Indice

Introduzione

I sistemi distribuiti sono da tempo oggetto di studio, e una attenzione particolare è stata dedicata loro negli ultimi anni, soprattutto per il crescente sviluppo di quella che può essere considerata la rete geografica per eccellenza: Internet. Questa tendenza è certamente connessa all'importanza assunta da numerosi servizi di rete, che, inducono ad un sempre crescente interesse del mercato verso la "interconnessione globale".

I modelli progettuali che erano stati concepiti per sistemi distribuiti convenzionali, formati da piccole reti locali, risultano non ottimizzati e quindi inefficienti per una distribuzione su larga scala, evidenziando problemi legati alla flessibilità, alla scalabilità e alla tolleranza ai guasti.

Questi problemi hanno portato alla ricerca di nuove soluzioni e nuovi modelli, in cui non è necessariamente presente un livello di trasparenza delle risorse, che risulta spesso di inefficiente realizzazione. In questo contesto nasce l'idea della mobilità del codice, che consente di cambiare dinamicamente il collegamento tra un frammento di codice e la locazione della sua esecuzione. All'interno di questa area, si inserisce una proposta recente, che si basa sul concetto di Agente Mobile: l'Agente è una unità di esecuzione che agisce in modo autonomo ed ha la caratteristica di essere Mobile, cioè ha l'abilità di muoversi all'interno di una rete durante la sua esecuzione. Progettare una applicazione distribuita secondo il paradigma ad agenti mobili significa realizzare una o più

di queste entità, capaci di coordinarsi al fine di ottenere un risultato comune.

Il modello ad agenti mobili può condurre ad una riduzione del traffico di rete e ad un aumento delle performance delle applicazioni distribuite; gli agenti possono offrire servizi flessibili, facilmente espandibili e adeguati alla dinamicità della rete. Le caratteristiche peculiari del modello ad agenti mobili, lo rendono particolarmente adatto ad una serie di domini applicativi, quali, ad esempio, l'Information Retrieval distribuito, il Commercio Elettronico o il Mobile Computing.

L'utilizzo di questo modello prevede l'esistenza di un framework che supporti l'esecuzione degli agenti: tale supporto deve essere scalabile, flessibile ed aperto, e deve rendere il più semplice possibile la modifica dinamica del sistema. L'insieme degli ambienti di esecuzione degli agenti deve poter variare dinamicamente: il framework deve permettere l'inserimento e l'eliminazione di località senza compromettere il funzionamento globale e deve essere capace di gestire gli sviluppi dinamici della configurazione. L'architettura deve fornire anche meccanismi e strumenti per il Network Management, indispensabile per una gestione dinamica della rete: il controllo deve essere decentralizzato tra agenti dislocati su più nodi del sistema e fra loro coordinati.

Nel primo capitolo, analizziamo i sistemi distribuiti, e i limiti che derivano dal loro uso secondo un approccio tradizionale; introduciamo il concetto di mobilità del codice e valutiamo come questa tecnologia riesca a superare alcuni svantaggi dell'approccio tradizionale. Analizziamo in seguito le caratteristiche principali dei linguaggi a codice mobile e i relativi domini applicativi, evidenziando i vantaggi di questa tecnologia ed i problemi di sicurezza connessi. Si fornisce una classificazione dei paradigmi di progettazione a codice mobile, confrontandola con il classico modello cliente-servitore.

Il secondo capitolo è dedicato ad un approfondimento del modello ad Agenti Mobili: descriviamo le caratteristiche principali di migrazione, comunicazione e sicurezza, e come Java sia un linguaggio adatto per la mobilità. Viene inoltre approfondito il tema del Network Management, e viene confrontato un approccio tradizionale di gestione e amministrazione di rete con uno innovativo basato sul paradigma ad agenti mobili.

Nel terzo capitolo introduciamo un'analisi di alto livello del supporto che abbiamo realizzato presso il dipartimento del DEIS dell'Università di Bologna: il sistema SOMA (Secure and Open Mobile Agent).

Infine, nel quarto capitolo, illustriamo con maggiore dettaglio le scelte implementative dell'architettura SOMA, prestando particolare attenzione alla gestione dinamica del sistema e al protocollo di comunicazione utilizzato.

Cap.1

Mobilità del codice nei sistemi distribuiti

1.1 I sistemi distribuiti

La larga diffusione di Internet e del Web ha indubbiamente fatto crescere, negli ultimi anni, l'interesse verso le reti di calcolatori (anche locali), cambiando la stessa percezione della macchina-computer, considerata non più come un'entità autonoma, ma come parte integrante di una struttura globale, costituita dalla sinergia di risorse locali e remote.

1.1.1 Nascita dei sistemi distribuiti

Prima degli anni '70 i sistemi informatici erano caratterizzati da strutture imponenti e molto costose, che solo importanti centri di ricerca potevano permettersi: le grosse risorse di calcolo erano rappresentate dai mainframe. Queste macchine imponenti non erano certo progettate per essere facilmente collegate tra loro, quindi lavoravano tipicamente in modo indipendente l'una dall'altra.

I progressi elettronici e le innovazioni tecnologiche avvenute nei primi anni '70 hanno portato notevoli miglioramenti di prestazioni e di dimensioni nell'ambito informatico, offrendo sul mercato macchine accessibili, economicamente, anche a livello di

piccole aziende. Questa novità ha rivoluzionato l'atteggiamento nei confronti dei sistemi informatici: nella tecnologia a mainframe, si doveva comprare la macchina più grande che ci si potesse permettere, perché con una spesa maggiore si ottenevano prestazioni notevolmente migliori [Tan94]. Con la nuova tecnologia dei microprocessori, la soluzione più efficace rispetto al costo è quella di raggruppare un numero maggiore di CPU (Central Process Unit) in un singolo sistema: con lo stesso investimento si può ottenere un rapporto prezzo prestazioni molto più favorevole rispetto ai grandi sistemi centralizzati, che può portare ad una potenza di calcolo teoricamente infinita, raggruppando un numero sempre crescente di microprocessori.

La diffusione di massa degli elaboratori ha spinto velocemente all'acquisto di più computer da affiancare sia per trarre benefici dall'incremento di prestazioni che ne consegue, sia per la natura intrinsecamente distribuita di alcune applicazioni (si pensi alla prenotazione aerea). È nata così la necessita di accedere da ogni macchina a tutte le risorse disponibili, come le stampanti o le unità di memorizzazione, indispensabili durante lo svolgimento delle attività lavorative: da qui la creazione di reti locali, dette LAN (Local Area Network), per collegare i calcolatori e quindi in generale tutte le risorse utilizzabili, hardware e software. In questo modo, è possibile acquistare componenti costosi come stampanti o altre periferiche, e sfruttarle con più macchine, aumentando anche l'efficienza totale del sistema.

I collegamenti di rete, consentono una distribuzione efficiente del carico su tutte le macchine disponibili, costruendo un sistema flessibile, affidabile e aperto: le scelte di gestione si adattano alle risorse disponibili, gli eventuali guasti non causano la totale caduta del sistema, e le aumentate esigenze sono soddisfatte con l'inserimento di nuovi elementi, che permettono una crescita incrementale della rete.

Il numero sempre crescente di macchine collegabili e la distanza sempre più elevata che li separa, ha introdotto la terminologia di **sistemi distribuiti**, per evidenziarne la contrapposizione alla tecnologia classica dei sistemi centralizzati; un sistema distribuito può essere definito come *un insieme di sistemi distinti per località che cooperano per ottenere risultati coordinati*.

Elemento	Descrizione
Economia	I microprocessori offrono un miglior rapporto prezzo prestazioni rispetto ai mainframe
Velocità	Un sistema distribuito può avere una potenza di calcolo complessiva superiore a quella di un mainframe
Distribuzione intrinseca	Alcune applicazioni richiedono macchine separate e distanti
Affidabilità	Se una macchina cade, il sistema può nel complesso sopravvivere
Flessibilità	Distribuisce il carico sulle macchine disponibili, ottenendo una maggiore efficienza
Condivisione di risorse	Permette a molti utenti di condividere risorse dispendiose e di accedere alla stessa base di dati
Crescita incrementale	Si può aggiungere potenza di calcolo in passi incrementali

Tabella 1 I vantaggi dei sistemi distribuiti.

I sistemi distribuiti hanno anche dei punti deboli. Un problema è la dipendenza del sistema dal canale trasmissivo: la rete di comunicazione può danneggiarsi o essere troppo carica (satura) da divenire inutilizzabile. In queste situazioni si perdono tutti i vantaggi dei collegamenti, su cui si basa il sistema stesso.

Un altro problema, molto sentito nei sistemi distribuiti è legato alla sicurezza: abbiamo accennato prima alla capacità di poter accedere alle informazioni disponibili su tutto il sistema, e lo abbiamo citato sotto la voce dei vantaggi. In questo modo,

comunque, si può reperire ogni tipo di dato, anche quello che non si vorrebbe diffondere: la sicurezza informatica è sicuramente un limite allo sviluppo di applicazioni distribuite, ma grossi sforzi sono dedicati a questo settore e sono già stati presentati notevoli passi avanti [KauPS95].

Elemento	Descrizione
Connessione di rete	La rete può saturarsi o causare altri problemi
Sicurezza	La facilità di accesso, vale anche per i dati riservati

Tabella 2 Gli svantaggi dei sistemi distribuiti.

Per ricavare un vantaggio reale e completo dall'utilizzo di sistemi distribuiti, si devono rispettare ed ottenere alcune caratteristiche chiave, che saranno analizzate nel seguito.

1.1.2 Caratteristiche chiave

Un sistema distribuito è costituito da una collezione di computer autonomi collegati in rete e correlati da un sistema software distribuito, che li coordina nello svolgimento delle loro attività. Una singola macchina collegata al sistema, è detta anche host o nodo del sistema e i componenti software e hardware fisicamente residenti o direttamente collegati ad essa sono le risorse locali, mentre tutte le risorse delle altre macchine si dicono remote.

Analizziamo le sette caratteristiche principali da cui deriva la reale utilità dei sistemi distribuiti: la condivisione delle risorse, la concorrenza, la tolleranza ai guasti, la scalabilità, la trasparenza, la sicurezza e l'apertura [CouDK94]. Queste caratteristiche non sono conseguenza diretta dei sistemi distribuiti, ma è compito del

progettista creare applicazioni nel rispetto di queste proprietà fondamentali.

1.1.2.1 Condivisione di risorse

Dalla definizione di sistema distribuito (*un insieme di sistemi distinti per località che cooperano per ottenere risultati coordinati*) emerge che una prima caratteristica chiave deve essere quella di fornire la possibilità di coordinare lavori differenti, svolti anche in località remote.

La condivisione di risorse, è solo uno dei possibili modi per realizzare coordinazione, ma è una caratteristica fondamentale dei sistemi distribuiti. Il termine risorsa è molto generico, e comprende un insieme molto vasto di elementi sia hardware che software: nel primo caso, si pensi alla possibilità di sfruttare stampanti remote o altre periferiche molto costose, nel secondo caso si pensi al fatto di condividere dati, caratteristica indispensabile per progetti che si basano su gruppi di lavoro, o più in generale al fatto di consultare e accedere a database remoti.

Una risorsa condivisa può essere utilizzata da entità diverse (per esempio, processi utente o processi di sistema), che potrebbero risiedere su altri nodi della rete: è importante garantire un meccanismo di comunicazione che permetta solo accessi affidabili e modifiche consistenti. Questo ruolo è svolto da un gestore della risorsa, detto *resource manager*, che è tipicamente rappresentato da un componente software e che si occupa di una certa quantità (non necessariamente unitaria) di risorse dello stesso tipo. Infatti ogni tipo di risorsa prevede una gestione personalizzata delle politiche e delle modalità di accesso, mentre esistono caratteristiche comuni a tutti i resource manager: per esempio, è indispensabile un modello di naming per catalogare le risorse, ed anche una coordinazione

specifica per gli accessi contemporanei alla risorsa, che rischiano di comprometterne la consistenza. Il resource manager fornisce dei servizi sulla risorsa condivisa, che possono essere richiesti da tutte le entità che devono interagire con la risorsa stessa.

1.1.2.2 Concorrenza

All'interno di una singola macchina, si parla di concorrenza quando esiste più di un processo in esecuzione. Se il nodo considerato è multiprocessore, si parla più precisamente di esecuzione parallela; nel caso monoprocessore per avere concorrenza è sufficiente che l'esecuzione di un processo inizi prima della terminazione di un altro, attraverso una gestione *interleaving*, in cui si intervalla l'esecuzione di una porzione di codice di un processo con quella di un altro.

Il concetto di concorrenza espresso in dettaglio per una macchina, può essere generalizzato in un sistema distribuito, formato da più host: in questo caso sono presenti più processori, che permettono ai vari processi di eseguire su macchine separate, in modo parallelo. In particolare, in un sistema basato sulla condivisione di risorse, l'esecuzione parallela può presentarsi qualora più utenti invochino contemporaneamente uno stesso programma applicativo, e, dal lato opposto, qualora più servitori (o in generale resource manager) esaudiscano richieste in modo simultaneo. Il vantaggio della concorrenza, deve essere controbilanciato da una gestione accurata della sincronizzazione: quando più entità vogliono accedere contemporaneamente ad una risorsa le loro azioni devono essere sincronizzate.

Quando, all'interno di una applicazioni sono disponibili frammenti di codice che possono essere svolti in modo autonomo e in parallelo, la loro esecuzione concorrente consente di ridurre il

tempo di attesa totale dell'applicazione (la legge di Amdhal [Cor98] conferma che un programma può essere suddiviso in una parte parallela e in un parte sequenziale).

1.1.2.3 Fault tolerance

Può capitare che in un sistema si verifichi un *fault* (errore, guasto), cioè un qualunque comportamento diverso da quello previsto dai requisiti, per esempio, un programma potrebbe produrre un risultato non corretto, o addirittura non riuscire a portare a termine la sua esecuzione. La causa di un errore può essere varia e in generale è definita con il termine *failure*, che rispecchia il modo di agire del sistema nel caso in cui si possono presentare errori.

Con il termine *fault tolerance*, si intende, in generale, la capacità del sistema di saper "tollerare gli errori", cioè di prevedere come reagire nel caso in sui si verifichi un comportamento indesiderato. Questa capacità del sistema, detta anche *dependability*, può essere espressa su due livelli: la *reliability*, che sottolinea la possibilità del sistema di fornire risposte corrette (l'accento è sulla correttezza delle risposte), e la *availability*, che evidenzia la possibilità del sistema di fornire comunque delle risposte, in un tempo limitato (l'accento è sul tempo di risposta).

Per realizzare un sistema tollerante ai guasti, sono possibili due tipi di approccio: l'uso di componenti hardware ridondanti, oppure la progettazione di programmi software capaci di ripristinare (recovery) una situazione consistente in caso di fault.

Nel primo caso, è possibile duplicare i componenti hardware, come una intera macchina, o elementi di granularità più fine, sfruttati nello svolgimento di una applicazione: mentre uno lavora, l'altro rimane in attesa di sostituirlo nel caso in cui si verifichino dei problemi. La soluzione presentata è molto costosa e si usa solo quando il livello di tolleranza ai guasti deve essere elevata; in altri

casi gli elementi replicati non sono lasciati inutilizzati, ma sono sfruttati per attività non critiche, quando non vi sono guasti. Per esempio, un database potrebbe essere replicato su più macchine, affinché la sua consultazione sia un servizio sempre garantito anche in caso di guasto di una di esse; quando il guasto dovesse accadere, le richieste dirette all'host caduto, saranno ridirette sugli altri nodi. Da questo si può capire che la tecnica di fault tolerance, basata sulla replicazione, se ben sfruttata, può anche offrire un incremento di efficienza, nella situazione di assenza di guasti.

Nel secondo caso, la tolleranza ai guasti è garantita da un recovery software [WahLAS93], cioè dalla progettazione di software capace di mantenere lo stato consistente dei dati, anche in caso di fault. Quando nel sistema si presenta un errore, i programmi in esecuzione potrebbero aver aggiornato solo parzialmente una struttura dati (un file, o un generico insieme di informazioni memorizzate) lasciandola in una situazione non consistente (si pensi, per esempio, a una transazione bancaria e ad un conto corrente non aggiornato). La soluzione è quella di ripristinare una situazione consistente: per questo scopo esistono alcuni meccanismi analizzati in [CouDK94].

1.1.2.4 Scalabilità

Un sistema scalabile prevede che le sue caratteristiche (tra cui le prestazioni, ma non solo) non siano influenzate in alcun modo dalla dimensione del sistema, cioè dal numero di macchine collegate. È facile comprendere che di assoluta scalabilità si possa parlare solo in termini teorici, mentre in termini pratici si può considerare già un buon risultato che al crescere delle dimensioni del sistema si rilevi una scarsa quantità di nuovi problemi.

Per esempio, consideriamo una piccola rete in cui sia disponibile una sola macchina con il servizio di "file server": al

crescere delle dimensioni della rete è possibile che la frequenza di richieste in arrivo al servitore aumenti a tal punto da rendere questo servizio un “collo di bottiglia” (cioè da costituire un rallentamento estremo dell’esecuzione di chi presenta la richiesta): una soluzione è quella di replicare i file anche su altre macchine e quindi mettere a disposizione altre fonti dello stesso servizio, per evitare di aumentare eccessivamente la frequenza media delle richieste. In queste situazioni, la replicazione dei dati è una buona soluzione, ma il sistema deve essere scalabile, nel senso che deve essere stato progettato per reagire all’inserimento dinamico, suddividendo equamente le richieste tra il vecchio e il nuovo servitore.

Da questo esempio si intuisce l’estrema importanza della scalabilità in un sistema distribuito su larga scala e in continua e rapida espansione come Internet, che costituisce una rete di migliaia o milioni di macchine: l’obiettivo sarebbe quello di fornire servizi in modo “quasi indipendente” dal numero di nodi connessi.

1.1.2.5 Trasparenza

Un sistema distribuito possiede la caratteristica della trasparenza quando la distribuzione dei suoi componenti su macchine differenti è nascosta all’utente e ai programmi applicativi, che percepiscono il sistema come una unica totalità. Per ottenere questa caratteristica è necessaria una coordinazione, realizzata sotto al livello utente, attraverso la comunicazione e la gestione esplicita dei componenti distribuiti nel sistema. In realtà, la trasparenza può essere ottenuta in modi differenti, in funzione della forma particolare di trasparenza fornita all’utente.

- *trasparenza all’accesso*: le risorse remote e le risorse locali sono accedute dall’utente nello stesso modo;

- *trasparenza alla locazione*: l'utente non è in grado di capire la locazione fisica delle risorse che utilizza;
- *trasparenza alla concorrenza*: l'utente non si accorge della presenza di eventuali altri utenti che usano le stesse risorse, poiché non vi sono interferenze tra loro;
- *trasparenza alla replicazione*: la presenza di un numero di copie della risorsa, per aumentare l'affidabilità e le prestazioni, non è noto all'utente;
- *trasparenza ai guasti*: l'utente non si accorge di eventuali problemi (hardware o software), terminando comunque la sua applicazione;
- *trasparenza alla migrazione*: le risorse possono spostarsi tra i nodi del sistema, senza che l'utente possa accorgersene;
- *trasparenza al parallelismo*: l'utente non si accorge che la sua applicazione sia suddivisa su più unità di elaborazione;
- *trasparenza alla scalabilità*: il sistema può aumentare la scala di distribuzione e l'utente non avverte nessun cambiamento.

La trasparenza nasconde e rende anonime le risorse usate dai programmi utente: questo potrebbe non essere sempre quello che si desidera. Per esempio, l'utente potrebbe voler conoscere esplicitamente la locazione di un componente sapendo di ridurre così il tempo di attesa di un servizio: una spiegazione più dettagliata sarà fornita nel paragrafo 1.2.1.

1.1.2.6 Sicurezza

Una caratteristica chiave per lo sviluppo di applicazioni distribuite, è la sicurezza: la capacità di poter accedere alle informazioni disponibili su tutto il sistema consente ad ogni utente di reperire ogni tipo di dato, anche quello che non si vorrebbe diffondere. Le

informazioni private, inserite in un sistema distribuito, devono essere protette da accessi indesiderati, per evitare di farle consultare o modificare. La sicurezza informatica è sicuramente un limite allo sviluppo di applicazioni distribuite: è compito del progettista scegliere adeguatamente le strutture dati da proteggere, sfruttando, per esempio, i numerosi algoritmi proposti per questo scopo [KauPS95].

1.1.2.7 Apertura

Una sistema distribuito gode della caratteristica di apertura se è dotato di una capacità evolutiva dinamica, cioè se consente di essere esteso in qualche modo, con l'introduzione di risorse hardware (per esempio, nuove periferiche, aggiunta di memoria o di una interfaccia di comunicazione) e software (per esempio, inserimento di nuove proprietà di sistema operativo, di protocolli di comunicazione o di servizi su risorse condivise).

Un sistema, per essere aperto deve utilizzare interfacce pubbliche, come possono essere gli standard riconosciuti a livello internazionale, ma più semplicemente, anche interfacce personalizzate, su cui però si esegua una adeguata diffusione e documentazione. La disponibilità delle interfacce permette agli sviluppatori e ai progettisti software e hardware di estendere il sistema con nuovi componenti adeguati alla struttura già esistente. In questo modo la caratteristica di apertura consente di sviluppare il sistema su una struttura eterogenea, sia dal lato hardware che software: una attenzione particolare deve essere rivolta al fatto che gli elementi inseriti in fasi successive, rispettino totalmente le interfacce fornite, per tutelare l'utilizzatore da indesiderati conflitti nati dalla non assoluta compatibilità di prodotti di marche differenti.

1.2 Mobilità del codice

Non sono state fornite definizioni universalmente riconosciute, né vi sono punti di accordo, su cosa si deve intendere per mobilità del codice. A nostro avviso, la definizione che segue ne racchiude in modo informale i concetti fondamentali: *la mobilità del codice è la capacità di cambiare dinamicamente il collegamento tra frammenti di codice e la locazione in cui sono eseguiti* [CarPV97]. Dalla definizione emerge che l'ambiente di azione naturale della mobilità del codice è rappresentato dai sistemi distribuiti, introdotti nel paragrafo precedente.

Sin dai primi esperimenti, la mobilità del codice ha suscitato notevole interesse in molti settori, sia accademici che industriali: questo interesse nasce dalla necessità di risolvere alcuni problemi e di superare alcuni limiti presenti con il **tradizionale approccio ai sistemi distribuiti**.

- La dimensione delle reti di calcolatori aumenta in modo vertiginoso: ne è una dimostrazione evidente l'incremento esponenziale degli utenti di Internet, ma anche a livello aziendale, le reti di gestione interna (intra-net) ed esterna (inter-net) richiedono continue e forti espansioni.
- L'aumento della dimensione dei sistemi distribuiti produce un inevitabile incremento del traffico di rete, che peggiora le prestazioni nelle comunicazioni remote.
- L'aumento del traffico di rete rende le interazioni con le risorse remote molto lente e costose in termini di tempi di attesa, di mancanza di concorrenza e possibilità di errori parziali.
- I collegamenti tra le macchine del sistema sono congestionate dal traffico eccessivo rischiando a volte di risultare inaffidabili

per lo svolgimento di operazioni critiche o totalmente inutilizzabili.

- Se una transazione è corrotta da un mal funzionamento risulta molto complicato il ripristino dello stato consistente su tutte le macchine coinvolte (*recovery distribuito*) [CouDK94].
- La struttura intrinseca dei sistemi è statica, senza possibilità di soluzioni dinamiche nel caso di guasti dei mezzi di comunicazione o delle macchine appartenenti alla rete.

Per chiarire, in generale, il concetto di mobilità nei sistemi distribuiti, citiamo l'esempio di una prima applicazione. In questo caso, la mobilità è sfruttata per fornire una gestione dinamica dei processi in esecuzione su un sistema distribuito: l'obiettivo è cercare di utilizzare in modo equo le risorse disponibili su tutte le macchine della rete ("load balancing") [Har94], per evitare di avere alcuni nodi troppo carichi di lavoro mentre altri rimangono inutilizzati. Per esempio, se all'interno di una rete aziendale sono disponibili due stampanti, si cerca di dividere in parti uguali il numero di stampe da realizzare, per evitare di dover attendere che si esaurisca una lunga coda di stampa prima di leggere il documento, quando l'altra stampante è inutilizzata. Questo esempio ci servirà anche nel seguito, per distinguere due metodi differenti per ottenere mobilità.

Le tecniche che si basano sulla mobilità del codice tentano di risolvere i principali problemi che caratterizzano il tradizionale approccio ai sistemi distribuiti: l'obiettivo ultimo è di offrire direttamente al programmatore forme di controllo sullo spostamento di ogni entità tra i vari nodi, inserendo un supporto run-time che implementi tutti i meccanismi necessari per la ricerca e la mobilità dei componenti, e che sia capace di adattarsi alla particolare politica di gestione scelta dall'amministratore di sistema.

Questa nuova visione del sistema appesantisce il ruolo del programmatore, che in prima persona deve saper sfruttare le caratteristiche ed i vantaggi della mobilità del codice. Infatti la mobilità permette di modificare ***dinamicamente*** il collegamento tra un frammento di codice e la locazione in cui eseguirlo, scegliendo la locazione in funzione delle esigenze attuali. Per trarre benefici, si può decidere di spostare l'esecuzione sul nodo meno carico, su quello in cui risiede la risorsa con cui interagire, su quello attualmente più stabile, su quello che offre i collegamenti più affidabili, ecc..

- Se il nodo su cui si risiede ha un "carico computazionale" elevato, mentre altri nodi limitrofi sono scarsamente utilizzati, la mobilità del codice consente di scegliere di proseguire l'esecuzione nell'*ambiente meno carico*, raggiungendo in minor tempo la conclusione del processo.
- Se è necessario interagire con una risorsa remota, si sposta il frammento di codice interessato sul nodo in cui risiede la risorsa, per aumentare le prestazione ("performance") attraverso il beneficio di una *interazione locale*, meno costosa di quella remota.
- Spostare l'esecuzione sulla macchina in cui risiede la risorsa da utilizzare, permette di *ridurre* anche *il traffico di rete*, evitando interazioni remote che sono la causa delle congestioni della rete.
- Sfruttando la mobilità del codice, lo svolgimento di operazioni critiche non è più minacciato da tratti di *connessioni inaffidabili*: è sufficiente spostare l'intera esecuzione nelle porzioni di rete in cui i collegamenti sono più stabili ed eventualmente restituire il risultato dell'operazione svolta in remoto.
- Non solo i collegamenti, ma anche una macchina può non fornire caratteristiche di *stabilità*: la mobilità consente di spostare dinamicamente l'esecuzione sul nodo che offre caratteristiche migliori in termini di stabilità.

- In conseguenza delle caratteristiche elencate, si introduce il concetto di "*mobile computing*": i nodi della rete non sono più vincolati ad essere situati in una locazione fisica fissa, e gli utenti possono muoversi insieme alle loro macchine su differenti posizioni, rimanendo connessi alla rete attraverso collegamenti senza fili ("wireless").

La mobilità del codice può essere sfruttata con grande interesse in sistemi distribuiti di ogni dimensione, *eterogenei* e *aperti*: l'esecuzione, diventa mobile, ed ogni sua parte può essere svolta nella locazione remota che offre in quel momento le migliori condizioni.

1.2.1 Controllo sulla locazione delle risorse

L'obiettivo tradizionale dei sistemi operativi distribuiti era quello di non fornire all'utente alcuna informazione sulla locazione delle risorse, anzi di non fare neppure trasparire l'esistenza stessa di nodi remoti. Le funzionalità di alto livello messe a disposizione da un sistema operativo realmente distribuito, consentono l'astrazione della posizione fisica degli oggetti nella rete, per cui tutte le risorse sono utilizzate come se fossero locali (Figura 1-1). Questo consente al programmatore di sviluppare applicazioni con facilità referenziando le risorse locali e remote con gli stessi meccanismi, indipendentemente dal nodo in cui risiedono.

Un noto esempio è rappresentato dall'architettura CORBA [CORBA], in cui la topologia della rete sottostante è nascosta al programmatore, e l'interazione con i componenti del sistema è realizzata senza alcuna informazione sulla locazione.

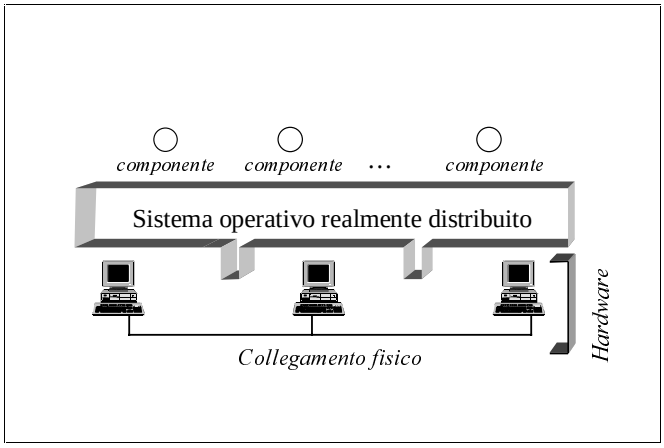


Figura 1-1 Nei sistemi distribuiti tradizionali la locazione dei componenti è trasparente per l'utente.

Nei sistemi distribuiti che supportano la mobilità, al contrario di quanto appena visto, è consentita la rilocazione dinamica dei componenti, intesi sia come unità di esecuzione (ad esempio un frammento di codice) sia come risorse utilizzate o condivise dalle unità appena citate (ad esempio un file).

Il codice mobile stravolge il tradizionale approccio verso i sistemi distribuiti: l'utente deve conoscere in modo esplicito la posizione fisica in cui le varie entità sono allocate nella rete, per poterle referenziare direttamente (Figura 1-2). Al programmatore è affidato il compito di sfruttare la conoscenza della locazione dei componenti per mettere in pratica i vantaggi della mobilità del codice. Per esempio l'utente deve sapere dove risiedono le risorse da utilizzare per trasferire, nella stessa macchina, il frammento di codice interessato all'interazione con la risorsa, per il beneficio di realizzare una esecuzione locale.

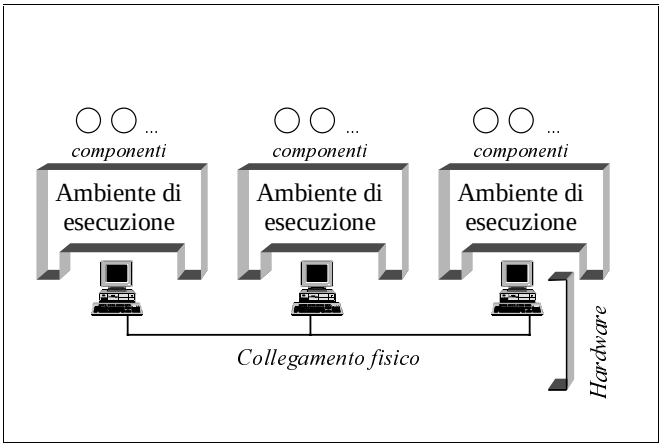


Figura 1-2 I sistemi basati sulla mobilità permettono la rilocazione dinamica dei componenti, per cui si deve avere conoscenza esplicita della posizione di ognuno di essi.

Per approfondire la differenza dei due contesti presentati, consideriamo l'esempio di "load balancing" introdotto nel paragrafo precedente. Se l'utente non ha coscienza dell'esistenza di nodi remoti (approccio tradizionale), la migrazione del carico avviene a livello di sistema operativo [Nut94] e serve un supporto che realizzi lo smistamento sapendo la struttura e la posizione delle risorse presenti (per esempio deve sapere come sono disposte le stampanti all'interno della rete aziendale): le modifiche dinamiche della composizione o del traffico della rete mettono in crisi l'obiettivo finale, cioè un buon bilanciamento del carico. Questo approccio, quindi, può essere utilizzato con soddisfazione solo in reti di piccole dimensioni, a causa della natura dinamica delle reti distribuite su larga scala, come Internet, che presentano una topologia non definita in modo statico.

Se invece lo smistamento del carico è realizzato dall'utente, in accordo con le caratteristiche della rete (come le modifiche topologiche, il traffico o la momentanea inaffidabilità di alcuni nodi) si realizza un bilanciamento di carico, che rispetta le scelte

considerate prioritarie dall'utente, anche in funzione della situazione corrente.

1.3 Mobilità di oggetti

Fino ad ora abbiamo parlato di mobilità di codice riferendoci genericamente allo spostamento di un frammento di codice. In realtà, insieme al codice si desiderano spostare anche informazioni aggiuntive, che derivano per esempio da interazioni con risorse locali o remote e che ci forniscono risultati intermedi (maggiori chiarimenti sono forniti nel paragrafo successivo). Quindi, quando si sfrutta la mobilità non si deve spostare solo il codice, ma anche altri elementi e sarebbe molto comodo poterli inglobare in un'unica entità per trasportarli facilmente. Questo obiettivo può essere realizzato riunendo gli elementi interessati in un unico "oggetto", che magari mantenga una struttura compatta anche durante la normale esecuzione.

È questo il motivo per cui la naturale architettura usata nello sviluppo di codice mobile è quella object-oriented, in cui un oggetto è l'unione di tutte le informazioni ed i suoi metodi rappresentano le modalità con cui accedervi [OOE95]: all'interno dell'oggetto sono presenti i suoi dati caratteristici e le informazioni su come poterli utilizzare. I dati dell'agente sono riservati e nessuno può accedervi in modo differente da quello messo a disposizione dall'oggetto stesso, attraverso l'interfaccia dei suoi metodi: la proprietà di incapsulamento, appena descritta, consente un livello significativo di protezione dei dati. L'enorme sviluppo dei linguaggi object-oriented negli ultimi anni, ha portato questa tecnologia ad essere quella prevalente nei progetti che sfruttano la mobilità del codice.

Tra i linguaggi object-oriented, citiamo l'esempio di Java [JAVA], che è un linguaggio abbastanza recente e che riscuote molto successo nel settore della mobilità (si veda il paragrafo 2.2).

1.4 Linguaggi a codice mobile

Sono stati proposti numerosi linguaggi di programmazione finalizzati ad un uso su Internet, con la caratteristica comune di fornire aspetti legati alla mobilità del codice, che a differenza dell'approccio tradizionale (si veda il paragrafo 1.2.1), integrano una mobilità su reti a larga scala.

Prima di confrontare i linguaggi tradizionali e quelli a codice mobile, definiamo alcuni concetti chiave indispensabili per proseguire [FugPV98]:

- *ambiente di esecuzione* (*CE: Computational Environment*): rappresenta l'identità della locazione in cui si svolge l'esecuzione.
- *unità di esecuzione (EU: Execution Unit)*: rappresenta un flusso sequenziale di esecuzione, che comprende oltre al codice, lo stato di esecuzione e lo spazio dei dati; esempi di EU sono un singolo Thread in un sistema multi-Tread o un qualunque processo sequenziale.
- *risorsa*: rappresenta un entità che può essere condivisa tra più unità di esecuzione, per esempio una variabile, un file o un oggetto in un sistema object-oriented.

Nei linguaggi tradizionali, come il C o il Pascal, ogni unità di esecuzione si svolge totalmente in un unico ambiente, ed il collegamento al suo codice è generalmente di tipo statico. Nei linguaggi basati sulla mobilità del codice [CugGPV97], invece, le parti fondamentali di una unita di esecuzione, cioè il segmento di

codice, lo stato di esecuzione e lo spazio dei dati, possono spostarsi dall'ambiente di esecuzione in cui risiedono ad un altro, in modo autonomo.

Distinguiamo i linguaggi di programmazione in funzione del tipo di mobilità che supportano [CarPV97], per evidenziare il livello di spostamento consentito.

- **Mobilità forte (*Strong Mobility*)**: un linguaggio supporta la mobilità forte se consente alle sue unità di esecuzione di spostarsi dall'ambiente in cui eseguono ad un altro, trasportando le loro parti caratteristiche: codice, stack e spazio dei dati. Prima della trasmissione, il flusso di esecuzione viene bloccato per essere poi ripristinato nell'ambiente di destinazione. Questo permette al flusso di esecuzione di bloccarsi non appena si esegue l'istruzione di trasferimento e di riprendere l'esecuzione nell'ambiente di destinazione esattamente dall'istruzione successiva: questa proprietà consente al linguaggio di essere più efficiente rispetto al caso successivo.

- **Mobilità debole (*Weak Mobility*)**: un linguaggio supporta la mobilità debole se consente lo spostamento di codice (ed eventualmente del suo spazio dei dati), o di oggetti nel caso di un linguaggio object-oriented. Le unità di esecuzione, al contrario del caso precedente, non hanno la capacità di spostarsi: all'interno dell'ambiente di esecuzione destinatario si effettua un collegamento dinamico tra un'unità di esecuzione dell'ambiente stesso e il codice proveniente da un altro ambiente o scaricato dalla rete. L'unità di esecuzione che si associa al codice in arrivo può essere creata appositamente, su necessità ("by need"), oppure se ne può sfruttare una preesistente.

Non abbiamo ancora approfondito come viene trattato lo spazio dei dati in un linguaggio basato sulla mobilità del codice. All'interno di un'unità di esecuzione potrebbero esserci dei riferimenti a variabili o più in generale a risorse caratteristiche dell'ambiente di esecuzione in cui si risiede e in caso di spostamento si deve scegliere come gestirli. Le strategie utilizzate si possono catalogare in due gruppi e sono da applicare in funzione del tipo di risorsa da considerare.

1. **Strategie di replicazione.** Si suddividono in replicazione statica e replicazione dinamica.

   - *Replicazione statica*: è una strategia utilizzata per risorse che si trovano staticamente in ogni ambiente di esecuzione, per esempio le variabili di sistema. In caso di spostamento il collegamento alla risorsa si sostituisce con uno locale. È importante notare che in questo modo si perde la storia di eventuali interazioni avvenute in precedenza: questo metodo deve quindi essere utilizzato solo per risorse prive di stato o per cui non interessa mantenere la consistenza tra i vari ambienti di esecuzione.

   - *Replicazione dinamica*: solo al momento dello spostamento, si agisce sulla risorsa utilizzata, inserendo nell'ambiente di destinazione la risorsa necessaria. Per ripristinare la consistenza dei dati si sfruttano due diversi criteri.

     - ✓ È possibile realizzare la replicazione dinamica costruendo una *copia* della risorsa originale e creare con essa lo stesso collegamento con cui era legata l'unità di esecuzione nell'ambiente precedente. Questa è una soluzione da utilizzare nel caso in cui la disponibilità

della risorsa stessa debba essere garantita su entrambi gli ambienti di esecuzione.

- ✓ In situazioni duali, è possibile realizzare la replicazione dinamica attraverso lo *spostamento* della risorsa dall'ambiente precedente a quello successivo. È chiaro che questa soluzione è adatta solo qualora l'elemento trasferito non sia più utilizzato da nessun'altra unità di esecuzione dell'ambiente di partenza, e prestando molta attenzione a non lasciare riferimenti pendenti (dangling reference).

2. **Strategie di condivisione.** La locazione fisica delle risorse referenziate non viene modificata: in caso di spostamento si genera un riferimento tra l'unità che ora esegue in un ambiente remoto e la risorsa che è rimasta in quello precedente. Così si creano risorse condivise tra unità che eseguono in ambienti differenti. Quando una unità può referenziare risorse remote, si dice che essa possiede uno *stato distribuito*.

1.5 Problemi di sicurezza nella mobilità

Non appena si introduce il concetto di mobilità del codice in un sistema distribuito, un problema importante da affrontare è quello di fornire una struttura adeguata di sicurezza: infatti, si inserisce in una macchina una entità in grado di eseguire ogni tipo di codice, che può anche avere un comportamento dannoso nei confronti delle risorse locali.

I sistemi che sfruttano la mobilità sono tipicamente sistemi aperti, in cui si deve garantire che i dati siano protetti e questo non è facile: la sicurezza, infatti, rappresenta uno dei limiti maggiori alla diffusione globale di questi sistemi.

Il problema della sicurezza comprende una molteplicità di aspetti differenti.

Innanzi tutto la necessità di fornire *un supporto di comunicazione affidabile*: per spostare del codice su un host remoto, si deve essere certi che il supporto di comunicazione sia sicuro, secondo tre differenti punti di vista: integrità, privacy e paternità.

- Un messaggio (sia questo codice, dati, ecc.) proveniente da una comunicazione di rete, deve essere ricevuto **integro**, cioè coincidente con il messaggio stesso al momento della spedizione. È necessario proteggere il contenuto informativo da alterazioni che potrebbe subire durante il percorso, per errori legati al processo di spedizione, oppure a causa di interventi esterni maligni.
- La **privacy**, o riservatezza, si garantisce escludendo la possibilità che terze parti non coinvolte, possano prendere visione del messaggio trasmesso sul canale di comunicazione.
- Infine, la **paternità** indica in modo oggettivo la provenienza di ogni messaggio, in modo che il mittente non possa negare di essere responsabile di eventuali conseguenze.

Nel caso di mobilità del codice, un altro aspetto della sicurezza è quello della *mutua autenticazione* tra unità in arrivo ed ambiente che dovrà ospitarla.

- Da un lato, *l'ambiente di esecuzione* deve proteggere le risorse locali da eventuali comportamenti dannosi da parte delle unità di esecuzione dei diversi utenti (anche possibili intrusioni in altre unità): per questo motivo si introducono diritti di accesso che vengono controllati prima di concedere autorizzazioni all'uso di risorse.
- D'altro canto, invece, non è affatto facile garantire una protezione duale. Le *unità di esecuzione* in arrivo su siti remoti devono iniziare la loro esecuzione, fornendo al nuovo ambiente

l’accesso al codice e alla rappresentazione run-time, esponendosi così ad attacchi come l’accesso a informazioni private, la modifica di codice e dati, oppure il pagamento sovrastimato di servizi locali, o addirittura la negazione dello svolgimento di un servizio.

Un ulteriore aspetto della sicurezza, riguarda l’*esecuzione sicura*: per proteggersi dall’abuso di risorse normalmente accessibili, un ambiente di esecuzione può scegliere di gestire una zona apposita ove lanciare programmi non fidati, provenienti da siti remoti. Un esempio è costituito dal linguaggio Java: quando si deve mettere in esecuzione un codice di dubbia provenienza o non fidato, gli si assegna un’area circoscritta, la *sand box,* in cui ogni tipo di comportamento evita di avere ripercussioni negative sul sistema. Ai processi che eseguono nella sand box è consentito un accesso limitato all’uso delle risorse (la cpu, un’area prefissata di memoria e un insieme limitato di periferiche, come il mouse e la tastiera) e sono assegnate capacità di esecuzione molto scarse: in questo modo la tutela del sistema è garantita, ma come contropartita si offrono scarse capacità anche a programmi potenzialmente innocui.

In generale, l’introduzione di criteri di sicurezza limita notevolmente le prestazioni dei sistemi, poiché introduce operazioni aggiuntive, non strettamente necessarie all’esecuzione delle applicazioni, ma importanti per le risorse da proteggere. D’altra parte un sistema non sicuro su cui realizzare mobilità del codice è un argomento privo di interesse (a parte il caso di reti completamente fidate), quindi è necessario trovare il giusto compromesso tra *performance* e *protezione*.

1.6 Codice compilato e codice interpretato

Nella mobilità del codice si presenta il problema di come trasferire il codice per eseguirlo sul nodo remoto, superando l'eterogeneità della rete: mettiamo in evidenza la differenza tra codice compilato e codice interpretato sulla base di elementi quali portabilità, prestazioni e sicurezza. Queste sono le caratteristiche più rilevanti degli ambienti di utilizzo della mobilità, cioè le reti di calcolatori locali o geografiche che collegano architetture eterogenee ed aperte. Ricordiamo che un codice è *portabile* se lo si può eseguire su tutti i tipi di macchina disponibili nella rete.

Un codice interpretato può supportare più facilmente la mobilità su un ambiente eterogeneo, perché è sufficiente fornire un interprete per ogni tipo di piattaforma su cui il programma dovrà eseguire. Nel caso invece di codice compilato, la macchina da cui si spedisce deve conoscere l'architettura dell'host remoto, perché in funzione di questa si deve inviare un appropriato codice nativo (cioè un linguaggio specifico dell'architettura presente): se è verificata questa condizione, l'uso di codice compilato offre prestazioni complessive molto migliori dell'uso di codice interpretato. In caso contrario, l'unica soluzione è trasmettere l'insieme di tutti i codici nativi disponibili. Con un approccio di tipo interpretato, l'ambiente di esecuzione può anche effettuare un controllo, a tempo di caricamento o di esecuzione, sul codice sorgente, per assicurare l'esecuzione delle sole istruzioni consentite, introducendo un livello aggiuntivo di sicurezza (vedere il paragrafo precedente).

Una soluzione comunemente adottata è quella di usare un approccio misto, per sfruttare i vantaggi di entrambi i metodi: la velocità del codice compilato e la portabilità di quello interpretato.

Per esempio, in *Java*, il codice scritto dal programmatore attraversa una prima fase in cui viene compilato e trasformato in un linguaggio intermedio, detto "Byte Code". Questo è il formato con

cui il programma dell'utente si sposta sulla rete per raggiungere i nodi remoti e ha la caratteristica di essere portabile perché, per eseguirlo, è sufficiente che nella macchina di destinazione sia presente l'interprete Java, ormai disponibile per ogni tipo di architettura. Quando il programma, in formato "byte code" raggiunge il sito obiettivo, attraversa una seconda fase in cui deve superare il controllo del "verificatore", che si occupa di eliminare le istruzioni illegali: questo controllo offre buone garanzie di sicurezza. A questo punto, nella terza ed ultima fase, l'interprete Java presente sulla macchina remota mette in esecuzione il programma dell'utente. Con questa tecnica Java media la velocità di esecuzione caratteristica della compilazione e la portabilità tipica dei linguaggi interpretati, inserendo anche un livello aggiuntivo di sicurezza.

1.7 Paradigmi di progettazione

Cominciamo con l'introdurre una definizione: un paradigma di progettazione è la specifica configurazione dei componenti e delle loro interazioni all'interno di un sistema, cioè la configurazione che il progettista usa per definire l'architettura di una applicazione [FugPV98].

La progettazione di applicazioni software richiede la scelta di un adeguato linguaggio di programmazione; quando si parla di mobilità del codice e l'ambiente di esecuzione è distribuito su larga scala, è anche importante considerare da subito quale paradigma di progettazione utilizzare. Preferire un paradigma piuttosto che un altro, è assolutamente indipendente dal linguaggio utilizzato e dalla architettura su cui l'applicazione sarà eseguita, mentre può influenzare notevolmente le prestazioni finali o provocare conseguenze non irrilevanti in termini di affidabilità.

I sistemi che usano la mobilità consentono ai componenti di modificare dinamicamente la loro posizione tra le macchine della rete: i concetti fondamentali quindi diventano la locazione e la distribuzione delle risorse nel sistema; per esempio, l'interazione tra due componenti che risiedono su host remoti è molto più onerosa del caso in cui la stessa interazione avvenga localmente.

Si fornisce di seguito, una classificazione concettuale dei paradigmi di progettazione basati sulla mobilità del codice, secondo i criteri descritti in [CarPV97]: una particolare attenzione deve essere rivolta alla locazione delle risorse, ai componenti responsabili dell'esecuzione del codice e a dove realmente il codice viene eseguito.

1.7.1 Client – Server

È il classico modello di programmazione distribuita, senza mobilità di codice.

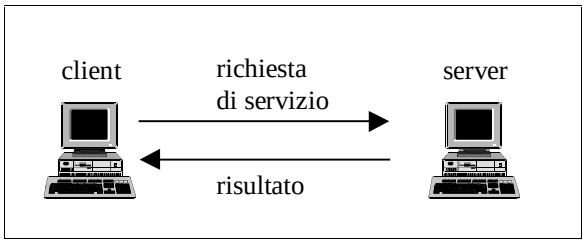


Figura 1-3 Client - Server. Il cliente richiede un servizio al server, disinteressandosi di come è svolto.

Le operazioni sono suddivise tra due punti della rete: un cliente che richiede un servizio ed un servitore che lo esegue (Figura 1-3). Deve essere garantita la connessione tra le macchine in cui risiedono le due entità, per consentire la comunicazione, che avviene seguendo un protocollo noto ad entrambi. In una tipica situazione, il cliente richiede un servizio ad un servitore che risiede

su una macchina più potente in cui sia possibile eseguire alcuni compiti standard per cui sia necessaria una maggiore potenza di calcolo.

Un generico esempio è rappresentato dal caso in cui sulla macchina del servitore risieda una risorsa a cui i clienti possono accedere solo con le funzionalità già implementate e messe a disposizione dal processo servitore: questa situazione esprime la scarsa elasticità del servitore a realizzare servizi lievemente diversi da quelli già presenti, obbligando all'introduzione di codice aggiuntivo, che aumenta le dimensioni del servitore, senza aggiungerne flessibilità.

Storicamente tutte le applicazioni distribuite erano sviluppate secondo il paradigma cliente-servitore. La Remote Procedure Call (RPC), che corrisponde all'invocazione di una procedura remota, è un esempio particolare: il processo cliente richiede a un servitore remoto l'esecuzione di un servizio attraverso l'invocazione di una sua procedura specifica, di cui conosce l'interfaccia e a cui può accedere direttamente.

I limiti di questo paradigma sono forniti dalla scarsa flessibilità ed estensibilità del servitore e dalla forte dipendenza da connessioni di rete affidabili. Nel caso in cui l'interazione tra cliente e servitore sia molto frequente, o l'esecuzione di un servizio richieda una prolungata fase di scambio di messaggi, questo modello risulta limitativo, soprattutto se l'applicazione considerata è distribuita su larga scala o la connessione di rete è lenta o di bassa qualità oppure se la rete è congestionata da un traffico eccessivo.

1.7.2 Remote Evaluation

Nel modello precedente, il cliente può richiedere un servizio, solo se appartenente a quelli messi a disposizione dal server. Infatti non

è possibile eseguire un codice arbitrario, poiché ad ogni servizio è associato un programma prestabilito, con cui non si può interagire.

Il paradigma Remote Evaluation permette ad un programma di sfruttare una entità remota, fornendo il codice da eseguire al sito in cui essa risiede (Figura 1-4). Così si rende più flessibile l'utilizzo delle risorse, lasciando al programma la scelta di come sfruttarle, attraverso una esecuzione più veloce, perché svolta localmente, ed eventualmente bloccandosi in attesa di un risultato.

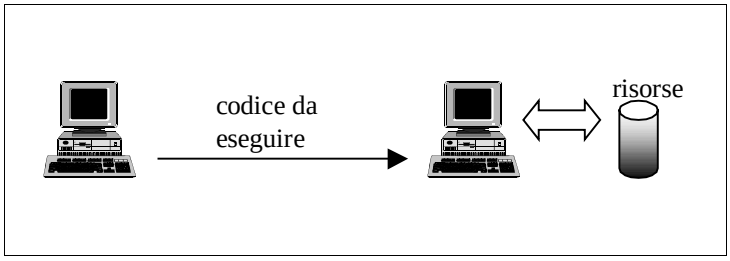


Figura 1-4 Remote Evaluation: il codice è inviato all'host remoto in cui sono presenti le risorse necessarie.

Se si presenta la necessità di creare un nuovo servizio, o di modificarne uno esistente, non si deve riorganizzare il servitore (cioè la macchina in cui risiede la risorsa), impedendo ad altri clienti di sfruttare le sue funzionalità durante la fase di riorganizzazione, ma è sufficiente realizzare un frammento di codice che soddisfi le nuove specifiche.

Il paradigma Remote Evaluation permette di realizzare un *servitore elastico*, cioè aperto e disponibile ad eseguire ogni tipo di codice, *dinamico*, poiché interagisce con le sue risorse in funzione dei cambiamenti e delle nuove esigenze dei clienti che gli si rivolgono, ed anche *globale*, perché può eseguire tutti i programmi presenti nel sistema.

Un semplice esempio di questo modello è la richiesta di stampa di un file postscript su una periferica remota: il documento in

questione rappresenta il codice da eseguire, sempre diverso ad ogni richiesta.

1.7.3 Code on Demand

Anche questo paradigma ha come obiettivo lo svolgimento di un servizio, sfruttando la mobilità del codice, ma a differenza di prima, l'esecuzione avviene all'interno della macchina che fa la richiesta (Figura 1-5).

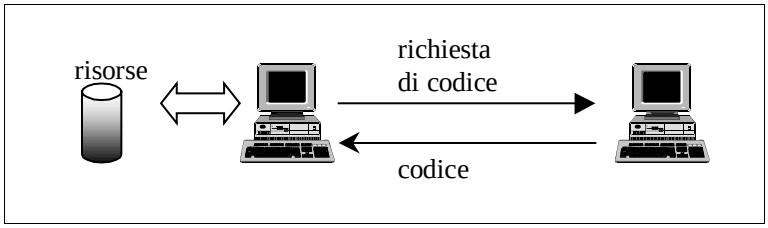


Figura 1-5 Code on Demand: la risorsa da utilizzare è disponibile, e quando è necessario, si richiede il codice.

La situazione iniziale è duale rispetto alla precedente: le risorse necessarie sono già presenti localmente, e manca la conoscenza di come sfruttarle. L'host remoto spedisce il codice in conseguenza di una esplicita richiesta, e questo è eseguito localmente, sulla macchina che ha iniziato l'interazione. Così, le entità in esecuzione si rendono dinamicamente estensibili, poiché possono ampliare la loro capacità di esecuzione, avvalendosi di qualunque codice disponibile sulla rete. Anche in questo caso, si realizza un *entità dinamica flessibile e globale*, che esegue i programmi dinamicamente disponibili nell'intero sistema.

Un'applicazione molto diffusa che sfrutta questo paradigma, è utilizzata ogni volta che un utente richiede al proprio browser di visualizzare una pagina web in cui è inserita una applet: il server

riceve la richiesta e spedisce, alla macchina dell'utente, il codice Java necessario da applicare alle risorse locali per visualizzare gli effetti dell'applet.

1.7.4 Agenti Mobili

Gli agenti mobili sono unità di esecuzione (EU) che agiscono in nome di un utente e che sono in grado di muoversi in modo autonomo all'interno di una rete.

Questo paradigma è molto differente dai precedenti: in Remote Evaluation e Code on Demand l'attenzione è rivolta allo spostamento di codice tra componenti in esecuzione; ora invece è il componente stesso che si sposta, trasportando anche i suoi dati, il suo codice e il suo stato di esecuzione (Figura 1-6).

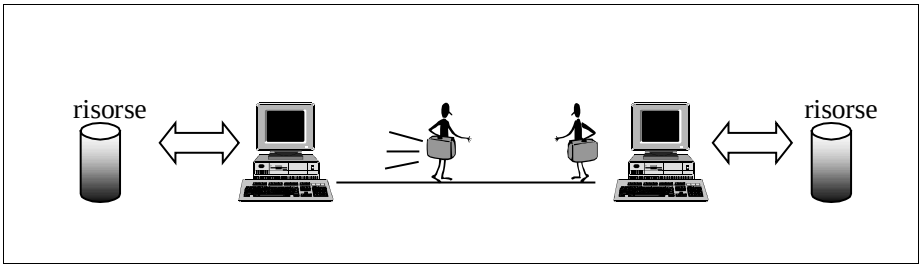


Figura 1-6 Agenti Mobili: sono programmi autonomi, che possono spostarsi nella rete, e continuare l'esecuzione sulla macchina in cui risiede la risorsa necessaria.

Questo paradigma, che sarà affrontato in modo più approfondito nel seguito (capitolo 2), riduce al minimo lo scambio di messaggi sulla rete, per evitare eventuali congestioni, e si avvantaggia della velocità delle esecuzioni locali.

Per capire questo paradigma, consideriamo un programma che a un certo punto della sua esecuzione, debba interagire con una

risorsa remota: questo, insieme allo stato e ai dati, che possono comprendere anche risultati intermedi raggiunti per esempio sfruttando elementi della macchina che lo ospitava, viene spostato nel sito in cui risiede la risorsa stessa e svolge con essa un'interazione locale.

Come si è potuto notare, nei paradigmi che utilizzano la mobilità del codice, diventano fondamentali i termini di locazione delle risorse, e di posizione relativa rispetto ai programmi che le devono sfruttare: infatti la capacità di spostamento consente al codice di avvicinarsi dinamicamente agli elementi con cui deve interagire.

1.8 Mobilità del codice vs. client-server

Il paradigma client-server rappresenta il modello di progettazione tradizionale, che prevede la richiesta del servizio da una parte e l'esecuzione di un codice prestabilito dall'altra. Con l'introduzione del concetto di mobilità si sfrutta un aspetto differente della progettazione, cioè la capacità di modificare dinamicamente il collegamento tra un frammento di codice e l'ambiente in cui sarà eseguito.

Nel caso in cui un cliente presenti una richiesta nuova o di poco differente dalle solite, se nessun servitore è in grado di soddisfarla, sarebbe necessario aggiungere questo ulteriore servizio a quelli già disponibili, aumentando la dimensione e la complessità del server, senza modificarne la flessibilità. Infatti, la mancanza di **flessibilità** è un grosso limite del paradigma client-server: con la mobilità questo problema è superato, offrendo la capacità di scegliere in modo dinamico quale tipo di codice eseguire su ogni risorsa remota, agendo direttamente sull'host in cui essa risiede. Questo permette di svolgere *funzioni personalizzate, sempre diverse, che*

corrispondono a tutte le necessità dei richiedenti e che si adattano alle trasformazioni dinamiche del sistema.

Con le tecnologie di mobilità del codice si introduce anche il concetto di **autonomia dei componenti**. Questa proprietà è utile quando l'ambiente applicativo si estende su piattaforme eterogenee che presentano collegamenti a volte inaffidabili o lenti. Con il paradigma cliente-servitore l'unica soluzione per ridurre al minimo il traffico di rete è agire sulla granularità dei servizi: questo implica svolgere un maggior numero di operazioni in conseguenza di un'unica richiesta, in modo da appesantire il meno possibile le connessioni di rete. Non è detto che si riesca sempre a riunire la sequenza di più operazioni sotto una unica richiesta, capace ancora di soddisfare i requisiti del cliente, e comunque questa soluzione accentua ancora di più l'aspetto delicato della flessibilità del servitore. La soluzione prevista sfruttando la mobilità è di consentire caratteristiche di autonomia alle applicazioni: piuttosto che attraversare più volte interconnessioni insicure o lente, l'intero codice si sposta laddove risiede la risorsa con cui interagire, porta a compimento l'esecuzione all'interno di una rete con migliori prestazioni ed attraversa nuovamente la zona meno affidabile solo se è necessario restituire un risultato all'ambiente iniziale.

Un benefico effetto collaterale della autonomia dei componenti, è una migliore **tolleranza ai guasti** espressa in termini di maggiore facilità nel ripristino di uno stato consistente. Infatti durante l'esecuzione di una applicazione distribuita, il client effettua frequenti interazioni con il server, ottenendo risultati intermedi indispensabili per proseguire. Se durante l'esecuzione di un servizio, si verifica un guasto sulla macchina del servitore, è molto difficile gestire il ripristino dei dati: è necessario sapere se il server ha già iniziato ad eseguire alcune operazioni e in caso affermativo è importante verificarne la consistenza, che può essere valutata solo se si interagisce con un servitore con stato. Questo è il tipico problema che si presenta per ogni applicazione distribuita tra più

ambienti di esecuzione, e in caso di mal funzionamento è necessaria una complicata operazione di recovery per ripristinare la consistenza tra gli ambienti coinvolti [CouDK94]. In caso di mobilità, è il frammento di codice stesso a spostarsi dove deve avvenire l'esecuzione e raggruppa direttamente al suo interno lo stato delle interazioni che, nel paradigma client-server, è distribuito tra più host. In caso di mal funzionamento, nel componente autonomo il recovery può essere eseguito localmente e senza la conoscenza dello stato globale. I problemi sono confinati solo nelle fasi di comunicazione iniziale del codice e di restituzione di un eventuale risultato.

L'aumento delle dimensioni delle reti e delle prestazioni che forniscono, conducono ad un importante fenomeno, che assegna alle reti la caratteristica della "onnipresenza": con questo termine si intende la capacità dei nodi di usufruire dei collegamenti di rete, anche da locazioni differenti. Infatti, lo sviluppo della tecnologia "senza fili" (*wireless*) svincola le macchine della rete dall'obbligo di essere collocati in una locazione fissa, ed introduce il nuovo concetto di "**mobile computing"**: gli utenti diventano mobili e sono capaci di spostarsi in locazioni fisiche remote, insieme alle macchine su cui lavorano e con cui interagiscono, connesse alla rete solo con collegamenti senza fili [FugPV98].

Non è detto che la mobilità rappresenti sempre la soluzione ottima di progettazione, soprattutto quando si richiede di svolgere un servizio standard, già noto al server, per cui la spedizione del codice risulterebbe solo un ulteriore appesantimento.

1.9 Domini applicativi

Abbiamo delineato quali siano i vantaggi introdotti con l'uso di codice mobile, utilizzabile attraverso paradigmi di progettazione che guidano il programmatore nella realizzazione di applicazioni

distribuite. In realtà non sono ancora presenti molti progetti reali implementati con questa tecnologia, poiché è ancora molto giovane, tuttavia esistono numerosi domini applicativi in cui è possibile impiegarla con profitto.

Il primo settore in cui è possibile sfruttare i vantaggi della mobilità del codice, è quello della ***ricerca di informazioni*** tra siti remoti. Un database racchiude spesso una quantità estesa di dati, tra i quali possono essere compresi quelli con cui si desidera interagire, che tipicamente costituiscono solo una minima parte dell'intero insieme. Piuttosto che affidarsi a metodi standard che selezionano solo alcuni dati da trasmettere, eventualmente inutili, richiedendo quindi numerosi tentativi, è sicuramente più conveniente inviare solo l'algoritmo di ricerca sul sito contenente il database e rioccupare i canali di comunicazione solo per la spedizione del risultato certamente corretto. In questo modo si evita di far passare grosse quantità di dati sui mezzi trasmissivi, ricordando che il traffico elevato rischia di congestionare la rete.

Nel caso in cui le informazioni siano distribuite su più macchine, con gli strumenti tradizionali il problema del trasferimento aumenta in proporzione, scaricando anche interi archivi che potrebbero risultare inutili.

Se invece la ricerca è eseguita da un'entità autonoma, è solo l'algoritmo di ricerca a navigare tra i vari host, e potrebbe anche scegliere dinamicamente su quali siti ricercare le informazioni, in funzione per esempio di indicazioni raccolte nel corso dei suoi spostamenti. La ricerca di informazioni svolta su un ambiente distribuito può trovare diversi campi applicativi: si pensi per esempio, alla consultazione degli orari di tutte le compagnie aeree o alla ricerca di particolari configurazioni software nelle macchine di una rete.

Un altro settore in cui la mobilità del codice risulta molto vantaggiosa è quello dell'***installazione ed aggiornamento del software***. Con gli strumenti tradizionali, è necessario che un operatore agisca personalmente su tutte le macchine interessate per inserire nuove funzionalità o aggiungere upgrade di applicazioni già presenti, che magari non verranno neanche utilizzate.

È chiaro che in questo settore l'uso della mobilità del codice rappresenta un'ottima alternativa alla presenza fisica di una persona presso tutte le postazioni da sistemare. Una entità autonoma potrebbe essere in grado di navigare sulla rete e di decidere in funzione della situazione corrente della macchina su cui si trova e del suo contesto di località se installare o aggiornare una applicazione software.

È possibile migliorare ancora la gestione dell'aggiornamento predisponendo una macchina a cui le entità mobili si rivolgano per scaricare l'upgrade solo in caso di reale necessità, in modo da evitare l'aggiornamento se tale upgrade non risultasse utile.

La ***gestione remota*** di dispositivi o in generale della configurazione di una rete, rappresenta un altro dominio applicativo in cui sfruttare i vantaggi della mobilità del codice. Questo settore racchiude in realtà aspetti molteplici di gestione, come il controllo di processi industriali, il network management, ecc.. Con gli strumenti tradizionali ogni tipo di gestione remota era realizzata attraverso la consultazione periodica dello stato della risorsa attraverso funzioni predefinite. Con l'utilizzo della mobilità, è possibile raggiungere l'elemento da gestire nell'host su cui risiede; dopo una verifica locale dello stato si sceglie dinamicamente se e quali operazioni sia necessario intraprendere. Una esposizione più esaustiva sull'argomento sarà fornita nel seguito (paragrafo 2.3).

Il ***commercio elettronico*****,** cioè l'acquisto di prodotti o servizi tramite una rete, come Internet, è un settore in forte espansione, che richiede flessibilità e dinamicità delle entità che lo gestiscono: infatti, come nel commercio tradizionale, l'abilità di un buon venditore risiede nel saper presentare gli aspetti che dovrebbero essere i più interessanti per il tipo di interlocutore a cui si rivolge. Queste sono le caratteristiche della mobilità: infatti, essa permette di ricercare l'occasione più promettente, di interagire con il sito che espone le offerte migliori e consente ai componenti mobili di accedere alle informazioni più aggiornate (si pensi alla velocità con cui si modificano le quotazioni della borsa).

Lo sviluppo delle tecnologie senza fili, liberano i nodi del sistema dal vincolo di rimanere collocati in una locazione fissa, stabilita senza dinamicità al momento dell'attivazione, e consentono l'introduzione di un nuovo concetto: il ***mobile computing***. In questa nuova situazione, l'utente mobile può decidere di spostarsi, affiancato dalla sua macchina (si pensi per esempio ai computer portatili) in locazioni fisiche differenti, in cui è ancora capace di connettersi alla rete, attraverso collegamenti senza fili. In questo modo, la rete diventa "onnipresente", ed è possibile sfruttarla, collegandosi ad essa indipendentemente dalla locazione fisica dell'utente [FugPV98].

Cap.2

Agenti Mobili

2.1 Agenti Mobili

Possiamo definire un **agente** come una unità di esecuzione (EU, cioè comprende oltre al codice, lo stato di esecuzione e lo spazio dei dati) che agisce autonomamente per conto di una persona o di una organizzazione [CheHK95]. Il termine “agente” deriva proprio dalla capacità di agire in nome dell’utente, per raggiungere l’obiettivo finale, scegliendo in modo autonomo quali operazioni intraprendere. Ogni agente ha il proprio flusso di esecuzione, in modo da eseguire su propria iniziativa i compiti assegnatigli.

La maggior parte degli agenti è realizzata in un linguaggio interpretato, per esempio Java, per sfruttarne la portabilità e con il vantaggio di poter eventualmente inserire un livello aggiuntivo di sicurezza (vedere paragrafo 1.6).

Un agente è *stazionario* se agisce esclusivamente sul nodo su cui ha iniziato la sua esecuzione; quando ha bisogno di informazioni remote o deve interagire con risorse non presenti localmente, può usare meccanismi di comunicazione come la Remote Procedure Call (RPC). Un agente, invece, è **mobile** se ha l’abilità di trasportare se stesso all’interno di una rete da una macchina ad un’altra, senza essere vincolato a quella in cui ha iniziato la sua esecuzione : si sposta nel sistema in cui si trova l’oggetto con cui vuole interagire, per trarre il beneficio di eseguire operazioni locali.

In realtà, nel campo della “Intelligenza Artificiale” si definisce anche un altro tipo di agenti, gli agenti intelligenti, che sono caratterizzati dalla capacità di inferire nuova conoscenza durante il loro tempo di vita. Gli agenti intelligenti sono tipicamente agenti stazionari, perché, a causa della loro natura, rischiano di avere dimensioni elevate; gli agenti mobili, invece, essendo trasferiti frequentemente devono essere snelli (“ligth weigth”), per evitare di subire spostamenti lenti, che congestionano la rete. Spesso i termini agenti mobili e agenti intelligenti sono usati come sinonimi e ci si riferisce ad essi in generale col termine agenti.

Le caratteristiche computazionali più importanti degli agenti possono essere riassunte in questi termini:

- Autonomi: ogni agente ha la capacità di scegliere, in funzione delle sue informazioni, cosa fare, dove andare e quando spostarsi.
- Asincroni: tutti gli agenti hanno un proprio flusso di esecuzione che possono eseguire in modo asincrono.
- Interazioni locali: i loro spostamenti sono finalizzati ad eseguire interazioni locali, per ridurne il costo, da effettuare con altri agenti mobili o in generale con oggetti stazionari che risiedono localmente.
- Efficienti: l’uso degli agenti consente di ridurre il traffico di rete, soprattutto nel realizzare interazioni remote complicate.
- Indipendenti dalla rete: gli agenti possono eseguire i loro compiti anche se la connessione di rete è al momento disattiva, interagendo con le risorse locali. Qualora sia necessario uno spostamento, si attende che il collegamento con la macchina remota si ripristini.
- Esecuzione parallela: la distribuzione degli agenti su più macchine permette di eseguire in parallelo i compiti intermedi che conducono al risultato finale, riducendo i tempi di attesa.
- Object-passing: la tecnologia più comoda per realizzare un agente mobile è quella object-oriented (paragrafo 1.3), in modo

che quando deve essere spostato, si trasferisce l'intero oggetto, che comprende i dati, il codice e lo stato di esecuzione dell'agente.

Queste caratteristiche permettono di ottimizzare ed ampliare i servizi disponibili sulla rete, riducendone il traffico e quindi il rischio di congestione.

Le caratteristiche degli agenti mobili che richiedono maggiori dettagli sono rappresentate dalla capacità di migrare su macchine remote, dalla possibilità di comunicare con altri agenti e dalla garanzia di sicurezza che devono fornire e che devono ricevere dall'ambiente di esecuzione. Analizziamo in dettaglio queste caratteristiche per avere una visione più completa delle proprietà generali di un agente mobile.

2.1.1 Migrazione

La caratteristica degli agenti mobili è di agire in modo autonomo, avendo conoscenza degli interessi dell'utente e cercando di soddisfarli, attraverso lo spostamento tra le macchine della rete. La migrazione è la conseguenza dell'invocazione di un'istruzione, che provoca il trasferimento del codice dell'agente e del suo stato di esecuzione. L'attuazione del comando di spostamento si sviluppa seguendo queste fasi:

- tutti i processi attivi dell'agente sono sospesi e insieme allo stato dell'agente (cioè i dati, il codice e lo stato di esecuzione) sono elaborati per essere incapsulati in un messaggio, espresso in una forma indipendente dalla macchina (per es. Abstract Syntax Notation);

- il messaggio così creato, può essere indirizzato esplicitamente al destinatario finale, o ad un nodo intermedio con funzioni di routing;
- quando arriva nel sistema di destinazione, il messaggio deve raggiungere l'ambiente di esecuzione a cui era destinato tra quelli presenti, e poi qui l'agente viene ricomposto;
- attraverso il suo stato di esecuzione e il valore dei suoi attributi, l'agente può riprendere la sua esecuzione dall'istruzione successiva a quella della migrazione .

Dopo la migrazione, l'agente può interagire con le funzioni applicative del nuovo ambiente (servitore) e procedendo nella sua esecuzione potrebbe intraprendere ancora una operazione di migrazione in modo analogo a quanto appena descritto.

Quando si dice che raggiunto l'ambiente di destinazione l'esecuzione dell'agente riprende dall'istruzione successiva a quella della migrazione, si intende semplicemente che il flusso di esecuzione si riattiva in accordo con lo stato e gli attributi che lo caratterizzano. È quindi una affermazione del tutto generica che dovrebbe essere approfondita in considerazione del caso specifico che il linguaggio considerato supporti un tipo di mobilità forte o debole (vedere paragrafo 1.4).

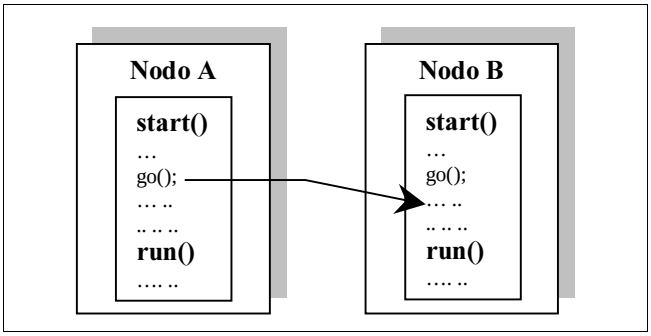


Figura 2-1 Mobilità forte

Infatti nel caso in cui la ripresa del flusso di esecuzione coincida esattamente con l'istruzione successiva a quella dello spostamento, significa che la mobilità supportata è di tipo forte (Figura 2-1).

Se invece la ripresa necessita di una fase di preambolo, che richiede la partenza da un punto prestabilito per consentire di ripristinare la consistenza, allora il tipo di mobilità supportata è debole (Figura 2-2).

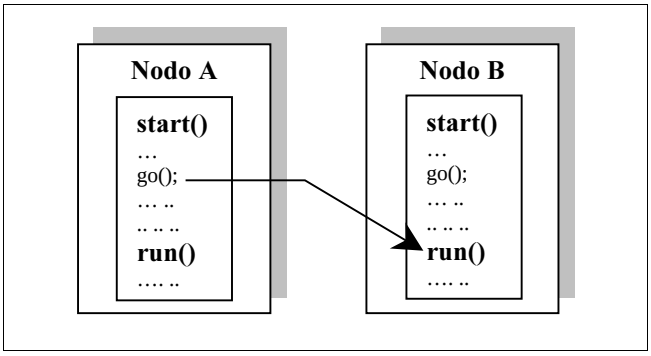


Figura 2-2 Mobilità debole

2.1.2 Comunicazione tra agenti

In un sistema ad agenti, è necessario definire il modo in cui gli agenti possono interagire tra loro per raggiungere il loro obiettivo. Gli stili di comunicazione possono essere suddivisi in due categorie, in funzione del tipo di interazione, stretta o lasca, che offrono:

- **Interazione stretta**: la comunicazione che si basa su una interazione stretta tra due agenti, richiede una loro conoscenza esplicita. Si ottiene attraverso la condivisione di oggetto o attraverso la diretta invocazione di un metodo dell'agente remoto (per esempio, in *Remote Method Invocation, RMI* [RMI98], il chiamante deve conoscere staticamente l'interfaccia

del metodo remoto da invocare [Java]). Questa è una comunicazione di alto livello e richiede una conoscenza specifica degli oggetti o degli agenti remoti con cui interagire: per inserire dinamicità, deve essere presente nel sistema un protocollo di acquisizione di informazioni remote, che si mantenga aggiornato sulle ultime modifiche e su eventuali spostamenti, senza ricadere nella visione classica dei sistemi distribuiti in cui si offre uno strato di trasparenza alla locazione. La realizzazione di interazioni strette è abbastanza costosa ed è quindi da utilizzare solo in casi realmente necessari.

- **Interazione lasca**: la comunicazione che si basa su una interazione lasca, non necessita di una conoscenza rigida tra gli agenti interessati. Un primo meccanismo per realizzare una interazione lasca, è lo **scambio di messaggi**: è ancora necessaria una forma di conoscenza tra i due interlocutori, ma è meno rigida, e può dipendere per esempio dal fatto di essere nati nello stesso ambiente di esecuzione o di discendere dalla stessa famiglia.

Un altro metodo per realizzare interazione lasca, è costituito dalla **comunicazione anonima**: quando due agenti devono comunicare, ma non hanno alcuna conoscenza reciproca, né hanno caratteristiche tali da potersi scambiare messaggi, si può realizzare un supporto che permetta di conoscere facilmente il nome di altri agenti, oppure è possibile realizzare astrazioni come la “Blackboard” oppure uno “spazio delle tuple” [CaGe89].

2.1.3 Sicurezza

Lo spostamento di un programma mette in evidenza immediatamente il delicato problema della sicurezza, che si estende su diversi aspetti.

Con il termine *autenticazione si intende l'operazione necessaria per riconoscere senza ambiguità l'entità in esame* [Coc98]: quando si intende condividere risorse proprie, offrire un servizio o, in generale, creare una forma di interazione con altre entità (che possono essere agenti, sistemi, ecc.) è bene conoscere l'identità di chi svolge l'interazione dal lato opposto (a meno che non si intendano realizzare interazioni anonime).

Con il termine *autorizzazione si intende l'operazione necessaria per controllare se l'interlocutore ha il diritto di svolgere l'azione richiesta*: compiuta la prima fase di autenticazione, conosciamo l'identità del nostro interlocutore e possiamo verificare se possiede l'autorizzazione ad eseguire l'azione richiesta.

Quando un agente si sposta su una nuova macchina, questa deve effettuare l'autenticazione dell'utente, che è il mittente dell'agente, e viceversa l'agente deve autenticare l'ambiente di esecuzione in cui si inserisce: ***mutua autenticazione*** (Figura 2-3).

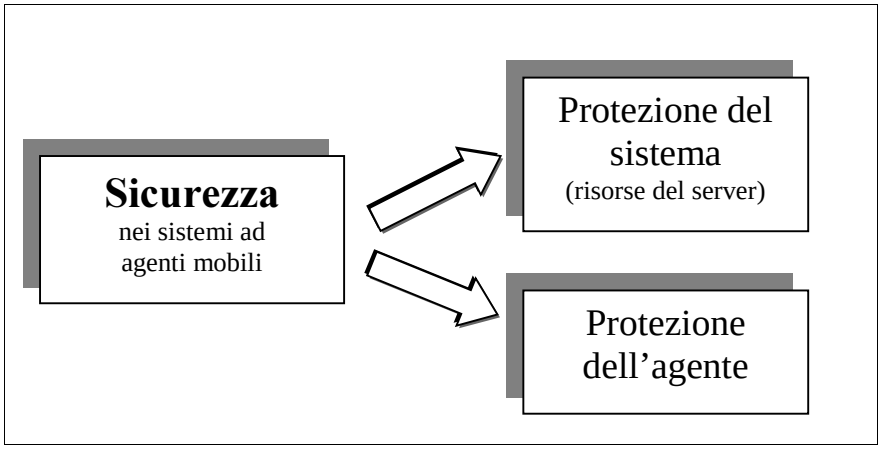


Figura 2-3 I due aspetti principali del problema sicurezza nei sistemi ad agenti mobili.

L'autenticazione dell'agente in arrivo è basato su informazioni di preambolo trasmesse insieme all'agente stesso, informazione che non forniscono invece nessuna indicazione su ciò che l'agente ha

intenzione di fare. Il server deve anche determinare se le intenzioni dell'agente sono corrette, controllando per esempio le risorse a cui vuole accedere, e negando il servizio qualora lo scopo sia danneggiare, in un qualunque modo, il sistema [Coc98]. Il server deve effettuare un ulteriore controllo sulla capacità e sulla volontà dell'agente di pagare per la funzione richiesta ; in egual misura l'utente deve assicurarsi che il servitore fornisca la prestazione per cui è stato eventualmente retribuito.

Cercare di rispettare tutti i criteri necessari alla realizzazione di un sistema ad agenti mobili sicuro, rischia di appesantire molto l'esecuzione, abbassando notevolmente le performance del sistema. È chiaro quindi che l'obiettivo sia raggiungere un adeguato compromesso tra efficienza e sicurezza, senza rinunciare a nessuna delle due componenti, in funzione del tipo di sistema che si vuole ottenere. Per esempio, in una piccola rete locale su cui agiscono solo utenti fidati, si può preferire di non rinunciare a prestazioni elevate, evitando ogni livello di sicurezza ("trust-net"). Parlando di agenti mobili, non si può non considerare il loro obiettivo di raggiungere larga diffusione su reti aperte e scalabili; in questo contesto, diventano indispensabili meccanismi e politiche che garantiscano un adeguato livello di sicurezza.

La difficoltà di garantire questo adeguato livello di sicurezza è un grosso limite allo sviluppo commerciale dei sistemi ad agenti mobili: su questo campo si evidenziano sforzi notevoli [KauPS94] per ottenere sempre migliori prestazioni.

2.2 Java: un linguaggio adatto per la mobilità

Java è un linguaggio object-oriented, progettato con lo scopo di fornire un prodotto di piccole dimensioni, semplice, facile da imparare, portabile su diverse piattaforme e adatto ad ogni dominio applicativo (general-purpose).

Java è un linguaggio, sintatticamente simile al C++, da cui derivano infatti parte della sintassi e della struttura ad oggetti, ma nonostante la somiglianza, le parti più complesse del linguaggio C++ sono state escluse, in modo da mantenere la semplicità, senza sacrificare le potenzialità. Infatti in Java, a differenza del C++, non è consentito l'uso dei puntatori a livello utente, ed è presente una gestione automatica della memoria: in C++ è compito del programmatore deallocare lo spazio di memoria nel momento in cui si elimina un puntatore, mentre in Java questo avviene automaticamente, attraverso l'introduzione di uno strumento detto "*garbage collector*" che periodicamente libera dalla memoria gli elementi non referenziati.

Nella programmazione ad oggetti, il meccanismo dell'*ereditarietà* consente di specificare nella definizione di una classe solo le differenze rispetto ad un'altra già esistente, considerando "ereditate" tutte le caratteristiche non espressamente citate. Legato a questo aspetto, nel C++ esiste un meccanismo non semplice da gestire, che è l'eredità multipla, cioè la possibilità per una classe di ereditare le caratteristiche da più di una classe già esistente. Per semplificarne la gestione, in Java questo strumento è disponibile solo per le *interfacce* che rappresentano le funzionalità disponibili in una classe senza la descrizione di come siano realizzate.

Limitiamo a questi pochi esempi le differenze a livello di programmazione tra Java e C++ (un confronto esaustivo è disponibile in [Nie98]) e soffermiamoci invece sulla caratteristica che rende *Java un linguaggio particolarmente adatto per la realizzazione di applicazioni distribuite, avendo la possibilità di superare l'eterogeneità intrinseca dei sistemi aperti*. La realizzazione di una applicazione in C++, o in qualunque altro linguaggio, prevede la fase di scrittura del codice e di compilazione: il compilatore traduce il programma in un linguaggio comprensibile alla macchina su cui si lavora (e a tutte

quelle compatibili), ma per eseguirlo su una architettura diversa si deve compilare nuovamente il codice sorgente con un compilatore diverso dal precedente e adatto al nuovo sistema.

In Java, invece, il codice sviluppato dal programmatore passa attraverso due componenti: un compilatore e un interprete. Il compilatore Java trasforma il codice sorgente (invece che in linguaggio macchina) in una forma intermedia e indipendente dalla piattaforma, il cosiddetto **byte code**; per riuscire ad eseguire il programma, realizzato dall'utente, il byte code deve essere interpretato da una macchina virtuale stack-based, detta *Java Virtual Machine* (*JVM*). L'introduzione dell'innovativo codice intermedio indipendente dalla piattaforma, permette a Java di essere un linguaggio altamente portabile, destinato ad essere utilizzato con successo in tutti i sistemi con architetture eterogenee, che sono l'ambiente di sviluppo naturale per le applicazioni ad agenti mobili.

Un fattore chiave del successo di Java è la totale **integrazione con la tecnologia WWW:** i browser web includono nelle loro estensioni la JVM e questo permette sia di realizzare con estrema facilità pagine interattive, animate con effetti speciali, da parte di programmatori principianti, sia progetti complessi per utenti più esperti. Così il linguaggio Java è entrato nel mondo informatico, sia a livello professionistico che dilettantistico, affermandosi come leader in tutte le applicazioni distribuite.

Vediamo in dettaglio le caratteristiche che offre questo linguaggio e verifichiamo se sono adatte per la realizzazione di un sistema ad agenti mobili.

Un primo vantaggio dell'utilizzo del byte code è l'**elevata portabilità** di Java, come di ogni altro linguaggio interpretato, che può eseguire in tutte le piattaforme per cui esista una JVM. Data la popolarità che ha acquistato in questi anni, si può dire che praticamente tutti i tipi di macchina forniscano un interprete Java, rendendo veritiero lo slogan che Sun usa a proposito di Java: "write

once, run anywhere". L'ambiente di esecuzione del paradigma ad agenti mobili è un sistema distribuito in cui convivono diversi tipi di macchine: usare Java come strumento di programmazione offre il beneficio di poter far migrare gli agenti in tutti i siti remoti in cui sia presente una JVM, e questo implica una elevata portabilità. È giusto sottolineare che i linguaggi interpretati offrono, come svantaggio, una diminuzione delle prestazioni, rispetto ai linguaggi compilati; per ottenere le stesse prestazioni, Java mette a disposizione un compilatore di byte code, detto JIT (Just In Time), per sfruttare il linguaggio macchina nativo [JAVA].

Il secondo vantaggio deriva dalla possibilità di svolgere un **processo di verifica** sul codice prima dell'esecuzione, con il quale si eliminano le operazioni che potrebbero scatenare un overflow dello stack, e si controlla che i metodi degli oggetti referenziati siano invocati con il corretto numero e tipo di argomenti. Questa verifica è agevolata dal fatto che in Java non è prevista l'esistenza di tipi di dato l'uso di strumenti, come i puntatori, che possono condurre all'accesso indiretto di aree non consentite.

Se durante l'esecuzione di un programma, si presenta un riferimento al nome non risolto di una classe, la JVM invoca a run-time il **class-loader**, che è un meccanismo di programmazione che ricerca e carica dinamicamente (*dynamic link*) le classi necessarie. Il class loader prima controlla che il frammento di codice non sia una classe di sistema, poi verifica che non sia già in memoria, altrimenti cerca di recuperarla da quale sito remoto; a questo punto il codice scaricato comincia la sua esecuzione [Coc98]. L'aspetto interessante è che Java fornisce al programmatore la possibilità di implementare un proprio class loader, ereditando le caratteristiche fondamentali da quello di sistema; con questo strumento possiamo caricare il codice di un agente da una qualunque macchina della rete.

Una caratteristica relativamente nuova in Java (perché introdotta solo con il jdk 1.1) è la possibilità di **serializzare** gli

oggetti: questo permette a qualunque oggetto di essere trasformato in modo reversibile in una rappresentazione adatta per la memorizzazione su disco o per la trasmissione in rete. Con questo meccanismo è possibile trasferire ad una macchina remota un Thread o un qualunque oggetto creato dal programmatore: l'unica e semplice condizione è che implementi l'interfaccia Serializable, cosa che tutte le classi possono fare. Nel paradigma ad agenti, la serializzazione è sfruttata per trasmettere lo stato dell'agente nel momento in cui vuole spostarsi.

Anche la gestione del **networking** è affrontata in Java, offrendo al programmatore un'interfaccia semplice e chiara, che facilita le operazioni su un canale di comunicazione con macchine remote (per esempio, il pacchetto di gestione delle *socket*, è più intuitivo rispetto ad altri linguaggi, come il C++). Quando si progetta un sistema che implementa un paradigma basato sulla mobilità, la gestione dei collegamenti di rete riveste un ruolo molto importante per lo spostamento del codice.

Il vantaggio della programmazione distribuita è la possibilità di sfruttare l'esecuzione concorrente e Java offre a questo riguardo caratteristiche di **sincronizzazione** a livello di linguaggio, cioè è possibile scrivere sezioni critiche in modo pulito ed elegante, con istruzioni specifiche: la clausola "*synchronized*" permette di gestire la sincronizzazione tra thread o risorse, ricordando però che è sempre compito del programmatore utilizzare in modo opportuno questo strumento.

Un punto debole di molti linguaggi è il livello di **sicurezza** che offrono: Java nella jdk 1.2 ha introdotto un nuovo modello di sicurezza, ampliando quello assai limitato costituito dalla sand box presentata nella versione iniziale del jdk 1.0, e già migliorato da una più efficiente granularità nella jdk 1.1 [Gong98] [Gong97]. In un sistema ad agenti mobili si devono prevedere meccanismi per proteggere sia gli host che gli agenti in arrivo su nuovi ambienti.

Esiste anche un principale limite nell'utilizzo di Java per gli agenti mobili: per ragioni di sicurezza non è possibile accedere allo stack di esecuzione di un programma, e quindi non è possibile ripristinarlo, per esempio, in seguito ad una serializzazione. Questo significa che quando un agente decide di spostarsi, non è possibile interrompere la sua esecuzione per farla riprendere dallo stesso punto sulla macchina remota: dalle definizioni viste in precedenza, possiamo affermare che Java non supporta la mobilità forte.

Se non si vuole comunque rinunciare alla "strong mobility", si deve accedere allo stato interno della JVM, realizzando una di queste due soluzioni.

- **Checkpointing**. Si deve monitorare completamente l'esecuzione del codice, per collezionare i dati necessari alla migrazione, dati che, quando l'agente si sposta, si spediscono alla macchina remota. Nel sito remoto, durante la fase di ripristino delle informazioni si devono sfruttare i *checkpoint* (inseriti dal sistema su cui l'agente eseguiva precedentemente) che segnalano dei punti stabili del codice da cui è possibile riprendere l'esecuzione. Per sfruttare questa tecnica, è necessario realizzare un post-compilatore in grado di inserire all'interno del byte code gli accorgimenti necessari per coordinare le due macchine [JAVA].
- **Modifica della macchina virtuale**. All'interno della macchina virtuale sono già presenti tutti i dati relativi allo stato di esecuzione degli agenti, che possono essere estratti con qualche modifica della stessa JVM. Ma questi cambiamenti comportano, come grave conseguenza, la perdita della portabilità.

Per non rinunciare comunque al vantaggio della portabilità che offre questo linguaggio, è possibile implementare una mobilità debole: nel nuovo ambiente gli agenti riprenderanno la loro esecuzione a partire da un punto stabilito a priori.

Dopo un bilancio totale delle caratteristiche positive e negative che Java offre allo sviluppo di sistemi ad agenti mobili, concludiamo affermando che, a nostro parere, si tratta di un linguaggio molto adatto con cui sviluppare applicazioni distribuite, realizzate con un qualunque paradigma di mobilità del codice.

2.3 Un dominio applicativo: Network Management

Negli ultimi anni, la dimensione e la diffusione delle reti di calcolatori sono cresciute a dismisura e la loro gestione deve essere finalizzata a raccogliere nel modo più efficiente ed efficace possibile le informazioni riguardanti il sistema stesso, in modo che su di esse si possa poi effettuare una buona attività di Network Management. Quando si parla di *Network Management*, si intende una *gestione della rete* per cui è necessario svolgere una sequenza di attività di *Monitoraggio*, cioè di ricerca delle informazioni distribuite sui nodi della rete, di *Interpretazione* dei dati ottenuti, ed eventualmente di *Azione* reattiva conseguente all'analisi riportata.

Gli approcci classici, reperibili in letteratura [Gol93], si basano sul paradigma cliente-servitore e sono caratterizzati da uno stile centralizzato, poco flessibile e poco elastico. Il nuovo paradigma ad agenti mobili, ha introdotto una visione innovativa, che propone un approccio complementare, senza stravolgere la tradizionale struttura delle applicazioni esistenti, e che sfrutta in modo adeguato la larghezza di banda, tipicamente limitata, del sistema, attraverso una gestione flessibile ed elastica.

Analizziamo rapidamente l'approccio standard al Network Management per capire i limiti che presenta e verifichiamo, poi, come il paradigma ad agenti mobili si proponga come possibile soluzione a tali limiti.

2.3.1 Gestione centralizzata

L'approccio alla gestione delle reti più utilizzato è quello proposto dalla *Internet Engineering Task Force* (IETF), che si basa sul *Simple Network Management Protocol* (SNMP) [CasMRW96] [CasFSD90]; di grande rilevanza è anche quello proposto da *Internation Organization for Standardization* (ISO), basato sul *Common Management Information Protocol* (CMIP) [ISO91]: entrambi sono realizzati secondo il paradigma cliente-servitore.

Questi protocolli prevedono clienti, che svolgono il ruolo di stazione di servizio fissa (*management station*), cioè di operatori centralizzati in cui si raccolgono i dati rilevati, e di servitori, detti *agent*, che hanno il compito di effettuare le rilevazioni, ma che non hanno nulla a che fare con gli agenti mobili oggetto di questo capitolo. Gli agenti risiedono su vari nodi della rete (router, workstation, ecc.) e devono estrarre informazioni sui dispositivi presenti localmente: i dati ottenuti vengono immagazzinati in una base di dati locale, detta MIB (management information base) da IETF e MIT (management information tree) da ISO. Il *management station* (cliente) interagisce con i servitori caricando da essi alcune delle informazioni che hanno ottenuto (polling): le informazioni sono elaborate dal cliente e come azione reattiva, quest'ultimo può richiedere agli agenti ulteriori dati o inviare loro comandi da eseguire in locale.

La granularità molto fine delle interazioni tra cliente e servitore (da qui il termine "*micro management"*) genera un elevato traffico di rete sui canali di comunicazione intorno al management station e affida totalmente a questi il carico computazionale di gestione.

L'approccio basato sul paradigma cliente-servitore ha origini storiche, infatti nasce per rispondere all'esigenza di gestire reti formate da macchine computazionalmente non potenti, su cui quindi potevano eseguire solo servitori con scarse capacità di elaborazione. Come risulta chiaro, questi protocolli di network

management soffrono di tutti i limiti caratteristici del paradigma cliente-servitore, e citiamo quelli che maggiormente vincolano la realizzazione di un buon sistema di gestione della rete.

- L'approccio fortemente centralizzato, blocca il sistema di monitoraggio nel caso in cui cada il nodo su cui risiede il cliente: infatti, in questo caso si perde l'unico punto in cui si elaborano le informazioni dello stato della rete.
- Come in ogni sistema centralizzato, il cliente diventa un collo di bottiglia a cui sono indirizzati tutti i messaggi, creando un traffico elevato che rischia di congestionare la rete.
- Il paradigma cliente-servitore, rende la gestione della rete non flessibile e non elastica a nuove esigenze rilevate dinamicamente nel sistema.
- Inoltre, come conseguenza dei punti precedenti, la scalabilità della gestione della rete risulta altamente limitata.

Per cercare di superare questi problemi, i protocolli di gestione della rete si stanno sviluppando inserendo caratteristiche di decentralizzazione: i servitori sono capaci di rilevare cambiamenti nel nodo su cui risiedono (per esempio la modifica di un collegamento) e di comunicarli direttamente al cliente senza attendere di essere interpellati. Ma attenzione: i servitori sono responsabili solo di notificare il verificarsi di un evento, e non di agire di conseguenza, con azioni di recovery. Quindi la conoscenza e la capacità di gestione è ancora totalmente "centralizzata" nel cliente.

2.3.2 Gestione ad agenti mobili

L'approccio ad agenti mobili, può essere descritto attraverso una gestione "per delega" (*by delegation*) [GolY95], che conduce a un effettivo aumento di flessibilità. La struttura del controllo può ancora comprendere una stazione di gestione fissa, ma si aggiungono elementi mobili, gli agenti, in grado di agire autonomamente e in modo totalmente asincrono rispetto all'entità non mobile.

Le funzionalità di gestione sono decentralizzate: si spediscono gli agenti nel sistema e questi si spostano tra gli ambienti di esecuzione. La stazione di gestione fissa invia gli agenti mobili sulla rete, con questi compiti: raggiungere i nodi che devono essere controllati, raccogliere le informazioni necessarie ed *elaborarle localmente*. Non è detto che la management station imponga la destinazione ai vari agenti: può anche essere decisa in modo autonomo, in funzione dello stato interno. In questo modo, gli agenti agiscono anche in considerazione della loro posizione attuale ed eventuali modifiche dinamiche della rete non ne pregiudicano la gestione.

Questo approccio al network management produce i seguenti vantaggi:

- incremento della flessibilità: l'accesso alle risorse dei vari nodi è eseguito dagli agenti, il cui comportamento può essere personalizzato, in funzione della macchina su cui agiscono;
- modifiche dinamiche del sistema: la topologia del sistema può cambiare dinamicamente, senza che la gestione ne sia compromessa;
- riduzione del traffico di rete: le informazioni sono elaborate localmente, e non spedite ad un gestore centralizzato;
- affidabilità: l'agente agisce in modo indipendente, anche se un tratto di connessione di rete dovesse cadere;

- alta scalabilità: si possono richiedere nuovi servizi di gestione, svolti per esempio, da un nuovo agente;
- carico computazionale distribuito: le operazioni di elaborazione dei dati non sono eseguite in modo centralizzato sulla stazione fissa, ma in locale, sui nodi controllati.

Consideriamo per esempio la situazione in cui un utente remoto sia addetto al network management di una LAN e confrontiamo la gestione centralizzata (Figura 2.4) e con quella ad agenti mobili (Figura 2.5). Assumiamo che all'interno della rete locale vi siano collegamenti affidabili e veloci, mentre quelli verso la network station siano molto lenti (tutto sommato, ipotesi abbastanza tipiche).

In questo caso, l'uso di agenti mobili comporta notevoli vantaggi, grazie alla loro capacità di analizzare le informazioni di nodi anche non direttamente collegati alla network station.

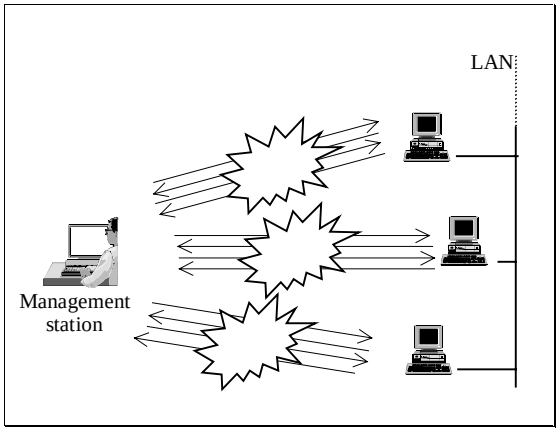


Figura 2-4 Gestione centralizzata, client-server.

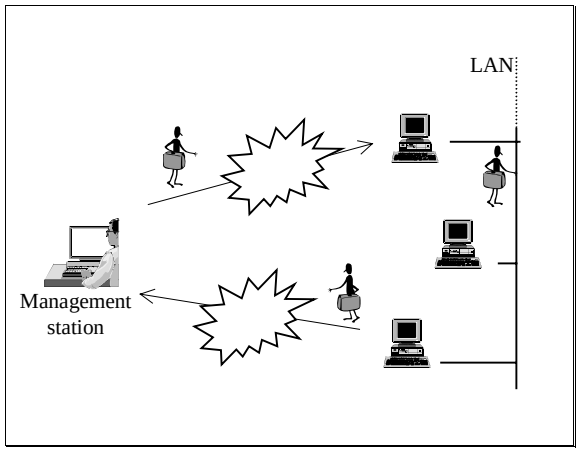


Figura 2-5 Gestione ad Agenti Mobili.

Anche se lo spostamento da un dispositivo ad un altro richiede una larghezza di banda maggiore per trasportare un agente rispetto al semplice spostamento di informazioni (della gestione centralizzata), si deve considerare che le migrazioni avvengono all'interno di una rete locale con collegamenti veloci. Invece, nella gestione centralizzata la necessità di scambiare messaggi impone molte comunicazioni, che si svolgono sempre su collegamenti lenti e inaffidabili.

2.4 Sistemi esistenti ad agenti mobili

La recente popolarità del codice mobile, ha portato alla nascita di un numero elevato di sistemi ad agenti.

Viene fornito di seguito un breve excursus, in cui si considerano solo i sistemi più recenti, fatta eccezione per il caso di Telescript che si può considerare come il fondatore della famiglia dei sistemi ad agenti mobili. In questa presentazione, in particolare, si è cercato di sottolineare le caratteristiche di mobilità attribuite agli

agenti e di estrarre le proprietà peculiari più significative di ogni sistema.

2.4.1 Telescript

Telescript [GenMag94], sviluppato dalla General Magic, è stato il primo sistema commerciale ad agenti mobili e quindi lo citiamo per l'importanza storica che riveste.

Telescript è realizzato con un linguaggio proprietario, chiamato *Low Telescript*, object-oriented e simile a Java e al C++. In ogni macchina della rete sono implementati uno o più ambienti di esecuzione virtuali, detti *place*. Gli agenti si spostano tra le macchine del sistema attraverso il metodo *go*, riprendendo la loro esecuzione dal punto in cui avevano terminato nel nodo precedente: la mobilità supportata è di tipo forte. L'interazione avviene in due modi distinti: il comando *meet* permette agli agenti di invocare metodi di altri agenti che risiedano sullo stesso place, mentre il comando *connect* consente il collegamento remoto tra agenti, per lo scambio di oggetti.

Questo sistema è stato considerato da subito un buon prodotto, anche per i meccanismi di sicurezza che fornisce. Quando un agente si sposta, presenta al place di destinazione delle informazioni, dette *credenziali*, che sono necessarie al nuovo ambiente di esecuzione, per autenticare l'identità dell'utente creatore e in funzione delle quali il sistema assegna all'agente stesso un adeguato grado di fiducia; durante la migrazione, le credenziali sono protette da eventuali manipolazioni esterne con l'uso di tecniche crittografiche. In più ad ogni agente sono assegnati dei permessi che autorizzano all'uso delle risorse disponibili: questi permessi sono concessi dai place, che in tal modo si proteggono da eventuali comportamenti maligni.

Telescript offre anche un buon sistema di tolleranza ai guasti: in ogni macchina è presente un server che con continuità memorizza su un supporto non volatile lo stato di esecuzione di tutti gli agenti locali, così che diventi semplice il ripristino dello stato consistente in caso di caduta del nodo.

Nonostante Telescript sia un prodotto con una buona tolleranza ai guasti, sicuro ed efficiente, la General Magic ha subìto una battuta d'arresto sul mercato a causa della rapida e sempre crescente attenzione dedicata a Java dal mondo informatico e a causa della mancanza di apertura del sistema, basato su un linguaggio proprietario.

2.4.2 Odissey

Odissey [Odissey98] è il sistema che la General Magic ha sviluppato per sostituire Telescript: in generale offre le stesse caratteristiche, a parte il fatto che il linguaggio utilizzato è Java. Questo cambiamento influisce anche sul tipo di mobilità supportato: come si è già notato (paragrafo 2.2), Java non fornisce le caratteristiche necessarie per catturare lo stato completo di un programma in esecuzione, fornendo, di default, una mobilità di tipo debole. Per mantenere la compatibilità con tutte le JVM, l'istruzione *go* non può più presentare le stesse caratteristiche offerte in Telescript. La General Magic ha imposto che gli agenti Odissey possano scegliere solo tra queste due alternative: la prima semplicemente consente di ricominciare da capo l'esecuzione dopo lo spostamento su un nodo remoto, mentre la seconda permette di creare, prima di iniziare a navigare per la rete, una lista di compiti da svolgere sulle determinate macchine che si intende visitare, lista che può essere anche modificata dinamicamente.

Odissey non è un prodotto commerciale, a differenza di Telescript, ed è usato dalla General Magic per svolgere la realizzazione di servizi ed applicazioni interni alla società.

2.4.3 Aglets

Aglets è il sistema realizzato da Ibm, che oggi sembra riscuotere il maggiore interesse tra quelli disponibili sul mercato [IBM96]. Il termine *Aglets* deriva dalla fusione di *agent* e *applet*, per il fatto che le API riprendono concetti e terminologia propri delle applet.

L'ambiente di esecuzione degli agenti, il *context*, è indipendente dal sistema su cui si lavora e per trasferire gli agenti sulla rete si utilizza l'*Agent Transfer Protocol* (ATP) [ATP97]. Questo protocollo è stato progettato per un uso su Internet, è uniforme e indipendente dalla piattaforma e permette quindi di trasferire agenti tra qualunque tipo di macchina. L'ATP cerca di porsi come punto di connessione tra i tanti sistemi presenti, ciascuno con le proprie scelte implementative e con i propri linguaggi di programmazione: un passo importante per raggiungere un primo livello di standardizzazione nel campo della mobilità.

Il sistema di Ibm è totalmente scritto in Java e supporta solo una mobilità di tipo debole: dopo ogni spostamento l'agente deve ricominciare l'esecuzione da un punto prestabilito, pur mantenendo i valori delle interazioni avvenute in precedenza.

Il trasferimento degli Aglet può avvenire secondo due modalità differenti. La primitiva *dispatch* sposta un frammento di codice o un'applicazione stand-alone sulla macchina il cui nome è fornito come argomento, in modo asincrono e immediato. Al contrario, la primitiva *retract* impone all'agente di ritornare nel contesto di origine, in cui ha iniziato la sua esecuzione.

2.4.4 Voyager

Voyager è il tool di sviluppo proposto da ObjectSpace [OBJ97a], realizzato in Java, con mobilità debole, e compatibile con lo standard proposto da Corba.

Voyager permette di costruire oggetti remoti, di spedire loro messaggi e di spostarli tra le applicazioni secondo necessità. Un agente è visto come un particolare tipo di oggetto e quindi, sfruttando la sintassi standard di Java, è possibile creare agenti remoti (caratteristica non prevista nei sistemi considerati sopra [OBJ97b]). Un'altra novità introdotta da Voyager è un servizio di nomi integrato, che associa ad ogni oggetto (e quindi in particolare anche agli agenti) un alias, con il quale in seguito è possibile referenziare dinamicamente l'oggetto stesso anche se è stato mosso (l'associazione infatti non si riferisce all'indirizzo della macchina in cui risiede l'elemento considerato).

Un agente per spostarsi si auto-spedisce il messaggio *moveTo* indicando la meta del trasferimento e il metodo da eseguire una volta arrivato. La destinazione può essere una locazione oppure un oggetto: questo gli permette di raggiungere la risorsa con cui interagire, anche quando questa si muove. Il tempo di vita degli agenti Voyager può essere determinato dall'utente, dando la preferenza ad una delle cinque possibilità disponibili: un agente può vivere finché ha riferimenti attivi (locali o remoti), per una certa quantità di tempo, fino ad un particolare istante, finché non rimane inattivo per un tempo prestabilito oppure per sempre. Di default, gli agenti Voyager vivono per un giorno.

2.4.5 Concordia

Concordia è il sistema sviluppato da Mitsubishi, realizzato interamente in Java [Con98]. Gli ambienti di esecuzione, detti *Concordia Server*, devono essere presenti su tutte le macchine e si

collegano tra loro solo “by need”, per lo spostamento di un agente. La lista delle destinazioni dei vari trasferimenti e i metodi da eseguire a destinazione sono espressi attraverso un oggetto che ogni agente possiede, detto *Itinerary*, che è controllato dal *Concordia Server* al momento dello spostamento per saper quale server remoto contattare. Un componente specifico, il *Persistent Manager*, memorizza lo stato dell’agente durante il suo trasferimento offrendo il beneficio di utili checkpoint da cui ricominciare l’esecuzione in caso di mal funzionamenti. In realtà, offre anche una granularità più fine, in termini di checkpoint sullo stato dei vari oggetti gestiti dall’agente, per le procedure più critiche.

Il *Security Manager* gestisce tre diversi livelli di sicurezza: per sistemi fidati non si richiede alcun tipo di controllo; per sistemi che attraversano reti pubbliche (o semi-pubbliche) gli agenti viaggiano criptati e le credenziali si usano solo per verificare l’identità degli utenti; infine, per sistemi aperti, sviluppati su Internet, deve essere fornito un livello di sicurezza più rigido da parte di qualche autorità esterna.

Dopo aver analizzato le caratteristiche generali dei sistemi ad agenti mobili e averne brevemente descritti alcuni presenti sul mercato, nel capitolo successivo approfondiremo l’analisi di una altro sistema: SOMA.

Cap.3

Una architettura dinamica ad agenti mobili: S.O.M.A.

L'architettura SOMA (Secure and Open Mobile Agent) è un ambiente di sviluppo per applicazioni ad agenti mobili, completamente in Java, realizzato presso il dipartimento del DEIS dell'Università di Ingegneria Informatica di Bologna [SOMA98].

L'architettura su cui lavorano gli agenti SOMA, presenta tutte le caratteristiche descritte nel capitolo precedente, in cui si è delineato la fisionomia di un sistema ad agenti. Le proprietà principali dell'architettura possono essere descritte nel seguente modo:

- *sicurezza*: è realizzata la protezione sia dal lato degli agenti che da quello degli ambienti in cui eseguono. L'accesso al sistema è protetto da password, così che solo utenti autorizzati possono creare agenti SOMA in grado di viaggiare nel sistema;
- *apertura*: il sistema aderisce alla standardizzazione proposta da OMG, cioè è prevista l'interazione di agenti SOMA con applicazioni CORBA-compliant e con gli agenti degli altri sistemi che aderiscono allo stesso standard [CorBel98];
- *dinamicità*: esistono strumenti per la gestione delle modifiche dinamiche della configurazione e dell'inserimento dinamico di nuovi utenti;
- *tool grafici*: tutte le operazioni utente (es., l'attivazione di un agente) sono accompagnate da una interfaccia grafica chiara ed intuitiva;

- *prestazioni*: l'uso degli agenti mobili consente di migliorare le performance di una applicazione, rispetto al tradizionale paradigma cliente-servitore; esiste flessibilità nel consentire diversi livelli di sicurezza, che permettono di ottenere migliori prestazioni;
- *portabilità*: lo sviluppo di applicazioni ad agenti può essere esteso su ambienti con piattaforme eterogenee.

Questo sistema cerca di fornire meccanismi adatti alla realizzazione di sistemi complessi, come Internet, dallo sviluppo incontrollato e dall'alta dinamicità, caratterizzati dal collegamento di reti eterogenee, ciascuna con le sue peculiarità (in termini di amministrazione, gestione e sicurezza). L'obiettivo è quello di fornire un sistema basato sul progetto di un modello che tenga in forte considerazione l'aggregazione di domini di rete estremamente differenti tra loro; per realizzare questo scopo, il sistema fornisce due livelli di astrazione per la descrizione architetturale: i place e i domini (Figura 3-1).

- I **place** rappresentano l'ambiente di esecuzione degli agenti: al loro interno gli agenti possono interagire con le risorse locali e possono cooperare in modo diretto con gli altri agenti presenti. Per esempio, un place può essere rappresentato da un nodo del sistema (si noti che anche più place possono essere implementati su un'unica macchina).

- Il **dominio** raggruppa un certo numero di ambienti di esecuzione correlati per motivi logici o fisici e quindi rappresenta un insieme di place con caratteristiche comuni. Per esempio, un dominio può essere una rete locale di calcolatori.

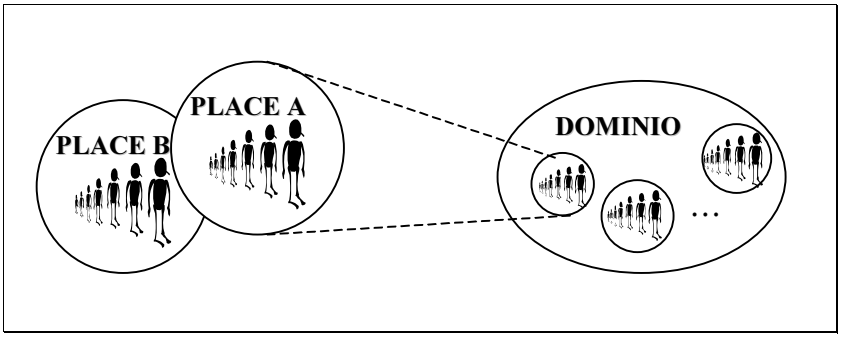


Figura 3-1 Ogni dominio raggruppa un certo numero di place correlati per motivi logici o fisici.

I place che appartengono a domini diversi sono concettualmente scorrelati e ogni tipo di interazione richiede un costo più elevato rispetto al caso in cui risiedano nello stesso dominio; questa scelta si giustifica col fatto che il numero di interazioni inter-dominio dovrebbe essere molto minore rispetto a quello relativo a interazioni intra-dominio.

Tra tutti i place contenuti all'interno di uno stesso dominio, ne esiste uno particolare, il gateway, a cui si affida un ruolo strategico poiché svolge la funzioni di "cancello" verso il mondo esterno: i collegamenti inter-dominio richiedono sempre una azione del gateway. Il place che svolge le funzionalità di gateway è detto "Default Place".

3.1 Mobilità degli agenti

Come si accennava all'inizio, il raggruppamento di place all'interno di un dominio deve essere effettuato secondo criteri logici, inserendo in uno stesso gruppo le entità in cui sono distribuite le risorse necessarie allo svolgimento di vari compiti. Per agevolare le loro frequenti interazioni, lo spostamento di agenti

tra gli ambienti di uno stesso dominio è diretto, mentre tra ambienti di domini diversi richiede l'intermediazione dei due gateway.

Quindi anche lo spostamento degli agenti rispetta le regole descritte sopra a proposito di una generica interazione e avviene in modo differente in funzione del fatto che l'ambiente di destinazione sia interno o esterno al dominio di partenza.

- Se un agente deve effettuare uno **spostamento intra-dominio** (cioè deve raggiungere un place interno al suo dominio), l'operazione coinvolge solo i due place interessati alla migrazione, senza far intervenire nessuna altra entità del sistema.
- Se invece l'agente deve effettuare uno **spostamento inter-dominio** (cioè deve raggiungere un place esterno al suo dominio), i due gateway svolgono il ruolo di intermediari (Figura 3-2), e l'agente è costretto a spostarsi seguendo questo percorso: partenza dal place mittente per raggiungere il Default Place del suo dominio (attraverso uno spostamento intra-dominio, quindi come quello descritto al punto precedente); da qui trasferimento al Default Place del dominio destinazione ed infine smistamento al place target.

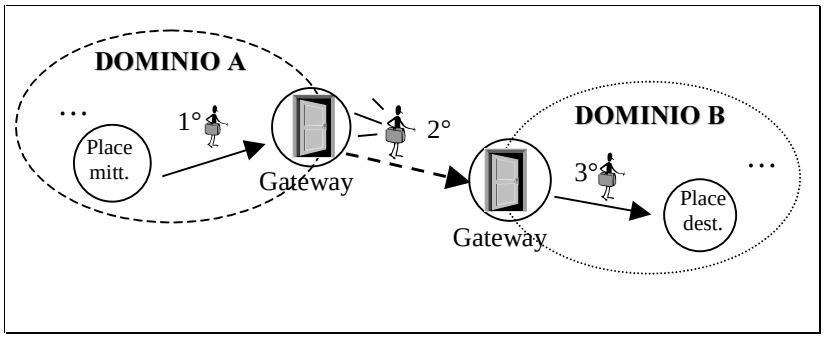


Figura 3-2 Spostamento inter- dominio di un agente.

Il Default Place rappresenta un passaggio obbligato per tutti gli agenti che desiderano raggiungere place di altri domini e, in modo

simmetrico, raccoglie gli agenti in arrivo per smistarli ai place interni. I due livelli di astrazione appena descritti hanno anche una importanza notevole per implementare un sistema di sicurezza: questo tema sarà affrontato in dettaglio nel seguito.

3.2 Comunicazione tra agenti

Nel nostro modello ogni cosa è rappresentata o come agente o come risorsa; gli agenti possono interagire con le risorse locali e spostarsi se la risorsa è remota. Quando invece, l'interazione deve svolgersi tra due agenti, il tipo di comunicazione che possono sfruttare dipende dalla locazione relativa dei due interlocutori.

- Quando due agenti eseguono all'interno dello stesso place, la comunicazione tra loro può avvenire attraverso la **condivisione di risorse** comuni, cioè appartenenti al campo di azione ("scope") del place stesso. Questo è un meccanismo di interazione stretta (si veda il paragrafo 2.1.2), con cui si può realizzare una comunicazione tra un numero illimitato di interlocutori, ma che rappresenta un servizio esclusivamente locale.

- Quando invece due agenti non eseguono all'interno dello stesso place, la comunicazione tra loro avviene solo attraverso **scambio di messaggi**: questo è consentito sia per interazioni intra-dominio che inter-dominio. È importante notare che un agente riceve i messaggi a lui destinati anche dopo un eventuale spostamento (cfr. paragrafo 4.3), poiché la sua locazione è trasparente al mittente. Questo è un meccanismo di interazione lasca che è realizzato inserendo nelle caratteristiche standard di ogni agente il possesso di una Mailbox, in cui ricevere e da cui spedire messaggi. Il meccanismo è realizzato attraverso un tipo

di spedizione asincrona: il mittente non attende che il messaggio sia recapitato correttamente, ma si limita ad affidarlo al sistema sottostante; dall'altro lato, la ricezione può essere sia bloccante che non bloccante: la richiesta di ricezione blocca l'agente finché non è arrivato un messaggio, ma è consentito anche di verificare se la Mailbox è vuota senza sospendere l'esecuzione.

- È implementato un ulteriore meccanismo attraverso cui gli agenti possono effettuare una **comunicazione anonima: la Blackboard**. Gli agenti in questo modo possono depositare un messaggio all'interno della Blackboard, differenziato da quelli già presenti attraverso una stringa di identificazione, e chiunque conosca questo identificatore può leggere (ed eventualmente cancellare) il messaggio originale. Con questo strumento è possibile mettere a disposizione dei servizi: i clienti senza avere conoscenza di come richiedere l'esecuzione di una prestazione, leggono dalla Blackboard le informazioni necessarie per lo svolgimento del servizio desiderato.

3.3 Dominio e Default Place

Chiariamo, ora, in modo dettagliato il ruolo strategico del Default Place all'interno del dominio, nel sistema SOMA. Il concetto di dominio è una astrazione di località, a livello di implementazione, che raggruppa un insieme di place concettualmente legati, e di cui ne rafforzare le politiche di sicurezza. Un dominio potrebbe essere rappresentato da una rete locale (LAN), o dalla sottorete di un dipartimento, all'interno della totalità del sistema che può essere una rete aziendale o in generale la rete della reti (Internet).

Come abbiamo già notato, lo spostamento degli agenti e l'interazione tra essi si distingue in funzione del fatto che appartengano o meno allo stesso dominio e la gestione di azioni

inter-dominio sono sempre controllate dal gateway (o Default Place). Per descrivere con chiarezza il ruolo del gateway, forniamo un elenco dei compiti che deve svolgere.

- Il Default Place deve intercettare tutti i messaggi provenienti dall'esterno del dominio, destinati all'interno e viceversa far uscire quelli interni che sono indirizzati a riceventi esterni: in questa funzione, il Default Place, può essere paragonato ai tradizionali "IP gateway". Se il sistema è stato suddiviso in domini, che rappresentano strutture logiche realmente correlate, il traffico inter-dominio dei messaggi non dovrebbe comunque essere eccessivo.
- Quando un agente vuole entrare nel dominio, viene prima fatto passare per il Default Place, che ha il compito di effettuare alcuni controlli di sicurezza per verificare il rispetto delle politiche caratteristiche del dominio stesso (paragrafo 3.4).
- Al gateway sono inviati tutti gli agenti che si muovono dall'interno del dominio su place remoti: da qui, questi agenti raggiungono il Default Place del dominio destinazione, che svolgerà a suo volta i controlli necessari.
- Terminata la fase dei controlli di sicurezza, gli agenti arrivati dall'esterno devono essere smistati all'interno del dominio: il gateway contiene un server di nomi che cataloga tutti i place del suo dominio. Questa fase non rappresenta un "collo di bottiglia" per la migrazione, perché è da applicare solo per gli spostamenti inter-dominio, mentre i trasferimenti intra-dominio sono diretti.

In pratica, il gateway svolge la funzione di ponte di connessione tra il suo dominio e il mondo esterno, comportandosi come un *proxy intelligente*, cioè capace di fare da collegamento tra più entità e capace di scegliere dinamicamente come comportarsi in funzione della situazione che si presenta. La capacità di affrontare scelte dinamiche, deriva dalla realizzazione di una struttura gerarchica tra i gateway, che è l'unica soluzione per consentire un grado di

dinamicità al sistema. La gerarchia è realizzata su due livelli: tra tutti i gateway ne esiste uno particolare, detto *Supervisor*, che svolge la funzione di coordinamento tra i domini esistenti. Il ruolo principale del Supervisor, è quello di gestire e divulgare nel sistema le modifiche dinamiche. L'introduzione di questa struttura gerarchica consente all'utente di sviluppare applicazioni distribuite su larga scala, cioè su reti con topologie dinamiche (come Internet).

3.4 Sicurezza: politiche di dominio e politiche di place

Quando avviene un'interazione tra due entità, ad esempio tra un agente e il sistema, entrambe la parti coinvolte devono essere tutelate da eventuali comportamenti illeciti. Da un lato si deve proteggere l'ambiente di esecuzione da azioni dell'agente che possono causare danni al sistema o accedere ad informazioni riservate; dall'altro è necessario proteggere gli agenti da ambienti di esecuzione che si rivelino ostili (Figura 2.3).

La struttura del sistema su due diversi livelli di astrazione, assume un ruolo fondamentale per garantirne la sicurezza nei confronti degli agenti. A livello di astrazione più basso, è possibile assegnare caratteristiche di sicurezza personalizzate ad ogni ambiente di esecuzione, permettendo ad ogni place di vedere rispettate le garanzie che ritiene necessarie. Ad un livello più alto di astrazione, il dominio, che coordina un insieme di place con proprietà affini, riunisce le caratteristiche generale dei suoi place per definire una strategia di sicurezza comune a tutto il gruppo. Questi due livelli di astrazione creano una protezione doppiamente rinforzata: ogni azione è controllata prima, per verificare se rispetta la politica di sicurezza imposta dal dominio, e poi, se il primo controllo va a buon fine, è valutata nel place in cui deve essere eseguita. Se uno dei due controlli non è superato, il permesso di svolgere l'azione non è concesso.

In *SOMA*, la protezione degli agenti è garantita in termini di privacy dei dati e di integrità del codice. È importante sottolineare, in questa circostanza, che non è possibile prevenire la completa distruzione di un agente, ma questo evento è simile a quello di un guasto in un sistema distribuito, e, se necessario, si può cercare di gestirlo, a livello applicativo, introducendo tecniche di fault-tolerance basate sulla replicazione.

3.5 Gestione dinamica

Nel sistema SOMA è compreso un *Tool di Gestione* ("*System Manager*") che consente l'amministrazione dinamica (Figura 3-3), in termini di:

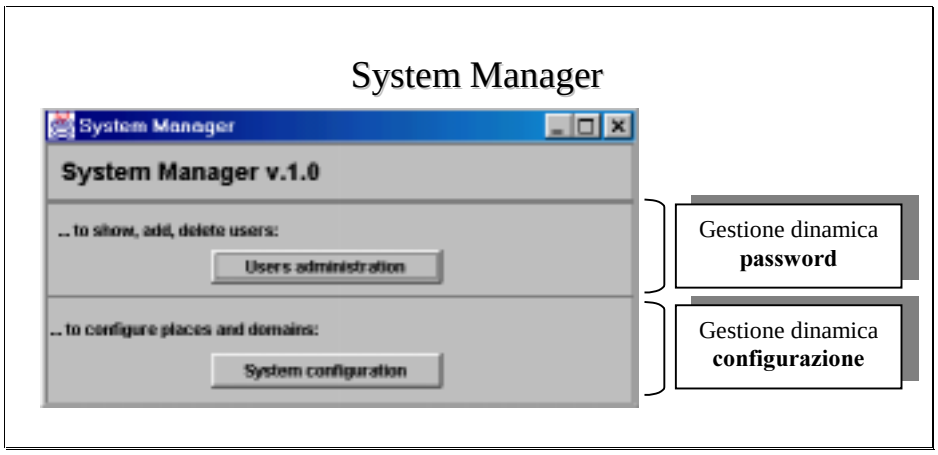


Figura 3-3 System Manager: accesso alla gestione dinamica delle password utente e della configurazione.

- **Configurazione delle località**: è possibile inserire (o eliminare) dinamicamente nuove località intese sia come domini che come place.

- **Gestione utenti**: è possibile inserire nuove password di accesso, per estendere dinamicamente l'uso del sistema SOMA a nuovi utenti interessati a sviluppare applicazioni ad agenti. È possibile anche eliminare utenti precedentemente inseriti se diventano indesiderati.

L'accesso alla gestione dinamica, è consentita solo ad utenti specializzati, per proteggere il sistema da operazioni incontrollate, come l'eliminazione di place attualmente sfruttati da utenti ignari delle modifiche (o addirittura la cancellazione totale della struttura del sistema), e per impedire l'inserimento o la cancellazione incontrollata di password di accesso.

Questo *Tool di Gestione*, progettato per facilitare le scelte dinamiche, può essere eseguito da un punto qualunque del sistema.

3.5.1 Gestione dinamica della configurazione

Analizziamo il *Tool di Gestione* ("*System Manager Tool*") compreso nel sistema SOMA, per quanto concerne l'aspetto della gestione dinamica della configurazione: le sue funzioni si impiegano sia per settare la configurazione iniziale del sistema, che per apportare eventuali modifiche dinamiche. È composto da due schermate principali: Domain Manager e Domain Configuration, riportate sotto rispettivamente nelle Figure 3.4 e 3.5.

È riportato nel seguito l'elenco delle operazioni messe a disposizione dal *Manager Tool*, con la descrizione di come realizzarle.

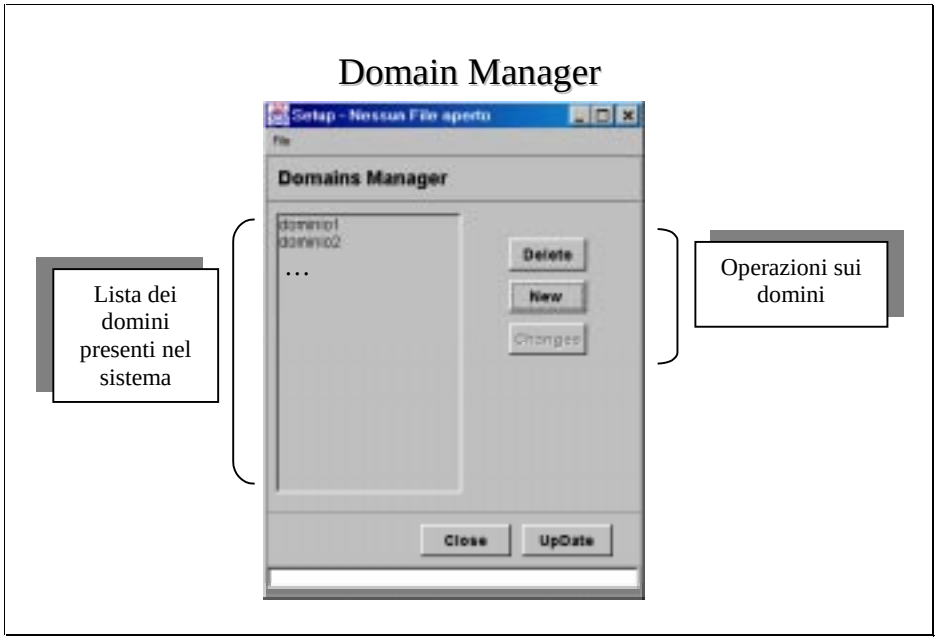


Figura 3-4 Domain Manager: gestione dinamica della configurazione dei domini.

- Consultazione dei domini esistenti: nella finestra "*Domain Manager*", appare la lista dei domini attualmente contenuti nel sistema (Figura 3-4).
- Creazione di un dominio: si preme sul pulsante "*New*" dalla finestra "*Domain Manager*" (Figura 3-4). Come secondo passo è necessario inserire il nome del dominio che si vuole creare e alcune informazioni sul place che svolge il ruolo di gateway.
- Consultazione dei place esistenti: nella finestra "*Domain Configuration*" (Figura 3-5), appare la lista dei place attualmente contenuti nel dominio selezionato (se si sta creando un nuovo dominio, la lista sarà vuota).
- Inserimento di place: si devono compilare i campi della sezione "*new place*" nella finestra "*Domain Configuration*" e quindi premere il pulsante "*Insert Place*" (in basso nella Figura 3-5). Nota: l'inserimento di un Place non è effettuato se vi sono

errori; è comunque presente una barra di stato che guida l'utente nella compilazione dei vari campi.

- Eliminazione di un place: per eliminare un Place, lo si deve selezionare nella finestra "*Domain Configuration*" e poi premere il pulsante "*Delete Place*" (un pulsante di comando della Figura 3-5).

Attivazione di un place: nella finestra "*Domain Configuration*" si seleziona il place da attivare dall'elenco, e quindi nel dominio considerato, e si preme il pulsante "*Switch on Place*" (un pulsante di comando della Figura 3-5).

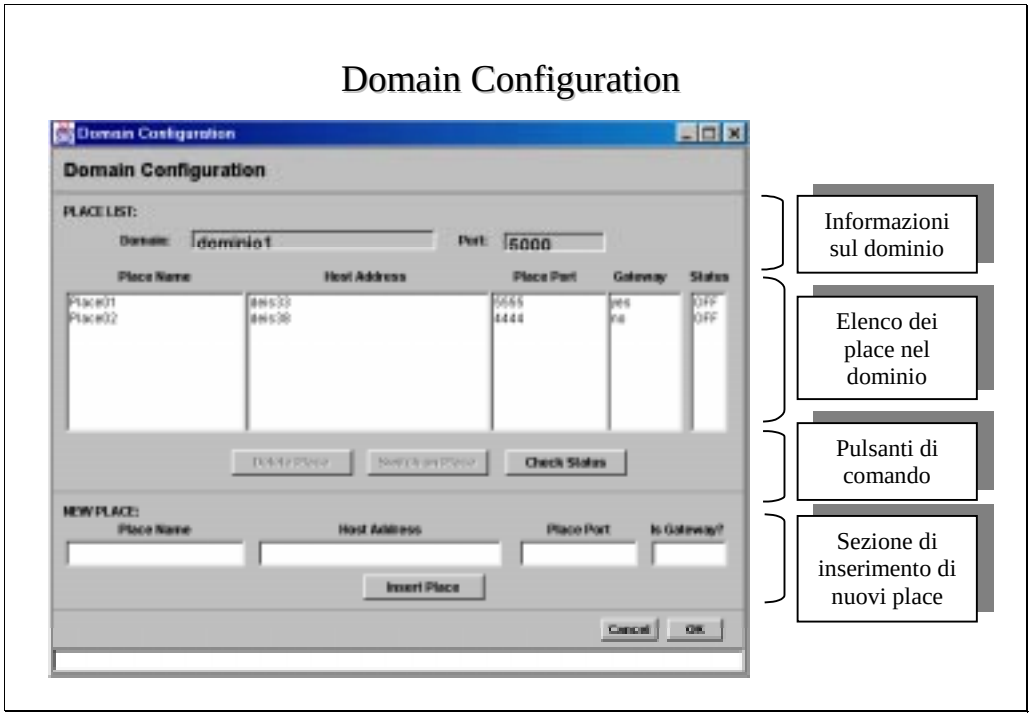


Figura 3-5 Domain Configuration: gestione dinamica della configurazione dei place di un dominio (in questo caso il "dominio1").

3.5.2 Gestione dinamica delle password

Il sistema SOMA offre una gestione dinamica delle password, attraverso lo strumento *Users Manager*. L'esistenza di password è introdotta per garantire un livello adeguato di sicurezza: tutti gli utenti devono possedere una password, che serve per essere riconosciuti dal sistema e acquisire quindi il diritto di lanciare agenti. Il System Manager Tool gestisce l'introduzione e l'eliminazione dinamica delle password, e consente di visualizzare l'elenco degli utenti già inseriti; l'interfaccia utente è semplice e intuitiva ed è illustrata in Figura 3-6.

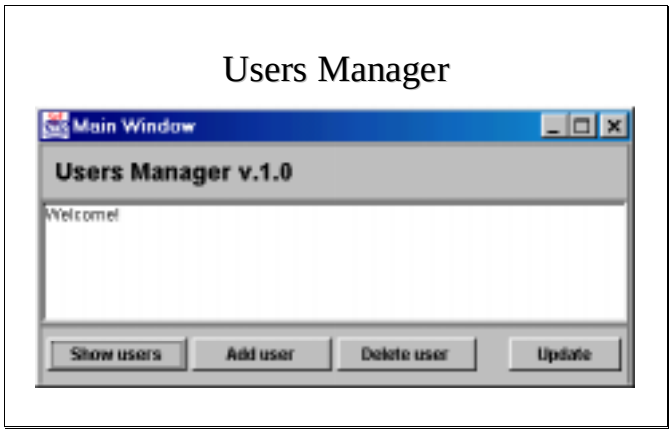


Figura 3-6 Users Manager: gestione dinamica delle password utente.

La sicurezza del sistema SOMA, è descrivibile attraverso i due aspetti fondamentali dei sistemi ad agenti: protezione del sistema e protezione dell'agente (Figura 2-3).

- **Protezione del sistema**. Gli agenti portano la firma del loro creatore (*principal*), per rivelarne l'identità ai place di destinazione: in questo modo, il nuovo ambiente potrà assegnare all'agente un adeguato grado di fiducia, in relazione alle caratteristiche del suo *principal*. Questo

controllo di accesso è indispensabile per proteggere l'ambiente di esecuzione, e quindi anche gli agenti presenti, dall'azione di utenti non fidati (cfr. paragrafo 4.5).

- **Protezione dell'agente**. La protezione sugli agenti è garantita in termini di protezione dei dati privati e di integrità del codice; durante gli spostamenti, gli agenti sono cifrati, per impedire attacchi esterni indesiderati.

Una analisi approfondita sulla sicurezza in SOMA, è disponibile in [Coc98].

3.6 "Agent Launcher"

SOMA comprende un Tool grafico con cui lanciare gli agenti nel sistema, l'Agent Launcher, riportato in Figura 3-7.

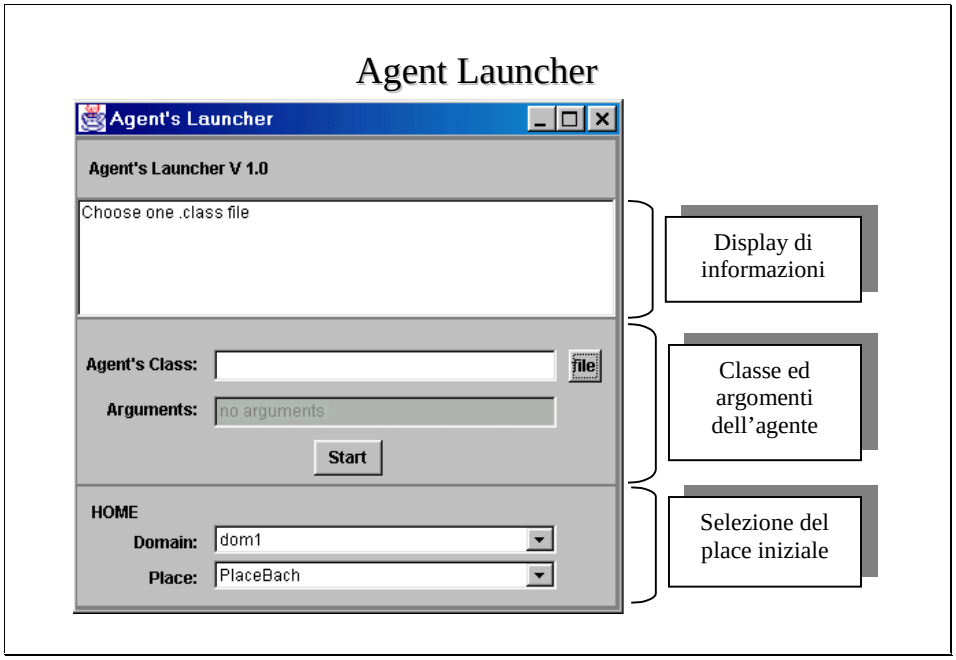


Figura 3-7 Agent Launcher: Tool Grafico di esecuzione degli agenti.

Per mettere in esecuzione un agente SOMA, si attraversano due fasi. Nella prima avviene l'identificazione dell'utente, attraverso l'inserimento di una password, necessaria per ragioni di sicurezza (paragrafo 3.5.2). Nel passo successivo, si specifica il tipo di agente da lanciare, cioè si seleziona il nome della classe (un file .class) che l'agente dovrà eseguire, selezionandolo dalla schermata che appare premendo il pulsante "file", nella sezione centrale dell'Agent Launcher.

Il Tool si collega al Supervisor per richiedergli la configurazione attuale del sistema: in questo modo l'utente può scegliere il place su cui fare iniziare l'esecuzione dell'agente, tra quelli attualmente attivi (un place è attivo se ha in esecuzione il supporto SOMA); è possibile specificare anche un place remoto, cioè non fisicamente localizzato sulla macchina corrente.

3.7 Network Management

In Soma, è presenta un Network Management a livello di sistema, realizzato, chiaramente, sfruttando gli agenti mobili: è fortemente decentralizzato e rispetta tutte le caratteristiche delineate nel paragrafo 2.3, per ottenere una efficiente gestione di rete.

Nel rispetto delle due astrazioni di località di place e dominio, presenti nel sistema, l'applicazione di monitoraggio e gestione è implementata attraverso l'interazione di tre tipi di agenti: uno che accetta le richieste, uno che coordina l'applicazione sul dominio e infine l'ultimo che realmente svolge le operazioni di monitoraggio (Figura 3-8). Questa suddivisione dei compiti, serve anche per affrontare con maggiore organizzazione il network management su reti di vaste dimensioni. Analizziamo ora, in dettaglio, gli elementi costitutivi.

- ***Amministratore generale*** *("SysAdmin")*. Svolge il ruolo di coordinatore dell'applicazione: attende le richieste dei clienti (anche remoti), lancia le operazioni di gestione della rete, raccoglie i risultati e li fornisce all'utente stesso. All'amministratore non sono assegnati particolari oneri decisionali, né carichi computazionali, in modo da non risultare un "collo di bottiglia". Non rappresenta, quindi, in alcun modo il componente centralizzato dell'approccio tradizionale a cliente-servitore ("network station"): se necessario, è possibile replicarlo (a causa del numero di utenti richiedenti, per l'elevata distribuzione del sistema, o per qualunque altra necessità), mettendo in esecuzione un amministratore su più macchine del sistema.
- ***Amministratore Locale di Dominio*** *("DomainServer")*. Ha il compito di coordinare l'attività di gestione che si svolge nel suo dominio. Possono essere presenti più amministratori locali per ogni dominio, nel caso in cui debbano dirigere le operazioni relative a più gestioni separate (per esempio, due gestioni richieste da due utenti differenti). L'amministratore locale è inviato da quello generale su un place del dominio (per esempio sul Default Place, oppure su un place qualunque), non appena riceve una nuova richiesta da un utente: giunto a destinazione, coordina le operazioni, raccoglie i risultati e li comunica all'amministratore generale.
- ***Agente incaricato***. È un agente mobile, creato "by need" dall'amministratore locale, in base alle richieste pervenute dall'amministratore generale: si sposta nei place da controllare e svolge la particolare mansione che gli è stata affidata. Esistono tanti tipi di agenti incaricati, che si differenziano per gli specifici compiti per cui sono utilizzati. Questi agenti hanno la capacità di elaborare localmente le informazioni, distribuendo il carico computazionale su tutta

la rete (caratteristica essenziale del network management basato su agenti mobili, come sottolineato nel paragrafo 2.3.2).

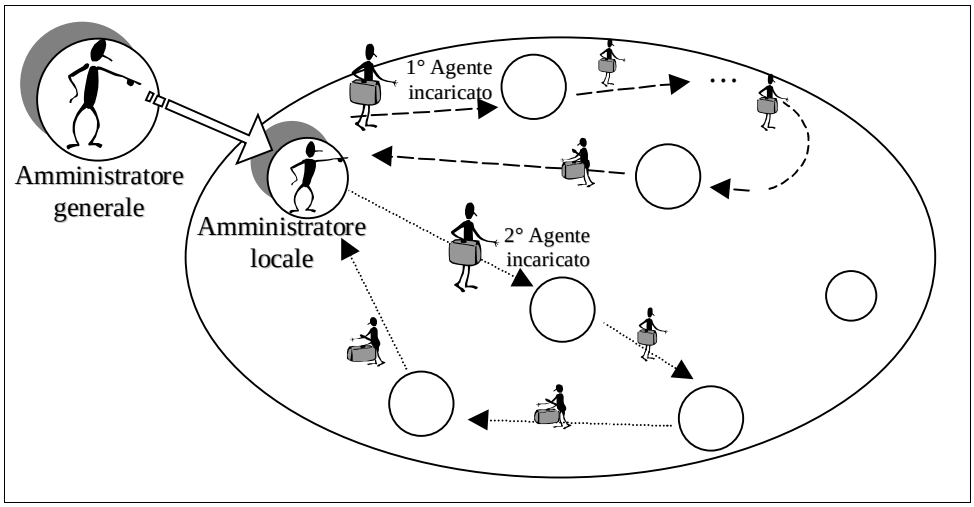


Figura 3-8 Network Manager di un dominio, realizzato con un amministratore generale, uno locale e due agenti incaricati, che si spostano sui place da controllare.

Quando l'amministratore generale riceve una richiesta di gestione da parte di un utente, crea un nuovo amministratore locale e lo invia sul dominio da analizzare, a meno che non ve ne sia già presente uno, disponibile a ricevere richieste di servizio. Successivamente l'amministratore generale comunica all'agente remoto il tipo di gestione da eseguire, attraverso lo scambio di un messaggio (tra nodi remoti, è consentita solo un'interazione lasca). A questo punto, l'amministratore locale genera degli agenti incaricati che si distribuiscano sul dominio per svolgere le operazioni di gestione; il numero di agenti creato è deciso dinamicamente in funzione della quantità di macchine da analizzare. Quando un agente incaricato ha terminato i suoi spostamenti e ha svolto le elaborazioni che gli erano state

commissionate, ritorna nel place in cui risiede l'amministratore locale per riferire il risultato del suo operato. In questo caso la comunicazione avviene con un'interazione stretta, attraverso la condivisione di risorse. Solo quando tutti gli agenti incaricati hanno portato a termine i loro compiti, l'amministratore locale informa l'amministratore generale dell'esito dell'operazione, attraverso un'interazione stretta. In generale però, i due agenti di amministrazione non risiedono sullo stesso nodo, vincolo inevitabile per un'interazione di quel tipo: per questo motivo si introduce un agente mobile che svolge il ruolo di "corriere", trasportando le informazioni raccolte dall'amministratore locale sulla macchina dell'amministratore generale e comunicando con questi attraverso la condivisione di risorse.

Il Network Management di SOMA mira ad ottenere la migliore performance attraverso una equa distribuzione degli agenti, che si spostano, per eseguire le operazioni di gestione, all'interno di un gruppo di place concettualmente o fisicamente correlati (il dominio), rappresentato per esempio nella realtà da una LAN. In generale, il Network Management può essere realizzato ad agenti mobili, anche con strutture più snelle e più flessibili.

Il sistema di Network Management realizzato in SOMA consente di sfruttare tutti i vantaggi che derivano dall'implementazione di una gestione basata su agenti mobili (paragrafo 2.3.2). Si è potuto dimostrare inoltre, la presenza di un reale incremento delle prestazioni, che permette di *diminuire il tempo di attesa delle rilevazioni, rispetto al tradizionale approccio centralizzato basato sul paradigma cliente-servitore* (Figura 3-9).

In particolare, è stato misurato il tempo impiegato per monitorare lo stato di un dominio costituito da N place attivi (considerando i casi di N=3 e N=6), con il modello ad agenti mobili e con una gestione centralizzata. L'applicazione di

monitoraggio è stata testata su una LAN di personal computer collegati da una rete Ethernet.

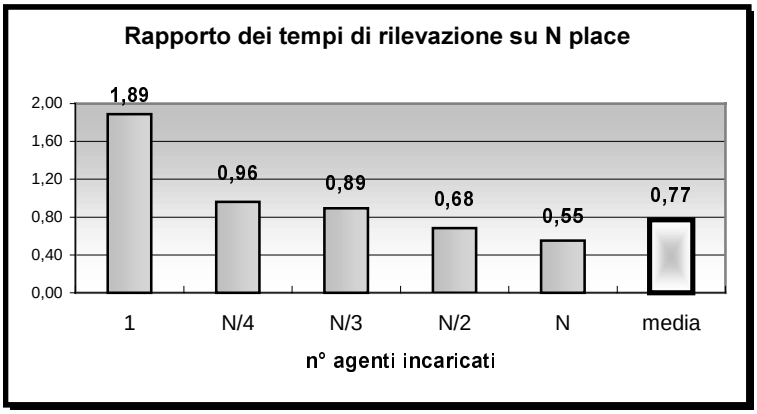


Figura 3-9 Rapporto dei tempi di rilevazione con agenti mobili rispetto alla gestione basata su cliente-servitore.

Il miglioramento ottimo si verifica quando esiste un agente per ogni ambiente da controllare (N agenti per N place), poiché si esegue una gestione completamente parallela, in cui tutti gli agenti eseguono contemporaneamente i loro compiti. Fino che ad ogni agente incaricato è affidato lo svolgimento del management su un massimo di due place (cioè se è presente un numero di agenti circa pari alla metà degli ambienti da controllare, quindi N/2 agenti per N place), il tempo di attesa medio è quasi la metà rispetto al tradizionale approccio centralizzato.

Questo miglioramento si ottiene solo se il sistema è utilizzato in modo ottimale: quando l'utente fa una richiesta di gestione, prima che questa sia effettivamente incominciata, si deve attendere che l'amministratore locale si sposti sul dominio da controllare e che da qui crei e coordini gli agenti incaricati. Questo tempo di attesa, deve essere ammortizzato, dalla realizzazione di un monitoraggio

“sufficientemente” parallelo. Infatti se l’agente incaricato fosse unico, l’overhead indispensabile per iniziare le operazioni non sarebbe compensato dall’efficienza della gestione. Per capire meglio, diciamo che sarebbe maggiore il tempo necessario ad organizzare il lavoro di quello indispensabile per eseguirlo.

Cap.4

Dinamicità del framework SOMA: architettura ed implementazione

Nel capitolo precedente abbiamo analizzato il sistema SOMA ad alto livello. Ora forniamo una descrizione dettagliata del framework SOMA: valutiamo l'implementazione dell'ambiente di esecuzione degli agenti, e come si realizza la coordinazione tra essi; approfondiamo le scelte di implementazione di un agente e delle sue caratteristiche principali. In seguito descriviamo la dinamicità dell'architettura e l'aspetto di comunicazione di basso livello.

4.1 Implementazione dell'astrazione di place

Il contesto di esecuzione, cioè il place, è una macchina virtuale su cui gli agenti possono eseguire: su uno stesso nodo fisico possono essere implementati più place, indipendenti tra loro, anche appartenenti a domini diversi. Il supporto SOMA fornisce ai place le strutture necessarie per la gestione dell'esecuzione degli agenti e mette a disposizione degli agenti stessi alcune funzioni.

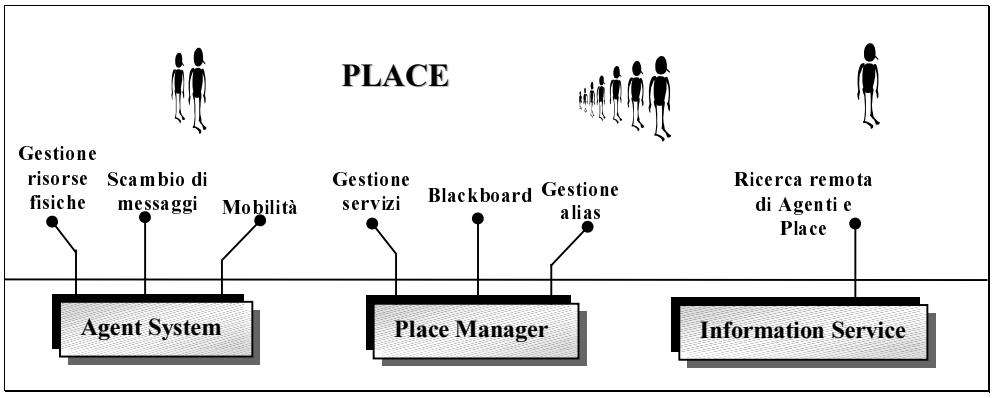


Figura 4-1 Il place interagisce con gli agenti attraverso i tre moduli: Agent System, Place Manager ed Information Service.

I moduli che realizzano l'astrazione di place sono tre (Figura 4-1):

- ***Agent System***. Gestisce tutte le funzionalità di base: implementa la *mobilità*, fornisce il supporto per lo *scambio di messaggi* e si occupa dell'*amministrazione delle risorse fisiche*. Un sottomodulo importante è costituito dal *NetworkManager*, che ha il compito di gestire tutte le connessioni con i place remoti.
- ***Place Manager***. Consente agli agenti di interagire con una *Blackboard*, coordina un *servizio di naming* locale, sfruttato dagli agenti "servitori" per mettere a disposizione una propria prestazione e offre agli agenti la possibilità di *registrarsi con un alias* presso il place (come avviene anche in Voyager).
- ***Information Service***. Consente la *ricerca remota di informazioni*: la posizione di un agente, la risoluzione di un alias o la localizzazione di un place. Il campo di azione è ristretto ad un dominio, e può essere quello di appartenenza o uno remoto.

4.1.1 Attivazione di un place

L'operazione più semplice per attivare un place è quella messa a disposizione dal Tool del "System Manager", descritto nel

paragrafo 3.5.1. Per completezza, citiamo anche un ulteriore metodo, che consiste nel lanciare una specifica istruzione a linea di comando, in cui è necessario distinguere tra place e gateway.

L'istruzione per mettere in esecuzione un place, è composta dal comando *java* ed è seguita da questi parametri:

1. AgentSystem.Main, che è la classe statica di partenza del sistema
2. Il numero della porta su cui attende il demone Main (per esempio, 7777)
3. Il nome del dominio a cui appartiene (per esempio, dom1)
4. Il nome del Place che si lancia (per esempio, Place1)
5. Il numero della porta su cui attende il demone del gateway (per esempio, 5001)
6. Il nome della macchina su cui è implementato il gateway (per esempio, deis33)

Per esempio,

java AgentSystem.Main 7777 dom2 PlaceGodel 5001 ronfo

L'istruzione per mettere in esecuzione un gateway (tra cui il Supervisor), è composta dal comando *java* ed è seguita da questi parametri:

1. AgentSystem.Main, che è la classe statica di pertenza del sistema
2. Il numero della porta su cui attende il demone Main (per esempio, 5555)
3. Il nome del dominio a cui appartiene (per esempio, dom1)
4. Il nome del Place che si lancia (per esempio, PlaceBach)
5. Il numero della porta su cui attende il demone del gateway (per esempio, 5001)
6. Il nome della macchina su cui è implementato il Supervisor (per esempio, deis33)
7. Il numero della porta su cui attende il demone del Supervisor (per esempio, 6001)

Per esempio,

java AgentSystem.Main 5555 dom1 PlaceBach 5001 deis33 6001

Se la macchina e la porta di attesa del Supervisor (i punti 6 e 7) coincidono con la macchina in cui si esegue il comando ("localhost") e con la porta del gateway stesso (il punto 5) allora il sistema deduce che il gateway appena messo in esecuzione è quello che svolge il ruolo di Supervisor.

4.2 Coordinazione tra i place

I servizi offerti agli agenti, richiedono una coordinazione strutturata tra i vari place, soprattutto tra quelli appartenenti allo stesso dominio. Per migliorare le prestazioni temporali dei servizi disponibili e dello spostamento degli agenti, i place di uno stesso dominio sono sempre in connessione tra loro. Si deve tenere in considerazione il fatto che non tutti gli ambienti di esecuzione dichiarati interni ad un dominio sono sempre attivi: possono essere disattivati per scelte di gestione o più semplicemente perché la macchina su cui sono implementati è fuori servizio. Quindi, a parte i place del dominio momentaneamente disattivi, gli altri sono tutti collegati tra loro. Non appena un place si attiva, informa immediatamente tutti i place del dominio già presenti e inizializza i canali di comunicazione con essi. Successivamente deve avviare un processo, un demone (implementato nella classe "ThServer"), che si metta in attesa di eventuali richieste di connessioni provenienti da nuovi place, a loro volta in fase di attivazione.

L'interazione tra due place può avvenire per due ragioni sostanziali: per l'esecuzione di un servizio o per lo spostamento di un agente. Per separare fisicamente lo svolgimento dei due tipi di operazioni e per non sequenzializzare richieste di natura diversa, esistono due canali di comunicazione separati.

Il primo canale di comunicazione è dedicato alla trasmissione di Comandi: infatti, l'interazione tra place è eseguita mediante lo scambio di *Comandi*. I Comandi sono oggetti, di sistema, eseguiti nell'ambiente di destinazione che racchiudono ciò che l'ambiente mittente desidera sia svolto in remoto. Tutti i Comandi ereditano da una classe astratta "Command" (Tabella 3), che implementa l'interfaccia "Serializable" per essere trasportabile tra gli ambienti che interagiscono: prevede un metodo exe() in cui è contenuto il codice da eseguire in remoto e la possibilità di restituire un risultato. Per creare un nuovo Comando, e quindi implementare un servizio, è sufficiente costruire una nuova classe che eredita da "Command", in cui specificare il codice da eseguire. Quando un place riceve un Comando dall'apposito canale di comunicazione, invoca il suo metodo exe() e soddisfa la richiesta del place mittente.

```
public abstract class Command implements Serializable{
        public Command() {}
        public abstract void exe();
        public abstract void Returned();
            }
```

Tabella 3 La classe astratta Command.

I comandi rispettano le regole imposte nel sistema e se devono raggiungere un place esterno al dominio devono passare attraverso il gateway, prima di uscire dal dominio di origine e prima di entrare in quello di destinazione.

Il secondo canale di comunicazione è dedicato al trasferimento degli agenti, che verrà trattato in modo approfondito nel paragrafo 4.4.

L'interazione tra i place è completamente asincrona, cioè non esistono momenti particolari in cui inviare o ricevere oggetti (intesi sia come Comandi che come agenti): quindi su entrambi i canali di comunicazione deve essere attivato un demone (rispettivamente

implementato in “CommandReader” e “AgentReader”), sempre in attesa di ricevere qualcosa.

Ogni place, quindi, ha due connessioni fisse con ogni altro place del dominio (Figura 4-2); il numero di collegamenti presenti in un place che appartiene a un dominio in cui esistono N place attivi, è pari a 2*(N-1).

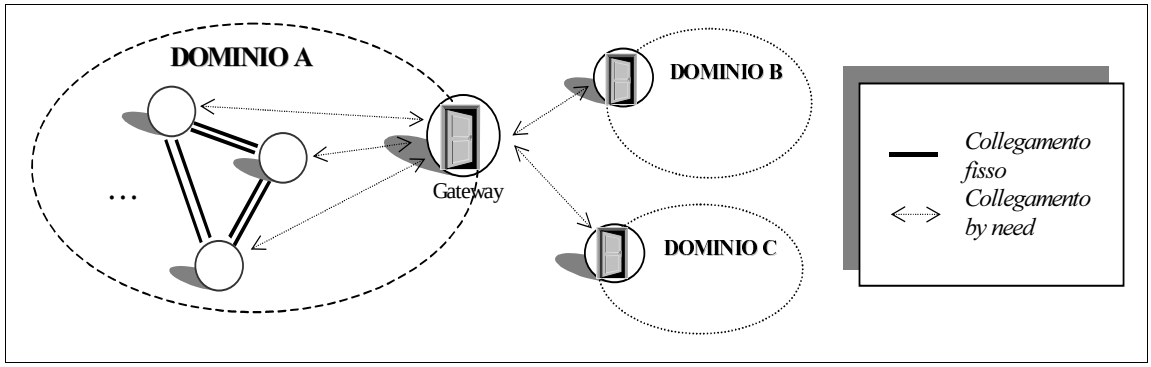


Figura 4-2 Connessioni tra place e tra gateway.

Citiamo anche un tipo di coordinazione, che non si svolge esattamente tra due place, ma tra un place e l’utente, quando si vuole mettere in esecuzione un agente nel sistema. Per facilitare questa operazione esiste un’interfaccia grafica, “Agent’s Launcher” (paragrafo 3.6), che deve chiedere all’ambiente prescelto di inserire un agente. Serve quindi un altro demone all’interno dei place, che rimanga in attesa di richieste utenti, e questo, dal lato utente, è sicuramente il demone principale (da qui, il nome della classe in cui è implementato “Main”).

C’è anche da considerare che, oltre alla coordinazione standard tra loro, tutti i place devono interagire in modo particolare con il gateway del loro dominio. Ricordando che un dominio raggruppa un insieme di place concettualmente o fisicamente correlati, si presume che il traffico dovuto alle interazioni sia concentrato particolarmente all’interno del dominio, mentre quello inter-dominio ne rappresenti solo una piccola parte. Questa

considerazione ci fa scegliere di non far avere ad ogni place un canale di comunicazione fisso verso il gateway, ma di crearlo solo by need (Figura 4-2).

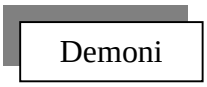
Quando un place deve svolgere un'operazione inter-dominio, richiede al suo gateway di aprire un canale di comunicazione, sul quale si sviluppa l'interazione. La richiesta di connessione è catturata sul gateway da un demone, che ha il solo compito di gestire le esigenze provenienti dall'interno del dominio (da qui, il nome della classe in cui è implementato "InternalManager").

Analogamente, quando il gateway ha necessità di dialogare con un place, per esempio per portare a termine un servizio richiesto dall'esterno, cioè da un altro dominio, apre un canale di comunicazione, rivolgendosi al demone del place, già citato, che ha il compito di ricevere richieste di connessione ("ThServer").

Anche il coordinamento tra i gateway dei vari domini avviene attraverso connessioni attivate by need, per lo stesso motivo già citato nel caso di coordinamento di place e gateway. La gestione delle richieste di connessione che arrivano dall'esterno, è affidata a un demone, addetto solo alla interazione con i Default Place remoti (implementato nella classe "ExternalManager").

Lasciamo per ultimo un tipo di coordinamento particolare, che si svolge tra il gateway e i suoi place: questa interazione è necessaria per far rispettare le regole di sicurezza imposte nel sistema. Rinviando al paragrafo 4.5 per una spiegazione dettagliata, è giusto citare qui, per dovere di completezza, l'esistenza di un ulteriore demone nel gateway, che attende richieste particolari da parte dei place che desiderano ricevere il database dei certificati per i Principal riconosciuti nel sistema: questo database è comunemente detto "keystore" e il demone è implementato nella classe "keystoreDEM".

Nella Tabella 4, sono riassunti il nome dei demoni e i rispettivi ruoli.



Place

ThServer	Attende richieste di connessione dai place che si stanno attivando e dal gateway
CommandReader	Attende l'arrivo di Comandi su una connessione già realizzata
AgentReader	Attende l'arrivo di Agenti su una connessione già realizzata
Main	Attende l'arrivo di richieste utenti, per inserire nuovi Agenti nel sistema

Gateway

InternalManager	Attende richieste di connessione dai place interni al dominio
ExternalManager	Attende richieste di connessione da altri gateway
KeystoreDEM	Attende richieste di connessione dai place che vogliono ricevere il keystore

Tabella 4 Demoni di SOMA per la coordinazione tra i place

4.3 Implementazione dell'Agente

Ogni agente SOMA è realizzato come sottoclasse dalla classe astratta "Agent" (Tabella 5): l'oggetto con cui si rappresenta un agente racchiude tutte le sue caratteristiche, così che, quando l'agente deve effettuare un trasferimento, vi sono già comprese tutte le informazioni che devono seguirlo nel nuovo ambiente e quindi è sufficiente inviare solo tale oggetto: questo vantaggio deriva dalla scelta di utilizzare un linguaggio object-oriented come Java (come descritto nel paragrafo 2.2). La classe "Agent" implementa l'interfaccia "Serializable" per consentire a tutte le

classi che ereditano da essa di essere trasportate facilmente come sequenza di byte.

```
public abstract class Agent implements Serializable {
        private AgentID My_ID;
        public String Start;
        public boolean Traceable;
        public Mailbox Mail;
        public ArrayList preferenze;
        public ArrayList credenziali;
}
```

Tabella 5 La classe astratta Agent.

Gli elementi caratteristici di un agente sono:

- **AgentID**: questo è un identificatore unico, all'interno di tutto il sistema. È formato da:
  - nome del place di origine
  - nome del dominio di origine
  - nome della classe che implementa l'agente
  - numero identificativo unico all'interno del place di nascita

  Questo identificatore è necessario per qualunque interazione si voglia effettuare con l'agente; la presenza dei dati sull'ambiente di origine all'interno dell'AgentID sono importanti, perché è proprio questo place che ha il compito di sapere sempre la posizione attuale dell'agente e di rintracciarlo quando è necessario (per esempio per recapitargli un messaggio). Quando l'agente cambia posizione, il place da cui si allontana informa il place di origine, attraverso un Comando e quest'ultimo aggiorna il campo adeguato. Ogni place conosce la posizione degli agenti che hanno avuto origine al suo interno: in questo modo, tutti gli agenti possono sempre essere rintracciabili.
- **Start**: questo campo contiene informazioni utili solo nel momento in cui l'agente deve spostarsi su un place remoto.

Come vedremo meglio nel prossimo paragrafo, l'esecuzione di un agente, dopo il suo trasferimento, riprende dal metodo il cui nome è contenuto in questo campo.

- **Traceable**: questo campo indica al sistema se deve tenere traccia degli spostamenti dell'agente, cioè se l'agente deve essere rintracciabile: è fondamentale che lo sia, se si desidera fargli arrivare dei messaggi durante il suo tempo di vita. Il fatto di creare agenti non rintracciabili, può causare problemi di gestione, quindi deve essere consentito solo ad agenti di sistema.
- **Mailbox**: questo è lo strumento che gli permette di ricevere e inviare messaggi. La comunicazione con altri agenti, quindi, è consentita solo se si conosce la "Mail" dell'agente a cui si vuole inviare il messaggio. Se si è deciso di rendere l'agente non rintracciabile, la gestione della Mailbox non è implementata e quindi l'agente diventa più snello e la sua migrazione risulta più efficiente, ma si perde anche la capacità di verificare la sua posizione.
- **Preferenze**: questo campo contiene una serie di entry che permettono al creatore di personalizzare il comportamento dell'agente, limitandone la capacità d'azione normalmente concessa a tutti gli altri.
- **Credenziali**: gli elementi contenuti all'interno di questo campo, servono nel momento in cui l'agente arriva su un nuovo place: infatti, il possesso di determinate credenziali può permettergli di ricevere un grado di fiducia superiore da parte del nuovo ambiente su cui deve eseguire. Le credenziali si basano sulla certificazione di un precedente rapporto dell'agente (per esempio, con un place fidato) e sono realizzate attraverso la firma digitale del codice dell'agente stesso.

Tutti gli agenti sono oggetti *passivi* (contrariamente a quanto si potrebbe pensare) e serializzabili: non appena un agente nasce,

viene affidato a un thread (**Worker**) con il solo compito di associare all'agente un flusso di esecuzione. Il Worker fa partire l'esecuzione dal metodo specificato nel campo Start dell'agente, che di default è il metodo run(), e attende che questo termini. Il tempo di vita dell'agente non può essere determinato a priori, né può essere specificato dall'utente (come accade in Voyager), ma dipende solo dal momento in cui termina la sua esecuzione.

4.4 Implementazione della mobilità

Il sistema è realizzato completamente in Java, che, come sappiamo, non supporta la mobilità forte (paragrafo 2.2). La nostra scelta è stata quella di mantenere la compatibilità assoluta con la JVM standard, per non perdere il vantaggio della portabilità del sistema: in SOMA, quindi, è *realizzata una mobilità debole*.

Come già detto al paragrafo 2.1.1, la mobilità debole impone che l'esecuzione nel place remoto riprenda a partire da un punto prestabilito: questo punto è determinato dal nome del metodo dinamicamente contenuto nel campo Start dell'agente stesso.

Quando un agente decide di intraprendere una operazione di migrazione, il Worker associato termina la sua esistenza e l'agente è affidato ad un nuovo Worker appositamente creato sul place di destinazione. Un agente, per comunicare l'intenzione di spostarsi su un altro ambiente di esecuzione, deve invocare un apposito comando SOMA: il comando go().

4.4.1 Il comando go()

Lo spostamento di un agente tra diversi ambienti di esecuzione, si ottiene in maniera semplice per il programmatore, grazie all'esistenza del metodo **go()** (Tabella 6), messo a disposizione dal supporto. Gli agenti invocano la go() nel momento

in cui desiderano migrare su un place remoto: devono specificare la destinazione dello spostamento e il punto da cui riprendere l'esecuzione una volta terminato il trasferimento. Analizziamo in dettaglio i due argomenti del comando go().

Public final void go(PlaceName p, String metodo) throws CantGoException { … }
Public final void go(DomainName d, String metodo) throws CantGoException { … }

Tabella 6 Interfaccia del comando go().

1. Il primo argomento del comando go() deve specificare la destinazione della migrazione:
   - se lo spostamento è intra-dominio, cioè interno al dominio di appartenenza, la destinazione è espressa con il nome del place da raggiungere (espresso con "PlaceName p") e l'esecuzione riprende sull'ambiente richiesto;
   - se lo spostamento è inter-dominio, cioè esterno al dominio di appartenenza, la destinazione è espressa con il nome del dominio da raggiungere (espresso con "DomainName d"): l'esecuzione riprende sul Default Place del nuovo dominio.

   Quando lo spostamento è inter-dominio, non è detto che sia già noto il nome del place di destinazione: è possibile che questo nome derivi dalla raccolta di informazioni prelevata internamente al dominio. Per esempio, un agente si può spostare su un dominio, verificare quali place sono attivi e poi, solo dopo, scegliere su quale spostarsi. Quindi, lo spostamento inter-dominio è composto da due passaggi: nel primo si trasporta l'agente dal place di origine fino al suo gateway; nel secondo l'agente è trasferito al Default Place del dominio di destinazione, dove riprende l'esecuzione; da qui l'agente può scegliere se spostarsi su un particolare place, con uno spostamento intra-dominio.

   Non è disponibile, a livello di sistema, la possibilità di muoversi "inseguendo" un altro agente, funzionalità messa

invece a disposizione agli agenti Voyager (paragrafo 2.4.4), ma è possibile realizzarla come servizio a livello applicativo.

2. Il secondo argomento del comando go() deve specificare il metodo da invocare, quando l'agente è arrivato a destinazione (espresso con "String metodo"): di default, il metodo invocato è "run()". Come è noto (spiegato in 2.1.1), la mobilità debole, supportata dal sistema, non consente di far riprendere l'esecuzione esattamente dal punto successivo alla richiesta di migrazione; l'esecuzione riprende dal metodo il cui nome è indicato nell'invocazione del comando go(), comunque senza limitare il potere espressivo del programmatore. Tipicamente, si sceglie di far corrispondere la richiesta di migrazione con l'ultima istruzione del metodo in cui è inserita: questa scelta introduce uno stile di programmazione strutturato, che può guidare l'utente alla stesura di un codice lineare e quindi con minore probabilità di errori concettuali. Per chiarire meglio questo punto, analizziamo un flusso di esecuzione simbolico di un agente SOMA (Tabella 7).

```
run() {                          ➔  eseguito sul place di origine
    … …
    /* valutazione dei compiti da eseguire */
    …
    /* spostamento sul dominio dom1 */
    go( dom1, "runDom1");
}
runDom1() {                      ➔  eseguito sul Default Place di dom1
    … …
    /* valutazione dei place attivi o esecuzione di un altro compito */
    /* spostamento sul place pippo */
    go( pippo, "runPippo");
}
runPippo() { … … }               ➔  eseguito sul place "pippo"
```

Tabella 7 Esempio di flusso di esecuzione di un agente SOMA.

Il flusso di esecuzione dell'agente risulta molto chiaro, poiché i compiti sono suddivisi in funzione dell'ambiente su cui devono essere svolti. Questa caratteristica fornisce una migliore leggibilità del codice e rende più facile all'utente la creazione di agenti SOMA che svolgono funzioni specifiche spostandosi anche su tutto il sistema distribuito.

4.4.2 CantGoException

L'operazione di migrazione su un place remoto da parte di un agente, che avviene con il comando go(), può non avere l'effetto desiderato. Infatti, la reazione alla chiamata del metodo di trasferimento può non provocare l'effettivo spostamento che l'agente desidera. Per segnalare all'agente che lo svolgimento del comando non è andato a buon fine, è stata introdotta una eccezione che l'agente può catturare e quindi gestire in base alla situazione attuale.

```
try{
        go(dom2,"run2");
  } catch (CantGoException e) {
        /* azioni reattive alla cattura dell'eccezione */
    }
```

Tabella 8 Gestione dell'eccezione CantGoException

Nella Tabella 8, è mostrato come catturare la CantGoException: all'interno dell'area in cui si intraprendono le azioni reattive, si deve esprimere il codice da eseguire nel caso in cui la migrazione non sia andata a buon fine.

Ma per sapere come reagire a un mancato trasferimento dell'agente, è bene sapere quali siano le **cause** che possono far scatenare la CantGoException.

- Una prima causa può essere il fatto che il place di destinazione, specificato come primo argomento della go(), non sia attivo: in questo caso l'esecuzione dell'agente riprende nello stesso ambiente precedente alla richiesta di migrazione (intra-dominio) ed esegue il codice esplicitato nell'area di azione reattiva.
- Una seconda causa può essere il mancato collegamento con il place di destinazione, a causa di una connessione non affidabile o, in generale, momentaneamente interrotta. L'esecuzione riprende nello stesso modo descritto nel caso precedente.
- Una situazione analoga alla prima, può presentarsi quando la richiesta di un trasferimento inter-dominio, non viene realizzata perché il gateway locale non è attivo (è infatti possibile che il lavoro all'interno di un dominio proceda, anche se il suo gateway è disattivo, l'unico limite è la completa sconnessione con l'esterno). Anche in questa situazione, la ripresa dell'esecuzione è la stessa vista nei casi precedenti.

4.4.3 Trasferimento inter-dominio: un caso particolare

Una situazione molto diversa da quelle descritte come cause di mancata migrazione, si verifica nel caso in cui un trasferimento inter-dominio non vada a buon fine perché il gateway remoto non è attivo: a differenza di tutti gli altri casi di impossibilità di migrazione, non si segnala all'agente nessuna eccezione.

Infatti, quando l'agente deve spostarsi su un altro dominio, il place su cui risiede lo affida al suo gateway e, terminato questo primo passaggio, considera l'operazione già conclusa correttamente. Ma in realtà la migrazione non è affatto conclusa e solo se l'agente riesce a raggiungere il dominio remoto, l'intero trasferimento è andato a buon fine. Quindi il place di partenza è totalmente disinteressato alla reale posizione dell'agente, una volta

che lo ha consegnato al suo gateway. Questo è diretta conseguenza della modalità di implementazione del comando go(), nel caso di trasferimento inter-dominio. Quando il Default Place locale ha ricevuto l'agente da spostare, e ha valutato che il gateway remoto non è attivo, non ha altra soluzione che rimettere in esecuzione l'agente nel proprio ambiente, dal momento che il suo place di origine ha già considerato chiusa l'operazione di trasferimento. In particolare, quando un gateway spedisce all'esterno (cioè ad un altro gateway) un oggetto (cioè l'istanza di una classe), attraverso il metodo "sendOutObject" (della classe "GateManager"), e la connessione non può realizzarsi, deve verificare se l'oggetto in questione è un agente, e in tal caso, deve rimetterlo in esecuzione nel proprio ambiente.

Questa scelta, impone al programmatore di agenti SOMA di controllare la posizione attuale dell'agente dopo ogni trasferimento inter-dominio: infatti, se il gateway remoto è disattivo, il flusso di esecuzione dell'agente procede normalmente, senza segnalare nessuna eccezione che possa far scegliere all'agente di intraprendere un percorso alternativo.

```
{  …
     try{
              go(new DomainName(dom),"newDom");
       } catch (Exception e) { }
}

public void newDom(){
      here = AgentSystem.getCurrentLocation();
      if (destinazione.equals (here)
            /* arrivato a destinazione → esecuzione normale*/
      else     /* NON arrivato a destinazione */
}
```

Tabella 9 Verifica conseguente ad uno spostamento inter-dominio.

Quindi, deve diventare una buona abitudine, *verificare il dominio in cui si risiede, dopo ogni spostamento inter-dominio*, come è mostrato nella Tabella 9.

4.5 Implementazione della Sicurezza

Un aspetto che non può essere trascurato in un sistema ad agenti è quello della sicurezza: per affrontare le scelte di implementazione di SOMA, si deve prima chiarire cosa si intende con il termine di "Principal": il *Principal* è il creatore dell'agente, o meglio il responsabile dell'inizio della sua esecuzione. Per esempio, un utente che inserisce un agente nel sistema, attraverso l'Agents Launcher, ne diventa il Principal; questo ruolo non è svolto solo da utenti, ma può essere interpretato anche da altri agenti [Coc98]. Il ruolo di Pricipal diventa molto importante per tutelare gli ambienti di esecuzione da comportamenti illeciti degli agenti.

Un primo passo è quello identificare l'agente che richiede un'interazione, per sapere che grado di fiducia assegnargli: ricordando che nella definizione di agente (paragrafo 2.1) si sottolinea la sua dipendenza ad agire secondo la volontà del suo creatore (Principal), l'operazione di identificazione dell'agente è strettamente correlata al suo Principal.

Un sistema ad agenti mobili, che si propone come supporto per applicazioni distribuite su reti aperte e a larga scala, deve fornire un buon sistema di sicurezza: analizziamone i principi alla luce della distinzione, ritenuta fondamentale, fra politiche e meccanismi.

4.5.1 Le politiche

L'operazione di identificazione di un agente permette al place di assegnargli un grado di fiducia: si deve quindi determinare quali

permessi gli sono assegnati e quali accessi sono invece esplicitamente vietati. Questo concetto è rappresentato nel sistema dall'introduzione di permessi positivi e negativi, cioè la distinzione tra azioni consentite e azioni assolutamente vietate: l'insieme delle associazioni tra un permesso e il soggetto a cui è riferito, costituisce la politica di sicurezza di un sistema.

La politica di sicurezza del sistema SOMA è definita su due differenti livelli di astrazione di località: il place e il dominio. La politica di default (o di dominio) raggruppa le caratteristiche comuni di tutti i place appartenenti, cioè esprime i permessi, positivi e negativi, da applicare in tutti i contesti di esecuzione interni al dominio. La politica locale (o di place) può "allargare" la politica di default, cioè può concedere permessi aggiuntivi, che non siano in contraddizione con quelli espressi dal dominio.

Un agente trasferito su un nuovo ambiente, che appartiene a un dominio diverso di quello di partenza, deve sottoporsi prima ai controlli imposti dalla politica di default, e, superati questi, deve rispettare la politica locale dell'ambiente su cui esegue. Invece, un agente trasferito su un ambiente che appartiene allo stesso dominio del place di partenza, dovrà prendere atto solo della nuova politica locale, perché già sottostava alla politica di dominio.

Infine, per differenziare il ruolo di agenti nati dallo stesso Principal e che svolgono lo stesso codice, è possibile in generale, associare loro delle *preferenze*: in questo modo si può personalizzare il comportamento dell'agente, imponendogli limitazioni ulteriori sui permessi concessi dalle politiche locali dei vari place.

4.5.2 I meccanismi

Il meccanismo per verificare l'identità di un agente, come abbiamo già accennato (paragrafo 4.3), può sfruttare il fatto che tutti gli

agenti sono caratterizzati da un identificatore unico nel sistema, che permette all'ambiente di esecuzione di distinguerlo senza ambiguità: ma, per decidere il grado di fiducia da assegnargli, l'ambiente ha bisogno di informazioni aggiuntive. Queste informazioni sono le *credenziali*, che costituiscono una sorta di garanzia sulle buone intenzioni dell'agente: la prima credenziale è rilasciata dal Principal, al momento della creazione. Questo avviene attraverso l'operazione di firma digitale del codice dell'agente, da parte del Principal. Ulteriori credenziali possono essere acquisite dagli agenti durante il loro tempo di vita, richiedendole ai place su cui eseguono e la presentazione di maggiori credenziali, può accrescere il grado di fiducia che l'ambiente concede all'agente.

L'operazione della firma digitale, è eseguita attraverso un algoritmo asimmetrico (Digital Signature Standard, DSS) [KauPS95], che sfrutta due chiavi, una pubblica e una privata, per ogni entità appartenente al sistema. Solo l'entità stessa può apporre la sua firma, attraverso l'uso della sua chiave privata, che è nota unicamente ad essa; chiunque nel sistema invece può verificare il proprietario di una firma, perché tale controllo si esegue con la chiave pubblica, che, come si deduce dal nome, è nota a tutti.

Quando l'agente arriva su un nuovo place, quest'ultimo valuta le credenziali dell'agente e deve quindi verificare la firma digitale apportata sul codice, da parte del Principal o da parte di un altro place con cui ha interagito in precedenza: questa operazione è possibile solo se l'ambiente possiede il database delle chiavi, detto *keystore* (ditribuito dal Supervisor [Coc98]), da cui estrarre la chiave pubblica di chi si pone come garante dell'agente.

Per garantire l'integrità dell'agente durante i suoi spostamenti, esso viene protetto attraverso l'uso di un algoritmo crittografico asimmetrico (RSA, dal nome degli autori: Rivest, Shamir, Adleman) [KauPS95], basato su chiavi pubbliche e chiavi private: le informazioni dell'agente, che devono essere trasportate su un

canale insicuro, sono mascherate (cifrate) dal place di partenza, utilizzando la chiave pubblica del place di destinazione. In questo modo, un ascoltatore mal intenzionato non è in grado di interpretare il contenuto delle informazioni trasmesse e non può neanche modificarle senza far rilevare la sua intromissione: solo il destinatario è in grado di decifrarle, sicuro dell'integrità della comunicazione.

Questi sono i meccanismi di sicurezza, utilizzati sia per proteggere l'ambiente di esecuzione da agenti non fidati, sia per garantire all'agente la protezione dei suoi dati, descritti in modo più approfondito in [Coc98]. Vedremo ora, come sono gestite, in un sistema dinamico, le strutture dati che consentono di realizzare questo livello di sicurezza.

4.6 Gestione dinamica

L'architettura del sistema deve prevedere la distribuzione e l'aggiornamento a place e gateway di alcune informazioni, necessarie per fornire i servizi di coordinazione e di sicurezza descritti nei paragrafi precedenti, in modo trasparente al programmatore.

Ogni place deve contenere al suo interno le informazioni sulla posizione fisica in cui sono implementati gli altri place del dominio (il nome del place, la macchina e la porta su cui attendono richieste di connessione). In più deve decidere che grado di fiducia assegnare agli agenti in arrivo, in funzione delle credenziali rilasciate dal Principal o da place fidati, di cui deve verificarne l'autenticità.

Ogni gateway, invece, deve contenere strutture dati per le informazioni sul nodo in cui sono implementati gli altri gateway (il nome del loro dominio, la macchina e la porta su cui attendono richieste di connessione), e tutte le informazioni sui place interni al

proprio dominio. In più, il gateway deve avere le conoscenze necessarie per svolgere il ruolo di coordinatore di alto livello nella gestione della sicurezza.

In definitiva, le strutture dati necessarie ai place e ai gateway riguardano gli aspetti della configurazione e della sicurezza, in misura diversa, a seconda del diverso tipo di astrazione di località che rappresentano: l'aspetto della configurazione comprende la posizione (fisica e logica) dei vari ambienti all'interno del sistema, mentre quello della sicurezza è relativo alle politiche da far rispettare e ai meccanismi con cui realizzarle (la gestione delle chiavi e del loro contenitore, il keystore).

4.6.1 Tre livelli gerarchici: Place, Gateway e Supervisor

Ricordiamo che, tra tutti i gateway, ne esiste uno che svolge una funzione di coordinamento, il "Supervisor": al suo interno esiste una struttura dati che racchiude le informazioni di tutto il sistema e in cui è memorizzata sempre la configurazione aggiornata. La scelta di realizzare un solo Supervisor è giustificata dalla semplicità di gestione che ne deriva, anche se diventa una entità centralizzata: questo problema potrebbe essere affrontato distribuendo e/o replicando i servizi che il supervisore mette a disposizione, ma si è scelto di non farlo, per non appesantire il sistema con la coordinazione di più strutture distribuite.

Il Supervisor risulta il punto di riferimento di tutte le entità del sistema e deve essere sempre attivo quando si eseguono modifiche dinamiche alla configurazione, per garantire la consistenza delle strutture dati interne a tutti gli ambienti di esecuzione.

Per questo scopo, è stato necessario considerare alcune **assunzioni**:

- ogni place conosce a priori il proprio gateway
- ogni gateway sa chi svolge il ruolo di Supervisor

- il Supervisor deve essere il primo place ad attivarsi.

Queste assunzioni impongono la conoscenza di una sola località, e per questo motivo si ritengono abbastanza ragionevoli.

All'interno del sistema, si crea una struttura gerarchica distribuita su tre livelli (Figura 4-3):

- il livello più alto è rappresentato dal **supervisore**, che raccoglie tutte le informazioni generali del sistema;
- il livello intermedio è costituito dai **gateway**, in cui sono contenuti i dettagli del dominio da coordinare e i dati sui gateway remoti;
- infine, al livello più basso, ci sono i **place**, che hanno conoscenza solo degli altri place interni al dominio.

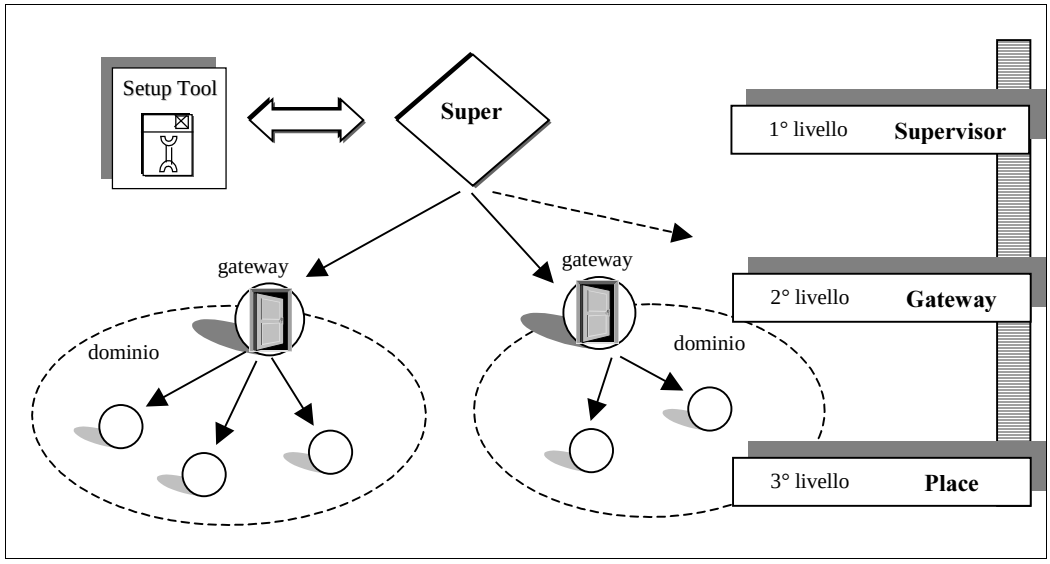


Figura 4-3 La propagazione delle modifiche dinamiche si sviluppa su tre livelli gerarchici: Place, Gateway e Supervisor.

La configurazione iniziale del sistema e le eventuali modifiche dinamiche che si intendono apportare, si realizzano utilizzando il package SetupTool. Questo Tool di Setup, progettato per facilitare

la gestione del sistema, avvia il “System Manager” (Figura 3-3) e può essere lanciato in ogni istante su un qualunque host SOMA: per visualizzare la configurazione attuale, deve interagire con il Supervisor, che è l’unico a possedere questa conoscenza. Si possono apportare le modifiche necessarie, e al termine, il pulsante “Update” aggiorna il sistema, rendendo attuale la nuova configurazione.

4.6.2 Inizializzazione del sistema

Per inizializzare il sistema si deve creare un file, di default “standard.conf”, in cui inserire la configurazione iniziale delle località: questo file creato attraverso il Tool di Setup del “System Manager” (in particolare in “Domain Manager” e “Domain Configuration”, riportati nelle Figure 3-4 e 3-5), è memorizzato nella directory di sistema ed è consultato solo al momento della inizializzazione (può essere utilizzato anche per eventuali inizializzazioni successive). Rispetto alle versioni precedenti [Tar98] si è esclusa la necessità di interpellare numerosi file di inizializzazione, residenti in modo statico in directory prestabilite.

Dopo che il Tool di Setup del “System Manager” ha memorizzato la configurazione iniziale, il sistema è pronto per partire. Nel file di configurazione sono compresi alcuni dati che vengono distribuiti ai vari ambienti quando si attivano. Il primo place che nasce deve essere il Supervisor (deve essere lanciato sempre da linea di comando, dalla directory di sistema): consulta il file di configurazione “standard.conf” memorizzato nella directory di sistema e inizializza le sue strutture interne che devono garantire sempre la configurazione globale aggiornata.

All’interno di un dominio, il primo place che si attiva deve essere il gateway: il primo passo per l’attivazione è quello di collegarsi subito al Supervisor per ricevere le informazioni sul

proprio dominio e su quelli remoti. Di queste informazioni, le prime sono necessarie per la gestione intra-dominio, mentre le seconde per quella inter-dominio.

Infine, quando un place si attiva, sa chi svolge il ruolo di gateway nel suo dominio (è la prima assunzione esposta in precedenza), e si rivolge a lui per chiedergli di essere informato sulla configurazione attuale; in questo modo può inizializzare le sue strutture dati, e può iniziare i tentativi di connessione con gli altri place del suo dominio.

In particolare, quando un gateway si attiva, si rivolge al Supervisor per ricevere le informazioni sulla configurazione inviandogli il Comando “ConfigurationRequestCommand” e si blocca in attesa dell’aggiornamento. Quando il Supervisor riceve la richiesta del gateway, invia a sua volta un Comando “ConfigurationUpdateCommand” che contiene i dati di tutti i domini presenti nel sistema.

In modo del tutto analogo a quanto appena descritto avviene anche l’attivazione di un place: esso invia la richiesta al suo gateway, che lo aggiorna limitando l’estensione delle informazioni al solo dominio di appartenenza.

Il comando “ConfigurationRequestCommand” (Tabella 10), inviato dal place al suo gateway per richiedere l’aggiornamento della configurazione, contiene nel suo metodo exe(), il codice da eseguire nell’ambiente di destinazione, cioè il metodo specifico del gateway (implementato nella classe “GateManager”) di nome “SendActualConfiguration”, con i parametri adeguati, che impone al gateway di aggiornare la configurazione di un place, di nome “NomePlace”.

```
public class ConfigurationRequestCommand extends Command {

  public ConfigurationRequestCommand(String place,boolean gate) {…}

  public void exe() {
    GateManager.SendActualConfiguration(NomePlace,isGate,true);
  }

  public void Returned() {}

}
```

Tabella 10 Il Comando “ConfigurationRequestCommand”.

Il Comando “ConfigurationUpdateCommand” (Tabella 11) è inviato come risposta alla ricezione del Comando “ConfigurationRequestCommand” e contiene le informazioni necessarie per l’aggiornamento richiesto: il suo metodo exe(), richiama il metodo “AggiornaConf” del gateway (implementato nella classe “GateManager”) o del place (contenuto nella classe “NetManager”) destinatario, che aggiorna le strutture dati interne.

```
public class ConfigurationUpdateCommand extends Command {

    private Location[] bagaglio=null;     /* info sui place del dominio */
    private GateInfo Domini[]=null;       /* info sui gateway del sistema */

    public ConfigurationUpdateCommand(…) {…}
    public void exe() {
        if (forGate)
                GateManager.aggiornaConf(bagaglio,Domini,rks);
            else
                NetManager.aggiornaConf(bagaglio);
            …
  }
    public void Returned() {}

}
```

Tabella 11 Il Comando “ConfigurationUpdateCommand”.

4.6.3 Modifiche dinamiche alla configurazione

Per apportare le modifiche alla configurazione del sistema in modo dinamico, si utilizza, come nella fase di inizializzazione, il Tool di Setup del “System Manager”: per confermare gli aggiornamenti è sufficiente premere il pulsante “Update” della schermata di “Domain Manager”. La lista delle operazioni che possono essere effettuate è presentata nel paragrafo 3.5.1: analizziamo come avviene l’aggiornamento di queste modifiche.

La diffusione dei cambiamenti applicati, si esegue seguendo la struttura gerarchica in cui è organizzato il sistema. Il Supervisor coordina l’aggiornamento, applicando, innanzi tutto, le modifiche alla sua struttura globale e poi comunicando ai gateway dei domini, coinvolti dalle modifiche, i cambiamenti da cui sono interessati. Inoltre, dal momento che, per poter realizzare interazioni inter-

dominio, ogni gateway conosce tutti gli altri gateway, è chiaro che in caso di nascita di un nuovo dominio, la notizia debba essere diffusa a tutti i gateway già presenti nel sistema.

In ogni dominio, poi, è il gateway che propaga ai suoi place le modifiche interne, così che ogni ambiente possa aggiornare le proprie strutture dati. Ricordiamo infatti, che tutti i place hanno conoscenza della configurazione interna al dominio. In particolare, il gateway elabora la nuova configurazione ricevuta dal Supervisor, e spedisce solo l'identità dei place inseriti o eliminati, per minimizzare il numero di dati inviati.

La propagazione delle modifiche consente a tutte le entità di avere le strutture dati sempre aggiornate. In questo modo, nella fase di attivazione, un place, riceve dal suo gateway le informazioni relative all'ultima configurazione e, risalendo la struttura gerarchica, anche al gateway sono trasmessi tutti i dati aggiornati dal supervisore. Tutte le modifiche sono propagate dal Supervisor ai vari gateway, attraverso l'invio dello stesso comando utilizzato in fase di inizializzazione "ConfigurationUpdateCommand", che può essere spedito in qualsiasi istante, in modo asincrono.

Consideriamo, per esempio, il caso di inserimento di un place: quando tutto il sistema è stato informato della modifica, il nuovo ambiente può anche essere attivato direttamente dal "Domain Configuration", attraverso il pulsante "*Switch on Place*". Il Tool di Configurazione gestisce questa richiesta, analizzando se la località da attivare corrisponde ad un gateway o ad un place: questa distinzione è importante, a causa delle diverse operazioni di inizializzazione da intraprendere (come descritto nel paragrafo precedente). Successivamente invia il Comando "CAttivaPlace" ad un place già attivo all'interno stessa macchina in cui mettere in esecuzione il nuovo ambiente. In generale, sarebbe necessaria la presenza di una entità sullo stesso host, che sia a conoscenza di un protocollo, in grado di realizzare i servizi richiesti: per evitare di inserire un processo demone ulteriore, si è posto il vincolo che

nella macchina di installazione del nuovo ambiente sia attivo almeno un place o un gateway. Il Comando “CattivaPlace” richiede all’ambiente a cui è inviato di eseguire una primitiva (Tabella 12) particolare, che consente l’attivazione del nuovo place (o gateway).

```
command={"java","AgentSystem.Main",portPlace,dominio,place,portGate,hostGate};
Runtime r=Runtime.getRuntime();
        try {
             p = r.exec(command);
         } catch (Exception e) {…}
```

Tabella 12 Primitiva “exec” per l’attivazione di un place.

4.6.4 Possibili soluzioni alternative: vantaggi e svantaggi

La soluzione descritta in precedenza prevede che ogni modifica sia propagata in tutto il sistema in tempo reale e che ogni place al momento dell’attivazione sia completamente ignaro della configurazione del suo dominio.

Una soluzione alternativa potrebbe alleggerire l’attivazione dei place: ogni place potrebbe memorizzare in una propria directory la struttura attuale del sistema ogni volta che questa viene modificata, così che, alla prossima attivazione, non sia necessario utilizzare il protocollo di richiesta al gateway. Durante il periodo in cui un place è disattivo, le eventuali modifiche del sistema potrebbero essere mantenute nel gateway, che avrebbe il compito di aggiornare il place non appena si riattiva.

Lo svantaggio di questa soluzione è che il ruolo del gateway risulterebbe appesantito da una struttura aggiuntiva in cui tenere traccia di quali place non siano stati avvisati delle modifiche,

mentre il vantaggio è costituito dal velocizzare l'attivazione dei place.

Si può quindi affermare che la soluzione appena descritta, sia particolarmente adatta per sistemi in cui i place si attivano e disattivano con una frequenza tale da giustificare la presenza di una gestione aggiuntiva nel gateway.

Nel caso di sistema ad agenti mobili, gli ambienti di esecuzione degli agenti (ovvero i place) è bene che si disattivino solo se strettamente necessario e che vi sia una struttura efficace per attivare modifiche dinamiche che possano ampliare o ridurre la configurazione del sistema

4.7 La comunicazione

Ogni tipo di coordinazione tra le entità del sistema, tra place o tra gateway, ma anche tra agenti, è realizzata attraverso un protocollo di più basso livello che sfrutta la connessione fisica tra gli ambienti di esecuzione. Questa connessione realizza un canale di comunicazione virtuale, su cui si spediscono messaggi di coordinazione. I protocolli utilizzabili per la comunicazione remota sono TCP (Transport Control Protocol) e UDP (Users Datagram Protocol).

4.7.1 Protocolli di comunicazione: TCP e UDP

Il protocollo TCP è un protocollo connection-oriented: consente di realizzare una astrazione di canale virtuale, costruita attraverso una prima fase di negoziazione in cui i due punti estremi, end-point (socket), si accordano su alcuni riferimenti e la suddivisione del flusso di informazioni in pacchetti distinti avviene in modo trasparente all'utilizzatore. TCP è progettato per una trasmissione

di dati affidabile: se una informazione è persa o danneggiata durante la trasmissione, TCP si occupa di ritrasmettere i dati mancanti. Se i pacchetti non arrivano nell'ordine giusto, li riordina in modo trasparente all'utente.

I vantaggi offerti da TCP presentano come inconveniente la limitata velocità; l'alternativa è rappresentata dal protocollo UDP, che risulta molto veloce e che offre quindi migliori prestazioni, ma è privo di affidabilità, poiché si solleva dal compito di controllare l'ordine dei pacchetti o la dispersione di alcuni di essi. UDP è un protocollo connection-less: non è presente l'astrazione di canale virtuale, ma è compito dell'utente suddividere il flusso di informazioni in pacchetti da spedire all'end-point (socket) remoto.

Il sistema SOMA, è realizzato completamente in Java che impone una limitazione sul numero di canali di comunicazione aperti contemporaneamente (al massimo 20), se basati su TCP: questo ostacolo impone l'uso del protocollo UDP, sul quale invece, non è presente nessuna limitazione, visto che l'obiettivo è realizzare un sistema scalabile.

Esiste una considerazione aggiuntiva che può guidare alla scelta di UDP come protocollo di comunicazione, specialmente quando il contesto è distribuito su larga scala: l'ambiente di sviluppo del sistema ad agenti mobili, può presentare la situazione in cui i place di uno stesso dominio siano fisicamente implementati su macchine remote, separate anche da connessioni non stabili o intermittenti. Come sappiamo, i place di uno stesso dominio devono avere collegamenti fissi, e se sono realizzati con TCP, qualora uno dei due punti di connessione cada, l'altro lo rileva immediatamente, per la caratteristica di essere connection-oriented. Da un lato questo è un vantaggio, poiché consente di avere un aggiornamento sempre attuale della situazione remota, ma impone anche di intraprendere azioni reattive immediate (come la chiusura inevitabile del canale di comunicazione), mentre se la disconnessione non viene segnalata, come accade con UDP, non è eseguita nessuna

operazione. Nell'ipotesi di lavorare su un canale intermittente, la necessità di chiudere il canale di comunicazione può presentarsi frequentemente, anche se, nel frattempo, il canale in realtà non si è utilizzato. Quindi, con UDP, in cui l'astrazione di canale non è presente, se la successiva ripresa del place remoto, avviene prima di qualunque interazione con quell'ambiente, si è risparmiata l'esecuzione delle azioni necessarie alla chiusura della connessione, altrimenti il costo in termini di operazioni reattive è uguale.

4.7.2 La comunicazione di basso livello in SOMA

Nella prima versione, il sistema realizzava la comunicazione con TCP, ma la limitazione imposta da Java sul numero di socket aperte e la crescente necessità di canali di comunicazione, come strumento indispensabile per la coordinazione tra i place, ha imposto la modifica del protocollo da utilizzare. Il passaggio da TCP a UDP è stato effettuato in modo trasparente, mantenendo la compatibilità totale del sistema, e realizzando un pacchetto autonomo (il package "Socket") da inserire come intermediario tra le classi core di SOMA già presenti e le classi core di Java.

In realtà, è indispensabile sottolineare che il pacchetto di comunicazione di basso livello che realizza il protocollo TCP, attraverso lo scambio di datagrammi, non è totalmente trasparente: esiste una osservazione da non dimenticare. A causa di problemi implementativi, spiegati nei punti 3 e 4 del paragrafo successivo, *quando si trasmettono oggetti serializzati, si devono sostituire i metodi writeObject e readObject (delle classi ObjectOutputStream e ObjectInputStream) con writeObj e readObj.*

Per realizzare una sostituzione di protocollo trasparente, analizziamo in dettaglio le differenze tra TCP e UDP, per capire

come implementare le caratteristiche fondamentali del primo con gli strumenti del secondo.

4.7.2.1 Apertura di una connessione

Con TCP, una richiesta di connessione va a buon fine solo se è già presente un servitore disposto ad esaudirla e come conseguenza si crea un collegamento diretto tra i due interlocutori. In UDP invece, non è presente alcuna nozione di ServerSocket (cioè un end-point che svolga il ruolo di server): si usa lo stesso tipo di socket sia per spedire messaggi che per ricevere richieste di connessione.

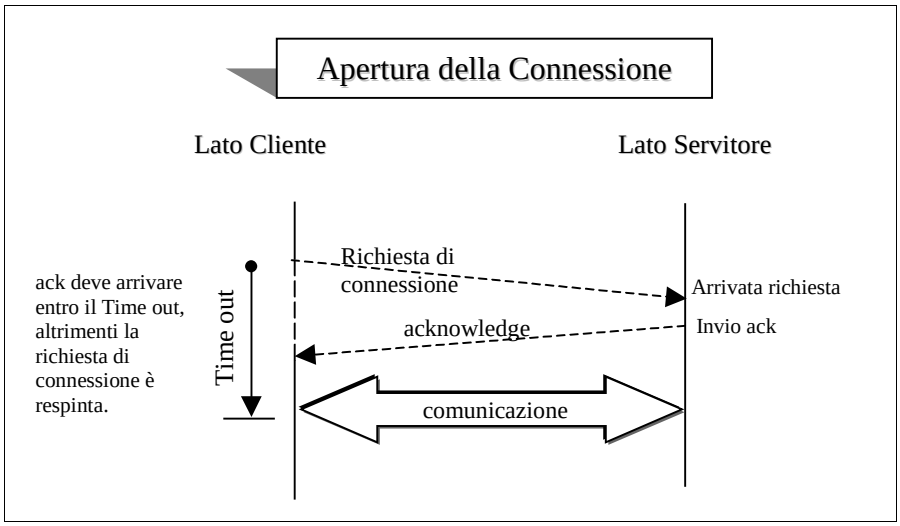


Figura 4-4 Protocollo di apertura di una connessione.

Con le socket Datagram è stata realizzata la richiesta di connessione, nonostante non sia presente la distinzione tra il lato cliente e il lato servitore: per considerare soddisfatta tale richiesta, si deve attendere il consenso da parte di chi svolge il ruolo del servitore. Questa autorizzazione si manifesta con un messaggio di

risposta, acknowledge (ack) e se non arriva entro un tempo prefissato, la connessione non si considera aperta (Figura 4-4).

4.7.2.2 Connessione dedicata

In TCP, ogni socket è dedicata ad una singola connessione, cioè un collegamento riservato solo ai due interlocutori che lo hanno creato. In UDP invece, una socket è usata per ricevere dati da qualunque mittente e per spedirne a qualunque destinatario.

Per simulare, con una socket Datagram, una connessione dedicata esclusivamente a due entità, si impone il vincolo di comunicare sempre e solo con lo stesso interlocutore. Ad ogni socket UDP, si aggiunge tra le sue caratteristiche l'identità dell'end-point a cui inviare e da cui ricevere messaggi: così non è più necessario specificare il destinatario e il mittente del datagramma ed ogni connessione si considera dedicata a solo due interlocutori, esattamente come accade con i canali di comunicazione TCP.

4.7.2.3 Protocollo di affidabilità

Come è noto, il protocollo UDP non presenta nessun criterio di affidabilità: non presenta nessuna astrazione di canale virtuale e lo scambio di messaggi avviene senza garantire il corretto ordine d'arrivo, né l'effettivo arrivo. Dal momento che l'utilizzo del protocollo UDP deve essere mascherato all'utente, l'obiettivo sarebbe quello di implementare lo stesso protocollo di affidabilità che caratterizza TCP.

È necessario, quindi, fornire affidabilità in termini di controllo dell'ordine dei messaggi e verifica della effettiva ricezione dei messaggi da parte del destinatario. Questi obiettivi si possono ottenere nel modo descritto di seguito:

- per controllare l'ordine di arrivo, si dovrebbero numerare tutti i datagrammi spediti dal mittente ed operare una ricostruzione completa del messaggio, seguendo, non l'ordine d'arrivo, ma la numerazione imposta alla partenza;
- per verificare l'effettiva ricezione dei datagrammi, si potrebbe introdurre il concetto di finestra scorrevole (sliding window) [Cor98]: la ricezione di un certo numero di informazioni (che sono in realtà quantificate con un numero prestabilito di byte, ma che per semplicità indicheremo in generale col termine "messaggio"), deve essere convalidata da una conferma (acknowledge), che informa il mittente del buon esito di quella parte di comunicazione. La conferma può includere anche l'avvenuta ricezione di un numero maggiore di informazioni: il numero massimo di messaggi che possono essere convalidati con un solo acknowledge, è deciso a priori e costituisce la dimensione della finestra scorrevole.

Il protocollo di affidabilità implementato nel package "Socket", ha l'obiettivo di non appesantire il compito svolto dagli ambienti tra cui avviene la comunicazione. Si è scelto quindi di realizzare una finestra scorrevole di dimensione minima, cioè unitaria: una dimensione maggiore infatti, avrebbe costretto il destinatario (e il mittente) a mantenere una copia di ogni messaggio ricevuto (e inviato) non ancora confermato, per poter riconoscere eventuali ritrasmissioni causate della perdita dell'acknowledge (e per sapere quale messaggio inviare nuovamente se la conferma non è ricevuta entro un tempo prestabilito, detto time out).

Nel caso in cui il messaggio inviato non sia ricevuto, oppure nel caso in cui la conferma non raggiunga la destinazione stabilita, è necessario ritrasmettere il messaggio originale: questa operazione può avvenire un numero di volte N, da stabilire a priori. Per esempio, se il destinatario del messaggio non è momentaneamente attivo, è importante che il numero N non sia elevato, perché il

tempo speso per le N ritrasmissioni risulta, in definitiva, inutile. D'altro canto, se il traffico della rete all'interno della quale avviene la comunicazione è elevato, aumentano le probabilità di perdita del messaggio, quindi è bene che il numero N di tentativi non sia troppo piccolo. Nel package "Socket", si è scelto di dimensionare il numero N uguale a 3.

Inoltre, la dimensione unitaria della finestra scorrevole, evita che si verifichi l'arrivo disordinato di messaggi: infatti non viene spedito nessuno messaggio finché non vi è la certezza che quello precedente è arrivato a destinazione e così non risulta necessaria la numerazione dei messaggi.

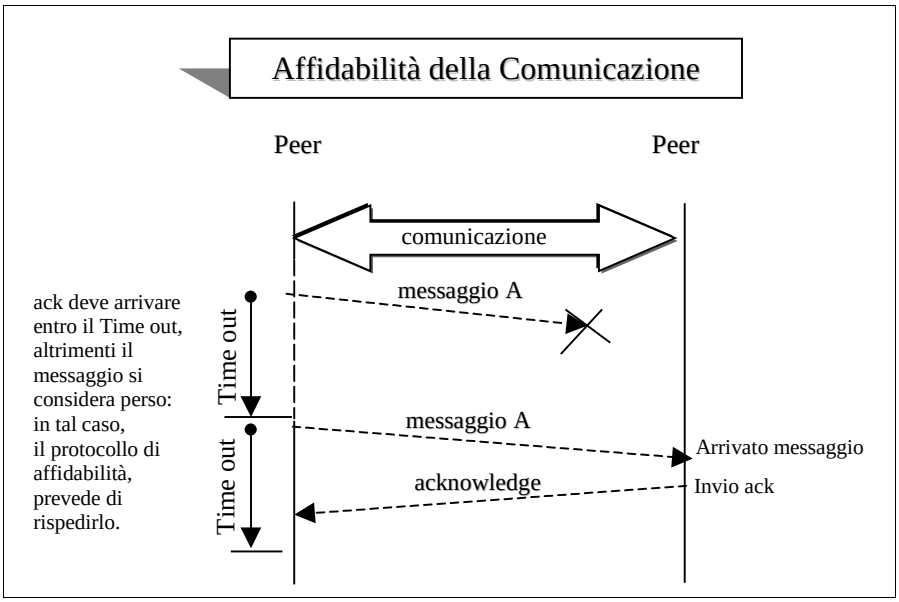


Figura 4-5 Protocolli di affidabilità per le comunicazioni in SOMA.

4.7.2.4 Sincronizzazione

Nelle applicazioni di rete è spesso presente un processo particolare, detto demone, con l'unico compito di attendere i messaggi in arrivo su una determinata socket.

Se un canale di comunicazione, su cui è presente un demone, è utilizzato anche per inviare messaggi (oltre che per riceverli), teoricamente non si presenterebbe alcun problema, perché i campi d'azione dei due processi sono distinti: uno utilizza il canale di ingresso e l'altro quello di uscita.

Nel modello descritto nei punti precedenti, si è scelto di inviare una risposta ad ogni messaggio, sulla stessa socket da cui si è ricevuto. Questo crea una situazione di condivisione del canale, anche quando le competenze dovrebbero essere distinte.

Una soluzione potrebbe essere utilizzare una socket differenziata per i messaggi di conferma, ma se l'applicazione è complessa si rischia di appesantirla eccessivamente, raddoppiando le vie di comunicazione (si ricordano i numerosi canali di comunicazione necessari per la coordinazione dei place, descritti nel paragrafo 4.2).

La soluzione scelta è quella di realizzare una sincronizzazione tra i processi in attesa di messaggi sullo stesso canale: un gestore cattura le informazioni in ingresso e distingue se sono dati o conferme di ricezione avvenuta (acknowledge), e sblocca solo il processo interessato.

4.7.3 Classi implementate

Abbiamo costruito un package che si sostituisce alle classi core di Java in modo trasparente, permettendo di usare la trasmissione di datagrammi in applicazioni già esistenti basate sul protocollo TCP. Per mascherare la presenza di UDP a livello sottostante, è

necessario ridefinire le classi indispensabili per la comunicazione, in modo che mantengano la stessa interfaccia offerte da TCP.

È stato considerato un gruppo di classi, e di queste un insieme limitato di metodi, che sono riportati nel seguito.

1. Classe Socket

   È stata ridefinita la classe Socket per realizzare la comunicazione attraverso Datagrammi in modo trasparente all'utente.

Il costruttore ha come argomenti l'indirizzo (InetAddress) del server remoto e il numero di porta su cui questi attende nuove connessioni:

public Socket(InetAddress ind, int porta) throws SocketException, IOException

Sono stati realizzati i metodi che restituiscono il canale di comunicazione d'ingresso e d'uscita della socket:

public OutputStream getOutputStream() throws IOException
public InputStream getInputStream() throws IOException

Sono stati realizzati i metodi che restituiscono la porta e l'indirizzo IP, remoti, a cui la socket è collegata:

public int ***getPort****()*
public InetAddress ***getInetAddress****()*

Infine è possibile invocare il metodo per chiudere la socket:

public void ***close****() throws IOException*

2. Classe ServerSocket

È stata ridefinita la classe ServerSocket per realizzare la comunicazione attraverso Datagrammi in modo trasparente all'utente.

Il costruttore ha come argomento il numero di porta su cui attende richieste di connessione:

public ***ServerSocket****(int porta) throws IOException*

È stato realizzato il metodo che sospende il processo in attesa di nuove richieste di connessione, e che restituisce la socket con cui gestire in seguito la comunicazione:

public Socket ***accept****() throws IOException*

Infine è possibile invocare il metodo per chiudere la socket:

public void ***close****() throws IOException*

3. Classe ObjectOutputStream

È stata realizzata la classe ObjectOutputStream, che eredita da java.io.ObjectOutputStream, per spedire attraverso socket di tipo datagram oggetti serializzati in modo trasparente all'utente.

Il costruttore ha come argomento il canale di uscita su cui si scrive:

public ***ObjectOutputStream****(OutputStream s) throws IOException*

Non è stato possibile ridefinire il metodo "writeObject", necessario per scrivere oggetti serializzati, perché dichiarato "final".

Per risolvere il problema si è scelto di realizzare un metodo con nome simile "**writeObj**" da utilizzare al posto del precedente, quando è necessario comunicare sul canale di uscita di una socket:

*public void **writeObj**(Object o) throws IOException*

4. Classe ObjectInputStream

In modo analogo al caso precedente, è stata realizzata la classe ObjectInputStream, che eredita da java.io.ObjectInputStream, per ricevere attraverso Datagrammi oggetti serializzati in modo trasparente all'utente.

Il costruttore ha come argomento il canale di ingresso su cui si legge:

*public **ObjectInputStream**(InputStream s) throws IOException,*
StreamCorruptedException

Non è stato possibile ridefinire il metodo "readObject", necessario per leggere oggetti serializzati, perché dichiarato "final".

Per risolvere il problema si è scelto di realizzare un metodo con nome simile "**readObj**" da utilizzare al posto del precedente, quando è necessario comunicare sul canale di ingresso di una socket:

*public Object **readObj**() throws OptionalDataException,*
ClassNotFoundException, IOException

4.7.4 Un esempio: la comunicazione di oggetti

È interessante notare il differente modo in cui avviene lo scambio di oggetti serializzati (cioè che implementano l'interfaccia "Serializable") con l'uso di TCP e UDP.

Nel sistema SOMA, lo scambio di oggetti è molto frequente: infatti, anche gli agenti stessi sono rappresentati come oggetti e la migrazione di un agente include, in una visione di basso livello, lo scambio di un oggetto serializzato tra gli ambienti coinvolti.

Si riporta di seguito come spedire un oggetto (serializzabile) nei due casi: TCP e UDP.

1. Per comunicare un oggetto con TCP, si deve estrarre lo stream di uscita della socket e su questo si deve scrivere l'oggetto serializzato:

```
Socket AgentSock;
OutputStream Asout = AgentSock.getOutputStream();
ObjectOutputStream AgentOut = new ObjectOutputStream(Asout);
AgentOut.writeObject( (Object) ag );
```

2. Per comunicare un oggetto con UDP, l'oggetto deve essere convertito in un array di byte: può essere una operazione difficile se la struttura dell'oggetto è complicata. La soluzione adottata evita un oneroso dispendio di codice: la Figura 4-6 mostra il flusso dei dati quando l'oggetto è trasmesso con Datagrammi. I passi da seguire per l'implementazione sono i seguenti:
   - creare un oggetto ByteArrayOutputStream;
   - costruire un ObjectOutputStream, usando lo stream del passo precedente;
   - scrivere l'oggetto da spedire usando il metodo writeObj (e non writeObject, per i motivi descritti nel paragrafo precedente al punto 3)
   - Recuperare l'array di byte dal contenuto del ByteArrayOutputStream, usando il metodo toByteArray;
   - costruire il pacchetto da spedire usando l'array ottenuto al passo precedente;
   - spedire il pacchetto finale, che contiene l'oggetto serializzato.

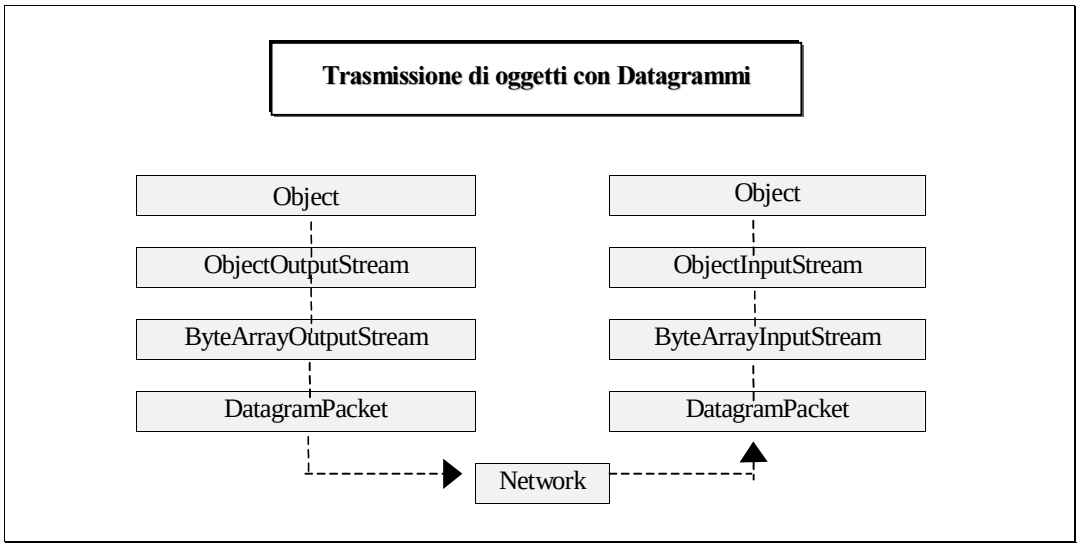


Figura 4-6 Trasmissione di oggetti serializzati con i datagrammi.

Ecco infine la formulazione del codice descritto nei passi precedenti:

si crea uno stream e su questo si scrivere l'oggetto serializzato:

```
DatagramSocket AgentSock;
ByteArrayOutputStream ASout = new ByteArrayOutputStream ();
ObjectOutputStream AgentOut = new ObjectOutputStream (ASout);
AgentOut.writeObj ( (Object) ag);
```

si crea un array di byte con il contenuto dello stream e con questo si costruisce il pacchetto da spedire:

```
byte[ ] sendBuf = ASout.ByteArray();
DatagramPacket packet = new DatagramPacket (sendBuf,
      sendBuf.length);
AgentSock.send(packet);
```

La lettura di un oggetto avviene in modo simmetrico al caso della scrittura.

Conclusioni

La progettazione di una architettura di supporto per sistemi ad agenti mobili ci ha permesso di valutare concretamente i vantaggi di questo paradigma, particolarmente adatto allo sviluppo di applicazioni distribuite su larga scala. Il supporto si è rivelato flessibile, efficiente e scalabile: anche in caso di guasto delle connessioni, la caratteristica di autonomia dell'agente ha concesso di eseguire sulla macchina locale e di attendere il ripristino del collegamento per raggiungere la destinazione successiva. La possibilità che ha un agente di spostarsi localmente alla risorsa cui vuole accedere ha consentito un incremento delle prestazioni e una diminuzione del traffico di rete. Inoltre, nel supporto realizzato è molto facile introdurre dinamicamente nuove località, su cui gli agenti possono spostarsi senza avvertire alcuna differenza rispetto alle località previste all'atto della configurazione iniziale.

La realizzazione di una architettura dinamica, ha richiesto l'inserimento di una struttura gerarchica tra le località del sistema: al livello più basso si sono inseriti gli ambienti di esecuzione degli agenti (i place), raggruppati per affinità logiche o fisiche (i domini); all'interno di ogni dominio è presente un gateway, che coordina, da un livello gerarchico superiore, i place interni; infine all'apice della struttura gerarchica risiede il Supervisor, che gestisce gli aggiornamenti dinamici del sistema.

Nel progetto di una architettura dinamica è risultato indispensabile l'inserimento di un monitoraggio finalizzato alla gestione delle astrazioni di località: la realizzazione di strumenti del Network Management ha coinvolto l'uso di più agenti coordinati, distribuiti tra gli ambienti da controllare e capaci di offrire un insieme di funzionalità flessibile e facilmente estensibile.

L’architettura è stata realizzata completamente in linguaggio Java, data l’importanza di avere un sistema ad agenti portabile in un contesto di reti aperte ed eterogenee. Il limite di Java è quello di offrire solo una mobilità di tipo debole, che vincola gli agenti a suddividere il loro flusso di esecuzione in moduli da eseguire nei diversi ambienti in cui si spostano. Tuttavia, questo limite costringe il programmatore di applicazioni ad agenti mobili a redigere il codice in modo strutturato e lineare, e non limita in modo significativo il suo potere espressivo.

Dopo questa esperienza possiamo dedurre che il modello ad agenti mobili sia una soluzione molto promettente per la progettazione e per l’implementazione di applicazioni distribuite su larga scala. Le potenzialità possono essere sfruttate appieno, però, solo se il supporto ad agenti fornisce anche un sistema di protezione e di sicurezza completo, flessibile ed affidabile, che sia in grado di proteggere sia l’agente che l’ambiente di esecuzione da attacchi o manomissioni.

Bibliografia

[ATP97] D.B. Lange, Y. Aridor: “Agent Transfer Protocol – ATP/0.1”, IBM Tokyo Research Laboratory.

[BalGP] M. Baldi, S. Gai, G.P. Picco: “Exploiting Code Mobility in Decentralized and-Flexible Network Management”, Mobile Agents: 1st International Workshop, MA’97, vol.1219, pp.13-26, Lecture Notes on Computer Science, April 1997.

[Bau97] J. Baumann et al.: “Comunication Concept for Mobile Agent Systems”, Mobile Agents, Proc. of the 1st International Workshop, MA’97, Springer 1997.

[CaGe89] N. Carriego, D. Gelernter: “Linda in Context”, Comunication of ACM, vol. 32, n°4, April 1989.

[CarPV97] A. Carzaniga, G.P. Picco, G. Vigna, “Designing Distributed Applications with Mobile Code Paradigms”, in Proc. of the 19th Int. Conf. on Software Engineerine (ICSE’97), R. Taylor, Ed. 1997, pp. 22–32, ACM Press.

[CasFSD90] J. D. Case, M. Fedor, M. L. Schoffstall, and C. Davin. “Simple Network Management Protocol”, RFC 1157, May 1990.

[CasMRW96] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. “Structure of Management Information for 2 of the Simple Network Management Protocol” RFC 1902, January 1996

[CheHK95] “Mobile Agents: Are They a Good Idea?” David Chees, Colin Harrison, Aaron Kershenbaum.

[Coc98] Fabrizio Coccia, “Modelli di sicurezza per sistemi ad agenti mobili”, tesi di Laurea presso l’Università di Ingegneria Informatica di Bologna, 1998.

[Con98] “Mobile Agent Computing: a White Paper”, Mitsubishi Electric ITA.

[Cor98] Antonio Corradi, Corso di “Reti di Calcolatori”, 1998.

[CORBA] R. Orfali and D. Harkey: “Client/Server Programming with JAVA and CORBA” WileyComputer Publishing, 1997.

[CouDK94] George Coulouris, Jean Dollimore, Tim KindBerg, “Distribuited Systems: Concepts and Design”, Addison-Wesley

[CugGPV97] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, Giavanni Vigna, “Analyzing Mobile Code Languages”.

[FugPV98] A. Fuggetta, G.P. Picco and G. Vigna “Understanding Code Mobility”, IEEE Transactions on Software Engineering, 1998.

[GenMag94] Jim White: “Telescript Tecnology: The Fondation for Electronic Marketplace”, General Magic Inc. Mountain View (CA), USA, 1994.

[Gol93] G. Goldszmidt: “On Distributed System Management”, Computer Science Department, Columbia University, Technical Report, October 1993.

[GolY93] G. Goldszmidt, Y. Yemini: “Evaluating Management Decision via Delegation”, Preceedings of IFIP International Symposium on Network Management, April 1993

[GolY95] G. Goldszmidt, Y. Yemini: “Distribuited Management by Delegation”, Preceedings of the 15th International Conference on Distribuited Computing System, June 1995.

[Gong97] Li Gong, M. Mueller, H. Prafullchandra and R. Schemers: “Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2”, in Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, Dec. 1997.

[Gong98] Li Gong: “Java Security Architecture (jdk1.2): Draft Document”, March 1998.

[Har94] M.Harchol-Balter: “Process Lifetimes are Not Exponential, more like 1/T: Implication Dynamic Load Balancing” Tech Report n.UCB/CSD-94-826 University of California, Berkeley, August 1994.

[IBM96] “Programming Mobile Agents in Java, A White Paper”, Danny B. Lange and Daniel T. Chang, September 1996

[ISO91] OSI. ISO 9595 Information Tecnology, Open System Interconnection, Common Management Information Protocol Specification, 1991.

[JAVA] http://www.sun.com

[Java] K.Arnold, J. Gosling: “The Java Programming Language”, Addison-Wesley 1996.

[Jworld] http://www.javaworld.com

[KauPS95] Charlie Kaufman, Radia Perlman, Mike Speciner, “Network Security, Private Comunication in a Public World”.

[Nie98] “An Introduction to Java”, Prof. Oscar Nierstrasz, Software Composition Group Institut fur Informatik (IAM) Universitat Bern

[Nut94] M. Nuttal, “Survey of system providing process or object migration”, Tech Rep. Doc 94/10, Dept. of Computing, Imperial College, May 1994.

[OBJ97a] “Voyager Core Package Thecnical Overwiew”, ObjectSpace press (March 1997)

[OBJ97b] “ObjectSpace Voyager, General Magic Odyssey, IBM Aglets: a Comparison”, ObjectSpace press (June 1997)

[Odissey98] “Introduction to the Odyssey API”, General Magic, 1998.

[OOE95] Object-Oriented Experiences and Future Trends, Special Issue di Comunication of ACM, Vol. 38, N°10, Ottobre 1995.

[RMI98] “Remote Method Invocation for Java”, Javasoft Corporation, http://chatsubo.javasoft.com/current/rmi/index.html.

[SOMA98] Sistema progettato nell’ambito del "Project Design Methodologies and Tools of High Performance Systems for Distributed Applications" fondato dal Ministero dell’Università e della Ricerca Scientifica e Tecnologica (MURST). Reperibile dal 1998 presso: http://www-lia.deis.unibo.it/Software/MA/.

[Tan94] Andrew S. Tanenbaum “I moderni sistemi operativi”, Prentice Hall International, Jackson Libri.

[Tar98] F. Tarantino, tesi di Laurea presso l’Università di Ingegneria Informatica di Bologna, 1998.

[WahLAS93] R. Wahbe, S. Lucco, T.E. Anderson and S.L. Graham: “Efficient software-based fault isolation”, in Proceedings of the 14th ACM Symposium on Operating Systems Principles, 1993.