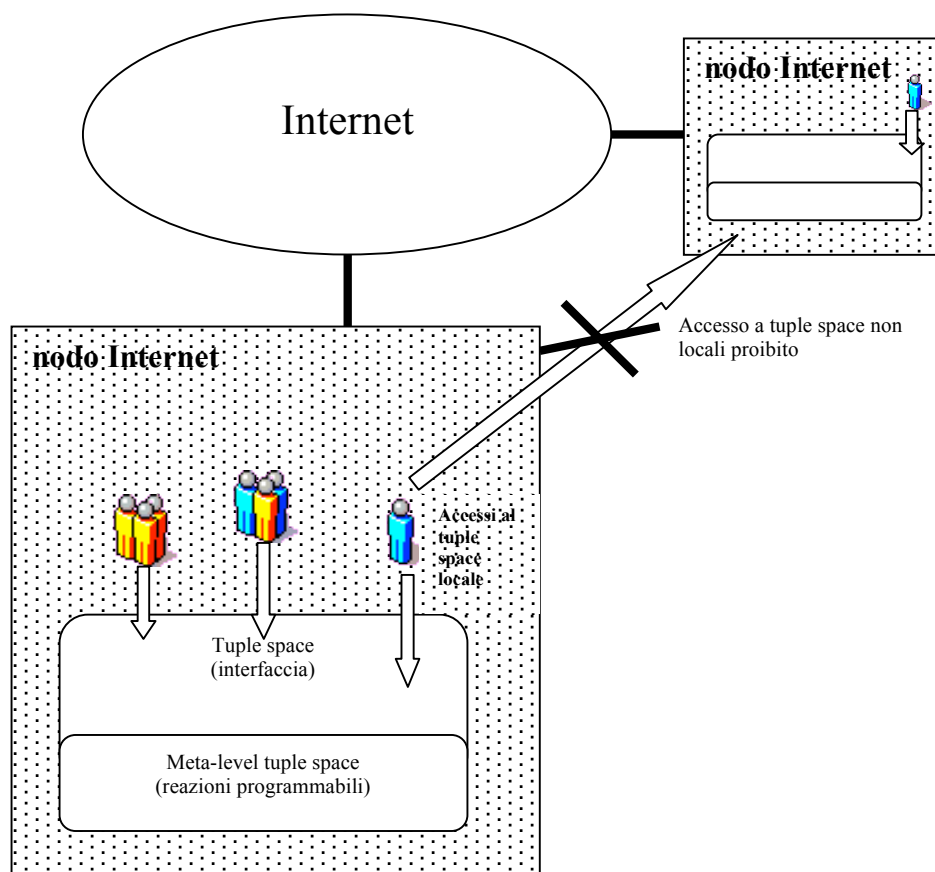


Figura 9 L'architettura Mars

Mars segue le specifiche JavaSpace, ormai quasi uno standard per quanto riguarda le interfacce Java per la gestione delle tuple; di fatto, le tuple Mars sono oggetti Java le cui

variabili di istanza rappresentano i campi della tupla stessa. Le primitive di tipo Linda che vengono fornite, e che riguardano le basilari operazioni che si possono eseguire su un tuple space, sono:

- *write*, che scrive una tupla, fornita come primo parametro, nel tuple space
- *read*, che legge una tupla compatibile al template fornito
- *take*, che lavora come read con l'eccezione che estrae fisicamente dal tuple space una tupla compatibile
- *readAll*
- *takeAll*

Le tuple usate come template con le operazioni *read*, *readAll*, *take* e *takeAll* possono avere sia campi con valori esplicitamente definiti che valori nulli. Una tupla T sarà compatibile (matching tupla) con un template U se i valori definiti di U sono uguali ai corrispondenti valori di T, seguendo le regole del pattern-matching di JavaSpace.

In aggiunta, ogni tuple space Mars può reagire agli accessi con un comportamento programmabile definito nel così detto *metalevel tuple space*. Infatti, Mars definisce un'architettura flessibile ed altamente controllabile per programmare le reazioni agli accessi degli agenti ai tuple spaces. Usando una tupla con quattro campi (4-ple) nella forma (Rct, t, Op, I), ad esempio, Mars può fare in modo da scatenare una reazione compatibile Rct quando l'agente I invoca l'operazione Op sulla tupla T. La scrittura o l'estrazione di una tupla nel metalevel tuple space da parte dell'amministratore di rete o di agenti autorizzati comporta rispettivamente l'installazione o la rimozione di particolari reazioni. Una reazione può influenzare gli effetti di determinate operazioni, in quanto si basa sul comportamento del tuple space corrente e sui precedenti accessi al medesimo. E' facile intuire come i vantaggi sottolineati dall'uso di questo modello siano importanti in termini di flessibilità, controllo e semplicità di programmazione.

L'intento del progetto Mars non è costruire un nuovo ambiente di esecuzione per gli agenti mobili che soppianti i frameworks esistenti, quanto lo sviluppo di un'architettura di coordinazione generale e portabile da affiancare ai motori già disponibili. Il tuple space Mars è debolmente connesso all'agent server, pertanto è associabile a diversi sistemi ad agenti mobili, come le Aglets IBM o il motore Odyssey, tanto per citarne alcuni.

3.5.1 L'interfaccia Mars

Le tuple Mars sono oggetti Java che rappresentano set ordinati di tipi, non necessariamente elementari. Per definire una tupla bisogna estendere la classe *Entry* (che definisce le proprietà base delle tuple) e definire, come variabili d'istanza, i specifici campi della tupla. Ogni campo della tupla può anche rappresentare tipi di dati primitivi (oggetti *wrapper*, nella terminologia Java), e possono presentare sia valori definiti o nulli.

Il tuple space è realizzato come oggetto Java, e precisamente è un'istanza della classe *Space* che implementa l'interfaccia con la quale gli agenti possono accedere al tuple space. L'interfaccia adottata estende quella descritta nelle specifiche JavaSpace, l'architettura candidata a diventare standard de facto in quest'ambito.

```

Public interface Mars extends JavaSpace {
    // metodi di interfaccia ereditati da JavaSpace
    // void write(Entry tup, Transaction txn, Identity who); // inserisce una tupla nel tuple space
    // Entry read(Entry req, Transaction txn, Identity who); // legge una tupla dal tuple space
    // Entry take(Entry req, Transaction txn, Identity who); // estrae una tupla dal tuple space

    // metodi aggiunti da Mars
    Vector readAll(Entry req, Transaction tx, Identity who); // legge tutte le tuple dal tuple space
    Vector takeAll(Entry req, Transaction txn, Identity who); // estrae tutte le tuple dal tuple
    space
}

```

L'interfaccia JavaSpace definisce tre operazioni per accedere al tuple space:

- *write*, per inserire una tupla, fornita come parametro, nello spazio;
- *read*, per leggere una tupla dallo spazio, sulla base di una tupla fornita come parametro e da usare come pattern per il meccanismo di matching
- *take*, che funziona come read con la differenza che estrae la tupla dallo spazio.

In aggiunta ai parametri di tipo tupla, il parametro *transaction* specifica le caratteristiche dell'operazione: in Mars viene usato per specificare se l'operazione read o take debba essere bloccante oppure no: un'operazione non bloccante che non trova alcuna matching tupla ritorna il valore NULL. Il parametro *identità* specifica il richiedente l'operazione ed è usato per scopi di sicurezza. In aggiunta l'interfaccia Mars presenta le due operazioni *readAll* e *takeAll* per estendere le operazioni a tutte le matching tuple che si incontrano nella ricerca.

La tupla usata come pattern per le operazioni di lettura ed estrazione può avere sia valori definiti che nulli: una tupla T sarà compatibile con la richiesta tupla R se i valori definiti di R sono uguali ai corrispondenti valori di T. Più in particolare, poiché in JavaSpace, così come in Mars, le tuple sono oggetti e gli elementi di una tupla possono essere a loro volta oggetti (non primitivi come i wrapper), le regole di matching devono tenere in considerazione questi casi: una tupla richiesta R è compatibile (match) una tupla T se e solo se si verificano le seguenti condizioni:

- R è un'istanza della classe T o di una sua superclasse; in questo senso JavaSpace estende il modello Linda permettendo il match anche tra due tuple di tipo diverso, supposto che appartengano alla stessa gerarchia
- i campi definiti di R che rappresentano tipi primitivi (integre, character, boolean ecc.) hanno gli stessi valori dei rispettivi campi in T
- i campi non primitivi definiti (cioè gli oggetti) di R sono uguali – in forma serializzata – ai corrispettivi di T
- un valore nullo di T corrisponde ad un valore nullo di R

Una volta che una tupla è stata ottenuta dallo spazio, i suoi campi attuali diventano accessibili ad ogni oggetto Java.

3.5.2 Modello reattivo

Diversamente da JavaSpace, Mars permette di associare reazioni programmabili agli eventi sul tuple space. Le reazioni sono oggetti con un singolo metodo che possono a loro volta accedere al tuple space, cambiarne il contenuto ed influenzare la semantica delle operazioni svolte dagli agenti.

Ci sono quattro parametri da considerare nell'associazione di una reazione: la reazione (Rct), l'elemento tupla (T), il tipo di operazione (O) e l'identità dell'agente (I); l'associazione avviene dunque mediante una 4-ple (Rct, T, O, I) e la reazione, cioè il metodo dell'oggetto Rct , viene eseguito quando l'agente I invoca l'operazione O sulla tupla T . Quando ci si riferisce all'associazione delle reazioni con le tuple si parla di *meta-level tuples* (Figura 10). Il meta-level tuple space associato segue gli stessi meccanismi del tuple space di base: una 4-ple ($ReactionObj, NULL, read, NULL$) specificherà di associare la reazione di $ReactionObj$ a tutte le operazioni di lettura ($read$), a prescindere dal particolare tipo di tupla e dall'identità dell'agente.

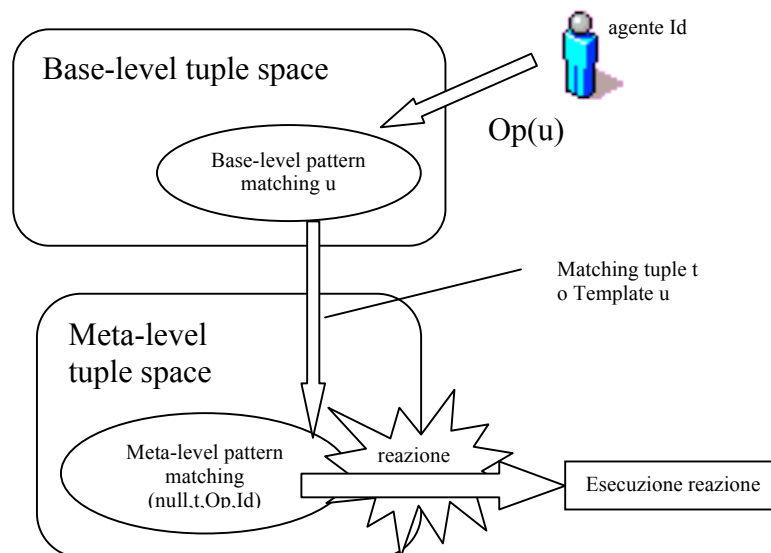


Figura 10 Attività del meta-level tuple space

Un aspetto che ancora non è stato approfondito è quello relativo all'autenticazione dell'agente ed ai permessi di accesso ai tuple space. Le specifiche JavaSpace definiscono un semplice meccanismo (basato su access control list, ACL) per rafforzare il controllo agli accessi allo spazio. In Mars, la capacità di programmare il tuple space e la possibilità di accedere e modificarne il contenuto tramite reazioni introduce problemi di sicurezza diversi. Ovviamente l'amministratore di un sito deve potere programmare liberamente e senza vincoli i propri tuple space; d'altro canto si può pensare di dotare gli agenti di un alto grado di flessibilità, lasciando loro la possibilità di programmare i loro meta-level space per installare le proprie reazioni, spostando il problema della sicurezza nella scelta di opportune politiche legate direttamente all'agente.

3.5.3 Esempio: Mars e il data retrieval

Per meglio comprendere le potenzialità del modello di coordinazione Mars, vediamo come viene affrontato il problema di data retrieval già visto negli altri modelli: supponiamo che l'ambiente di esecuzione fornisca l'accesso alle pagine html immagazzinate tramite tupla che ne identifichino le caratteristiche generali, come pathname, estensione, dimensione e data dell'ultima modifica ed infine un riferimento ad un oggetto *File* usato per accedere al contenuto del file in questione. Ognuna di queste tuple sarà un'istanza della classe *FileEntry*, il cui codice sarà del tipo di seguito.

```

Class FileEntry extends Entry {                // Entry è la classe alla base della gerarchia

    // campi dell'oggetto
    String PathName;                            // pathname del file rappresentato da questa tupla
    String Extension;                           // estensione del file
    Date ModificationTime;                       // data dell'ultimo update
    File ActualFile;                            // oggetto per accedere ai contenuti del file

    // costruttore della tupla. L'ordine dei parametri definisce l'ordine dei campi
    public FileEntry(String name, String extensione, Date modificationTime, File file) {
        PathName = name;
        Extension = extension;
        ModificationTime = modificationTime;
        ActualFile = file;
    }
}

```

Gli agenti su di un nodo in cerca di pagine html devono ottenere il riferimento accedendo al tuple space: nel nostro esempio, gli agenti otterranno tali riferimenti invocando l'operazione `readAll` in modo non bloccante (per evitare blocchi indefiniti nel caso che il sito locale non abbia alcuna tupla html) usando una tupla da usare come pattern nella quale il campo `Extension` presenta il valore "html"; l'operazione restituirà un vettore con ogni elemento trovato. Un frammento del codice è il seguente:

```

...
FileEntry FilePattern = new FileEntry(null, "html", null, null); // creazione della tupla
richiesta

Vector HTMLFiles = LocalSpace.readAll(FilePattern, NO_BLOCK, myIdentity);
// operazione readAll: legge tutte le matching tupla e ritorna un vettore di riferimenti alle tuple

if (HTMLFiles.isEmpty()) // nessuna matching tupla trovata
    terminate();         // l'agente non ha nulla da fare e può terminare

```

```

else
  for (int i = 0; i < HTMLFiles.size(); i++) {           // per ogni matching tupla
    FileEntry Hfile = (FileEntry) HTMLFiles.elementAt(i); // Hfile è la tupla del file
    if (this.SearchKeyword(keyword, Hfile)) {           // cerca la keyword nel file
      FoundFiles.addElement(LocalHost, Hfile);         // salva il file su un vettore
    }
  }
privato
  this.SearchLink(Hfile);                               // cerca i link e clona se stesso
}
}
...

```

Per evitare visite multiple sullo stesso sito è necessario un qualche criterio di coordinazione inter-agente: l'applicazione potrebbe ad esempio installare una reazione che non eviti semplicemente la duplicazione delle informazioni ma che si occupi anche dell'aggiornamento delle pagine html: quando un agente accede al tuple space per recuperare i riferimenti alle pagine html, la reazione controlla – per ogni tupla FileEntry compatibile – se il file corrispondente è stato modificato oppure no dall'ultima visita. Se il file non è stato modificato, la tupla corrispondente non viene restituita all'agente. Il codice seguente mostra tale reazione:

```

Class IncrementalVisit implements Reactivity {

  private Date visit;           // data dell'ultima visita

  public IncrementalVisit() {   // costruttore
    visit = new Date();         // quando una reazione viene installata da un agente, la
    data                        // dell'ultima visita è automaticamente settata dal
    costruttore
  }

  public Entry reaction (Space s, Entry Fe, Operation Op, Identity Id) {
    if ( (FileEntry) Fe.ModificationTime.before(visit) ) // docum. Modificato prima
    dell'ultima visita?
      return null;
    else {
      visit = new Date();           //setta la data dell'ultima visita
      return Fe;
    }
  }
}

```

3.5.4 Coordinazione context-dependent in Mars

Il sistema Mars si offre naturalmente a descrivere anche il modello di coordinazione context-dependent: supponiamo ad esempio che un'applicazione ad agenti mobili debba reperire delle pagine web da un sito, e che quest'ultimo salvi tali pagine col formato htm piuttosto che col classico html. Anziché forzare l'amministratore locale del sito a cambiare tutte le estensioni oppure a duplicare le tuple relative per renderle disponibili nelle due forme, l'amministratore può semplicemente installare una reazione che trasformi ogni richiesta a file html in una richiesta a file htm. La reazione potrà essere installata scrivendo la meta-level tupla che segue:

```
class HTML2HTM implements Reactivity {
  Public Entry[] reaction( Space s,Entry InputTuple[],Entry Template,Operation Op,
    Identity Id) {
    // se il sito ha solo files con estensione htm, modifica l'estensione nel template
    ((FileTuple) Template).Extension="htm";

    // richiede il matching con la nuova estensione
    Entry[] result = s.readAll(Template,null,NO_WAIT);
    for (int i=0; i<result.length; i++)

      // ripristina l'estensione nelle tuple trovate
      ((FileTuple) result[i]).Extension = "html";
    return result;
  } // fine del metodo reazione
} // fine della classe HTML2HTM
```

Per quanto riguarda la coordinazione environment-dependent, un tipico problema si presenta allorché un agente potrebbe tenti di effettuare un'operazione takeAll (cioè una estrazione dal tuple space) dei files html, eludendo così il controllo dell'amministratore. Quest'ultimo d'altronde può ovviare a tale inconveniente con una semplice reazione che trasformi l'operazione takeAll in un readAll con gli stessi parametri, restituendo così i dati richiesti in maniera del tutto trasparente all'agente. La reazione può essere implementata come segue:

```
class ReadOnly implements Reactivity {
  public Entry[] reaction(Space s,Entry InputTuples[],Entry Template,Operation Op,
    Identity id) {

    // controlla l'identità dell'agente
    if (Id.equals(manager))
      // l'amministratore può eliminare le tuple
      return s.takeAll(Template,null,NO_WAIT);

    // altrimenti le tuples non vengono estratte ma semplicemente restituite
    else return InputTuples;
  } // fine del metodo reazione
} // fine della classe ReadOnly
```

3.6 Confronto tra modelli

Come già visto, tutti i modelli di coordinazione sono caratterizzati da tre elementi fondamentali: i *coordinables*, i *coordination media* e le *coordination laws*. Sintetizzando, possiamo dire che gli obiettivi principali sui quali focalizzare l'attenzione sono:

Coordinables: è necessario assegnare un ruolo ai servizi Internet per far sì che il modello sia propriamente integrato nel Web. In questa direzione, l'unico tentativo eseguito è quello di PageSpace: ogni servizio Internet può trovare il suo ruolo come *special-purpose coordinable*. In ogni caso, l'idea di nascondere i servizi Internet dietro un tuple space non dovrebbe essere esclusa a priori, perché potrebbe risultare utile nel fornire un accesso semplice e tuple-based a servizi già esistenti.

Coordination Media: la coordinazione dovrebbe essere basata su una molteplicità di astrazioni di tipo tuple-based ed indipendenti, che siano distribuite nei siti Internet e che possibilmente vengano gestite ognuna nel modo più appropriato. Un'architettura network-aware comporta senza dubbio vantaggi, dato che gli agenti mobili possono fare riferimento ai tuple spaces implicitamente, a seconda della loro posizione. In più, si potrebbe concedere loro la possibilità di fare riferimento esplicito ad un tuple space, per esempio grazie ad un URL, come in TuCSon. Tuttavia, nel caso di reti di piccole dimensioni (es: reti interne) è la trasparenza a comportare i maggiori vantaggi. Un'altra soluzione interessante è quella prospettata da LIME: gli agenti possono agire trasparentemente anche rispetto alle risorse degli altri nodi, come se fossero locali.

Coordination Language: un set limitato di primitive di tipo Linda è spesso sufficiente. Inoltre, un coordination language non dovrebbe essere dinamicamente estendibile, al fine di non rendere più complessa la gestione delle applicazioni. Nel caso nuove funzionalità debbano essere aggiunte, vi è la possibilità di incorporarle come coordination laws, senza modificare il coordination language. Più difficile è la progettazione del *data model*. La soluzione migliore sarebbe definire un'architettura in cui tuple spaces diversi, basati su *data models* diversi, possano coesistere.

Coordination Laws: mentre la semplicità dei coordination language Linda-like è ottimale per le applicazioni Internet, le coordination laws di Linda potrebbero essere troppo restrittive. Per questo motivo, un tuple space programmabile dovrebbe integrare una *Web coordination architecture*. Occorre, infatti, che i sistemi siano capaci di modificare il comportamento in risposta a determinati eventi di comunicazione, e senza dover modificare il coordination language. In quest'ottica, coordination media programmabili basati su diversi modelli, come quello logico di TuCSon o quello orientato ad oggetti di Mars, dovrebbero poter coesistere. È chiaro che tutte le proposte dovrebbero tener conto degli standard esistenti, così come di quelli che si stanno per affermare, al fine di essere più facilmente accettate. Per esempio, poiché la tecnologia Xml diventerà lo standard *de facto* per la rappresentazione delle strutture dati e lo scambio di informazioni attraverso Internet, ogni modello dovrebbe tenerne conto. A questo proposito, nel paragrafo 5 presentiamo Mars-X e XmlSpaces.

Un'altra problematica interessante riguarda la sicurezza. Mentre un coordination language dovrebbe consentire agli agenti di sfruttare tutte le primitive di comunicazione e di accedere a tutte le strutture di dati disponibili, una qualsiasi security policy limiterebbe il protocollo di interazione, per esempio proibendo l'accesso ad alcune risorse o mezzi di

comunicazione. Bisogna cercare il miglior compromesso tra potere espressivo e sicurezza nelle interazioni.

Infine, un coordination model dovrebbe tenere conto di problematiche economiche, per esempio assegnando determinati prezzi alle interazioni, fronteggiando situazioni di risorse limitate, ecc.

Vediamo ora come i diversi modelli di coordinazione hanno interpretato questi concetti, e cerchiamo di capire quali sono i pro ed i contro di ognuna di queste scelte [2][24].

3.6.1 Definizione dei Coordinables

In generale, possiamo ritenere impensabile definire i coordinables senza che siano definite anche le entità non mobili, che potremmo chiamare non-mobile coordinables. La distinzione è opportuna, poiché un'applicazione Internet include anche diverse entità non-mobili, quali i WWW servers ed i CORBA-compliant services [25]. È possibile considerare l'insieme delle entità non mobili come un insieme generale di servizi Internet.

L'utilità di un sistema Linda-like, che utilizza tuple spaces, può essere meglio compresa se lo si considera come un modello di coordinazione incorporato (embedded) in un linguaggio di programmazione convenzionale. Per esempio, nel caso di Linda embedded in Java, i coordinables potrebbero essere Java objects attivi, il coordination medium un multiset di oggetti Java, e le coordination laws potrebbero essere quelle che descrivono la semantica delle primitive Linda-like.

Il tuple space può essere utilizzato come mezzo di coordinazione tra agenti e servizi Internet. La Figura 11 schematizza questa situazione:

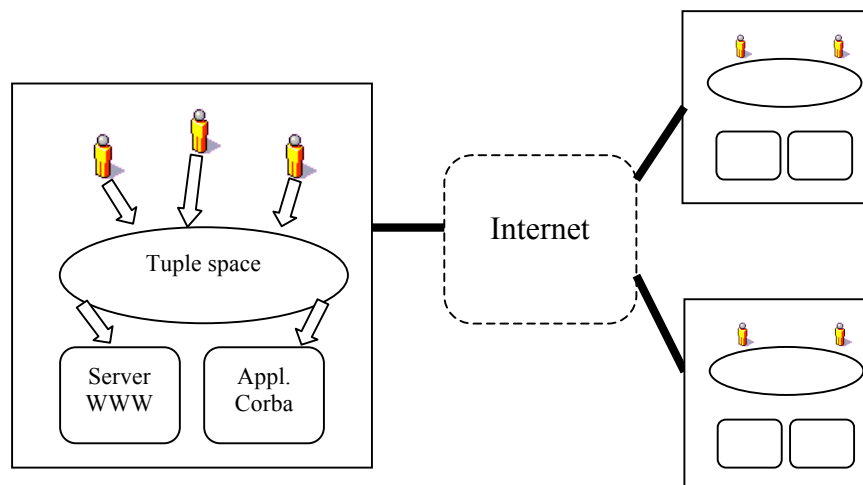


Figura 11 Tuple spaces come mezzi di coordinazione

Un simile approccio, si noti, non porta ad una definizione esplicita e precisa dei coordinables. Tale scelta è adottata da Mars, che associa un unico tuple space globale ad ogni nodo della rete capace di accettare ed eseguire agenti. Mars dà ad un agente la possibilità di interagire con l'ambiente di esecuzione solo grazie al tuple space. Questa soluzione consente di adottare uno stile di programmazione molto semplice, e quindi porta

alla definizione di un altrettanto semplice linguaggio di programmazione, rendendo la coordinazione di tipo Linda l'unico modo possibile di interagire con le applicazioni Internet. Una strategia simile potrebbe però risultare non appropriata per componenti, quali i proxy-servers, che potrebbero aver bisogno di agire sia come client, sia come server [26].

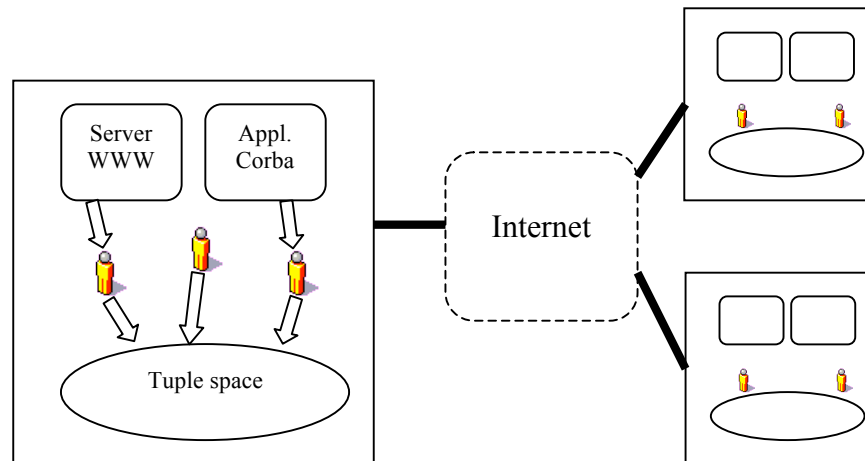


Figura 12 Tuple space come middleware

Un'alternativa consiste nelle scegliere come coordinables le entità del Web. Quest'approccio, schematizzato in Figura 12, è adottato da PageSpace.

PageSpace è attualmente l'unico sistema che si preoccupa di definire e caratterizzare sia i coordinables (es: gli agenti Internet), sia tutte le altre entità costituenti il modello. È importante cogliere la differenza rispetto a JavaSpace (e quindi Mars) e T Spaces, che definiscono soltanto l'architettura per il mezzo di coordinazione, dando per scontata la presenza di coordinables non meglio determinati.

3.6.2 Definizione dei Coordination Media

Anche se le applicazioni più semplici potrebbero trarre vantaggio dall'uso di un singolo tuple space, ciò non renderebbe possibile sviluppare applicazioni complesse e widely-distributed. Tali applicazioni necessitano, infatti, di molteplici di tuple spaces distribuiti in Internet ed accessibili tramite agenti. D'altro canto, una simile disponibilità di tuple spaces multipli ed indipendenti permette di ottenere decentralizzazione e modularità, ma introduce nuovi problemi. In particolare, un modello di coordinazione comprendente tuple spaces multipli richiede che gli agenti abbiano un modo efficace di riferirsi e di accedere agli spazi stessi. Molti sistemi di coordinazione (JavaSpace, Mars, PageSpace) suppongono che gli stessi tuple spaces siano oggetti. Gli agenti possono allora scambiarsi le referenze ai tuple spaces ed usarle per accedere ad un dato tuple space, trasparentemente alla loro location. Questa soluzione, estremamente semplice, ci fornisce un modo di accedere al tuple space sia trasparente sia location independent.

Tuttavia, nel contesto delle applicazioni Internet, la network unawareness non è la scelta più naturale, perché i tempi di latenza imprevedibili tipici di Internet renderebbero le prestazioni delle applicazioni difficili da prevedere.

L'organizzazione dei tuple spaces, così come il modo relativo di accedervi, devono tenere conto della mobilità network-aware degli agenti. In quest'ottica, sono possibili due approcci: uno implicito ed uno esplicito.

L'approccio implicito evita all'agente la necessità di fare esplicitamente riferimento al tuple space, e lascia che l'agente acceda automaticamente ad un tuple space di default che varia secondo la posizione in Internet. In questo modo l'approccio implicito connette le questioni del coordination model alla mobilità degli agenti.

Il sistema Mars adotta un approccio implicito e context-dependent per identificare il tuple space ed accedervi. Il tuple space può essere utilizzato per accedere alle risorse ed ai servizi di un certo nodo in modo implicito (senza nominarlo né farvi riferimento), oltre che per consentire la coordinazione tra agenti. Quando un agente arriva su un nuovo nodo, viene stabilita una nuova connessione tra l'agente ed il tuple space situato su quel nodo, mentre la vecchia connessione viene persa. Un agente non ha nessun modo di accedere ad un tuple space remoto, se non quello di migrare esplicitamente verso il nodo su cui quel tuple space è situato.

LIME adotta un approccio diverso: ogni agente porta con sé un tuple space, e questo è l'unico tuple space a cui può accedere nel corso della sua esistenza. Non appena un agente arriva su un nodo, il suo tuple space privato è automaticamente fuso con quello associato all'ambiente di esecuzione. In questo modo, il tuple space privato può essere utilizzato per accedere implicitamente alle risorse di quel nodo. Lo svantaggio principale dell'approccio implicito è che obbliga gli agenti a migrare al fine di accedere ad un tuple space. Questa può non essere la scelta migliore, come è evidente nel caso l'agente debba fare un singolo accesso al tuple space; in una situazione simile sarebbe più opportuno l'accesso remoto, in modo da L'approccio esplicito, mantenendo un'architettura network-aware, consente accesso remoto ai tuple spaces grazie ad un *global naming scheme* (es: URL).

L'adozione di un approccio esplicito non implica il rigetto dell'approccio implicito: TuCSoN, per esempio, abilita gli agenti all'accesso sia implicito, sia esplicito.

3.6.3 Definizione delle Coordination Laws

Nel contesto di Internet, il modello di tuple space object-oriented adottato da JavaSpace sostituisce il classico meccanismo pattern-matching di Linda. Due oggetti corrispondono se le loro *serialized forms* sono uguali. Al contrario, l'unificazione logica è presa come basic matching-mechanism da TuCSoN. La maggior parte dei sistemi presentati in letteratura (tra cui PageSpace, LIME e JavaSpace) adotta l'approccio di Linda che consiste nell'incorporare regole fisse di coordinazione nel coordination medium. Tuttavia, in un contesto vasto ed imprevedibile come Internet, questa scelta potrebbe non essere la migliore, dato che gli agenti mobili devono interagire e migrare attraverso ambienti di esecuzione eterogenei.

T Spaces, TuCSoN e Mars concepiscono invece il coordination medium come un kernel configurabile, in modo che nuove coordination laws possano essere definite, a vantaggio della flessibilità. Mentre T Spaces rende possibile definire anche nuove primitive, TuCSoN

e Mars proibiscono sia l'adozione di nuove primitive che la modifica dei meccanismi di matching esistenti. Essi rendono però possibile programmare completamente le coordination laws, consentendo pertanto la definizione di nuovi comportamenti di risposta agli eventi della comunicazione. In TuCSon i coordination media possono essere programmati da agenti intelligenti di meta-livello. Mars, in coerenza con il suo modello di tuple space orientato ad oggetti, segue un approccio di programmazione delle coordination laws orientato ad oggetti: uno specifico metodo di una specifica classe-interfaccia può essere implementato ed associato ad uno specifico evento di comunicazione. Questo rende la programmabilità di Mars auspicabile per i servizi di alto livello ed il *system management*.

Per concludere la riflessione sulle coordination laws, riteniamo opportuno affrontare brevemente la questione del coordination language, che abbiamo precedentemente definito come l'insieme delle primitive di interazione. Il trend attuale, seguito da PageSpace, JavaSpace e Mars, è quello di definire modelli object-oriented di tuple spaces, in cui sia i tuple spaces che i componenti sono oggetti Java. Tuttavia, Java potrebbe non essere il linguaggio più adatto per tutte le applicazioni. Per esempio, T Spaces, usato per gestire dati di grandi dimensioni, adotta un modello relazionale: ogni tupla è una riga in una tabella; ogni tipo di tupla identifica un tipo di tabella.

Per quanto riguarda le primitive di comunicazione, la maggior parte dei sistemi identifica un set limitato di primitive aventi la stessa semantica delle primitive Linda. Per le applicazioni basate sugli agenti autonomi, operazioni asincrone simili ai meccanismi di notifica di JavaSpace sono utili: un agente può richiedere una tupla ad un tuple space senza restare bloccato in attesa di una tupla appropriata; alla notifica della disponibilità di una tupla appropriata, può recuperare tale tuple. Quando però vengono coinvolti gli agenti mobili, i meccanismi di notifica non sono opportuni, dato che richiederebbero il supporto run-time per localizzare i *mobile agents*.

4. Confronto critico

In questa sezione viene presentato un tipico esempio applicativo distribuito, realizzato attraverso il paradigma ad agenti mobili, per analizzarne gli aspetti peculiari ed i problemi che possono insorgere dall'approccio con i modelli di coordinazione attualmente disponibili. In particolare vengono trattati i due modelli agli estremi della tassonomia già presentata nella sez. 2.

4.1 Un esempio applicativo: le aste su Internet

L'utilizzo di agenti mobili può portare innumerevoli vantaggi per gli utenti interessati alle aste Internet, in particolare nel monitoraggio automatico di eventi esterni, nell'implementazione di complesse strategie decisionali e nella delega di compiti noiosi per

l'uomo. Inoltre, le caratteristiche di mobilità, che incrementano l'autonomia degli agenti, determinano ulteriori vantaggi.

Vi sono diversi tipi di aste, in base al numero dei partecipanti, ai criteri per l'assegnazione delle risorse e così via. In questo ambito ci soffermeremo sulle aste con un solo venditore e più compratori contemporaneamente, anche se le possibilità che lo scenario introduce sono molteplici [27].

4.1.1 Implementazione di aste

In generale le aste possono essere implementate sia mediante il modello message-passing che per mezzo di tuple space programmabile [26][28]. Nel primo caso, alcuni agenti fissi agiscono da venditori (*auctioneers*) ed accettano le richieste per iniziare un'asta ricevendo messaggi direttamente dai compratori. Devono inoltre notificare a tutti gli agenti interessati la loro presenza ed il tipo di beni/risorse in vendita. Generalmente, gli agenti che partecipano all'asta devono sottoscrivere delle liste pubbliche per informare i venditori circa i propri interessi, costringendo questi ultimi a tenere aggiornate tali liste per conoscere sempre i potenziali acquirenti. Gli agenti acquirenti fanno un'offerta per mezzo di un messaggio inviato direttamente al venditore, il quale dovrà rendere nota l'offerta agli agenti coinvolti nell'asta.

In presenza del modello di tuple space programmabile, invece, l'asta ha inizio quando un venditore scrive una tupla che descriva il bene/la risorsa in vendita (Figura 13a), in aggiunta ad altre informazioni utili. In maniera del tutto simile, le offerte vengono effettuate scrivendo appropriate tuple nello stesso tuple space (Figura 13b). La presenza del modello reattivo permette inoltre di associare azioni da intraprendere ogni qual volta viene fatta un'offerta, quali la verifica della correttezza dell'offerta, l'aggiornamento dello stato dell'asta e la notifica agli altri agenti dell'avvenuto evento.

Alla fine dell'asta, un'ulteriore reazione decide il vincitore e notifica i risultati a tutti gli agenti coinvolti (Figura 13c). Di seguito vengono analizzati alcuni peculiari problemi di coordinazione, per vedere come vengono affrontati dai due modelli.

4.1.2 Coordinazione inter-agent

Una situazione che può richiedere complessi protocolli di interazione tra gli acquirenti si presenta quando il venditore vuole vendere un set di beni/risorse come un tutt'uno. E' ragionevolmente improbabile che un singolo agente voglia acquistare l'intero set, quindi potrebbe ricercare altri agenti interessati a parti del set per farli entrare nell'asta al fine di acquistare il tutto come se fossero un team di compratori e poi dividersi le rispettive parti. L'accordo dovrebbe includere anche il limite massimo delle offerte che l'agente sarebbe disposto a fare. Il caso più semplice è quello in cui il set in vendita è composto da due elementi X e Y, e l'agente A1 interessato all'elemento X/Y (elemento X in relazione all'elemento Y) cerca un secondo agente A2 interessato all'elemento Y/X, per effettuare un'offerta combinata. Viene mostrato di seguito come l'agente A1 vada alla ricerca

dell'agente A2 usando, rispettivamente, i modelli message-passing e tuple space programmabile.

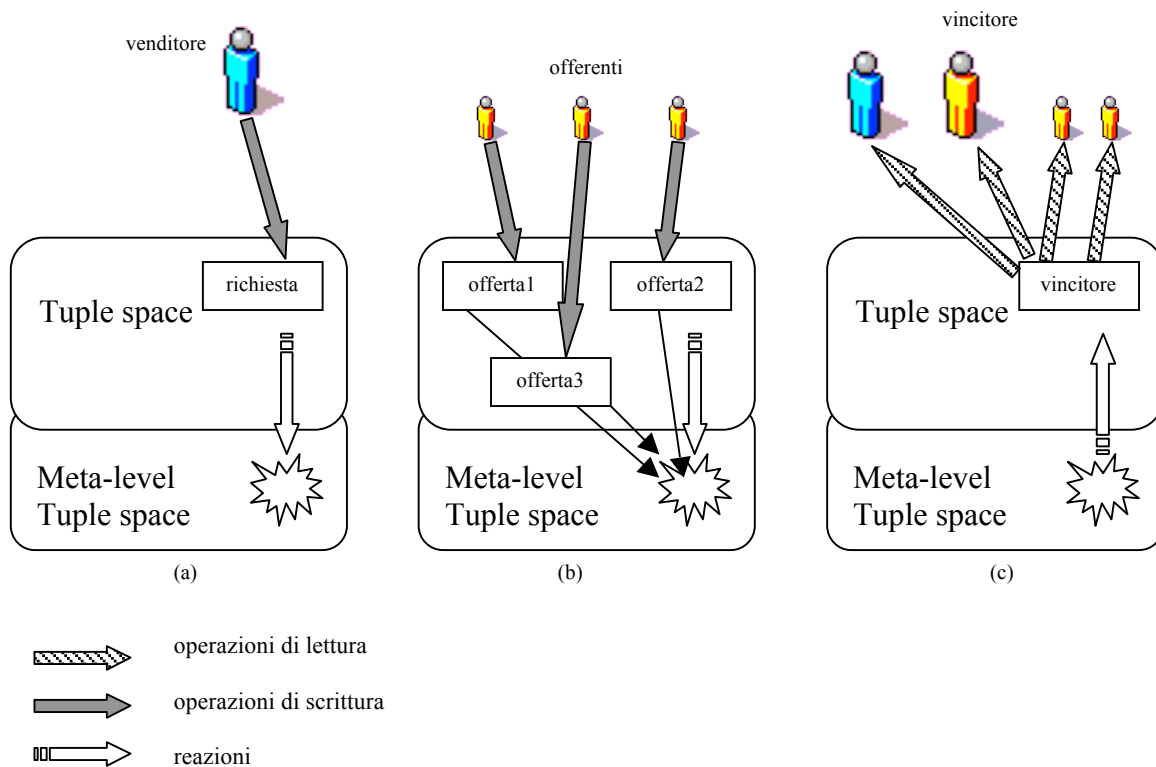


Figura 13 Un'asta implementata attraverso tuple space programmabile: un agente venditore mette in vendita un bene (a), gli agenti compratori fanno le loro offerte (b), una reazione decide il vincitore (c)

Nel caso di coordinazione message-passing, supponiamo che il primo agente A1 sia appena arrivato al sito dove si svolge l'asta; il primo passo che deve eseguire è la ricerca degli altri agenti presenti nel sito. Con il sistema Aglets, ciò è realizzabile per mezzo della richiesta dei proxy di tutti gli agenti al context locale. Il secondo passo consiste nella ricerca degli agenti coinvolti nella specifica asta, operazione abbastanza agevole se il sito è organizzato in maniera tale da associare ad ogni asta un proprio context. Se così non fosse, l'agente A1 dovrebbe chiedere ad ogni agente presente se è interessato alla specifica asta oppure no. Ad ogni modo, il terzo passo consiste nella ricerca degli agenti interessati all'elemento Y/X e nella richiesta della massima offerta che questi sono disposti ad affrontare. Quest'ultima è l'operazione più complessa, sia perché l'interazione deve essere attentamente progettata e sia perché l'esecuzione di interazioni multiple rallenta sensibilmente l'esecuzione degli agenti. Infine, una volta noti tutti gli agenti interessati ad una possibile partnership, A1 può scegliere il miglior partner, A2.

L'uso di tuple space programmabile semplifica le interazioni con gli altri agenti per mezzo di semplici letture e scritture di tuple. Usando Mars, ad esempio, l'agente compratore scrive una tupla che può essere espressa nella forma (*auction*, *element*, *agent_id*, *max_bid*, *percentage*), dove *auction* è l'asta a cui l'agente è interessato, *element* è l'elemento desiderato, *agent_id* l'identificatore dell'agente compratore, *max_bid* la massima offerta che può essere affrontata e *percentage* la parte dell'offerta che si è disposti a pagare per l'elemento specificato. L'agente A1 cerca di leggere tutte le tuple compatibili al template (*auction_id*, *elem*, *NULL*, *NULL*, *NULL*), dove *elem* può essere X o Y e *NULL* rappresenta un valore formale. Se la lettura ha successo, l'agente A1 viene subito a conoscenza di tutti gli agenti disponibili ad un accordo e può sceglierne uno in base alla propria strategia.

L'uso della reattività comporta numerosi vantaggi, ad esempio nella selezione delle tuple: quando un template fornito dall'agente A1 presenta i valori di *max_bid* e *percentage* indefiniti, un comportamento non reattivo causerebbe il ritorno di tutte le tuple compatibili e, su questa base, l'agente A1 sceglierebbe il miglior partner. Le reazioni possono invece semplificare il compito di selezione: in particolare, in Mars una specifica reazione potrebbe essere definita per effettuare una selezione preliminare delle tuple compatibili sulla base del valore di *max_bid*, ad esempio.

4.1.3 Coordinazione agent-to-hosting-environment

Un'altro punto importante nello scenario delle aste riguarda l'interazione tra gli agenti ed il loro ambiente ospitante. Considereremo in questo ambito il caso dell'arrivo di un agente nel sito dell'asta che voglia sapere quali aste siano attualmente attive ed in che stato si trovino; in particolare, l'agente potrebbe essere interessato in alcuni particolari beni/risorse di una particolare asta e/o in tutte le aste le cui offerte più alte sono attualmente ad un determinato valore.

Adottando l'infrastruttura di coordinazione message-passing delle Aglets, l'agente deve necessariamente venire a conoscenza dell'entità che funge da battitore d'asta, ad esempio un agente fisso, FA per semplicità. In tal caso l'agente in arrivo, chiamato IA, riceve un riferimento a FA per mezzo di una richiesta al context locale. Quindi IA manda un messaggio a FA per sapere quali aste sono attualmente attive e selezionare quelle a cui è interessato. Anche se queste interazioni possono effettuarsi tramite un singolo messaggio, la vastità dello scenario richiede spesso parecchi messaggi per definire con cura e precisione i risultati che dovranno essere ottenuti.

Usando tuple space, l'agente IA non si interessa di chi controlla attualmente l'asta – sia esso un agente fisso, una reazione o qualsiasi altro processo del sistema – purché essa sia descritta mediante una tupla nel tuple space. Così facendo, la selezione delle informazioni necessarie avviene mediante il meccanismo nativo del pattern-matching. In Mars, ad esempio, l'agente IA può semplicemente eseguire un'operazione `readAll` fornendo un template che definisca solo i valori che ha fissato e lasciando nulli i campi che vuole ricevere. Ad esempio, usando un template nella forma (asta, risorsa, descrizione, prezzo_iniziale, fine_asta), se l'agente fosse interessato a tutte le aste con scadenza il 10 Ottobre 2002 userebbe (NULL, NULL, NULL, NULL, 10/10/2002). Questo approccio sposta l'implementazione di un adeguato algoritmo dall'agente al tuple space, portando ad

una più semplice programmazione di entrambi e ad una più flessibile migrazione dell'agente sulla rete.

Ancora una volta, la reattività migliora le caratteristiche del modello coordinativo: se un agente vuole conoscere tutte le aste la cui massima offerta risulta minore di un determinato prezzo, una reazione Mars potrebbe essere installata per fornire tale servizio.

5. Progetti futuri

La tecnologia ad agenti mobili non ha finora saputo imporsi al di fuori dell'ambito di ricerca universitario non tanto per una mancata maturità delle soluzioni disponibili, tanto per la carenza di uno standard che chiarisse i concetti basilari attorno al quale un'architettura deve basarsi: coordinazione, comunicazione, fault-tolerance, sicurezza ecc. Per quanto riguarda la coordinazione e l'interoperabilità, la situazione sembra essere leggermente migliore, nel senso che da qualche anno appositi gruppi, primo fra tutti il consorzio OMG, stanno studiando soluzioni intese come standard aperti a tutti i sistemi.

In quest'ottica si inquadrano molti dei progetti futuri che riguardano i modelli di coordinazione uniti ai meccanismi di interoperabilità, primo fra tutti il promettente uso della tecnologia Xml che già riscuote vastissimo successo nell'ambito dello scambio di contenuti su Internet.

Come già ampiamente visto, i modelli di coordinazione basati su tuple space rappresentano un'interessante tecnologia per facilitare lo sviluppo di applicazioni distribuite basate su agenti mobili, ed il crescente numero di proposte che sfruttano la coordinazione Linda-like, affrontandola con diversi approcci, non può che confermare questo trend. Ad ogni modo, tali metodi devono ancora trovare un riscontro pratico nelle applicazioni realmente usate su Internet. Uno dei punti da tenere in considerazione è rappresentato dalla sicurezza, che è intrinsecamente legata alla coordinazione. Infatti, mentre le tecnologie di coordinazione per le applicazioni distribuite spingono nella direzione di una maggiore interazione per renderla più fruibile, la sicurezza tende a limitare tali interazioni, per avere un controllo più diretto. In altre parole, mentre un linguaggio di coordinazione tende a consentire a qualsiasi agente di accedere ad ogni risorsa disponibile, una politica di sicurezza tende a ridurre e limitare tali accessi, impedendo ad esempio l'accesso ad alcuni dati o dispositivi.

Standard esistenti così come altri emergenti si stanno affacciando nel panorama degli agenti mobili. Ad esempio, dal momento che la tecnologia Xml rappresenta uno standard de-facto per la rappresentazione dei dati e lo scambio attraverso Internet, è facile predire che ogni modello di coordinazione diventerà compatibile con Xml. Infatti è possibile rappresentare in Xml non solo dati ma anche agenti e servizi, così come protocolli d'interazione ad alto livello (Figura 14). Questo non significa che Xml soppianderà le esistenti architetture di coordinazione, ma tali modelli, integrando la tecnologia Xml, porteranno ad un framework nel quale un vasto tipo di proposte agent-based troveranno un ruolo coerente.

In questo ambito si inquadra questa sezione, nella quale vengono presentati Mars-X e XmlSpaces, due architetture integrate per la coordinazione di agenti mobili attraverso la tecnologia Xml. Per comprendere meglio le ragioni che hanno spinto verso tale unione, e contemporaneamente fornire una panoramica sulle funzionalità di Xml che risultano utili al paradigma ad agenti mobili, rimandiamo all'appendice B.

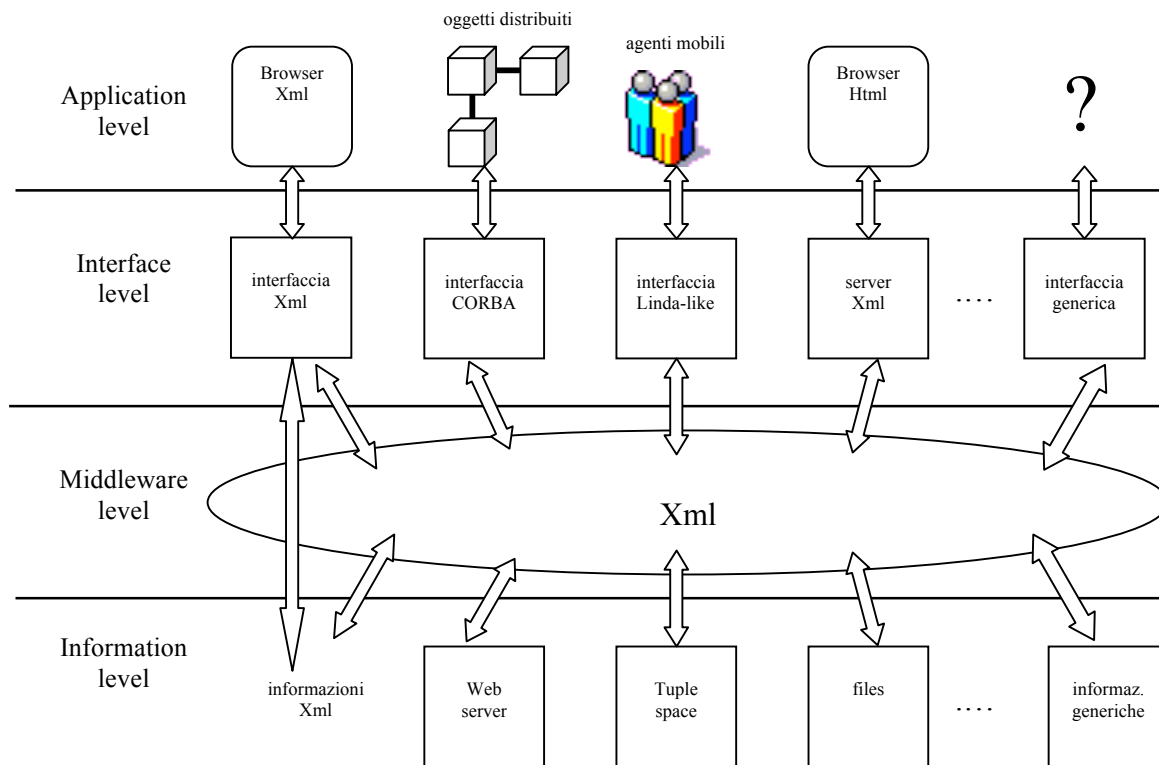


Figura 14 L'architettura Xml

5.1 Mars-X

Dall'integrazione tra Xml e Mars nasce come già detto Mars-X [29], un'architettura di coordinazione per agenti mobili che unisce i vantaggi del linguaggio Xml (interoperatività tra fonti di informazione eterogenee) e della coordinazione Linda-like (dinamicità ed interazioni completamente disaccoppiate). In Mars-X gli agenti si coordinano, sia tra di loro che con l'ambiente di esecuzione, grazie a dataspace Xml programmabili. Ognuno di questi dataspace è associato ad un determinato ambiente di esecuzione ed è accessibile agli agenti in modo Linda-like, come se fosse un normale tuple space. Poiché il comportamento dei dataspace Xml in risposta agli accessi può essere completamente programmato, Mars-X permette di includere nei dataspace regole di coordinazione sia application-specific che coordination-specific.

Nel caso di Mars-X, l'interfaccia Linda-like che consente agli agenti di accedere ai dataspace Xml è un'interfaccia di tipo JavaSpace standard (Figura 15). Ciò significa che un

tuple space di Mars-X assume la forma di un oggetto Java, il quale rende disponibili le operazioni read, write e take, oltre alle readAll e takeAll che Mars aggiunge a JavaSpace.

La scelta di implementare un'interfaccia JavaSpace invece di definirne una Xml-oriented è stata dettata dal fatto che, come abbiamo già detto, la tecnologia JavaSpace sarà molto importante nel contesto delle applicazioni Java distribuite del futuro più prossimo. Inoltre, questa scelta si adatta meglio all'approccio degli agenti Java, che vedono le tuple come Java objects le cui variabili sono i campi delle tuple stesse. Le operazioni dell'interfaccia devono essere in grado di tradurre la rappresentazione ad oggetti delle tuple nella corrispondente rappresentazione Xml e viceversa.

In JavaSpace, e perciò in Mars-X come in Mars, si implementa l'interfaccia Entry derivando ogni tuple class dalla classe AbstractEntry. Inoltre, in Mars-X ogni tuple class deve avere un campo aggiuntivo che specifichi la DTD (Document Type Definition), la quale descrive la struttura del documento Xml corrispondente alle istanze di quella classe.

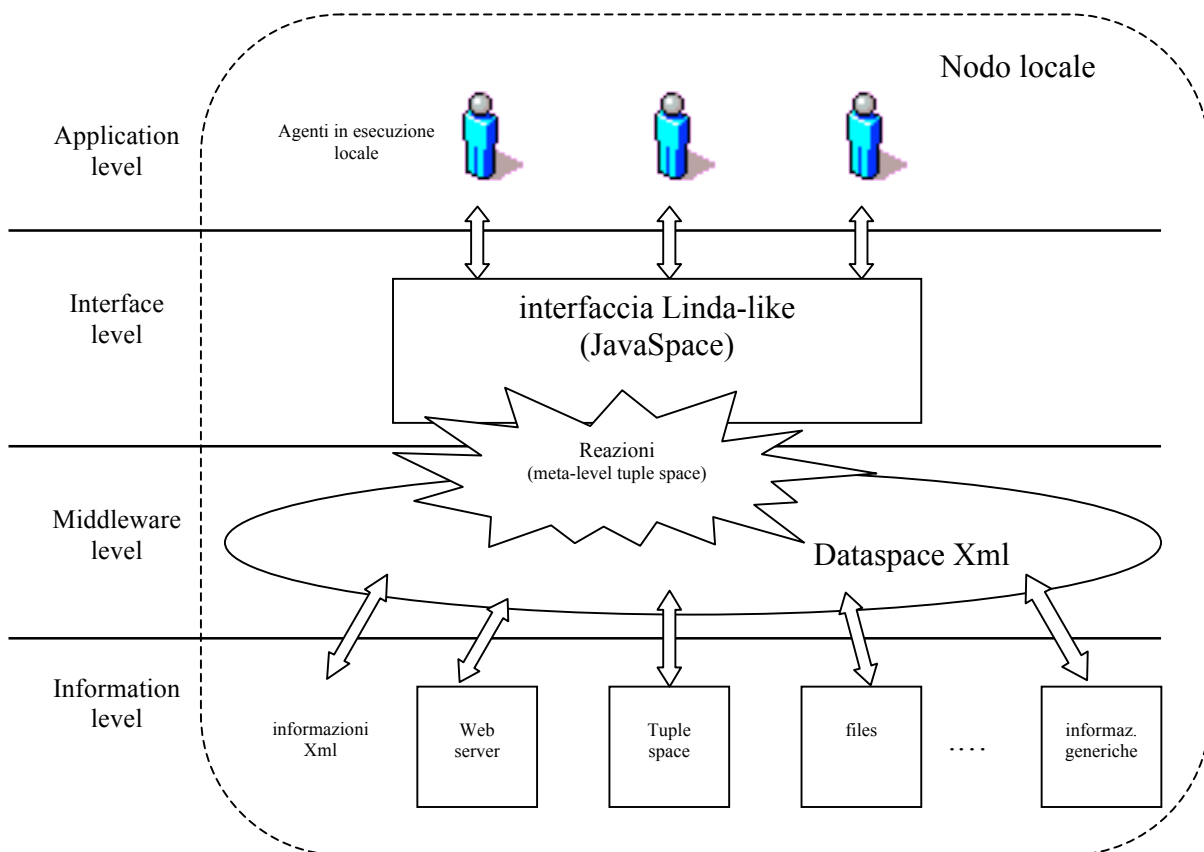


Figura 15 L'architettura Mars-X

Ad ogni tupla di un generico tuple space corrisponde uno ed un solo elemento di un documento Xml. In particolare: 1) il nome della classe, che inizia con un underscore, corrisponde, una volta eliminato l'underscore, al nome del tag che definisce l'elemento Xml; 2) i nomi delle instance variables corrispondono, nell'ordine, ai dati compresi i valori delle instance variables corrispondono, nell'ordine, ai dati compresi nei tags.

Quando un'operazione di input è invocata da un agente, Mars-X ricerca nel dataspace Xml un elemento in un documento Xml tale che:

1. il DTD del documento sia uguale a quello specificato nel campo della tupla;
2. la tupla tradotta in formato Xml corrisponda ad almeno un elemento del documento;
3. i valori dei campi della tupla corrispondano ai valori nei tags dell'elemento.

Nell'ottica della mobilità, ed in particolare allo scopo di ottenere località di accesso, i dataspace Xml devono essere considerati risorse locali associate ad un nodo. In Mars-X, ogni nodo Internet deve definire il proprio dataspace Xml e l'interfaccia Linda-like ad esso associato. Quando arriva su un nodo, ad un agente viene fornito un riferimento al dataspace Xml del nodo, in modo che sia libero di accedervi in modo Linda-like. Inoltre, domini locali di nodi (es: reti locali), possono implementare un unico dataspace Xml.

In Mars-X, la reattività è implementata esattamente come in Mars, cioè attraverso reazioni, eventi e 4-ple. Si noti che Mars-X, esattamente come Mars, non implementa un nuovo agent system Java, ma è stato progettato per completare le funzioni degli agent system già esistenti, senza essere legato ad un'implementazione specifica.

L'implementazione corrente di Xml soffre di alcune limitazioni: a) gli attributi dei tags sono considerati tags addizionali, per cui non c'è corrispondenza "uno-a-uno" tra gli elementi Xml e gli oggetti Entry; b) non tratta gli Xml namespaces; c) la sincronizzazione degli accessi concorrenti è fatta in politica MR/SW (Multiple Readers Single Writers) a livello del singolo documento Xml, e questa non è la scelta ottimale.

5.2 XmlSpaces

In maniera del tutto simile a Mars-X, il sistema XmlSpaces [31], sviluppato alla Technische Universität di Berlino, estende il modello di base Linda, precisamente l'implementazione TSpaces del modello Linda, sotto parecchi punti di vista:

1. i documenti Xml vengono usati come campi delle tuple nello spazio di coordinazione; ne segue che possono essere trattate le tuple ordinarie già incontrate nonché puri documenti Xml intesi come tuple con un singolo campo
2. può essere usata una moltitudine di relazioni tra i documenti Xml per le funzioni di matching, alcune delle quali sono supportate in maniera nativa dall'architettura mentre altre estensioni possono essere aggiunte totalmente liberamente
3. XmlSpaces è un'architettura distribuita, nel senso che server multipli in locazioni diverse usati come dataspace vengono visti come un unico dataspace logico; contestualmente possono essere adottate diverse politiche di sicurezza
4. vengono supportati gli eventi distribuiti, che permettono la notifica immediata a tutti i client dell'inserimento/rimozione di una tupla

In XmlSpaces, i campi actuals di una tupla possono contenere, come già anticipato, interi documenti Xml, mentre i campi formals possono contenere ulteriori descrizioni sul documento, come ad esempio una query espressa in un apposito Xml query language. La relazione di matching segue le stesse regole del pattern-matching Linda, anche se in questo

caso vengono supportate relazioni multiple sui documenti Xml. In particolare, le regole di matching si basano sul motore XQL, e prevedono che:

- un documento Xml equivale (match) ad un altro sulla base della uguaglianza di contenuti o sull'uguaglianza di attributi nei singoli elementi
- un documento Xml equivale ad un altro sulla base di una minima grammatica comune con o senza l'uguaglianza dei rimanenti elementi e relativi attributi
- un documento Xml può essere comparato ad una matching query che segua una delle semantiche supportate, come Xml-QL, XQL o XPath/Pointer

L'utilizzo di XmlSpaces nell'ambito della applicazioni distribuite, oltre all'innegabile vantaggio dell'estensione del linguaggio Linda, trova il suo punto di forza nell'impiego di schemi distribuiti diversi a seconda delle esigenze particolari; in pratica, l'architettura può essere di tipo:

- *centralized*: un server incapsula tutto il dataspace
- *distributed*: tutti i server incapsulano un sottoinsieme del dataspace intero
- *full replication*: tutti i server posseggono copie persistenti del dataspace intero
- *partial replication*: sottoinsiemi di server conservano copie consistenti di subset dell'intero dataspace
- *hashing*: le matching tuples ed i templates vengono conservati nello stesso server e selezionati in base ad una qualche funzione hash

In XmlSpaces ad ogni modo non viene prescritta una specifica strategia di utilizzo, quanto piuttosto viene incapsulata la strategia distribuita nel cosiddetto *distributor object*, alla cui interfaccia vengono fornite le primitive Linda. Grazie all'uso del distributor object, XmlSpaces risulta totalmente aperto, in quanto qualsiasi server può unirsi o uscire in qualsiasi istante, poiché il distributor object deve semplicemente registrare i server correntemente attivi per mezzo dei metodi di registrazione/cancellazione inclusi nella propria interfaccia. Sarà poi ancora per mezzo del distributor object che avverrà la notifica degli eventi distribuiti a tutti i server che hanno sottoscritto l'evento in questione, rendendo così il multicasting basato su dataspace Xml distribuiti facilmente implementabile.

Appendice A: il sistema Linda

Linda è il capostipite dei linguaggi di coordinazione di questo tipo [2][8]. Linda non è un linguaggio di programmazione ma, più propriamente, un'estensione che può integrare quasi tutti i linguaggi al fine di consentire creazione di processi, comunicazione e sincronizzazione. Si può definire Linda un *coordination language* (si veda il paragrafo 2.1). Come C è un linguaggio computazionale completo, così Linda è un linguaggio di coordinazione completo. L'idea alla base di Linda, come già detto, è quella di definire un modello di coordinazione indipendente che possa essere aggiunto ad ogni linguaggio di base. Linda implementa una memoria condivisa di tipo associativo. L'informazione è organizzata in *tupla* (o *n-ple*) di campi, ognuno dei quali ha un tipo ed un valore. Le tuple vengono raggruppate in *tuple spaces*. Questi tuple spaces, che risiedono su hosts fissi, sono condivisi tra diversi agenti. L'accesso alle tuple avviene in modo associativo: i processi specificano che tipo di informazione vogliono e non usano né identificatori, né nomi simbolici di variabili. Il risultato è quello di avere una comunicazione disaccoppiata ed implicita (*decoupled and implicit*). Tali caratteristiche sono particolarmente opportune in ambienti mobili, in cui le entità cambiano spesso contesto e non sanno con quali altre entità possono mettersi in contatto in un certo istante.

Linda può essere inoltre un modello di file system e database, dato che le tuple sono oggetti permanenti. La similarità tra tuple spaces e databases è stata ben presto evidenziata; ciò nonostante, pochissimi progetti sono stati realizzati per esplorarne le corrispondenze. *Persistent Linda* rappresenta il primo tentativo di dotare Linda delle funzionalità dei databases.

Le tuple vengono aggiunte al tuple space attraverso l'operazione *out(t)*, dopo la quale la tupla *t* è disponibile a qualsiasi altra operazione. L'update del tuple space avviene atomicamente.

Le tuple possono essere rimosse dal tuple space tramite l'operazione *in(p)*, che è bloccante. L'argomento *p* è chiamato *template*, ed i suoi campi possono essere *actuals* (valori) o *formals* (ovvero wild cards come integer, char, ecc.). Il sistema verifica l'esistenza di tupla che corrispondano (*match*) al template, il che avviene se:

1. i campi della tupla sono uguali in numero e, ordinatamente, in tipo a quelli del template;
2. nel caso i template abbia campi con un valore specificato, il valore dei corrispondenti campi della tupla deve essere lo stesso.

Se più tupla corrispondono al template, quella che viene rimossa dalla *in(p)* è scelta in modo non deterministico. La lettura può avvenire anche tramite la funzione *rd(p)*. Con questa funzione, anch'essa bloccante, la tupla non viene però rimossa dal tuple space.

Un'altra operazione fondamentale di Linda è *eval*, che consente la *process creation* e, dunque, la reattività.

I modelli basati su Linda sono in genere carenti in termini di flessibilità e controllo delle interazioni. La ragione è che sia le interazioni agente-agente, sia quelle agente-ambiente di esecuzione, avvengono grazie ai meccanismi di pattern matching suddetti, i quali non si prestano, se non forzatamente, a tutti i tipi di interazione. Inoltre, tali meccanismi potrebbero rendere difficile realizzare protocolli di interazione complessi. Infine, il modello di interazione previsto da Linda non consente soluzioni semplici per accedere ai servizi forniti da un sito.

Appendice B: l'architettura Xml

L'incredibile successo di html ha spinto il consorzio Xml Working Group (originariamente noto come SGML Editorial Review Board) costituitosi sotto gli auspici del World Wide Web Consortium (W3C) allo sviluppo nel 1996 di *Xml (Extensible Markup Language)*, un linguaggio di rappresentazione che probabilmente diventerà lo standard per l'interoperatività in Internet [29]. L'obiettivo di questo gruppo di lavoro era di portare il linguaggio SGML nel Web. L'SGML è un linguaggio per la specifica dei linguaggi di markup ed è il genitore del ben noto HTML. La progettazione dell'Xml venne eseguita esaminando i punti di forza e di debolezza dell'SGML. Il risultato è uno standard per i linguaggi di markup che contiene tutta la potenza dell'SGML ma non tutte le funzioni complesse e raramente utilizzate.

L'Xml venne mostrato per la prima volta al pubblico quando l'SGML celebrò il suo decimo anno. Innanzitutto, Xml rappresenta i dati in una familiare forma testuale e *tagged* (html-like). Al contrario di html, che determina come un documento debba essere processato da un browser, Xml si limita a specificare quali siano le strutture di dati, lasciando ogni decisione in materia di processing (elaborazione o visualizzazione) al livello applicativo. In questo modo si ottengono sia la platform-independence necessaria per Internet, sia, nel contesto dei Web servers, un approccio più modulare alla concezione del server. In sintesi, Xml è un linguaggio di markup aperto e basato su testo che fornisce informazioni di tipo strutturale e semantico relative ai dati veri e propri. Questi "dati sui dati", o *metadati*, offrono un contesto aggiuntivo all'applicazione che utilizza i dati e consente un nuovo livello di gestione e manipolazione delle informazioni basate su Web. A basso livello, il server administrator può memorizzare le informazioni in Xml, adottando per i dati il formato più naturale e senza preoccuparsi della loro visualizzazione.

Ad un livello più alto, l'informazione in html per essere visualizzata tramite un browser secondo specifiche Xml. È dunque necessario un livello software aggiuntivo che traduca Xml in html al fine di dotare i browser del normale interfaccia di un server HTTP. Inoltre, l'information level può memorizzare in formato Xml informazioni relative al server management, che il server può leggere ed interpretare nel corso delle sue attività. Infine, poiché il set dei tags di Xml è liberamente estendibile, Xml consente di rappresentare tutti i tipi di dati e di entità che si possono trovare in Internet, dai documenti complessi agli agenti. Per esempio, nel caso di un DBMS che usa una rappresentazione proprietaria dei dati, si potrebbero tradurre in formato Xml tutte le informazioni prodotte dalle query, lasciando che sia l'Xml server a tradurle successivamente in html. In questo caso, l'approccio classico prevedrebbe un'applicazione CGI per accedere al DBMS e tradurre i risultati in html. Gli obiettivi progettuali di Xml sono:

1. **Xml deve essere utilizzabile in modo semplice su Internet:** in primo luogo, l'Xml deve operare in maniera efficiente su Internet e soddisfare le esigenze delle applicazioni eseguite in un ambiente di rete distribuito.
2. **Xml deve supportare un gran numero di applicazioni:** deve essere possibile utilizzare l'Xml con un'ampia gamma di applicazioni, tra cui strumenti di creazione, motori per la visualizzazione di contenuti, strumenti di traduzione e applicazioni di database.

3. **Xml deve essere compatibile con SGML:** questo obiettivo è stato definito sulla base del presupposto che un documento Xml valido debba anche essere un documento SGML valido, in modo tale che gli strumenti SGML esistenti possano essere utilizzati con l'Xml e siano in grado di *analizzare* il codice Xml.
4. **Deve essere facile lo sviluppo di programmi che elaborino documenti Xml:** l'adozione del linguaggio è proporzionale alla disponibilità di strumenti e la proliferazione di questi è la dimostrazione che questo obiettivo è stato raggiunto.
5. **Il numero di caratteristiche opzionali deve essere mantenuto al minimo possibile:** al contrario dell'SGML, l'Xml elimina le opzioni, in tal modo qualsiasi elaboratore potrà pertanto analizzare qualunque documento Xml, indipendentemente dai dati e dalla struttura contenuti nel documento.
6. **I documenti Xml dovrebbero essere leggibili da un utente e ragionevolmente chiari:** poiché utilizza il testo normale per descrivere i dati e le relazioni tra i dati, l'Xml è più semplice da utilizzare e da leggere del formato binario che esegue la stessa operazione; inoltre poiché il codice è formattato in modo diretto, è utile che l'Xml sia facilmente leggibile da parte sia degli utenti che dei computer.
7. **La progettazione di Xml dovrebbe essere rapida:** l'Xml è stato sviluppato per soddisfare l'esigenza di un linguaggio estensibile per il Web. Questo obiettivo è stato definito dopo aver considerato l'eventualità che se l'Xml non fosse stato reso disponibile rapidamente come metodo per estendere l'HTML, altre organizzazioni avrebbero potuto provvedere a fornire una soluzione proprietaria, binaria o entrambe.
8. **La progettazione di Xml deve essere formale e concisa:** questo obiettivo deriva dall'esigenza di rendere il linguaggio il più possibile conciso, formalizzando la formulazione della specifica.
9. **I documenti Xml devono essere facili da creare:** i documenti Xml possono essere creati facendo ricorso a strumenti di semplice utilizzo, quali editor di testo normale.
10. **Non è di nessuna importanza l'economicità nel markup Xml:** nell'SGML e nell'HTML la presenza di un tag di apertura è sufficiente per segnalare che l'elemento precedente deve essere chiuso. Benché così sia possibile ridurre il lavoro degli autori, questa soluzione potrebbe essere fonte di confusione per i lettori, nell'Xml la chiarezza ha in ogni caso la precedenza sulla concisione.

Appendice C: glossario dei termini usati

- **Acl:** acronimo di Access Control List, set di dati che informa il sistema operativo di un computer su quali permessi e diritti di accesso possiede ogni utente o gruppo in uno specifico ambito.
- **Corba:** acronimo di Common Object Broker Architecture, architettura che abilita pezzi di programmi, chiamati oggetti, a comunicare con altri oggetti indipendentemente dal linguaggio di programmazione col quale sono stati scritti o dal sistema operativo sul quale sono in esecuzione. Corba è stato sviluppato dal consorzio OMG. Tra le implementazioni di Corba, le più diffuse sono SOM di IBM, DCOM di Microsoft e RMI di Sun Microsystems
- **DTD:** acronimo di Document Type Definition, definisce i blocchi corretti di un documento Xml e la struttura attraverso essi. Un DTD può essere dichiarato inline nel documento Xml oppure come riferimento esterno. Un DTD deve inoltre a definire una grammatica libera dal contesto alla quale i documenti Xml devono attenersi
- **Fipa:** acronimo di Foundation for Intelligent Physical Agents, organizzazione non-profit nata per creare standards per l'interoperabilità di agenti software eterogenei.
- **KQML:** acronimo di Knowledge Query and Manipulation Language, linguaggio e protocollo per lo scambio di informazioni e conoscenza usato dai programmi per comunicare tra di loro. Può essere usato oltre che come linguaggio come sistema intelligente per il supporto al cooperative problem solving. È complementare al sistema Corba
- **OMG:** acronimo di Object Management Group, consorzio che racchiude più di 700 industrie col compito di sviluppare un framework comune per lo sviluppo di applicazioni basate sul paradigma object-oriented. È responsabile delle specifiche Corba.
- **Pattern matching:** tecnica che consente complessi controlli decisionali sulle strutture dati da esprimere in maniera puntuale; è implementata in parecchi linguaggi moderni, come Standard ML, Haskell e Miranda. La forma base di un'espressione pattern matching è "*match exp [pat body]...*", dove *exp* è un'espressione, *pat* è un pattern e *body* una o più espressioni.
- **RMI:** acronimo di Remote Method Invocation, set di protocolli sviluppato dalla divisione JavaSoft di Sun Microsystems, abilita gli oggetti Java a comunicare in maniera remota con altri oggetti Java
- **Tupla:** nel gergo dei Relational Database Management Systems, sinonimo di record; set completo di informazioni composto da campi (fields) ognuno dei quali contiene un oggetto dell'informazione

Appendice D: riferimenti bibliografici

- [1] H.S.Nwana, L.Lee, N.R.Jennings, "Co-ordination in software agent systems", *BT Technol J* Vol 14, 1996
- [2] Giulio Guaitoli, "Infrastrutture di coordinazione dipendenti dal contesto in mabito wireless", <http://polaris.ing.unimo.it/didattica/curriculum/letizia/tesi/guaitoli/Tesi.pdf>, 2000
- [3] Luca Cardelli, Andrew D.Gordon, "Mobile Ambients", *Formal Methods for Distributed Processing, A Survey of Object-Oriented Approaches*, H. Bowman and J. Derrick Editors, Cambridge University Press, 2001. ISBN 0-521-77184-6. pp 198-229.
- [4] Gelernter, Carriero, "Coordination languages and their significance", *Communications of the ACM*, vol. 35 No. 2, 1992, pp. 96-107
- [5] Paolo Ciancarini, "Agent Technology", <http://www.cs.unibo.it/~cianca>, 2000
- [6] Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, "Coordination infrastructures for mobile agents", *Elsevier Microprocessors and Microsystems* 25 (2001) 85-92
- [7] Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, "How to Coordinate Internet Applications based on Mobile Agents", *Enabling Technologies: Infrastructure for Collaborative Enterprises*, 1998. (WET ICE '98) Proceedings., Seventh IEEE International Workshops on , 1998 Page(s): 104 -109
- [8] Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, "The Impact of the Coordination Model in the Design of Mobile Agent Applications", *IEEE Computer Software and Applications Conference*, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International , 1998 Page(s): 436 -441
- [9] L. Lee H.S. Nwana and N.R. Jennings, "Co-ordination in Multi-Agent Systems", in H. S. Nwana and N. Azarmi, editors, *Software Agents and Soft Computing*, number 1198 in LNAI. Springer Verlag, 1997
- [10] S. Bussmann and J. Muller, "A Negotiation Framework for Co-operating Agents", in S. M. Deen, editor, *Proceedings of CKBS-SIG*, pages 1-17. Dake Centre, University of Keele, 1992
- [11] C. Khunboa, R. Simon, "On the Performance of Coordination Spaces for Distributed Agent Systems", aiga.cs.gmu.edu/papers/sim_symposium.pdf
- [12] E. Denti, A. Natali, A. Omicini, "On the Expressive Power of a Language for Programmable Coordination Media", *ACM Symposium on Applied Computing*, Atlanta, 1998
- [13] Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, "Reactive Tuple spaces for Mobile Agent Coordination", *2nd International Workshop on Mobile Agents*, LNCS, No. 1477, Springer-Verlag, 1998
- [14] Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, "Engineering Mobile-agent Applications via Context-dependet Coordination", *IEEE Software Engineering*, 2001. ICSE 2001. Proceedings of the 23rd International Conference on, 2001 Page(s): 371 - 380
- [15] R. Tolksdorf, "Coordinating Java agents with multiple coordination language on the Berlinda platform", *Enabling Technologies: Infrastructure for Collaborative Enterprises*, 1997. Proceedings., Sixth IEEE Workshops on , 1997 Page(s): 121 -126
- [16] D. B. Lange, D. T. Chang, *IBM Aglets Workbench - Programming Mobile Agents in Java*, IBM Corporation White Paper, 1996

- [17] Holger Peine, “Application and Programming Experience with the Ara Mobile Agent System”, preprint of an article accepted for publication in IEEE Software – Practice and Experience, 2002
- [18] Lingnau, “ffMAIN: using Tcl and the TclHttpd Web Server to implement a mobile agent infrastructure”, www.tu-harburg.de/skf/tcltk/papers2000/paper.pdf, 2000
- [19] Sun Microsystems, “The JavaSpace Specification”, <http://www.sun.com/jini/specs/js-spec.html>, 1999
- [20] The MOON Home Page, University of Modena, <http://sirio.dsi.unimo.it/MOON>
- [21] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, A. Knoche, “Coordinating Multi-Agents Applications on the WWW: a Reference Architecture”, Software Engineering, IEEE Transactions on , Volume: 24 Issue: 5 , May 1998 Page(s): 362 -375
- [22] G. Cabri, L. Leonardi, F. Zambonelli, “Mobile-Agent Coordination Models for Internet Applications”, IEEE Computer , Volume: 33 Issue: 2 , Feb. 2000 Page(s): 82 -89
- [23] Gabriele Reggiani, Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, “Design and Implementation of a Programmable Coordination Architecture for Mobile Agents”, IEEE Technology of Object-Oriented Languages and Systems, 1999. Proceedings of , 1999 Page(s): 10 -19
- [24] Paolo Ciancarini, Andrea Omicidi, Franco Zambonelli, “Coordination Models for Multi-Agent Systems”, <http://www.agentlink.org> newsletter 3, 1999
- [25] J.Baumann, F.Hohl, M.Straßer, “Beyond Java: merging Corbabaased Mobile Agents and WWW”, a position paper for the Joint W3C/OMG Workshop on Distributed Objects and Mobile Code, 1996
- [26] P.Ciancarini, A.Omicini, F.Zambonelli, “Coordination Technologies for Internet Agents”, Nordic Journal of Computer 6, 1999
- [27] G. Cabri, L. Leonardi, F. Zambonelli, “Implementing Agent Auctions using MARS”, <http://sirio.dsi.unimo.it/MOON/papers/papers.html>, 2000
- [28] Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, “Auction-based Negotiation via Programmable Tuple spaces”, <http://dsi.unimo.it/MOON/papers/cia00.pdf>
- [29] Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, “Xml Dataspaces for Mobile Agent Coordination”, ACM 1-58113-239-5/00/003, 2000
- [30] Paolo Bellavista, Antonio Corradi, Cesare Stefanelli, “A Mobile Agent Infrastructure for the Mobility Support” <http://deis.unibo.it/Staff/PaoloBellavista/papers/sac00b.pdf>
- [31] R.Tolksdorf, D.Glaubitz, “XmlSpaces for Coordination in Web-based Systems”, flp.cs.tu-berlin.de/~tolk/xmlspaces , 2001

Indice

- 1 Introduzione**
 - 1.1 La coordinazione nei sistemi ad agenti mobili
- 2 I modelli di coordinazione**
 - 2.1 Tassonomia dei modelli di coordinazione
 - 2.1.1 Coordinazione Direct
 - 2.1.2 Coordinazione Meeting-Oriented
 - 2.1.3 Coordinazione Blackboard-based
 - 2.1.4 Coordinazione Linda-like
 - 2.2 Coordinazione context-dependent
 - 2.2.1 Coordinazione environment-dependent
 - 2.2.2 Coordinazione application-dependent
 - 2.3 I linguaggi di coordinazione e Berlinda
 - 2.3.1 Berlinda
- 3 Implementazione dei modelli di coordinazione**
 - 3.1 IBM Aglets
 - 3.2 Ara
 - 3.3 ffMAIN
 - 3.4 JavaSpace
 - 3.5 Mars
 - 3.5.1 L'interfaccia Mars
 - 3.5.2 Modello reattivo
 - 3.5.3 Esempio: Mars e il data retrieval
 - 3.5.4 Coordinazione context-dependent in Mars
 - 3.6 Confronto tra modelli
 - 3.6.1 Definizione dei Coordinables
 - 3.6.2 Definizione dei Coordination Media
 - 3.6.3 Definizione delle Coordination Laws
- 4 Confronto critico**
 - 4.1 Un esempio applicativo: le aste su Internet
 - 4.1.1 Implementazione di aste
 - 4.1.2 Coordinazione inter-agent
 - 4.1.3 Coordinazione agent-to-hosting environment
- 5 Progetti futuri**
 - 5.1 Mars-X
 - 5.2 XmlSpaces

Appendice A: il sistema Linda

Appendice B: l'architettura Xml

Appendice C: glossario dei termini usati

Appendice C: riferimenti bibliografici