Indice

Introduzione

Lo sviluppo di Internet come mezzo di comunicazione globale ha portato a una grande domanda di applicazioni distribuite per le più diverse necessità. La progettazione di applicazioni destinate ad ambienti aperti presenta però diverse problematiche, alcune delle quali strettamente legate al modello di programmazione. L'approccio classico, basato sul modello Client/Server, necessita una rigorosa e rigida organizzazione dei ruoli per ogni entità partecipante; questo vincolo può creare grosse difficoltà nello sviluppo di applicazioni complesse in special modo nel caso di specifiche forti sulla scalabilità e sulla tolleranza ai guasti. Questi problemi hanno portato alla ricerca di nuove soluzioni e nuovi modelli in grado di semplificare la programmazione senza per questo motivo nascondere le risorse sotto forti astrazioni spesso inefficienti.

Una delle proposte più recenti si basa sul concetto di ***Agente Mobile:*** un **Agente** è un'entità attiva, un processo in grado di assumere dinamicamente il ruolo di Cliente o Servitore a seconda delle necessità. Questo Agente ha la proprietà fondamentale di essere **Mobile**, cioè è in grado di spostarsi da un nodo a un altro della rete per eseguire compiti specifici, attuabili solo localmente. Realizzare applicazioni come insieme di Agenti che si coordinano per ottenere un obbiettivo comune, risulta più intuitivo e più semplice perché non è necessario scomporre gli algoritmi in una parte per il Client e una parte per il Server.

Gli Agenti permettono maggiore flessibilità e riducono i costi di coordinamento; per questo motivo le aree applicative che possono trarre vantaggio da questo modello sono molteplici: L'*Information Retrieval* e il *Network Management* sono alcuni esempi. Per la realizzazione di qualsiasi

applicazione è però fondamentale fornire agli Agenti gli strumenti opportuni per interagire con il sistema sottostante; è importante quindi considerare attentamente la possibilità di realizzare funzioni di monitoraggio e rilevazione dello stato del sistema.

Il Capitolo 1 analizza i problemi per lo sviluppo di applicazioni distribuite per ambienti dinamici ed eterogenei; descrive inoltre le peculiarità della mobilità evidenziandone i problemi ed i vantaggi.

Attraverso il confronto tra alcuni sistemi attualmente disponibili che permettono la programmazione ad Agenti il Capitolo 1 poi evidenzia quali caratteristiche debba avere un supporto per la programmazione in presenza di mobilità .

Il Capitolo 2 descrive la problematica del *Monitoraggio Distribuito*, ingrediente fondamentale per la *percezione del mondo* da parte degli Agenti e le tecniche per realizzare un supporto di monitoraggio con particolare attenzione alla rilevabilità di determinati eventi.

Quando si parla di Internet e Sistemi Aperti, il linguaggio di programmazione di riferimento è ormai Java considerato uno standard *de facto*. È un linguaggio ad oggetti interpretato ideato esplicitamente per la programmazione di rete. Il Capitolo 3 analizza Java dal punto di vista della mobilità, descrivendone le caratteristiche, i limiti e le possibilità che offre.

L'obbiettivo di questo lavoro di analisi è la realizzazione di un ambiente per l'esecuzione di Agenti in Java; il Capitolo 4 presenta la realizzazione di un supporto interamente scritto in Java, analizzando i modelli e le astrazioni che vengono forniti ai programmatori di Agenti; inoltre descrive, con un livello di dettaglio superiore, i particolari implementativi più significativi.

Come verifica delle funzionalità del supporto progettato è stata realizzata un'applicazione di *Network Management*, composta da un monitor distribuito per la rilevazione dello stato dei nodi e un insieme di strumenti di amministrazione come l'installazione remota di pacchetti software oppure il backup distribuito di dati residenti su macchine diverse.

1 Evoluzione nel Progetto del Software di rete

La diffusione di grandi sistemi in rete anche nelle piccole medie aziende e lo sviluppo di Internet come mezzo di comunicazione per fini commerciali e di coordinamento, ha portato una grande diffusione di applicazioni di rete e distribuite per le più diverse necessità. Il design e la progettazione di tali applicazioni sono però costosi, poiché le tecniche ed i supporti a disposizione sono ancora molto complessi e non esistono strumenti sufficientemente flessibili. Una applicazione distribuita, infatti, ha necessità peculiari che non sono facilmente ottenibili senza un adeguato supporto a livello di Sistema Operativo o di un Middleware specifico. Queste necessità possono essere riassunte nelle seguenti proprietà:

- **Apertura:** una applicazione deve essere in grado di adattarsi a variazioni anche sostanziali del sistema su cui opera; un sistema in rete infatti è intrinsecamente variabile come numero di partecipanti, posizione delle risorse specifiche, configurazione della rete e di tutti quei particolari che ne caratterizzano la gestione. Una qualità necessaria per ottenere questo tipo di adattabilità è l'**eterogeneità**; un sistema aperto, infatti, comprende unità diverse tra loro, sia dal punto di vista software che hardware: queste differenze non devono essere un ostacolo ed è quindi necessario tenerne conto. Possedere questa qualità è estremamente difficile, ma lo sviluppo di Internet, sistema aperto per eccellenza, la ha resa non solo importante, ma anche necessaria.

- **Scalabilità:** l'efficienza e l'efficacia dell'applicazione non deve essere dipendente dalla grandezza del sistema; deve quindi essere in grado di

adattarsi sia a piccoli sistemi che a grandi reti articolate. In particolare, deve essere in grado di sopportare un eventuale *upgrade* del sistema per sopravvenute necessità, senza interventi costosi o complessi.

- **Tolleranza ai guasti:** la probabilità che in un sistema ci siano guasti è direttamente proporzionale al numero di componenti; è evidente quindi che la probabilità che ci siano guasti di vario genere e natura nel sistema è elevatissima. Una applicazione deve essere messa in grado di sopravvivere ai guasti di minore entità e mantenere i suoi dati consistenti in caso di errori più gravi.

Queste proprietà sono fondamentali per la qualità del software prodotto e spesso non sono ancora ottenibili in modo automatico da un supporto di programmazione, ma devono essere realizzate dall'utente. Nel seguito si analizzeranno le caratteristiche ed il design delle applicazioni distribuite; si cercherà di evidenziare quali e come siano le qualità fondamentali che deve possedere un supporto di sviluppo sia dal punto di vista del linguaggio che del Middleware necessario.

1.1 Progetto di applicazioni distribuite

Nello sviluppo di un Software, le fasi di progetto possono essere schematizzate nel seguente modo:

- Definizione degli obbiettivi e progettazione dello schema di principio.
- **Progettazione e parallelizzazione:** definizione della struttura del programma con particolare attenzione alla scomposizione del problema in entità che saranno eseguite parallelamente, sia per usufruire di risorse remote specifiche (come ad esempio uno specifico database oppure una particolare periferica) che per migliorare le prestazioni complessive.
- **Sviluppo del Codice:** fase di scrittura ed articolazione del codice; si risolvono i problemi di natura algoritmica e si definisce completamente il coordinamento fra le entità che devono eseguire concorrentemente e parallelamente.
- **Debugging:** si verifica la correttezza di quanto prodotto ed il rispetto da parte del programma delle caratteristiche desiderate; in ambiente parallelo, sono necessari strumenti che permettano il controllo della coordinazione tra entità e ne determini eventuali lacune.
- **Mapping:** questa fase è completamente assente nello sviluppo di applicazioni tradizionali; si tratta di decidere dove allocare le varie entità che compongono il programma, affinché l'uso delle risorse a disposizione sia equilibrato ed efficiente. Questa fase è estremamente complessa e dipende non solo dalla applicazione e dalle sue necessità, ma anche dall'architettura del sistema.

1.1.1 Il Problema del Mapping

Una generica applicazione può essere vista come un insieme di entità indipendenti (da ora in poi saranno dette per semplicità **EU** - *Execution Unit*) che, coordinate, eseguono per raggiungere un obbiettivo. Il *problema del Mapping* può essere quindi così definito:

Siano R_i con $i=1..n$ le risorse a disposizione nel sistema, EU_j $j=1..m$, le EU che devono essere eseguite, N_k $k=1..nn$, i possibili nodi dove possono essere allocate le EU per utilizzare le diverse risorse; il problema del mapping consiste nel trovare m destinazioni N_k per EU_j tali che l'efficienza nell'uso delle risorse R_i sia massima o, analogamente, che il tempo complessivo di esecuzione sia minimo. La soluzione ha una forte dipendenza dal tempo: varia a seconda delle necessità delle singole EU e quindi si può immaginare, per ogni EU_j, una successione di destinazioni, un percorso che porta alla massima efficienza $\mathbf{P}_j=[N_0,N_a,N_b,..]$.

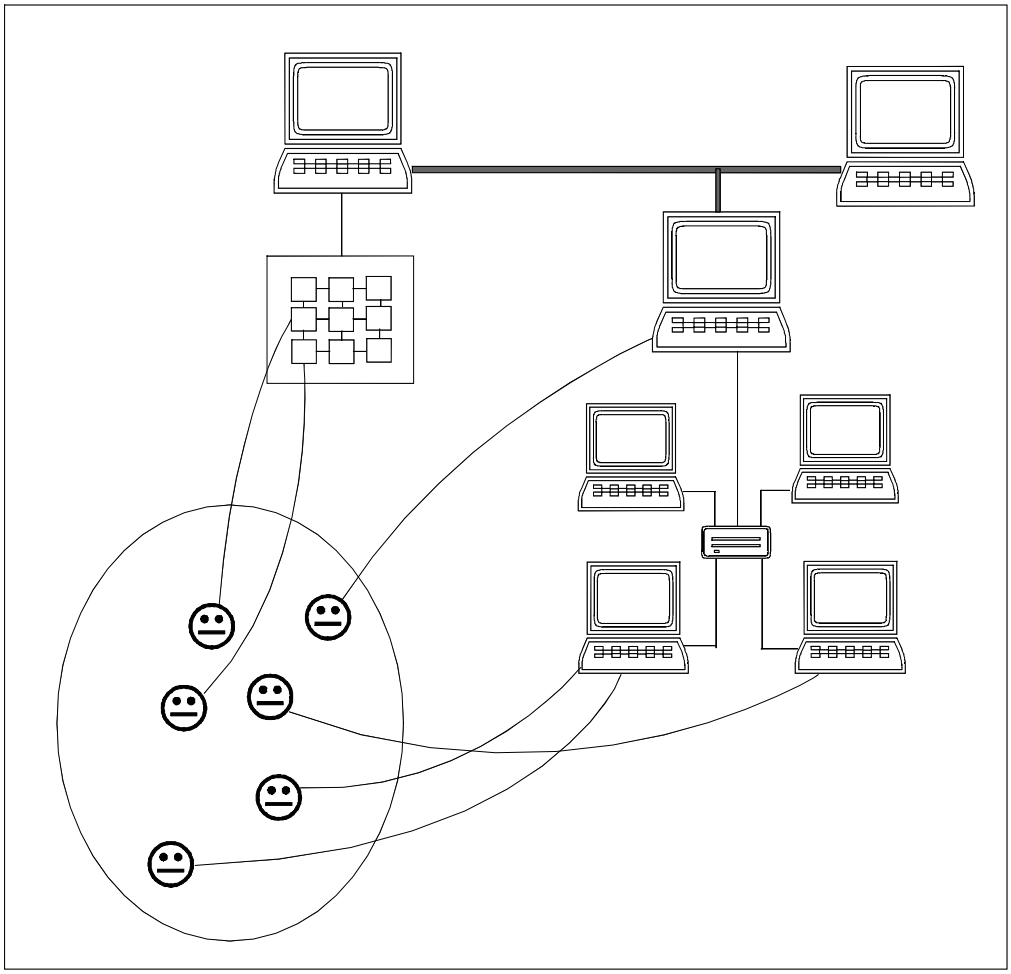

Figura 1: Il problema del Mapping è strettamente dipendente dalla tipologia della rete e dalle caratteristiche della applicazione; al crescere della complessità diventa sempre più difficile trovare la soluzione ottima.

La configurazione ottima può essere rappresentata come l'insieme dei vettori percorso $\mathbf{P}_j \quad \forall j \in [1;n]$, questa forma evidenzia la dipendenza della configurazione non solo dalle EU ma anche dalla successione di locazioni date da $\mathbf{P}_j$.

La soluzione del problema spesso non tiene conto di molti fattori per ragioni di semplicità; la comunicazione, ad esempio, viene considerata implicitamente come istantanea e gratuita (non consuma cioè risorse destinate all'applicazione), semplificazione ovviamente lontana dal vero. Non è possibile rendere questo modello sufficientemente articolato e tanto preciso da permettere un approccio formale ed ottenere soluzioni ottime applicabili alla realtà. Attraverso l'uso di tecniche di tipo adattativo, basate su successive approssimazioni, è possibile comunque ottenere delle soluzioni accettabili (anche se subottimali) al problema.

1.1.2 Load Balancing e Load Sharing

Il problema presenta uno spazio delle possibili soluzioni molto vasto. Sarebbe quindi interessante che il Mapping fosse eseguito in maniera automatica dal sistema, al di sopra di un supporto che renda trasparente l'accesso a qualsiasi risorsa da qualsiasi locazione. In questa direzione si sono spinti molti sistemi operativi (Charlotte [ArF89], Sprite [DoOu91], DEMOS/MP [PoMi83] ad esempio); questi sistemi si basano sul raggiungimento di un obbiettivo più semplice della configurazione ottima: **il bilanciamento del carico**. Utilizzando le risorse a disposizione in maniera equilibrata ed uniforme, infatti, è possibile ottenere una soluzione abbastanza vicina a quella ideale. Le tecniche per ottenere il bilanciamento sono diverse, distinguibili in due famiglie:

- **Load Sharing:** si può decidere dove allocare una EU solo all'atto della sua creazione; esiste un meccanismo che permette la mobilità del codice della EU ed una politica di decisione. Durante tutto il suo ciclo di vita, ogni EU è staticamente legata al sito di nascita, il vettore percorso $\mathbf{P}_j$ è caratterizzato da un solo elemento, il nodo di nascita: $\mathbf{P}_j=[N_0]$.

- **Load Balancing:** siccome la configurazione ottima dipende dal tempo, può non essere sufficiente una allocazione all'atto della creazione.

Affinché sia possibile muovere dinamicamente una EU *attiva* da una locazione ad un'altra, è necessario realizzare un meccanismo di migrazione che permetta di conservare lo stato della EU. Le politiche sono più complesse, puntano ad ottenere una configurazione più vicina possibile a quella del bilanciamento perfetto con il minor numero possibile di passaggi e, soprattutto, con il minor numero di errori, di deviazioni dal vettore percorso ideale $\mathbf{P}_j=[N_0,N_a,..]$.

Le tecniche basate sul Load Sharing sono più semplici ed hanno un'intrusione minore, sono però incapaci di ottenere il bilanciamento del carico in molte situazioni, poiché non sono in grado di seguire le variazioni temporali indipendenti dalla nascita di nuove EU. Con il Load Balancing, per contro, è possibile ottenere risultati migliori, ma il meccanismo di migrazione molto costoso impone alle politiche scelte molto precise ed è quindi necessario un supporto di monitoring molto completo, per ottenere sufficienti dati per le decisioni.

L'esperienza dei molti sistemi dotati di Load Balancing hanno evidenziato che non esiste una politica adatta ad ogni situazione di carico; esistono varie famiglie di politiche, ma i loro risultati sono strettamente correlati alla topologia del sistema ed alle peculiarità dell'applicazione [Kok96]. Gli scarsi risultati dipendono dalla difficoltà di caratterizzazione delle EU e quindi dall'impossibilità di poter prevedere quali saranno le necessità dell'applicazione nel prossimo futuro [Har94].

1.1.3 Integrazione della migrazione nel linguaggio

Il sistema operativo può conoscere solo il codice eseguibile di un programma, un compilatore invece conosce molti più particolari sulla struttura e l'organizzazione dell'applicazione; portando l'implementazione della migrazione a livello del linguaggio è possibile quindi avere molti dati di tipo statico sulle singole EU. In ORCA ad esempio [BaKT92], esistono tipi di dati predefiniti in grado di rappresentare qualsiasi astrazione cosicché anche alberi e liste possano essere implicitamente gestiti. In questo modo, il compilatore conosce sempre la forma di ogni dato e quindi può rendere più agile il meccanismo di migrazione (che verrà analizzato in seguito) e permette di generare dei vincoli statici per le politiche, che quindi si

muovono in uno spazio delle soluzioni inferiore. Questi vincoli possono diventare anche una peculiarità del linguaggio: in ACL [CoLZ96] il programmatore ha la possibilità di *consigliare* ma anche *obbligare* al sistema alcune caratteristiche di allocazione dei vari oggetti attivi; ad esempio, si possono vincolare due EU come **coresident,** allocabili solamente sullo stesso nodo, oppure come **neighbour**, obbligate a risiedere in due nodi *vicini* (inteso come velocità di comunicazione).

Un approccio di questo tipo evidenzia come il problema del Mapping possa essere più o meno esplicitamente risolto nella fase di sviluppo del Software; questa ottica spinge a considerare la migrazione e la mobilità in generale come una proprietà del linguaggio, da integrare nel modello stesso di programmazione, piuttosto che un tool di libreria.

1.2 Il concetto di mobilità

Per mobilità si intende la possibilità di muovere, da un nodo ad un altro, dei componenti dell'applicazione; questi componenti possono essere sia dati, sia codice , sia lo stato congelato di una EU. La mobilità è alla base di ogni paradigma di programmazione di rete. Nel concetto di Remote Procedure Call, ad esempio, è fondamentale la necessità di trasferire i dati argomento della chiamata verso il server e i dati risposta verso il client.

La mobilità del codice, sebbene più complessa da realizzare, permette di superare molti problemi caratteristici della staticità del programma eseguibile: potendo creare componenti dinamicamente modificabili si accresce notevolmente la potenziale flessibilità dell'applicazione.

Per ottenere meccanismi snelli di movimento, i componenti mobili devono essere isolati gli uni dagli altri e devono avere il minor numero possibile di legami con il sistema locale, in modo che non sia necessario tenerne traccia.

I linguaggi basati sul paradigma Object Oriented, sono sicuramente i più adatti ad integrare la mobilità; infatti, l'assenza di uno scope globale ed il concetto di incapsulamento dei componenti sia attivi che passivi del programma permettono lo sviluppo di meccanismi molto più semplici ed efficaci.

L'analisi dei linguaggi che permettono Code Mobility, evidenzia come essi siano essenzialmente divisibili in due famiglie, in base al grado di mobilità che forniscono [CaPV96]: si parla di **Weak Mobility** quando è possibile associare dinamicamente il codice ad una EU, si parla invece di **Strong Mobility** quando una EU stessa è in grado di muoversi da un sito ad un altro; la EU viene congelata, trasferita e rimessa in esecuzione dove si era interrotta.

In presenza di **Weak Mobility**, le EU possono dinamicamente scambiarsi dei moduli di codice (in un paradigma ad Oggetti tipicamente degli Oggetti membro); in questo scenario, il problema del Mapping viene essenzialmente risolto in analogia al Load Sharing: ogni EU viene associata all'atto della sua creazione ad una locazione dove rimane per tutto il suo ciclo di vita. Al contrario del Load Sharing però, le EU possono cedere

dinamicamente alcune loro funzionalità ad altre che si trovano in una situazione più vantaggiosa per eseguire.

Una generica EU_j, caratterizzata inizialmente da mm moduli $M_{j,h}$, è rappresentabile in forma vettoriale $\mathbf{EU}_j=[M_{j,1},..,M_{j,mm}]$; i moduli non sono staticamente legati alla EU, ma possono essere ceduti o acquisiti dinamicamente per funzionalità o efficienza. La configurazione varia con una granularità più fine: c'è un legame statico tra N_0 ed $\mathbf{EU}_j$, la quale però varia al variare dei moduli che la compongono. La soluzione del problema del Mapping a questo punto è rappresentata dal vettore percorso degenere $\mathbf{P}_j=[N_0]$ e dalla configurazione delle diverse EU che può variare nel tempo.

In caso di **Strong Mobility**, vale tutto quello già esposto per la mobilità Weak; il vettore percorso non è più degenere: $\mathbf{P}_j=[N_0,N_a,\ldots]$ quindi la configurazione nel tempo ha un grado di libertà in più. La flessibilità è superiore, lo spazio delle soluzioni è infatti più vasto ed articolato ed è quindi possibile ottenere una soluzione più precisa del problema del Mapping.

Parlando di *Code Mobility* quindi ci si riferisce essenzialmente a due forme di mobilità: *migrazione di modulo* o, in un contesto ad oggetti, **migrazione di oggetto passivo** e *migrazione di una EU*, la **migrazione di un oggetto attivo**. Evidentemente non sono due concetti ortogonali, ma sono significativamente differenti sia dal punto di vista del modello che dal punto di vista realizzativo.

La ricerca della soluzione al problema del Mapping, con l'aumento della granularità, non viene sicuramente semplificata; l'aumento dei gradi di libertà rende più acute le problematiche di bilanciamento automatico. Il programmatore deve quindi essere parte attiva nella ricerca della configurazione ottima di esecuzione. La mobilità diventa così qualcosa di più di uno strumento a disposizione del progettista, deve essere integrata nei paradigmi stessi di programmazione distribuita.

1.2.1 I Modelli di programmazione in presenza di Code Mobility

Attraverso l'uso di un linguaggio dotato di Code Mobility è possibile utilizzare dei paradigmi di programmazione diversi rispetto ai linguaggi

classici. Di seguito è riportata una classificazione ordinata in funzione del grado di mobilità [CaPV96] (Figura 2):

- **Client Server (*CS*):** è il paradigma classico, le EU si dividono in Clienti che chiedono un servizio (remoto) e Servitori che lo forniscono; non è necessaria mobilità del codice ed infatti è l'unico modello usato nei linguaggi tradizionali.

- **Remote Evaluation (*REV*):** il cliente ha la possibilità di fornire il codice necessario per ottenere il servizio ad un generico servitore oppure, dualmente, il cliente può chiedere il codice ad un servitore per completare un dato compito. Si tratta di due aspetti dello stesso modello, nel primo caso si ha una azione di tipo *PUSH*, nel secondo caso una azione di tipo *PULL*. Per questo tipo di modello è sufficiente che il linguaggio supporti mobilità di tipo *Weak*.

- **Mobile Agent (*MA*):** se una EU ha bisogno di una determinata risorsa remota semplicemente si muove essa stessa per utilizzarla. Questo modello è estremamente flessibile ed è in realtà l'uso diretto del meccanismo di migrazione da parte del programmatore. Questo paradigma necessità sicuramente di *Strong Mobility* e, per questo motivo, solo pochi linguaggi consentono, al momento, una implementazione diretta.

Confrontando il classico modello CS con gli altri due dotati di mobilità si nota che REV è in realtà un'evoluzione di CS verso una maggiore dinamicità. Esistono attualmente varie applicazioni basate su questo modello: gli *Applet* di Java ad esempio rispecchiano un modello di tipo *PULL* (il cliente chiede il codice al Web Server), la possibilità di eseguire codice attraverso una shell remota nel mondo Unix (**rsh**) è un esempio di REV di tipo *PUSH*. MA invece è qualcosa di completamente nuovo. La realizzazione di una applicazione, utilizzando il modello MA, risulta completamente svincolata da ogni problematica programmazione di rete; indipendentemente dalla sua struttura, il sistema viene visto dall'agente come un insieme di contesti (noti come *place*) dove è possibile interagire con le più svariate risorse fisiche locali (ci può essere l'accesso ad un Database ma anche una persona fisica

che aspetta informazioni) oppure con altri agenti (sia che siano *parenti* o *stranieri*) con le più svariate tecniche di comunicazione.

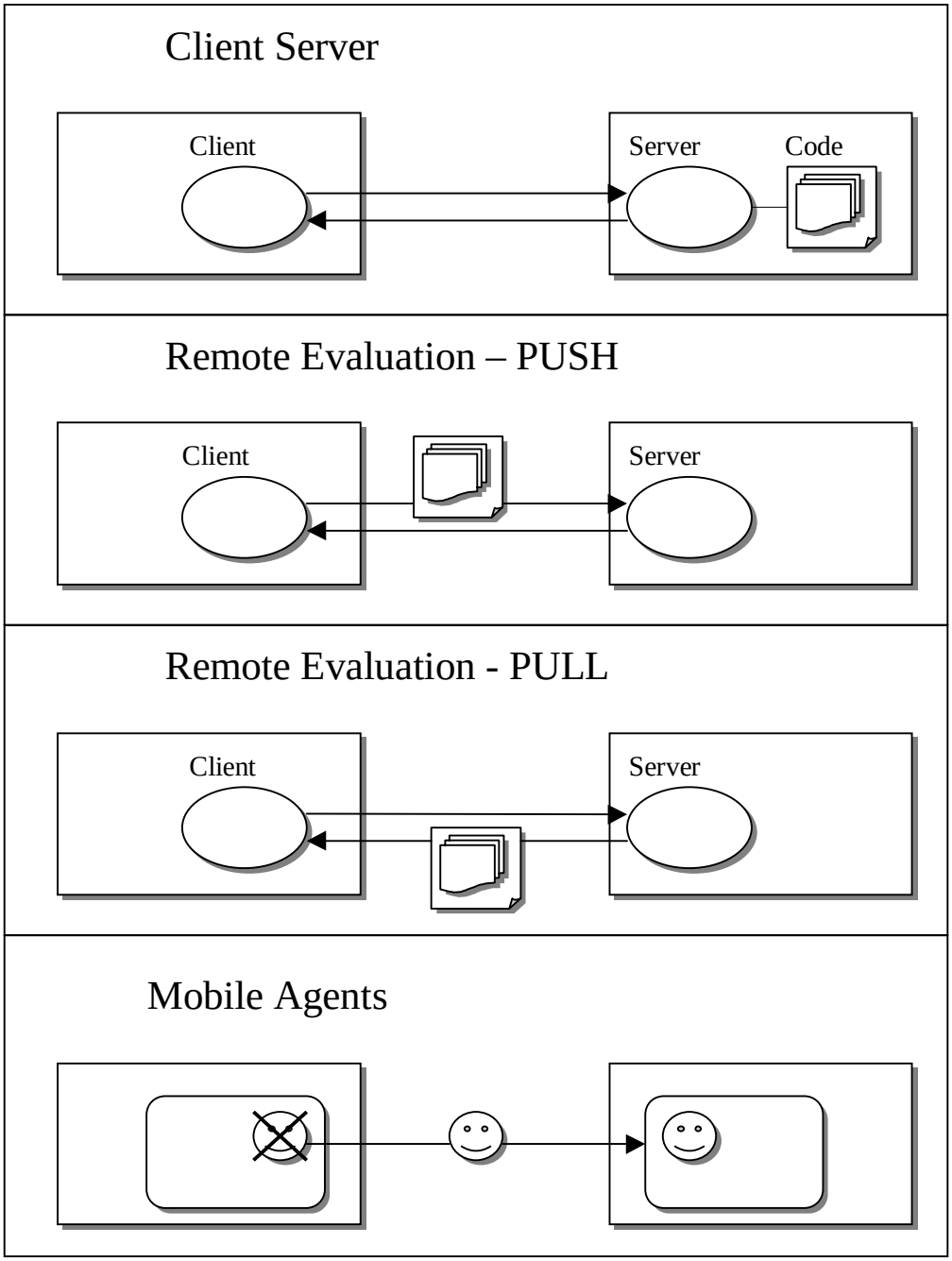

Figura 2: I modelli di programmazione in presenza di mobilità.

1.2.2 Confronto tra i Modelli

Supponiamo di voler realizzare un *News Searcher*, una applicazione in grado di cercare specifiche news nella rete Usenet.

Secondo il paradigma **CS** (figura 3), il programma chiede il servizio di indice ai News Servers (a), ottiene quindi il download di tutti i soggetti e riferimenti delle news pervenute (b), analizza e seleziona un ristretto numero di documenti da richiedere (c). Se fosse necessaria una ricerca di tipo più avanzato, magari sui documenti e non sui soggetti, dovrebbe essere realizzata dai Server e quindi il cliente non avrebbe nessuna possibilità di intervento.

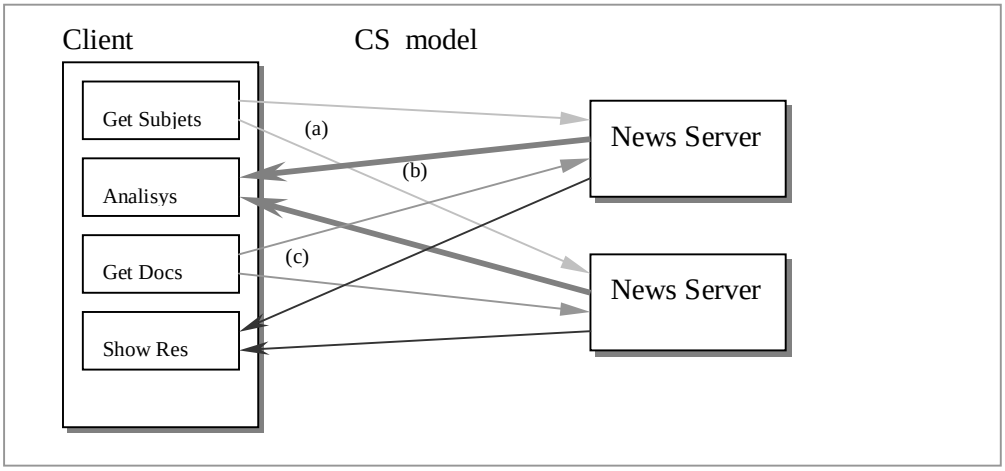

Figura 3: News Searcher realizzato con paradigma CS; la quantità di dati trasferiti al cliente è decisamente elevata.

Se si realizza la stessa applicazione con il modello **REV** (figura 4), il Cliente invia al News Server il codice (sotto forma di un Oggetto ad esempio) necessario per eseguire la specifica ricerca (a). Il Server, effettua la ricerca richiesta e poi spedisce direttamente i documenti trovati al Cliente (b). Ipotizzando il database molto vasto, e considerando quindi al grandezza del codice da trasferire decisamente minore della quantità di dati necessari per una elaborazione locale, l'occupazione di banda risulta decisamente inferiore rispetto ad un approccio in CS. Le ricerche possono essere personalizzate dal Cliente senza l'intervento del Server; in caso di successive raffinazioni però, non è possibile riutilizzare il lavoro di ricerca già svolto: ogni volta che

chiedo un servizio questo viene ripetuto da zero e non può essere mantenuto implicitamente nessuno stato.

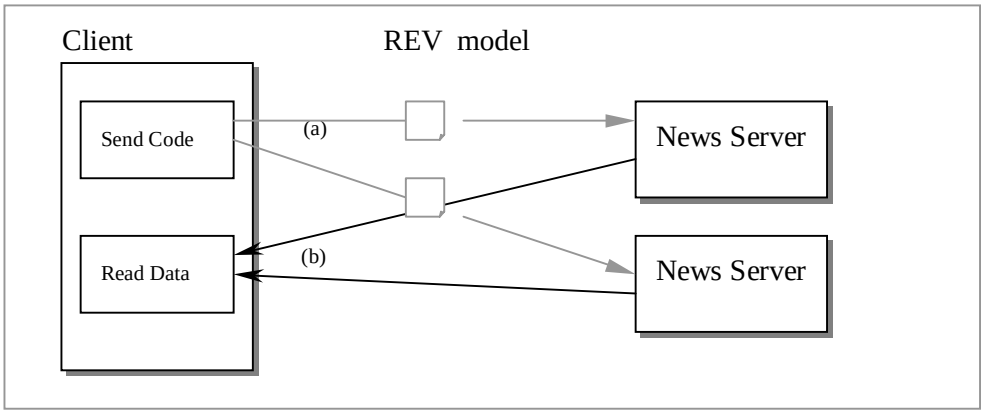

Figura 4: News Searcher realizzato secondo il modello REV; l'uso della rete ha guadagnato in efficienza.

Con un approccio basato su **MA**, l'applicazione è caratterizzata da un agente che si muove verso il primo Server (a), si clona e si sposta sul secondo Server, dove effettua la ricerca; il clone, effettua la ricerca e poi segue l'altro agente per comunicare i risultati (figura 5). A questo punto il clone muore e l'agente originario torna indietro trasportando i documenti ritrovati (b). Se sono necessarie successive raffinazioni, con informazioni aggiuntive, l'agente può tornare sul News Server per completare la sua ricerca partendo da quella precedente, cioè dal suo stato interno. Anche con questo modello il risparmio di banda non è assoluto; dipende ovviamente dalla grandezza dell'agente (il suo stato potrebbe non essere trascurabile) confrontata con le necessità di trasferimento dati in un approccio classico.

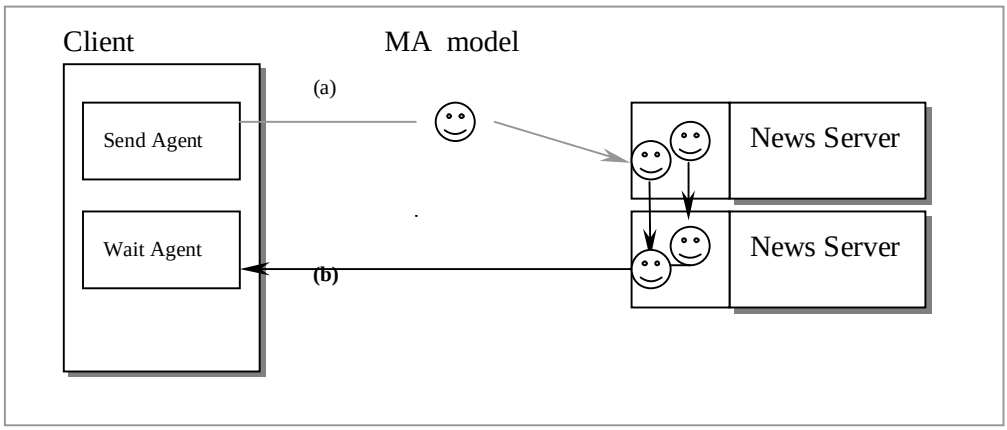

Figura 5: News Searcher realizzato con il modello ad Agenti; l'occupazione di banda risulta decisamente ridotta nell'ipotesi che l'agente abbia dimensioni inferiori ai dati da trasferire per l'elaborazione.

Questo esempio, è solo un possibile soluzione del problema con paradigma MA. Sono possibili infinite varianti, che possono essere più o meno utili in diverse configurazioni; può essere l'agente stesso a scegliere la strategia in base a come percepisce l'ambiente dove deve operare.

Se confrontato con l'approccio CS, gli agenti risultano decisamente più semplici ed intuitivi. Anche rispetto al caso REV si nota come sia maggiore la flessibilità e la duttilità e come l'attenzione del programmatore sia completamente spostata sul problema da risolvere e non sui particolari legati al trasporto ed alla rete.

1.3 I meccanismi che permettono Strong Mobility

Per muovere una EU da un sito ad un altro sono necessari 3 passi fondamentali (Figura 5):

- Salvataggio dello stato della EU (*Store*)
- Trasporto della EU sul nuovo nodo (*Serialize, Transfer, Deserialize)*
- Ripristino dello stato e messa in esecuzione (*Restore*)

In certi casi si distingue tra *Moving* e *Cloning* ma il meccanismo è esattamente lo stesso, l'unica differenza sta nella eliminazione o meno della EU sorgente.

La prima e la terza fase sono duali, per *stato di una EU* si intende tutto ciò che è caratteristico della esecuzione in corso e quindi dipende dal tipo di EU. Se si parla di processi di tipo pesante (come ad esempio i processi Unix), lo stato comprende tutte le aree dati utente sia statiche (data segment) che dinamiche (stack e heap), le tabelle di sistema proprie del task (ad esempio la gestione dei file) ed infine il contenuto dei registri della CPU. Quando invece si intende migrare processi leggeri e si parla di *Oggetti Attivi* (i *Threads* Java ad esempio), lo stato si riduce allo stack, all'heap, ai registri di sistema più i legami con gli oggetti che rimangono nella locazione sorgente. Nelle aree di memoria dinamiche in generale sono contenuti Oggetti passivi, questi possono avere riferimenti ad altri oggetti non membri del thread, bisogna tenere traccia del grafo di legami che si produce ai fini della consistenza dei riferimenti remoti.

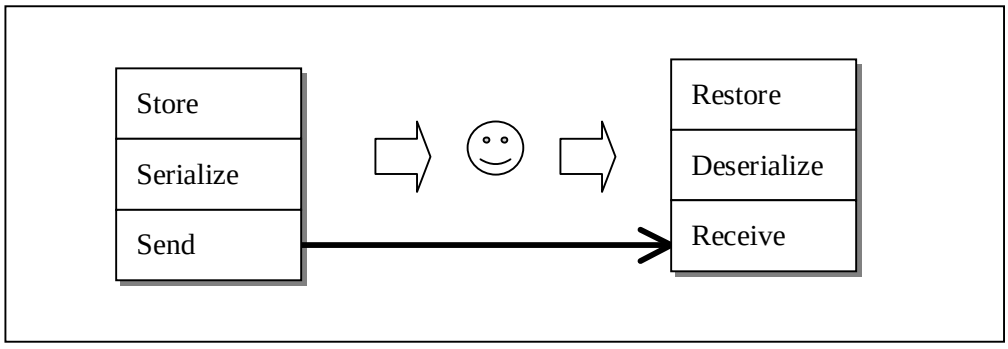

Figura 6: Fasi che caratterizzano il meccanismo di migrazione.

Poiché i puntatori devono mantenere il loro significato, si deve provvedere ad una ridirezione sui nuovi indirizzi oppure a ripristinare le aree dati nelle stesse pagine fisiche. La seconda soluzione non è praticamente realizzabile se non in alcuni specifici casi, la ridirezione deve essere implementata a livello di sistema, altrimenti il costo necessario per tenere traccia di tutti i riferimenti per poterli gestire diventa troppo grande.

La seconda fase, il trasporto, è un meccanismo noto: si tratta di trasformare i dati da trasferire in una forma sequenziale (*serializing*) affinché possano essere trasmessi su un canale fisico e poi rigenerati nella forma originale (*deserializing*). Se il meccanismo deve funzionare tra ambienti eterogenei, è necessario che il trasporto venga effettuato in una forma standard cosicché tutti i partecipanti siano in grado di capirne e tradurne il significato. Il costo di questa fase è direttamente proporzionale alla quantità di dati che devono essere trasferiti, in un contesto ad Oggetti Attivi sarà sensibilmente inferiore rispetto al caso dei processi pesanti.

1.3.1 Le gestione dei puntatori e dei riferimenti remoti

La gestione della consistenza dei puntatori e dei riferimenti durante le fasi di *Store e Restore* è fondamentale per il buon funzionamento del meccanismo.

I dati che appartengono alla EU, come ad esempio le variabili dinamiche o in generale gli *Oggetti membro*, possono essere mantenuti consistenti con le due tecniche già accennate:

- Tracciare e gestire direttamente tutti gli accessi dinamici, in questo modo si aggiunge un livello di indirezione, disaccoppiando gli indirizzi dalla memoria fisica.

- Ripristinare dalla parte del ricevente le aree dati negli stessi indirizzi fisici del nodo mittente, così tutti i riferimenti rimangono consistenti.

La prima tecnica è fortemente intrusiva, penalizza molto i tempi di accesso alla memoria; se però l'indirezione è già presente per altri motivi (ad esempio nella gestione della memoria da parte di un interprete), può essere facilmente implementabile ed efficiente.

Ripristinare le aree dati nelle stesse locazioni, invece, sarebbe molto efficiente; presenta però grossi problemi di tipo realizzativo, poiché non è sempre possibile ottenere una specifica area di memoria dal sistema, giacché potrebbe essere già occupata. Solo se le EU sono processi pesanti ed il sistema fornisce l'astrazione di memoria virtuale, è possibile l'uso di questa tecnica; ogni processo infatti, *vede* sempre l'intero spazio virtuale tutto a sua disposizione.

Per mantenere la consistenza dei puntatori remoti, cioè i riferimenti a tutto ciò che non si muove con la EU, la tecnica di base consiste nel riferirsi localmente ad un qualche meccanismo che fornisca trasparentemente l'accesso all'area remota. Anche in questo caso si tratta di aggiungere un livello di indirezione per una gestione trasparente [Ach97].

Il costo di una accesso remoto è sempre superiore al costo di un accesso locale di qualche ordine di grandezza, quando risulta possibile utilizzare tecniche di replicazione le prestazioni ne risentono sensibilmente.

1.3.2 Eterogeneità e Code Mobility

Un sistema aperto è intrisecamente eterogeneo, si è già accennato che in queste condizioni è necessario prevedere dei formati standard che vengano intesi da tutti i partecipanti. Per i dati puri, questo approccio non presenta particolari problemi poiché le differenze sono sempre meccanicamente codificabili; se invece deve essere trasferito del *Codice Eseguibile* il discorso è più complesso. Si deve distinguere anche in questo caso tra mobilità Weak e Strong: nel caso Weak, cioè di mobilità di moduli passivi, il codice può essere trasferito in forma sorgente ed eventualmente compilato sulla macchina destinazione. Non essendo necessario nessun legame con uno stato attivo dipendente dalla macchina il problema dell'eterogeneità viene risolto implicitamente dal progetto di un linguaggio svincolato dall'architettura.

In caso di mobilità Strong, è necessario trovare una corrispondenza tra gli stati attivi su diverse piattaforme e quindi una equivalenza a livello di codice binario. La definizione di standard a livello di codice binario, vista l'impossibilità di modificare fisicamente le CPU, può essere affrontata in due modi:

- Creazione di una Macchina Virtuale per il codice standard: le applicazioni vengono sempre interpretate, svincolandosi completamente dal supporto fisico sottostante. In questo modo si risolve anche il problema dei riferimenti, integrando nel meccanismo di gestione della memoria il ripristino dei riferimenti locali ed il meccanismo di accesso ai riferimenti remoti.

- Ricompilazione del codice all'arrivo sulla macchina destinazione: si traduce il metacodice nel formato nativo all'arrivo e lo si manda in esecuzione.

L'uso di una macchina virtuale comporta un drastico peggioramento delle prestazioni a causa dell'uso indiretto delle risorse computazionali, permette però di ottenere a basso costo tutti i meccanismi necessari per la migrazione.

La riconversione del codice nel formato nativo invece permette il pieno sfruttamento delle CPU ma rende più complesse le fasi di *Storing e Restoring*; è necessario infatti che lo stato attivo della EU venga anch'esso trasformato in un metaformato, in modo che la macchina destinazione sia in grado di ottenere uno stato consistente con le sue caratteristiche di esecuzione. Esistono delle sezioni critiche, lungo il filo dell'esecuzione, dove non è possibile astrarre uno stato indipendente dalla macchina [SteJu95], per questo motivo, il sistema deve essere in grado di trovare nel codice dei punti stabili (detti C*heckpoint* o *Bus Stop*) e permettere la migrazione solo in quelle specifiche posizioni. Se la CPU sorgente è molto diversa da quella di riferimento, possono non essere identificabili dei checkpoint; in questi casi allora è necessario che il sistema provveda a costruire dei *bridge* di codice che la macchina destinazione deve eseguire prima di poter completare il *Restoring*

Dati questi limiti, appare attualmente difficile pensare alla realizzazione di un sistema aperto, prescindendo dall'uso di un linguaggio interpretato da una macchina virtuale anche se ci sono proposte in questo senso [SteJu95].

1.4 Gli Agenti Mobili

Un *Agente* è una entità attiva caratterizzata da un comportamento autonomo e variabile a seconda delle situazioni; spesso si parla di agenti riferendosi ad una qualsiasi EU che sfugge al concetto Cliente - Servitore ed assume dinamicamente l'uno o l'altro ruolo a seconda delle necessità. I primi agenti dotati di mobilità vennero concepiti come una versione *soft* di un robot (da qui il termine *Softbot*) quindi dotati di caratteristiche cognitive e capacità di interazione con l'ambiente che nel loro caso è virtuale [EW94]. Successivamente, privato delle sue caratteristiche cognitive, l'agente mobile è stato inquadrato come un modello di programmazione di rete forte dell'interazione con l'ambiente.

1.4.1 I *place*

L'ambiente di gestione degli agenti in ogni sito è detto **place** [Pein97]; qui devono essere disponibili un certo numero di servizi standardizzati, tali da permettere l'interazione tra agenti e risorse locali ed eventualmente remote. A livello di principio, ci possono essere più places associati ad un nodo fisico o più nodi fisici associati ad un place. Il place è come una *piazza virtuale* dove l'agente può cercare, *tra le bancarelle*, quello che ha bisogno.

La prima necessità è quella di trovare le *bancarelle*; deve essere disponibile un servizio di informazioni, che permetta il riconoscimento delle risorse disponibili e il ritrovamento della loro locazione. Una volta trovato il servizio, è necessario poter interagire con esso per ottenere quello di cui si ha bisogno; in un sistema aperto, questa interazione deve essere più omogenea possibile, in questo modo si rende attuabile l'accesso a risorse anche non note a priori. Per semplicità di progettazione sia del supporto che degli agenti, la soluzione adottata più frequentemente è quella di implementare i servitori come agenti eventualmente statici, in questo modo l'interazione tra servizi e agenti risulta più semplice ed uniforme.

1.4.2 La Comunicazione

Le tecniche di comunicazione tra EU sono numerose, per gli Agenti si è visto come, a seconda delle necessità, siano più utili alcune piuttosto che altre. Le esigenze di comunicazione di un agente possono essenzialmente classificate in due tipi:

- **Interazione Stretta:** quando due agenti devono collaborare strettamente in un rapporto client-server o peer-to-peer, è necessaria una comunicazione di alto livello attraverso *shared objects* o la chiamata diretta dei metodi del partner. Il meccanismo di base per questo tipo di comunicazione è la cosiddetta *Remote Method Invocation* (*RMI)* [Java] : la capacità di poter eseguire il metodo di un oggetto che non risiede esplicitamente nello stesso sito. Questo tipo di meccanismo è comunque legato alla conoscenza specifica degli Agenti che vogliono interagire: un accesso RMI prevede che il chiamante conosca staticamente (nella sua fase di progettazione) i tipi ed il numero degli argomenti della chiamata. Per ottenere chiamate completamente dinamiche è necessario un protocollo di acquisizione delle informazioni e degli argomenti standardizzato che permetta questo tipo di interazione anche se gli Agenti non sono stati progettati ad hoc. Non si tratta comunque di fornire l'accesso delle risorse in maniera trasparente poiché sarebbe contro il modello. La comunicazione remota di questo tipo è inefficiente, deve quindi essere limitata in uso ma disponibile in alcuni casi, per evitare grosse complicazioni nella programmazione degli Agenti soprattutto nel caso di uso di Oggetti Condivisi.

- **Interazione Lasca:** spesso non è necessario avere un contatto così stretto. A volte, è sufficiente un singolo messaggio per fornire un dato o una informazione; in questi casi risulta molto comodo il modello *Message Passing*, che può essere utilizzato senza nessuna ipotesi di località. Una forma di comunicazione di questo tipo molto importante nel mondo degli Agenti è la **Comunicazione Anonima:** quando due entità non si conoscono, perché non hanno avuto origine nello stesso sito o semplicemente perché non sono parenti, non sono in grado di comunicare con metodi espliciti, hanno bisogno di un supporto che permetta di conoscere con facilità i nomi di altri agenti e di astrazioni

come la *Blackboard* oppure uno *Spazio delle Tuple* [CaGe89] per poter interagire in maniera asincrona. Questo tipo di interazione può essere vista come una prima forma di comunicazione in vista della creazione di un eventuale legame stretto; è importante che sia disponibile sia in locale che in remoto e, almeno nella sua forma più semplice, anche in maniera trasparente, per incoraggiare solo le piccole interazioni in remoto.

All'interno dei place, è necessaria una qualche forma di interazione con i servizi a disposizione e la soluzione più semplice consiste nel realizzare i gestori dei servizi come Agenti. In questo modo, si ottiene una interfaccia omogenea e non è quindi necessario realizzare meccanismi ad hoc.

Un aspetto fondamentale della comunicazione riguarda la **sicurezza:** spesso in una interazione risulta fondamentale l'autenticazione del partner di comunicazione, specialmente se in funzione della erogazione di un servizio riservato o a pagamento. Siccome l'Agente ha la possibilità di muoversi, è necessario che il sistema integri nella mobilità il controllo dell'identità dell'Agente ed i suoi permessi [Pein97]; in questo modo si verifica un eventuale intruso prima di inserirlo nel place e non sono quindi necessarie altri controlli all'interno del sito se non quelli sui permessi per la cui veridicità garantisce il sistema.

1.5 Analisi degli ambienti che permettono Code Mobility

Per cercare di analizzare appieno quali siano le caratteristiche principali che deve possedere un supporto ad Agenti, sono stati analizzati alcuni dei più noti ambienti che permettono Code Mobility. In tutti questi sistemi si parla di Agenti, però in alcuni il paradigma MA viene solo simulato sopra un supporto REV (solitamente dato da Java). Questa anomalia è dovuta alla difficoltà di realizzare un supporto che permetta Strong Mobility ed all'indubbio vantaggio di poter scrivere in un linguaggio affermato e indipendente dalla piattaforma.

Nell'analisi è stata fatta particolare attenzione a come sono realizzati gli Agenti, al tipo di mobilità e come è stata ottenuta; di particolare importanza è da considerarsi anche la comunicazione, in particolare quella Anonima che è essenzialmente una novità.

I sistemi di seguito analizzati non sono gli unici noti, ci sono molte varianti ad agenti sia nel mondo commerciale che accademico; questi sistemi contengono le caratteristiche essenziali più significative e possono essere considerati come un campione.

1.5.1 Agent Tcl

Agent Tcl [RuGK97] è una estensione del linguaggio interpretato Tcl [Ous94], abbastanza noto nel mondo Unix. Questo sistema, dotato di Strong Mobility, è stato concepito per realizzare applicazioni in grado di adattarsi dinamicamente ai mutamenti della rete. I progettisti si sono quindi focalizzati sulla *percezione del mondo* da parte degli Agenti; una infrastruttura di Monitor fornisce una serie di indicatori sullo stato della rete e dei nodi. Insieme al servizio di Monitoring, nel place è anche disponibile un servizio di *yellow pages*, basato su un database distribuito in grado di fornire l'esatta posizione delle risorse e degli Agenti nell'intero sistema.

La scelta del linguaggio Tcl per la realizzazione degli *Agenti trasportabili* è dovuta alla maturità dell'interprete dal punto di vista della portabilità (in particolare nel mondo Unix) e della sicurezza (il problema

dell'esecuzione di script Tcl provenienti da siti *untrusted* è stato ampiamente sviluppato ed approfondito). Poiché Tcl è uno *Script Language*, è molto semplice e diffuso ed il suo interprete è facilmente manipolabile.

Un Agente Tcl può muoversi da un sito ad un altro in qualsiasi momento: mediante il comando **agent_jump** l'Agente si sospende, viene catturato il suo stato interno e spedito al server di destinazione che lo ripristina.

La comunicazione tra Agenti si ottiene via message passing, è però possibile aprire delle connessioni dirette per il trasferimento dati attraverso i comandi **agent_meet, agent_send, agent_receive**. Per comunicare, un Agente deve conoscere sia la locazione del destinatario che il suo nome simbolico (scelto dall'Agente stesso dinamicamente); entrambe le informazioni sono ottenibile mediante il servizio di *yellow page*.

Un Agente può utilizzare tutte le potenzialità del linguaggio Tcl nel sito dove si trova: è possibile aprire file o interfacce grafiche nei limiti permessi dalle politiche di protezione del sito. Tutto ciò che è legato al nodo (finestre, file descriptor etc. possono essere definiti come lo ***stato esterno*** di una EU) viene implicitamente perso all'atto della migrazione per evitare di mantenere tracce nei siti visitati.

Affinché un Agente sia indipendente, abbia cioè la possibilità di essere autonomo fino alla fine del suo compito, è necessario che sia in grado di percepire le caratteristiche dell'ambiente in cui si muove. Per questo motivo è stato realizzato un piccolo sistema di monitoring che verrà brevemente descritto nel prossimo capitolo.

Il sistema dispone di un servizio di *Yellow Page* con il compito di fornire le informazioni a proposito dei servizi e delle risorse del sistema globale. È realizzato come un database distribuito gestito da un gruppo di Agenti specializzati organizzati gerarchicamente: quando un Agente ha bisogno di informazioni, esegue una query, basata su keyword, ad un Agente addetto presente in ogni nodo; questo Agente interpella i suoi superiori sparsi per la rete in modo da ottenere le informazioni cercate. I servizi a disposizione sono in generale replicati e multipli, è compito degli Agenti addetti alle Yellow Page selezionare dinamicamente i siti migliori (accesso più veloce, percorso più breve…).

1.5.2 ARA

ARA, il cui nome è un acronimo per *Agents for Remote Actions,* è un supporto sviluppato all'Università di Kaiserlautern [Pein97]. È strutturato in modo da essere indipendente dal linguaggio di utilizzo per programmare gli Agenti; al di sopra di un livello comune, detto *Ara core*, sono costruibili interpreti per qualsiasi linguaggio (figura 6).

La scelta modulare, permette di svincolarsi completamente dal linguaggio ottenendo così un grado superiore di apertura e permettendo al sistema di sopravvivere all'evoluzione dei linguaggi sempre molto veloce.

Al contrario di Agent Tcl, ARA fornisce l'astrazione di **place**, un ambiente virtuale all'interno del quale gli Agenti possono interagire. Concettualmente, il place è svincolato dal mondo fisico, nasconde completamente la natura del nodo quindi permette di superare eventuali eterogeneità.

Ara core provvede alla gestione dei places, alla comunicazione, alla migrazione ed al *cloning*, con un orientamento spiccato sulla sicurezza, intesa come la possibilità di controllare l'uso delle risorse ed evitare che Agenti *mal intenzionati* possano abusare del place in cui si trovano.

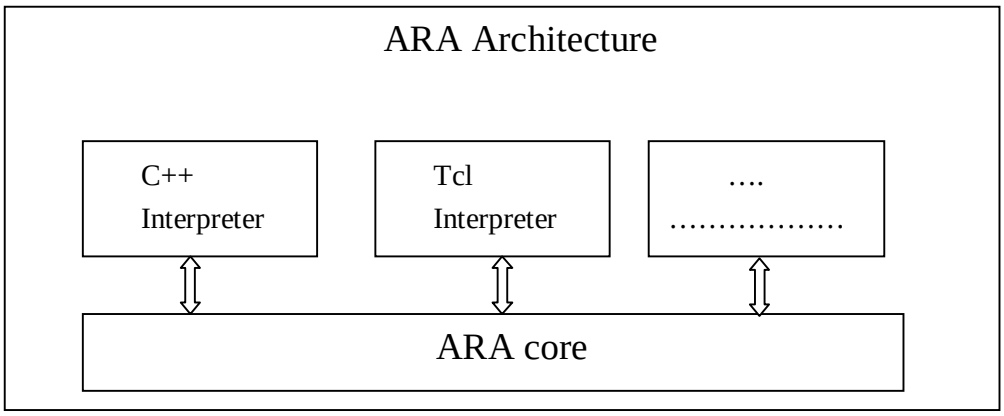

Figura 7: Ara prevede diversi moduli per consentire la realizzazione degli Agenti in più linguaggi.

La sicurezza e la protezione sono fondamentali per un supporto pensato all'uso in reti geografiche: gli Agenti devono essere controllati in ogni movimento e situazione. Il modulo dell'interprete ha il compito di verificare

le cosiddette *allowance*: il permesso di poter fare determinate azioni o utilizzare determinati servizi. Una allowance può essere di tipo quantitativo (uso massimo di memoria…) o di tipo qualitativo (permesso di leggere un determinato file…). Gli Agenti sono organizzati in gruppi, in un gruppo vengono condivise le allowance ma ci possono essere scambi (o acquisti) con altri gruppi.

Le allowance non hanno solo lo scopo di evitare accessi non desiderati a risorse private; il creatore dell'applicazione, per fini di protezione, può imporre determinati vincoli di comportamento di tipo generale *(Global Allowance)*; l'Agente stesso può porsi dei limiti runtime in un determinato place *(Local Allowance)*.
La migrazione si ottiene attraverso la chiamata **ara_go**[1]:

```
ara_go ? –la  <local allowance at destination> ? \
          -ga  <global allowance at destination> ? \
          <destination place name> ? <agent> ?
```

La mobilità è di tipo Strong, l'unica restrizione consiste nell'impossibilità di mantenere lo stato esterno all'atto della migrazione (come in Agent Tcl). Come destinazione è necessario specificare Uno scheduler not-preemptive all'interno di ARA core seleziona il modulo interprete al quale fornire la CPU ed a livello di interprete viene gestito lo scheduling preemptive tra i vari Agenti in quel linguaggio.

La mobilità integrata nel linguaggio permette di realizzare programmi in cui i place sono considerate delle risorse al pari delle altre:

```
foreach place{ moira kismet fatum} {
    ara_go $place
    puts "Hello at $place, I'm coming from $prevplace"
    set $prevplace $place
    }
```

[1] Il linguaggio in questi esempi è Tcl: nella notazione, il simbolo ? è il separatore tra gli argomenti la cui descrizione è racchiusa tra < >.

In questo esempio, scritto in Tcl e tratto da [Pein97] è evidente come la locazione venga gestita implicitamente nel flusso del linguaggio.

La comunicazione si basa sul concetto di **Service Point**: un Agente, con il compito di Servitore, annuncia che è disponibile al servizio in un particolare place; l'Agente che ha bisogno di questo servizio, può chiedere al sistema dove si trova il Servitore e muoversi per incontrarlo. Una volta incontrati, la comunicazione è basata sullo scambio di messaggi e lo stile è spiccatamente Client-Server. I Service Point sono completamente dinamici, un Cliente può anche bloccarsi in attesa della nascita di uno specifico Servitore.

Un Service Point si crea quando l'Agente servitore lo annuncia assegnandogli un nome simbolico che deve essere unico nel place: *ara_announce <name>*.

Il cliente può incontrare un Service Point specificandone il nome, se il servitore non lo ha ancora annunciato la chiamata è a default bloccante ma è possibile l'opzione non bloccante: *ara_meet ? –dontwait ? <name>.*

Ci possono essere un qualsiasi numero di clienti ma un solo servitore; un cliente sottopone delle richieste al servitore con un protocollo Request/Reply con chiamate bloccanti sincrone.

Le risorse disponibili agli Agenti nei place sono gestite da due tipi di Agenti speciali:

- **Proxy Agents:** sono agenti immobili fidati che hanno il compito di rappresentare nel mondo ARA le applicazioni esterne.

- **System Agents:** forniscono l'interfaccia alle risorse fisiche del sito, sono immobili e compilati in codice nativo per ottenere migliori prestazioni.

Non solo gli Agenti di sistema possono essere compilati, ma è possibile creare Agenti in codice nativo per ottenere migliori prestazioni a scapito della mobilità. Gli Agenti compilati infatti hanno una mobilità di tipo Weak attraverso il movimento dei sorgenti e la ricompilazione *Just in Time*. Il by-pass della macchina virtuale svincola l'Agente anche dai controlli delle allowance: è importante che il sistema accetti sorgenti solo da siti fidati.

1.5.3 Mole

Mole è un ambiente scritto in Java per Agenti scritti in Java, è stato sviluppato dall'Università di Stoccarda [StrBH96] e viene considerato come un prototipo per lo sviluppo degli strumenti di cui necessità un sistema ad Agenti.

Ogni Agente, ha un nome unico fornito all'atto della creazione; la mobilità è di tipo Weak (un Agente riparte sempre dal metodo **start()** ogni volta che cambia locazione), viene ottenuta mediante il metodo **goto()**, che accetta come argomenti sia il nome di una *abstract location* (place) che quello di un altro Agente, in questo modo è possibile muoversi seguendo un eventuale partner di comunicazione svincolandosi completamente dalla necessità di conoscere le locazioni fisiche dei partecipanti.

Il place è considerato un accesso trasparente alle risorse fisiche, ci possono essere più place sullo stesso nodo oppure più nodi possono formare un solo place.

Per Agente, si intende un classico Oggetto Attivo, un contesto chiuso la cui informazione con l'esterno è standardizzata attraverso i metodi pubblici al cui interno possono girare uno o più thread in paradigma *Shared Memory*. Questo modello crea dei problemi di natura semantica: quando un thread decide di migrare, infatti, costringe gli altri a muoversi senza poter decidere autonomamente; nella attuale implementazione, i thread vengono sospesi e viene demandata al programmatore la gestione di questo evento. In questo senso, i thread all'interno dell'Oggetto sono solo parallelizzazioni temporanee nell'esecuzione di un determinato compito, il programmatore ha il compito di verificarne il completamento prima di dare il comando di migrazione.

La natura di prototipo dell'ambiente ha spinto i progettisti a realizzare varie forme di comunicazione. Nel rispetto del modello Object Oriented di Java, è disponibile la Chiamata a Metodo Remoto (RMI): gli Agenti possono comunicare tra loro invocando con RMI i metodi pubblici l'uno dell'altro. Il Message Passing fornisce una interazione più semplice ed in più viene fornito un metodo di comunicazione anonima [Baum97] detto **Session Oriented Communication.** Questa astrazione è simile al concetto di *Service Point* in ARA: si stabilisce una connessione, un rapporto di riconoscimento tra due o più agenti che a priori non hanno nessuna conoscenza l'uno dell'altro. Un

agente, che vuole essere riconosciuto da altri per uno specifico ruolo in una interazione, *indossa* un **badge**, una *etichetta* (ad esempio *Coordinator* oppure *Worker XX*) cosicché gli altri possono riconoscerlo. Un badge è altamente dinamico, può essere *indossato* o *tolto* in qualsiasi momento, si può utilizzare contemporaneamente un qualsiasi numero ed il mantenimento della sessione dipende da essi. Una sessione può coinvolgere anche Agenti remoti; quando però un partecipante decide di muoversi, viene implicitamente staccato per semplificare il meccanismo, che così non deve tenere traccia degli spostamenti. La comunicazione, all'interno della sessione, avviene mediante **com** (common) **objects** che possono essere di due tipi:

- **RMI Object:** viene esportato da un membro della sessione e viene usato come Shared Object.

- **Messaging Object:** sono delle Mailbox associate ad ogni membro della sessione.

In [Baum97] si parla anche di sincronizzazione degli Agenti mediante gestione degli eventi secondo lo standard OMG [OMG94] e dell'astrazione di tupla (presente anche in [LDD95]) come futuri sviluppi per Mole nell'ambito della comunicazione anonima.

1.5.4 Aglets

Aglets Workbench, un prodotto di IBM [Aglets96], è un ambiente scritto interamente in Java per Agenti Java e, come Mole, fornisce un meccanismo di mobilità di tipo **Weak**. In questo sistema l'Agente rappresenta un'estensione del concetto di Applet Java; per applet si intende un modulo di codice che può muoversi dal Server verso il Client mentre un *aglet* è in grado di mantenere il suo stato.

È un ambiente creato per fornire uno strumento più potente e flessibile per la creazione di siti Web, per questo motivo pone un particolare accento sui problemi di sicurezza, di programmabilità e di standard. Dal punto di vista della sicurezza, Aglets utilizza un protocollo di autenticazione, questo permette la verifica dell'identità del proprietario dell'Agente e quindi la sua classificazione come *Trusted* oppure *Untrusted* permettendo così l'accesso vincolato alle risorse. Il meccanismo di migrazione verifica inoltre se nel

trasferimento ci sono stati eventuali danni quali possono essere l'introduzione di virus o la manipolazione del codice degli agenti, attraverso l'uso di un m*essage digest*.

Per quanto riguarda la semplicità di programmazione, Aglets cerca di mantenere la stessa logica di programmazione ad eventi del package AWT di Java secondo il modello della versione 1.1 [Ven97]: la Classe **Aglet** infatti presenta dei metodi che vengono implicitamente invocati nelle varie fasi di vita dell'agente; in questo modo è possibile, tramite overloading, programmare e personalizzare il comportamento dell'agente in ogni situazione. Se consideriamo ad esempio la creazione di un nuovo agente, si nota come vengano chiamati implicitamente due metodi oltre al costruttore:

CreateAglet()→	*Aglet()*
	onCreation()
	run()

Il meccanismo di migrazione, di tipo Weak, riparte sempre dal metodo **run()**, viene invocato attraverso il metodo **dispatch(URL)** e provoca la seguente cascata di invocazioni:

dispatch()→	*onDispatching()*	(eseguito localmente)
	onArrival()	(eseguito a destinazione)
	run()	

Il metodo **onDispatching()** viene invocato all'atto del salvataggio dello stato dell'agente, in questo modo il programmatore ha la possibilità di preparare la sua partenza. Il metodo **onArrival()** viene invece invocato all'arrivo sul nodo destinazione, permette così la personalizzazione dell'arrivo in un nuovo place.

Questa tecnica di programmazione ad eventi, risulta molto semplice e naturale soprattutto per chi ha esperienza in programmazione di GUI (Graphics User Interface), permette così un più facile apprendimento da parte dei creatori delle pagine Web che hanno già maturato esperienza in Java con gli applet. I principali eventi che caratterizzano la vita di un Aglet sono riassunti in Tabella I: oltre agli eventi legati alla mobilità ci possono essere

situazione di copia (clone) oppure di attivazione/disattivazione (Aglets fornisce un servizio di persistenza).

Momento di invocazione	Oggetto di Gestione	Listener	Metodo invocato
Inizio clonazione	CloneEvent	CloneListener	onCloning
Creazione clone	CloneEvent	CloneListener	onClone
Dopo creato clone	CloneEvent	CloneListener	onCloned
Inizio migrazione	MobilityEvent	MobilityListener	onDispatching
Inizio Retract	MobilityEvent	MobilityListener	onReverting
Arrivato a destinazione	MobilityEvent	MobilityListener	onArrival
Arrivo Messaggio	-	-	handleMessage
Inizio disattivazione	PersistencyEvent	PersistencyListener	onDeactivating
Dopo la riattivazione	PersistencyEvent	PersistencyListener	onActivation

Tabella I: I principali Eventi gestiti da Aglets; per **Listener** si intende il gestore del servizio dell'evento presso cui è necessario registrarsi per poterlo ricevere (in accordo con il modello ad Eventi della versione 1.1 di Java [Java]).

La comunicazione tra Agenti è realizzata da due meccanismi:

- **Message passing:** è l'unica tecnica di comunicazione diretta, è di tipo sincrono o asincrono e ci deve essere conoscenza tra i partner di comunicazione senza però nessuna ipotesi di località. Per semplificarne l'uso vengono forniti alcuni protocolli di più alto livello come *Master-Slave, Messenger-Receiver, Notifier-Notification.*

- **Whiteboard:** si tratta di un'area comune in ogni place dove qualsiasi agente può lasciare un messaggio; chiunque può accedere in lettura e leggere i messaggi depositati per verificare se è un destinatario. Questo tipo di comunicazione è anonima ed asincrona con ipotesi di località. In un ambiente ad agenti è uno strumento fondamentale per permettere l'interazione e la conoscenza di agenti che non sono parenti.

Aglets è realizzato in puro Java, svincolandosi così da problemi di eterogeneità. È interesse di IBM creare le premesse per una piattaforma indipendente dall'implementazione degli Agenti; nella gestione e migrazione degli agenti c'è stato uno sforzo di standardizzazione e integrazione con altri sistemi commerciali: IBM ha proposto uno standard, **Agent Transfer Protocol (ATP)** [ATP97], presentato all'OMG affinché sia integrato in CORBA. ATP dovrebbe essere la controparte per agenti del protocollo HTTP per le pagine Web; si tratta di un meccanismo che permette la trasmissione e l'interazione degli agenti indipendemente dal supporto ove sono stati realizzati ed eseguiti. Nella stesura di questo standard, ancora in fase di definizione, l'intento è quello di ottenere un protocollo in grado di soddisfare ogni possibile necessità. Attualmente sono state trattate le seguenti aree:

- **Agent Identifiers:** affinché gli agenti possano essere determinati univocamente deve essere associato un identificatore (in ATP 0.1 una stringa alfanumerica).

- **Naming degli Agenti Servitori:** si basa sul concetto di *Uniform Resource Identifier (URI)*, caratterizzato da un URL più l'identificatore di agente.

- **Agent trasportation:** la tecnica di comunicazione di più basso livello consiste nello scambio di messaggi; su questo modello viene basato il protocollo di comunicazione in grado di trasportare gli Agenti ed i loro messaggi remoti.

Questo protocollo aprirebbe Internet agli Agenti indipendemente dal supporto in cui sono realizzati, permettendone un controllo diretto che non sarebbe ottenibile se ogni supporto realizzasse un suo meccanismo di comunicazione e trasporto.

1.5.5 Sumatra

Sumatra è un ambiente sviluppato all'Università del Maryland [Ach97] che supporta Agenti Java. Al contrario di Mole e Aglets, Sumatra fornisce un meccanismo di Strong Mobility; per ottenerlo però è stata necessario modificare la macchina virtuale Java (in seguito verrà specificatamente

analizzata la mobilità in Java) e questo ha comportato di fatto la perdita della portabilità assoluta.

È un ambiente orientato alla creazione di *Resource Aware Mobile Programs;* di applicazioni in grado di adattarsi dinamicamente ai mutamenti del sistema per ottenere migliori performance. Come in Agent Tcl, Sumatra si focalizza sulla percezione della dinamica del sistema, in particolare, sulle prestazioni della rete e delle connessioni (il sistema di monitoring è noto come **Komodo**). Un Agente deve possedere tre proprietà:

- **Awareness:** consapevolezza dell'ambiente in cui si trova; deve quindi essere in grado di monitorare quantitativamente e qualitativamente il sistema.

- **Agility:** è la capacità di reagire prontamente ai mutamenti del sistema; ci deve essere un supporto in grado di informare l'Agente dei mutamenti a tempo di esecuzione.

- **Authority:** deve essere possibile controllare e verificare l'uso delle risorse.

La prima proprietà è ottenibile attraverso un supporto di monitoring distribuito; le risorse possono essere monitorate sia *su richiesta (on-demand)* che *continuamente* (come sarà evidenziato nel prossimo capitolo).

L'**agilità** è caratterizzata da due meccanismi fondamentali: la possibilità di ricevere eventi in maniera asincrona e la capacità di muoversi in qualsiasi momento verso un sito remoto. Il primo meccanismo, simile ai segnali Unix, è contrario al modello base ad Agenti, che prevede che il controllo sia sempre gestito dall'agente nel suo flusso primario di esecuzione. In questo caso invece un evento può provocare l'invocazione di un Handler che obblighi il movimento dell'agente su un altro sito forzando la migrazione dell'agente in maniera trasparente (senza quindi intervento da parte dell'agente). La seconda proprietà si risolve nel possedere la mobilità di tipo Strong: è evidente che una mobilità Weak sarebbe incompatibile con un meccanismo asincrono di gestione degli eventi.

L'**authority** può essere gestita attraverso due forme di controllo: lanciando una eccezione in caso di ogni fallimento di autorizzazione o

considerando le violazioni come un evento asincrono e premettendo al programmatore di associare handler per la eventuale gestione.

Un agente in Sumatra è realizzato con un Thread Java in grado di utilizzare il comando **go()** per cambiare sito; la comunicazione avviene tramite Oggetti condivisi ed RMI, questi oggetti vengono aggregati in gruppi dinamici e possono essere mossi nella rete attraverso il comando **moveTo().**

1.5.6 Comparazione tra i Sistemi esistenti

Dall'analisi effettuata è possibile tentare una comparazione delle principali caratteristiche per evidenziare dove i diversi autori sono d'accordo su soluzioni ormai affermate e dove invece c'è divergenza di opinioni e c'è spazio per eventuali sviluppi e miglioramenti. I diversi sistemi sono stati analizzati confrontandoli sulle seguenti proprietà:

- **La Mobilità**
- **La Comunicazione**
- **Il Monitoraggio**
- **La Sicurezza**

La caratteristica principale è sicuramente la **Mobilità**, come cioè i vari sistemi forniscono la possibilità di migrazione, con che granularità, flessibilità ed astrazione; un elemento strettamente correlato è sicuramente il **Naming,** in quanto non è possibile una gestione distribuita degli agenti senza un sistema di nomi che la supporti. Di seguito vengono evidenziate le caratteristiche di Naming (Tab II) e Mobilità (Tab. III) dei vari Sistemi.

Come evidenziato in Tabella II, tutti i sistemi possiedono un identificatore unico e globale per l'agente. Risulta inoltre importante avere la possibilità di poter assumere dinamicamente diversi alias per semplificare l'interazione con gli altri Agenti. La mancanza di un servizio di informazioni in Sumatra e Aglets è dovuto all'idea dei progettisti di questi due sistemi che un servizio di questo tipo debba essere fornito a livello applicativo con, eventualmente, anche la possibilità di alias astratti.

	Identificatore Agente	Naming Dinamico	Servizio di Informazioni
Agent Tcl	Unico e globale	Sì globale (Yellow Pages)	Yellow Pages
ARA	Unico e globale	Sì locale (Place)	Sì
Mole	Unico e globale	Sì locale (Badge)	Sì (solo locazione agenti)
Aglets	Unico e globale	No	No
Sumatra	--	No	No

Tabella II: Caratteristiche principali del naming dei sistemi analizzati.

	Tipo Mobilità	Astrazione Nodo	Tipo Destinazione
Agent Tcl	Strong	No	URL
ARA	Strong	Place	URL o Place
Mole	Weak	Place	URL, Place o Agente
Aglets	Weak	AgletContext	URL
Sumatra	Strong	No	URL

Tabella III: Caratteristiche della Mobilità.

La Tabella III mostra come, per tutti i sistemi, sarebbe desiderabile fornire una mobilità di tipo Strong; ciò non è però ottenibile in Java standard (Sez. 3.2), quindi Aglets e Mole cercano di simulare il modello ad Agenti su un modello REV. La possibilità di gestire dei contesti svincolati dai nodi fisici sottostanti è molto diffusa e utile. In Mole viene fornita la possibilità di muoversi verso un determinato Agente, una funzionalità di più alto livello che necessita un'integrazione con un sistema di nomi globale in grado di verificare le posizioni degli agenti, ma permette una maggiore flessibilità nella programmazione degli Agenti.

Un'altra caratteristica molto importante per un ambiente ad Agenti è la **Comunicazione** non avendo grandi potenzialità un sistema in cui gli Agenti non possono interagire tra. Nella Tabelle IV e V sono evidenziate le peculiarità principali della comunicazione nei vari sistemi, come già evidenziato nel par 1.4.2 si può classificare la forma di comunicazionein base a quanto intenso sia lo scambio di informazioni, distinguendo così tra

interazione stretta e lasca. Un'altra classificazione possibile si basa sulla possibilità di comunicare con o senza la completa conoscenza del partner, si parla di *comunicazione anonima* quando i partecipanti dell'interazione non hanno nessun legame di conoscenza se non l'interazione stessa.

Non in tutti i sistemi vengono forniti diversi modelli di comunicazione per le diverse esigenze; in Agent Tcl, per esempio, tutto si basa sullo scambio di messaggi, mentre in Sumatra invece esistono solo gli oggetti condivisi. I meccanismi di comunicazione strettamente anonima sono la Blackboard e lo spazio delle Tuple, la prima astrazione è facilmente realizzabile in un contesto locale e quindi viene messa a disposizione da molti, lo spazio delle tuple invece necessita di un meccanismo di pattern matching più complesso ed attualmente solo Mole intende integrarlo nel supporto.

	Modello di Comunicazione	Necessità conoscenza reciproca	Trasparenza alla Locazione
Agent Tcl	Message Passing con Connessione fissa	Sì (si può ottenere dalle YP)	No
ARA	Message Passing in un Service Point	Sì	No
Mole	Shared Objects in un Meeting	No	Sì (RMI)
Aglets	Shared Objects	Sì	No
Sumatra	Shared Objects	Sì	Sì

Tabella IV: Caratteristiche dell'interazione stretta tra Agenti.

Per quanto riguarda le capacità di **Monitoraggio** i vari sistemi sono profondamente diversi: alcuni ne sono completamente sprovvisti, altri invece hanno un vero e proprio supporto dedicato al rilevamento dello stato del sistema essenzialmente per due obbiettivi principali: fornire informazioni sullo stato del sistema agli agenti (Sumatra e Agent Tcl) e controllo più dettagliato possibile dell'esecuzione per garantire la sicurezza del sistema.

	Modello di Comunicazione	Necessità conoscenza reciproca	Trasparenza alla Locazione
Agent Tcl	Message Passing	Sì (Y.P.)	No
ARA	-Message Passing -Blackboard	Sì No	No
Mole	-Message Passing -Spazio delle Tuple	Sì No	Sì No
Aglets	-Message Passing -Blackboard	Sì No	No No
Sumatra	--	--	--

Tabella V: Caratteristiche dell'interazione lasca tra Agenti.

Nella Tabella VI sono riassunte le caratteristiche principali del monitor nei diversi ambienti. In ARA l'attenzione è incentrata sulla sicurezza ed il monitor è un servizio di sistema e non concepito espressamente per gli Agenti. In Sumatra e Agent Tcl invece, lo scopo stesso degli Agenti è quello di sfruttare le informazioni sul sistema per migliorare il proprio comportamento ed il monitor in questo caso viene esplicitamente creato per gli Agenti.

La **Sicurezza** nel modello ad Agenti è fondamentale, non è pensabile fornire un servizio di esecuzione remota senza verificare l'affidabilità di cosa viene eseguito. Quasi tutti i Sistemi analizzati forniscono, o si impegnano a fornire a breve, meccanismi che permettano l'autenticazione degli agenti. In particolare, ARA fornisce un supporto completo anche per quanto riguarda i permessi di uso delle singole risorse con la possibilità di avere crediti temporanei che si consumano con l'uso delle risorse oppure hanno scadenze temporali.

	Presenza	Scopo	Precisione	Sonde
Agent Tcl	Sì	Servizio fornito agli Agenti	Qualitativo	-Rete -Stato Agenti
ARA	Sì	Controllo e Sicurezza	Non specificata	-CPU -Memoria -Rete -Agenti Servitori
Mole	No	--	--	--
Aglets	No	--	--	--
Sumatra	Sì	Servizio fornito agli Agenti	Quantitativo dipendente dalla sonda	-CPU -Rete (Latenza)

Tabella VI: Caratteristiche di Monitor nei diversi Sistemi.

1.6 Applicazioni nel modello ad Agenti

A livello di principio, qualsiasi tipo di Software distribuito può essere realizzato in paradigma MA; però alcune necessità implementative del supporto pongono dei significativi handicap in determinate condizioni. Abbiamo detto che, per superare l'eterogeneità, è necessario ricorrere ad un interprete anziché compilare gli Agenti in codice nativo; non ci si può certo aspettare che la velocità di esecuzione di un Agente interpretato sia paragonabile a quella di una qualsiasi applicazione nativa.

Le applicazioni, dovranno essere tipicamente *Net Bound*, cioè significativamente limitate dai tempi di comunicazione e non dalle risorse computazionali; questo non significa che le risorse computazionali siano inutilizzabili, ma semplicemente che solo gli algoritmi, in cui i costi di comunicazione tra entità sono molto elevati, possono trarne reale giovamento. Dato l'elevato grado di astrazione del modello, i costi di progettazione e sviluppo del Software dovrebbero essere nettamente inferiori al caso tradizionale; tutte le applicazioni in cui la prestazione pura non ha importanza ma è importante la duttilità e l'economicità, sono decisamente orientate verso questo modello.

1.6.1 Network Management

Il controllo e l'amministrazione di reti di grandi dimensioni necessita lo sviluppo di strumenti che permettano di eseguire in remoto le procedure di normale manutenzione dei nodi.

Una delle necessità quotidiane, ad esempio, è la verifica del corretto funzionamento di tutti i componenti della rete. Uno strumento di monitoring di questo tipo necessità di grande flessibilità e tolleranza ai guasti, in quanto uno dei suoi compiti primari è appunto la notifica ed eventualmente il ripristino di situazioni di errore attraverso comportamenti adattativi

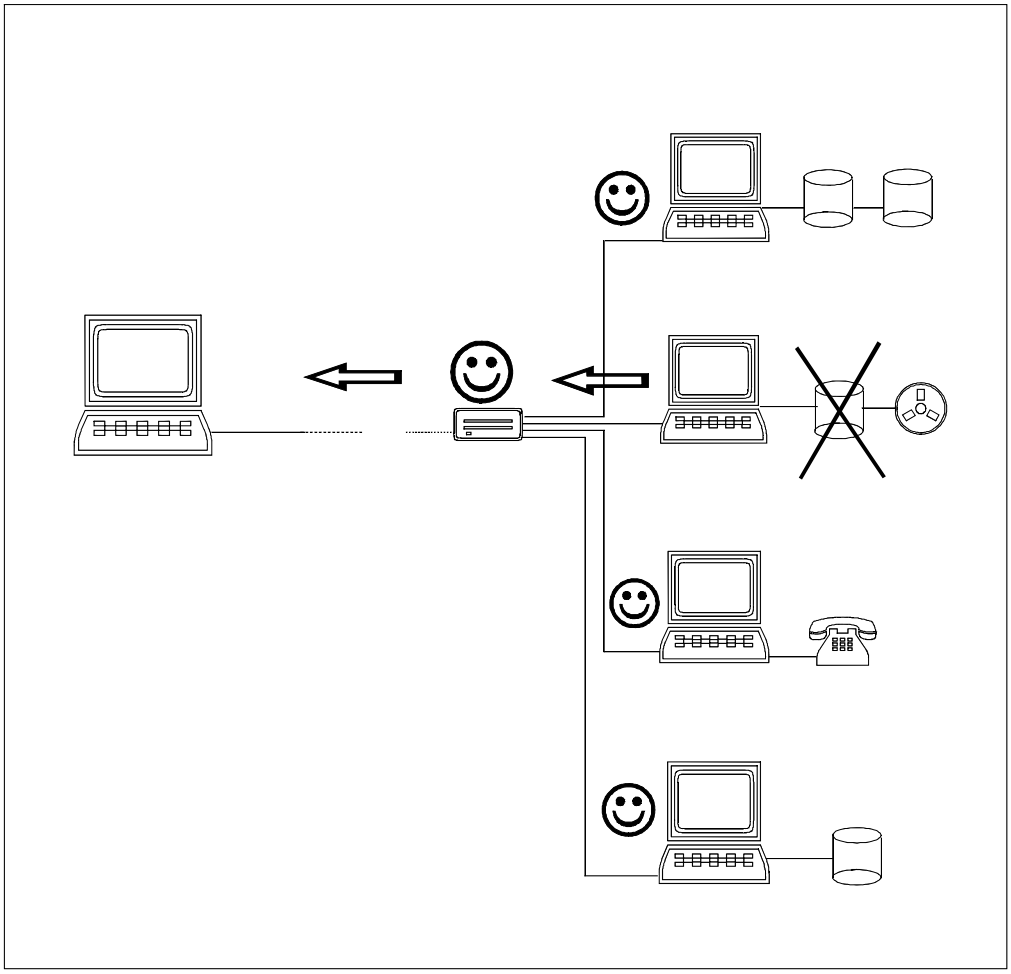

Figura 8: Un Network Manager può essere caratterizzato da un gruppo di *Agenti sentinella* che verificano il corretto funzionamento dei nodi ed eventualmente, in caso di anomalie, provvede a notificare l'errore ad un nodo di riferimento.

I sistemi esistenti e gli standard proposti [BaGP97], sono caratterizzati da un alto grado di centralizzazione, per la difficoltà di gestire un supporto completamente distribuito con un paradigma Client-Server.

Un sistema ad Agenti potrebbe soddisfare i requisiti necessari predisponendo delle *sentinelle*, che si muovono nella rete (o stazionano nei nodi) controllando il comportamento del sistema. In caso di anomalie o guasti possono muoversi verso la *Stazione di Controllo* e notificare l'avvenuto. Un

approccio di questo tipo, è completamente decentrato ed autonomo. La tolleranza ai guasti può essere ottenuta a vari livelli: a livello di meccanismo, assicurandosi che l'agente sia arrivato correttamente sul nodo destinazione, prima di eliminare la sua copia sorgente; a livello di programmazione degli Agenti, implementando ad hoc specifici comportamenti per il recupero e la notifica dei guasti.

1.6.2 Information Retrieval

Come già esposto precedentemente, la ricerca ed il recupero di informazioni remote è nettamente *Net Bound.* Strutturando l'applicazione ad Agenti, non solo si riduce enormemente la quantità di dati trasferiti, ma è possibile ottimizzare e specializzare la ricerca oltre gli strumenti forniti dal server. Quando si utilizzano connessioni non permanenti (accessi Modem o ISDN), è anche possibile ridurre i costi di esercizio dando un *appuntamento* all'Agente Cercatore, non mantenendo la connessione durante l'elaborazione remota. La ricerca *off-line*, è possibile anche in sistema Client/Server: è necessaria però una gestione specifica attraverso una richiesta asincrona ed una seguente richiesta di download dei risultati dopo il ripristino della connessione; nel modello ad Agenti il programmatore non deve invece preoccuparsi dei dettagli del meccanismo di ritorno dell'Agente che può essere integrato in un supporto orientato a questo tipo di applicazioni.

1.6.3 Applicazioni coordinate per Gruppi di lavoro

Un campo di importanza crescente è quello delle applicazioni orientate al coordinamento e all'interazione di più utenti in un lavoro di équipe.

La possibilità di utilizzare i più diversi strumenti, dal Word Processor al CAD, in più utenti contemporaneamente sugli stessi lavori, è essenzialmente legata ad un problema di sincronizzazione per il mantenimento della consistenza dei dati condivisi. Un coordinamento gestito ad Agenti (per esempio potrebbe essere associato un agente per ogni lavoro e l'utente, quando desidera operare delle modifiche allo stato, deve prima chiamare a se l'agente), può essere vantaggioso e permettere la condivisione anche di parziali elaborazioni o di stati di esecuzione.

Un'altra necessità molto sentita nel lavoro di équipe è la forte interattività nella comunicazione e nello scambio di informazioni. La condivisione di File System o la Posta Elettronica sono poco immediati e per nulla interattivi. Possono essere quindi molto interessanti strumenti come la *Scrivania Distribuita*, un ambiente in cui gli utenti possono porre note, messaggi, file di vario genere e natura rendendoli disponibili ai componenti del gruppo. Uno strumento del genere può essere realizzato come sistema di place, dove alcuni Agenti si occupano dell'interfaccia con gli Utenti (remoti) ed altri della gestione degli oggetti che dinamicamente entrano ed escono.

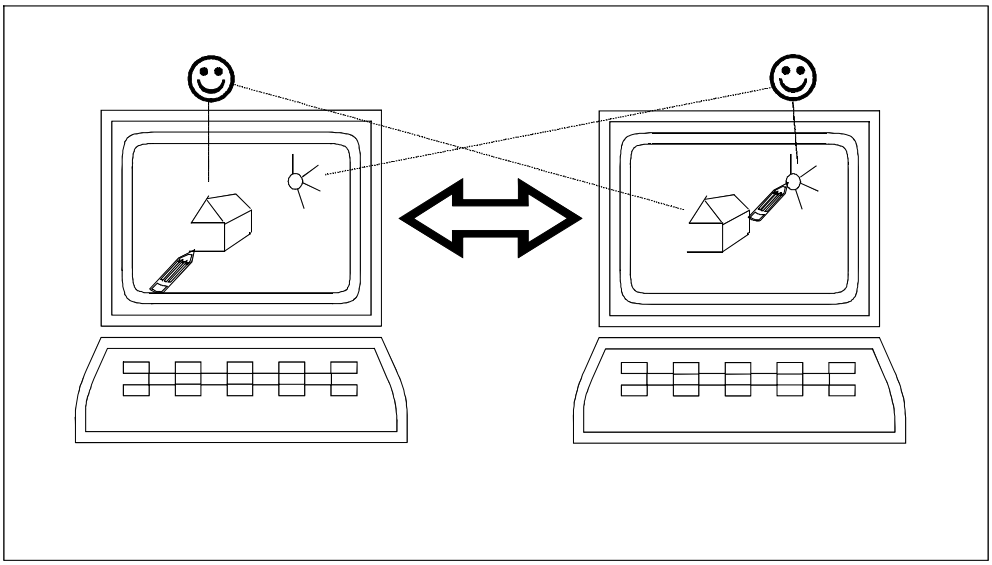

Figura 9: Per mantenere la consistenza ed il coordinamento in un lavoro di gruppo, si può associare ad ogni oggetto un Agente che ha il compito di verificarne la sincronia in tutti i nodi.

1.6.4 Soluzioni per il Commercio Elettronico

Per Commercio Elettronico si intende la possibilità di effettuare transazioni commerciali senza la necessità di depositare un contratto firmato su supporto cartaceo e con forme di pagamento non tradizionali. Le problematiche sono molteplici, tuttora in fase di studio e sono essenzialmente

correlate a necessità di sicurezza nei pagamenti e all'autenticazione dell'identità dei partecipanti alla transazione; ciò evita rischi di potenziali truffe o illeciti. Si tratta quindi di problematiche di coordinamento *sicuro* nelle interazioni estremamente variabili tra diversi utenti. Un approccio ad Agenti potrebbe permettere non solo una gestione sicura del passaggio dati, ma anche una notevole flessibilità di adattamento alle diverse esigenze. Per esempio, si potrebbero progettare agenti in grado di ricercare su Internet prodotti con particolari caratteristiche, o per scegliere, definito un prodotto, quello con il prezzo migliore.

1.6.5 Interazione IntraNet-InterNet

Al crescere della complessità e della struttura dei sistemi di rete e con la necessità di controllo e sicurezza dei sottosistemi collegati, l'uso di risorse remote (come un database aziendale) è diventato sempre più problematico. L'adozione della tecnologia HTML e della suite TCP/IP ha portato all'integrazione dei sistemi locali in Internet (da qui il termine *IntraNet*) ma non ha semplificato l'uso delle risorse dei sistemi remoti. Le moderne tecniche di protezione sono basate sull'uso di *Firewall* i quali permettono, previa autenticazione, l'accesso alle risorse interne solamente attraverso protocolli standard e non consente nessun tipo di connessione di tipo applicativo. Il meccanismo di migrazione può usufruire di protocolli standard per il trasporto; un sistema ad Agenti, può rappresentare il miglior strumento per la programmazione in questo tipo di ambienti, fornendo la possibilità, non solo di creare applicazioni *Trans Firewall* ma anche servizi integrati per l'uso di risorse, sia su scala locale che geografica.

1.6.6 Giochi multi utente

Ci sono molti tipi di giochi di simulazione che necessitano l'interazione di più utenti: non solo quelli puramente ludici, ma anche molte applicazioni educative o di training aziendale. Le problematiche sono essenzialmente simili a quelle già viste per le applicazioni per il lavoro di gruppo. Si tratta di ottenere degli ambienti distribuiti all'interno dei quali gli utenti devono

essere messi in condizione di agire nel modo più naturale ed intuitivo possibile.

I *MUD (Multi User Dungeons)*, ad esempio, sono intuitivamente realizzabili ad Agenti; i place possono rappresentare i vari ambienti del gioco, alcuni Agenti si occupano di gestire i personaggi non-giocatori, mentre altri dell'interfaccia remota con gli Utenti.

1.6.7 Place Distribuiti e creazione di Politiche di Load Balancing

In alcuni casi, può essere interessante realizzare dei place gestiti da più nodi, in questo caso il supporto deve fornire la trasparenza all'allocazione sui sotto-nodi. Esistono numerose situazioni in cui i servizi forniti da un place possono essere gestiti in maniera distribuita: il place può rappresentare l'interfaccia di utilizzo di una macchina parallela, come risorsa computazionale, oppure l'accesso ad un Database distribuito. In questa situazione l'Agente deve essere svincolato dal controllo della sua posizione, e deve poter *vedere* il place in maniera unitaria. Il problema torna ad essere focalizzato sull'uso omogeneo delle risorse e sul bilanciamento del carico.

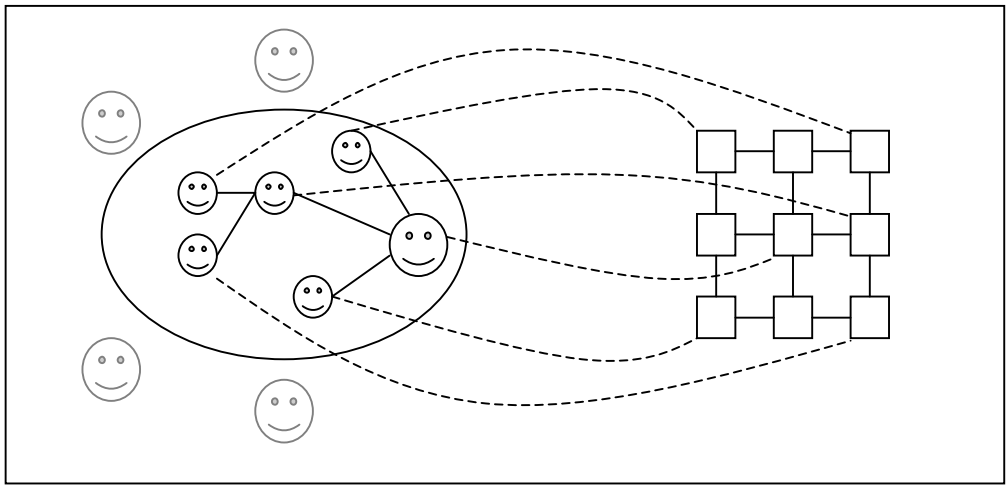

Figura 10: Place realizzato per gestire una macchina parallela come risorsa computazionale; gli *Agenti di Sistema* (in chiaro) si occupano del bilanciamento del carico e della gestione delle risorse; gli *Agenti Utente* si disinteressano completamente della loro posizione locale.

Le politiche di Load Balancing possono essere realizzate attraverso *Agenti di Sistema*, che lavorano ad un livello più basso fornendo l'astrazione di Place Distribuito agli *Agenti Utente*. In questo modo è possibile realizzare politiche completamente decentrate e distribuite in una logica completamente innovativa rispetto a quelle tradizionali.

1.7 Caratteristiche essenziali di un ambiente ad Agenti

Un sistema ad Agenti, in grado di supportare i diversi tipi di applicazioni evidenziate nella sezione 1.6, deve possedere varie caratteristiche, molte delle quali già sperimentate nei sistemi analizzati nella sezione 1.5. Le problematiche fondamentali, per la qualità del supporto sono essenzialmente: l'**Allocazione,** la **Comunicazione,** i **Place,** la **Sicurezza,** il **Monitoring** e la **Gestione delle informazioni** (figura 11)**;** possono essere affrontate in vario modo e di seguito vengono analizzate nel particolare (per quanto riguarda il Monitoring, verrà trattato per esteso nel cap.2).

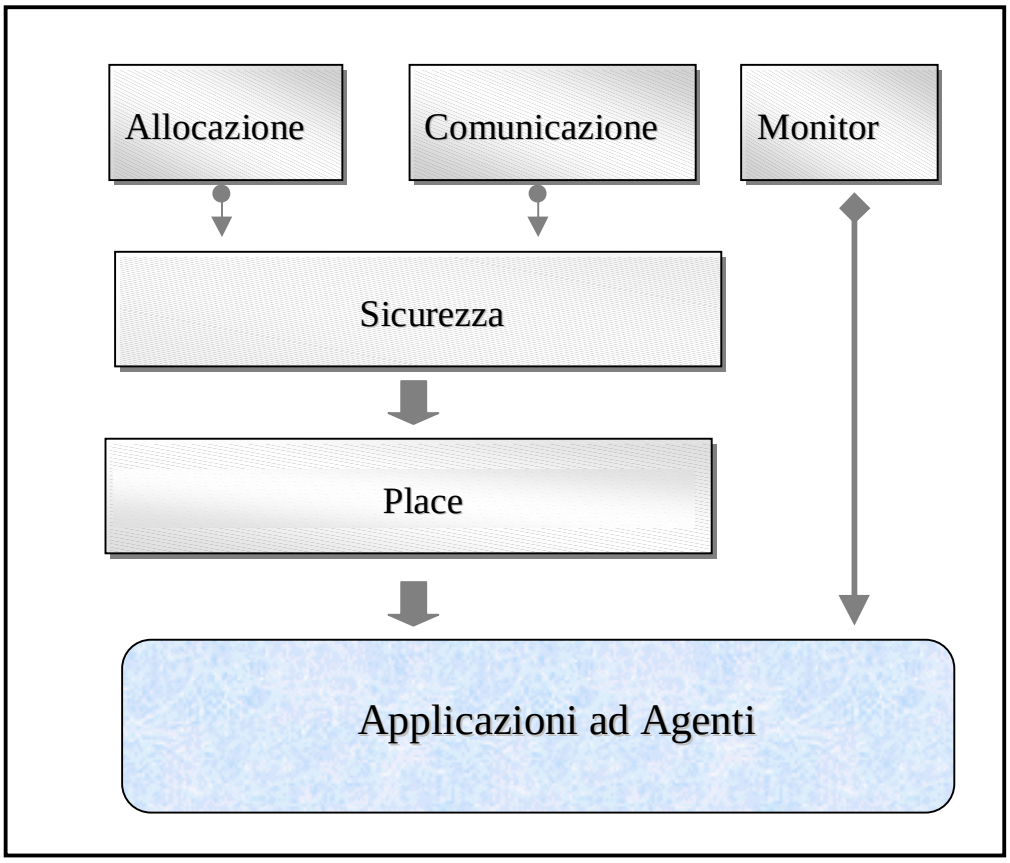

Figura 11: Le problematiche fondamentali sono strettamente correlate tra loro, imponendone una risoluzione unificata.

1.7.1 L'Allocazione

Il servizio più importante per la funzionalità dell'ambiente è la gestione della mobilità. Si tratta di mettere a disposizione degli Agenti la possibilità di muoversi in maniera esplicita da un sito ad un altro superando eventuali scogli quali l'eterogeneità o l'inaffidabilità delle linee di comunicazione. Il tipo di mobilità desiderabile è sicuramente di tipo Strong, un meccanismo molto complesso che impone l'uso di un interprete. Infatti, come già evidenziato nel par.1.3.2, le problematiche nell'uso di codice nativo in presenza di eterogeneità sono molteplici e di difficile soluzione, ragione per cui la scelta di eseguire gli Agenti in una macchina virtuale risulta sicuramente la più economica se non la più efficiente.

Per quanto riguarda la scelta dell'interprete, bisogna considerare che non esistono attualmente molti linguaggi che permettono Strong Mobility, si tratta per lo più di prototipi o esperimenti che non hanno ancora raggiunto la maturità necessaria per permetterne l'uso a terze parti per lo sviluppo di un supporto. La scelta quindi non può che cadere sul linguaggio dotato di mobilità più diffuso attualmente, **Java** [Java]. Java purtroppo non è dotato di Strong Mobility, non è quindi possibile realizzare perfettamente il modello ad Agenti; si tratta di ottenere un compromesso migliore possibile come in Aglets e Mole, confidando in una eventuale integrazione della mobilità Strong nelle future release.

Un altro particolare significativo per la gestione della migrazione è la scelta del protocollo di interazione tra i gestori dei diversi siti. La scelta di adottare un protocollo standard come HTTP è molto interessante per quanto riguarda la gestione di eventuali FireWall poiché questi sistemi di protezione sono in grado di gestire efficacemente solo protocolli di tipo standard quindi non permettono altri tipi di passaggio. Il protocollo HTTP però presenta dei grossi limiti espressivi che rendono molto inefficiente l'interazione. Il protocollo proposto da IBM, ATP non è ancora definitivo ne largamente accettato [ATP97], la soluzione ideale potrebbe essere la realizzazione di un supporto modulare in grado di agganciare dinamicamente diversi tipi di protocollo in maniera trasparente al gestore della migrazione; in questo modo è possibile creare, a seconda delle necessità, una connessione proprietaria oppure standard senza che il sistema debba subire delle modifiche.

Un'altra problematica legata alla migrazione è il livello di astrazione disponibile: è importante infatti rendere disponibili diversi contesti astratti al di sopra del supporto fisico e dare all'agente la possibilità di muoversi verso una destinazione non solo attraverso l'esplicitazione del sito ma anche attraverso il riferimento ad un servizio oppure ad un agente (*vai dove si trova l'agente XX*). Queste funzionalità di più alto livello semplificano moltissimo la programmazione dell'agente, coinvolgendo però il meccanismo di gestione delle informazioni ed il naming: per questo motivo devono essere realizzate a livello superiore, al di sopra di un *core* che fornisca solo i meccanismi di base.

1.7.2 I Place

L'astrazione di Place permette di identificare dei contesti ristretti all'interno dei quali è possibile ottenere specifici servizi. Nei place gli agenti possono avere interazioni strette tra loro in ipotesi di località: indipendemente dalla natura fisica del nodo infatti il place rappresenta una località per gli Agenti. Per omogeneità di gestione, è utile che tutti i servizi disponibili (oltre a quelli standard del linguaggio) vengano forniti da Agenti Servitori, i quali, hanno il compito di creare una interfaccia nel place di tutti i possibili servizi erogabili da un determinato sito, indipendemente che questi siano stati pensati originariamente per gli Agenti o meno.

Un Agente può quindi poter chiedere di aprire dinamicamente un proprio place che può ereditare o meno alcune delle proprietà del place padre. Si tratta quindi di realizzare un place di default statico, dove sono disponibili solo i servizi di basso livello del sistema (migrazione, comunicazione e monitoring) e permettere poi, ad Agenti fidati, la creazione dinamica di contesti per le diverse possibili necessità.

1.7.3 La Comunicazione

Anche la comunicazione deve essere disponibile a diversi livelli di astrazione e di trasparenza. Il modello ad Agenti suggerisce che tutti i servizi che coinvolgano la rete debbano essere forniti in maniera non trasparente. In questo modo l'agente stesso è consapevole di effettuare un accesso remoto e

quindi è in grado di prevederne il costo. Tutti i servizi di interazione devono essere trasparenti in locale e necessitano di una richiesta esplicita per il remoto. L'unica eccezione può essere rappresentata dal message passing: è il meccanismo di più basso livello, la sua implementazione trasparente non è particolarmente costosa quindi può essere significativo rendere disponibile il servizio in ogni occasione per semplificare il coordinamento lasco tra agenti che, eventualmente, possono poi muoversi per una interazione più stretta.

Per risolvere tutte le necessità degli agenti, il sistema deve supportare un meccanismo di *interazione lasca*, uno di *interazione stretta* e permettere la *comunicazione anonima*:

- *Message Passing*: interazione lasca eventualmente trasparente alla locazione.
- *Condivisione Oggetti*: interazione stretta in ipotesi di località.
- *Spazio delle Tuple*: permette la comunicazione anonima tra entità in modo più avanzato di una *Blackboard* non strutturata.

Il modello di interazione stretta più generale è la condivisione di oggetti locali. In un linguaggio ad oggetti come Java, l'*oggetto sincronizzato*, può essere usato localmente attraverso il suo riferimento che può essere facilmente condiviso. Per quanto riguarda un eventuale accesso remoto la funzionalità può essere fornita da un agente servitore che si sposta localmente per accedere e poi torna per fornire la risposta.

Come modello base per la comunicazione anonima, può essere reso disponibile un meccanismo di *pattern matching* per ottenere l'astrazione di **spazio delle Tuple** che permette la massima espressività. Anche in questo caso la gestione deve essere strettamente locale, anzi se lo spazio delle tuple viene fornito come servizio di sistema, è possibile creare più spazi indipendenti in vari contesti e permetterne così l'uso non solo come mezzo di comunicazione comune, ma anche come astrazione per realizzare diversi servizi come ad esempio il sistema di gestione delle informazioni.

1.7.4 La Gestione delle Informazioni

In un sistema aperto, il servizio di informazioni è fondamentale per poter ottenere in modo dinamico informazioni di vario genere soprattutto a proposito dei siti su cui è possibile trovare i vari servizi e le posizioni, i nomi degli Agenti correntemente in vita, le posizioni di specifici place che si desidera raggiungere.

Questo tipo di servizio si ottiene tramite un database distribuito, la gestione del quale è realizzata ad Agenti ad esempio in modo gerarchico come in Agent Tcl. Per quanto riguarda la forma del database, potrebbe essere realizzato con diversi spazi delle tuple divisi nei vari nodi. Questa realizzazione permette un accesso ai dati per attributi rendendo molto più potente e flessibile il servizio.

1.7.5 La Sicurezza

Il problema della sicurezza è essenzialmente legato al *permesso* di un Agente di utilizzare o meno determinate risorse. Prima di tutto deve esserci il modo di poter autenticare un Agente quando arriva in un sito; questo è ottenibile attraverso la verifica di una firma digitale del mittente che permetta di verificare la provenienza dell'agente. In seguito bisogna verificare se e quanto un agente può usufruire delle risorse locali in base ai permessi che possiede il suo proprietario, *consumando* eventualmente i suoi crediti di uso di una specifica risorsa.

Dal punto di vista dell'agente, si deve poi garantire la privacy del suo stato interno e l'integrità del suo codice da accidentali o maliziose interazioni da parte di altri agenti. In questo caso è necessario ricorrere alla crittografia ed alla verifica della correttezza del codice trasmesso tra un sito e l'altro mediante *message digest.*

Una gestione simile a quella presentata in ARA permette di verificare efficientemente l'identità degli agenti in arrivo e di gestire successivamente gli accessi alle risorse locali, con il sistema nel ruolo del garante delle credenziali del cliente. Un Agente ha a disposizione una *allowance* per ogni risorsa e, una volta consumata viene sollevata una eccezione di sicurezza oppure, nei casi più gravi, viene abortito l'agente trasgressore.

2 Rilevazione dello Stato del Sistema

Per Monitor si intende un sistema, distribuito o meno, software o hardware od entrambi, che raccolga informazioni, tramite appositi *sensori*, sullo stato del sistema *target* e che, quindi, fornisca una interpretazione di queste ultime, rendendone eventualmente una visualizzazione. Le *sonde*, o sensori, reagiscono a variazioni di stato del sistema monitorato, variazioni notificate al monitor, il quale quindi deve fornirne una interpretazione, molto spesso resa in forma di *Eventi* (operazioni o insiemi di operazioni, eseguite all'interno dell'applicazione monitorata, che vengono ritenute di interesse ai fini della comprensione della stessa) e reagire di conseguenza. A seconda delle funzioni del Monitor si può trattare di semplici visualizzazioni oppure di vere e proprie azioni; infatti, come si vede in seguito, il sistema di monitoraggio può essere progettato per compiere determinate azioni al verificarsi di determinati Eventi.

Un particolare di estrema importanza è la decisione di quale livello di astrazione usare, cioè definire quale classe di eventi sarà di interesse, ovvero il livello di dettaglio con cui analizzare l'applicazione.

I problemi principali del monitoraggio sono:

- **Osservabilità**: la possibilità concreta, da parte del Monitor, di raccogliere ed interpretare i dati.

- **Intrusione:** le alterazioni che il Monitor stesso può introdurre nel sistema monitorato.

- **Completezza** dell'insieme degli eventi monitorati: la capacità di descrivere ogni possibile comportamento del sistema *target* (in pratica difficilmente raggiungibile).

- **Correttezza** delle interpretazioni fornite.
- **Temporizzazione:** il problema riguardante i modi in cui misurare i tempi e, a sua volta, i tempi tipici del Monitor.

2.1 Classificazione dei sistemi di Monitoraggio

Esistono differenti modi per classificare i Monitor, secondo sotto quali aspetti si analizzi il sistema.

Come enunciato in [GuVS94], a seconda del modello scelto per descrivere il sistema monitorato cambia di conseguenza il Monitor; esistono essenzialmente due tipi di approccio.

2.1.1 Approccio orientato agli eventi

Come già esposto in precedenza un evento è un *comportamento* caratteristico dell'applicazione osservata. La modellazione orientata agli eventi consente la composizione di comportamenti del programma, a più alto livello, partendo da un flusso di eventi che rappresentano le attività del sistema monitorato. I vantaggi di questo tipo di approccio sono la semplicità nella formulazione e nella interpretazione degli eventi da parte dell'utente e la possibilità di standardizzare il formato degli eventi con possibilità di condivisioni di informazioni tra diversi sistemi di analisi e visualizzazione.

Gli esempi di questo tipo di approccio sono innumerevoli quali ZM4/SIMPLE [Daup92], JEWEL [LaKG92] e molti altri.

2.1.2 Approccio relazionale

Questo tipo di approccio, proposto per la prima volta da Snodgrass in [Sno82], tratta le informazioni estratte (dati a tempo d'esecuzione, stati dei processi, messaggi, etc.) come relazioni, come dati di un database

relazionale. La specifica delle attività e degli stati del programma osservati dal monitor vengono specificate tramite un *query language*, ovvero una estensione dei linguaggi di interrogazione dei database; il primo logico ampliamento è stato fare riferimento a basi di dati temporali.

Esempi di questo tipo di approccio sono il Tquel di Snodgrass [Sno87] e il ChaosMON [KiSc91] di Kilpatrick e Schwan.

2.2 Raccolta delle informazioni

Un altra suddivisione tra gli approcci scelti per il monitoraggio riguarda l'orientamento seguito per la raccolta delle informazioni. Si noti, comunque, che, indipendentemente dalla scelta dell'orientamento, entrambi i casi, non prevedono una raccolta completa di informazioni:

- **Orientato ai tempi (time-driven):** in questo caso la raccolta dei dati scelti come caratteristici avviene ad intervalli di tempo prefissati, funzione della frequenza di campionamento scelta. Questo approccio facilita una valutazione di carattere statistico del sistema.

- **Orientato agli eventi (event-driven):** seguendo questo orientamento verranno registrate le esecuzioni di tutte le operazioni riguardanti eventi scelti (coerentemente col livello di astrazione). Questo orientamento consente una maggior comprensione del funzionamento e del comportamento del sistema monitorato.

2.2.1 Analisi dei dati

A seconda del momento in cui i dati raccolti vengono analizzati ed i risultati divengono disponibili parla di:

- **Analisi post-mortem o Monitor off-line:** le informazioni, raccolte durante l'esecuzione del programma, saranno analizzate ed i risultati

restituiti soltanto *dopo la morte* dell'applicazione stessa (come ad esempio in Instant Replay [LeMC87]).

- **Analisi on-line o Monitor on-line:** le informazioni ed i risultati dell'analisi delle stesse sono rese disponibili durante l'esecuzione del programma e possono essere quindi usate per la conoscenza, aggiornata dinamicamente, dello stato attuale del sistema monitorato. Molto spesso questo secondo tipo di approccio porta a considerare il Monitor parte integrante e quindi permanente dell'applicazione. Con ciò si può considerare differentemente la perturbazione introdotta da un Monitor off-line ed uno on-line; infatti nei casi in cui il Monitor non sia un elemento permanente bisogna sempre ritenere che il comportamento dell'applicazione , in presenza od in assenza dello stesso, sia differente (basti pensare all'introduzione di alterazioni delle temporizzazioni).

2.3 Mapping e Monitoring

Come già precedentemente esposto, il monitoring è fondamentale nella realizzazione di soluzioni adattative al problema del Mapping; affinché sia utile in queste applicazioni, il supporto di rilevazione deve avere le seguenti caratteristiche:

- **On-line:** è evidente che, per il mapping, il monitor debba fornire i dati prima possibile e sicuramente on-line.

- **Non intrusione:** come per tutti i servizi *di sistema* vale il principio di non intrusione: *l'uso delle risorse per eseguire il servizio deve essere trascurabile rispetto al consumo delle applicazioni*. La presenza o meno del supporto non deve influenzare le prestazioni delle applicazioni in esecuzione. Nel caso del monitoring, questa necessità diventa una specifica forte di progetto: la rilevazione dei dati è spesso molto costosa ed il costo cresce con la precisione che si desidera ottenere. È importante trovare il giusto trade-off tra costo e prestazioni a seconda delle necessità del caso.

- **Indipendenza dal nodo di rilevazione:** essendo lo scopo principale la rilevazione dello stato del sistema per fini di adattamento, è necessario che il monitor sia distribuito, in grado non solo di collezionare lo stato di un nodo, ma anche quello degli altri (per lo meno dei vicini) e della rete. In presenza di eterogeneità, intesa non solo come diversità di architetture ma anche come semplice differenza di configurazione tra macchine compatibili, è necessario che i dati rilevati in un nodo siano comparabili con quelli rilevati in altri nodi; solo in questo caso è possibile avere un'idea globale dell'intero sistema.

- **Astrazioni e filtri:** non ha senso fornire grandi moli di dati la cui interpretazione è spesso difficile ed in generale costosa. È importante vengano forniti vari livelli di astrazione che permettano di ottenere quadri di insieme anche se meno precisi. La possibilità di inserire filtri programmabili per selezionare solo alcune informazioni piuttosto che

altre o per collezionare statistiche riassumibili in pochi indici permette inoltre una maggiore flessibilità e risposta del sistema.

Queste caratteristiche sono ottenibili con varie tecniche la cui validità non è in generale assoluta, dipende spesso da tutto quello con cui interagisce il monitor: dal livello in cui si colloca (di sistema o utente) e dal tipo di risorse che intende monitorare.

2.4 Il monitor On-line

I monitor on-line, può adottare due diverse tecniche per la rilevazione dei dati [Ach97]:

- **Continuos Monitoring:** si rileva costantemente (con una certa granularità temporale) lo stato della risorsa in esame; quando un cliente richiede i dati prodotti, possono essere forniti sotto forma di indici statistici o di un flusso di informazioni. Questa forma di rilevamento è significativa se il costo di analisi della risorsa è molto basso e sono necessari dati che coinvolgono un determinato lasso di tempo (ad esempio, lo stato di carico della CPU è interessante come media di un determinato periodo e non come rilevazione di un istante perché la risorsa è soggetta a forti fluttuazioni temporanee non interessanti).

- **On-demand Monitoring:** se la risorsa è sondabile solo ad un costo molto elevato oppure il suo stato è immediatamente rilevabile (non sono necessarie medie temporali), non è logico un monitoring di tipo continuo. In questo caso la tecnica più efficiente consiste nell'effettuare la rilevazione all'atto della richiesta da parte di un cliente. Questo tipo di rilevazione ha il vantaggio di essere *stateless* (il suo valore non dipende da dati passati) semplificando la gestione stessa del monitor.

A seconda della natura della risorsa è importante scegliere la forma di rilevazione più adatta; il vincolo principale rimane sempre quello della minima intrusione, ma le necessità di rilevamento non vanno trascurate affinché il supporto possa essere di una qualche utilità.

2.4.1 La granularità

Un altro fattore significativo per l'efficacia del sistema è la granularità di rilevazione: è importante ottenere il giusto equilibrio tra costo e precisione.

La precisione con cui si desidera ottenere determinate informazioni può essere estremamente variabile: in alcuni casi può essere sufficiente una risposta binaria (si/no), in altre situazioni invece può essere importante ottenere un dato con molti bit di precisione.

Se il monitoring serve per risolvere il problema del mapping, in realtà la precisione non è veramente importante, interessa molto di più la prontezza e l'efficienza del sistema. La granularità sarà quindi abbastanza grossa, cosicché sia possibile una semplice acquisizione ed elaborazione dei risultati ottenuti.

2.4.2 Il monitor nei sistemi ad Agenti

In alcuni sistemi ad Agenti è disponibile un sistema di monitoring che permette di realizzare applicazioni in grado di adattarsi dinamicamente alle caratteristiche del sistema. I servizi di monitoring possono in questo caso essere divisi in due tipi: verifiche **qualitative** ed analisi **quantitative.** Il primo tipo di servizio permette di verificare l'esistenza o meno di una determinata risorsa (*esiste il nodo XX?*) mentre il secondo fornisce informazioni di tipo quantitativo (*quanto costa arrivare al nodo XX?*). Nel modello ad Agenti, sono considerate risorse qualsiasi tipo di servitore e quindi in generale qualsiasi altro Agente: deve quindi essere disponibile del monitoring sugli Agenti.

In **Komodo** [Rang97], il supporto di monitoring del sistema Sumatra, viene fornito il servizio continuo e on-demand, con una particolare attenzione alla massima efficienza ed alla minima intrusione.

Il monitor on-demand viene utilizzato soprattutto quando lo stato della risorsa varia lentamente oppure il costo di rilevazione è molto elevato rispetto al possibile vantaggio ottenibile con un monitoring continuo. Questo tipo di monitor è necessariamente sincrono, ha un costo limitato, ma non permette di rilevare veloci cambiamenti del sistema o di tracciare andamenti temporali di una risorsa.

Il monitor continuo è molto utile se si devono controllare risorse molto usate, può avere una interfaccia sia sincrona che asincrona ed è in generale molto costoso; per questo motivo è molto importante determinare la giusta granularità nel rilevamento dei dati. Siccome la granularità dipende essenzialmente dal tipo di risorsa in esame, la soluzione più flessibile consiste nel realizzare dei filtri che estrapolino indici significativi a partire dal monitor continuo sulla risorsa.

In **Agent Tcl** [RuGK97], esiste un semplice supporto di monitoraggio, caratterizzato da una granularità molto grossa e fornisce informazioni soprattutto di natura qualitativa.

Ci sono tre tipi fondamentali di *sensori*, focalizzati sullo stato dell'Hardware, del Software e degli altri Agenti. I sensori per l'Hardware sono orientati soprattutto allo stato della rete:

- **Local Connectivity:** verifica la connessione dell'host locale alla rete, viene effettuato un *ping* all'indirizzo broadcast della sottorete locale per verificarne il corretto funzionamento.

- **Site Reachability:** verifica se uno specificato sito sia *up* o *down* sempre attraverso il comando Unix *ping*.

- **Network Load:** verifica la banda passante per un nodo specifico, il cui indice di riferimento è la latenza che viene estrapolata dalle tabelle locali sulle informazioni di traffico. La classificazione è a soglie (0-15sec, 15-30sec,…, più di 2 settimane).

Attraverso questi sensori, l'Agente può avere comportamenti adattativi all'evoluzione dello stato dell'ambiente; può però anche realizzare politiche di tolleranza e rilevamento dei guasti e funzionalità generiche di Network Management. Non è stato però preso in considerazione il caso in cui la sonda possa andare incontro a malfunzionamento in seguito al *crash* di una collegamento durante il rilevamento; questo può eventualmente indebolire eventuali applicazioni orientate alla tolleranza ai guasti.

I sensori per il Software si riferiscono al controllo dello stato dei servizi e dei dati. È possibile verificare se un server è attivo o meno, se un file è stato modificato di recente; queste funzionalità danno agli Agenti capacità di Monitoring e controllo su risorse distribuite.

2.4.3 Il superamento dell'eterogeneità

La necessità di confrontare dati rilevati su macchine diverse rende necessario ideare un qualche metodo di paragone, bisogna determinare delle situazioni campione su cui **normalizzare** i risultati ottenuti su diverse piattaforme [CoSt97]. La maggiore difficoltà consiste nel trovare questi

campioni: non devono infatti avere caratteristiche tali da privilegiare una architettura piuttosto che un'altra ma devono essere più possibile obbiettivi.

Consideriamo ad esempio la rilevazione del carico della CPU: se interessa decidere quale tra un gruppo di macchine è la più scarica perché si vuole prendere una decisione di bilanciamento del carico è necessario confrontare i dati ottenuti dai diversi rilevamenti. Una CPU molto veloce con un carico del 90% potrebbe essere un ricevente migliore di una molto lenta con un carico del 20%; è necessario normalizzare i risultati rispetto alle velocità relative delle due macchine. Supponiamo che la macchina A, con un carico del 90% abbia una velocità di 1000 Op/sec (Operazioni al secondo) e che la macchina B invece abbia un carico del 20% ed una velocità di 10 Op/sec; in un secondo la macchina A può eseguire ancora 100 Op mentre la B solo 8.

La misura del termine di paragone, il calcolo cioè di quante operazioni al secondo è in grado di eseguire un CPU è determinante. La scelta dell'operazione di riferimento potrebbe favorire una macchina che sa fare velocemente solo quella specifica operazione (o sequenza di operazioni) e minimizzare le capacità di altre architetture in realtà più potenti.

Superare l'eterogeneità quindi significa trovare dei campioni, delle unità di misura su cui uniformare i risultati ottenuti su diverse piattaforme. Non tutti i tipi di rilevazioni necessitano di questo tipo di gestione, molte risorse infatti sono astratte e quindi spesso sono già state progettate per nascondere eventuali differenze di sistema; in questo caso il monitor non ha bisogno di nessun tipo di accorgimento.

2.5 La rilevazione dei dati

I meccanismi per la rilevazione dei dati sono detti *sonde*: una sonda ha il compito di ottenere lo stato di una specifica risorsa e comunicare il risultato al sistema di registrazione e notifica. La forma della sonda è strettamente dipendente dalla risorsa e dalle necessità del monitoring. Nei prossimi paragrafi verranno analizzate le risorse più interessanti in un sistema ad Agenti e le tecniche con cui è possibile ottenere diverse informazioni.

2.5.1 La CPU

Rilevare lo stato della CPU significa misurare la percentuale di tempo in cui esegue in un determinato intervallo. In un sistema multi-tasking in time sharing, la CPU viene gestita in quanti di tempo: una EU ottiene un quanto per eseguire ed allo scadere deve rendere il controllo del processore al sistema. Quando non ci sono EU pronte ad andare in esecuzione (*ready*) lo scheduler si blocca in attesa di una richiesta e la CPU entra in *idle*.

Per effettuare questa misura a livello di sistema, è sufficiente contare in un tempo fissato il numero di quanti utilizzati rispetto a quelli disponibili, in questo modo si ottiene direttamente la percentuale d'uso.

A livello utente la gestione della CPU è completamente nascosta, non è possibile ottenere il conteggio dei quanti di tempo direttamente, è necessario un qualche meccanismo che estrapoli indirettamente questo dato.

La tecnica più comune consiste nel creare un processo *dummy* ad alta priorità (più alta dei processi da monitorare). Ogni volta che avviene uno scheduling, esso viene sempre eseguito per primo: è così possibile misurare quanto tempo è passato dal precedente cambio di contesto. Attraverso questa misura indiretta si ottiene, con una certa approssimazione, per quanto tempo eseguono le EU. Quando dummy, che si trova sempre in uno stato ready, viene messo in esecuzione immediatamente dopo che ha ceduto il controllo, significa che non ci sono EU pronte per eseguire e quindi la CPU è idle.

La precisione di questo metodo dipende fortemente dall'algoritmo di scheduling; non sempre infatti si può avere la certezza che dummy venga messo in esecuzione ad ogni cambio di contesto, soprattutto se l'algoritmo è a priorità variabile.

2.5.2 La memoria

Rilevare l'occupazione di memoria può avere molteplici utilità, il controllo di come viene utilizzata è importante ai fini dell'*efficienza*, della *protezione* e della *sicurezza*:

- Quando la memoria in uso supera la memoria centrale, il sistema è costretto ad una gestione in swapping anche per le pagine di uso più frequente, provocando un drastico peggioramento delle prestazioni. Un monitoring sulla memoria permette quindi di evidenziare condizioni di bassa efficienza che altrimenti non potrebbero essere identificate.

- A causa di possibili errori nella programmazione, può accadere che una EU continui ad allocare memoria oltre le reali necessità. Questo tipo di problema, difficile da riconoscere, può essere facilmente identificato attraverso il monitoraggio della memoria.

- Se il nodo è pubblico, permette l'esecuzione anche ad EU che provengono da siti *untrusted*, è necessario controllare e vincolare uso massimo delle risorse.

Per misurare l'uso di memoria dal punto di vista globale (senza distinguere tra le diverse EU) sono in genere disponibili, anche a livello utente, System call da cui è possibile ottenere direttamente il dato; per conoscere l'occupazione di memoria di una singola EU, se non esiste una chiamata ad hoc, è necessario tracciare tutte le chiamate di gestione e raccogliere di volta in volta i dati.

2.5.3 La comunicazione

Le risorse di comunicazione dipendono dal tipo di servizi che vengono messi a disposizione dal sistema. Considerando lo scambio di messaggi, modello sopra il quale è possibile realizzare qualsiasi tipo di comunicazione, è possibile eseguire due tipi di misure:

- **Quantità:** è possibile misurare quanti messaggi vengono spediti/ricevuti da una determinata EU. Il numero dei messaggi è facilmente misurabile

attraverso l'uso di contatori nei servitori addetti allo scambio di messaggi.

- **Costo:** la misura di quanto costa (in termini di tempo) eseguire una comunicazione (locale o remota). Questo tipo di misura è più complesso, non può essere verificato direttamente ma attraverso simulazioni o estrapolazioni dai dati di traffico della rete.

Il numero di messaggi scambiati tra due EU permette di quantificare il vincolo tra due diverse EU, quindi di evidenziare quali EU hanno la necessità di convivere sullo stesso nodo. Questo tipo di misura è fortemente intrusiva, è necessario infatti intervenire in ogni comunicazione e registrare l'evento per tutti i partecipanti della interazione.

Una misura di costo invece, può essere utile per verificare se vale o meno la pena di effettuare una comunicazione oppure se conviene optare per altre soluzioni.

In un ambiente ad Agenti, la misura di tipo quantitativo non riveste una particolare importanza, l'elevato costo quindi ne sconsiglia l'uso. Il calcolo del costo invece è molto interessante come *percezione delle distanze* da parte dell'agente.

Il tempo necessario per scambiare un messaggio in locale è strettamente dipendente dall'architettura sia Hardware che Software, non dipende, se non debolmente, dalla grandezza del messaggio ed è quindi praticamente costante.

Se si considera una comunicazione remota, il suo costo dipende da diversi fattori:

- **Banda disponibile** (B)**:** ogni canale fisico ha una certa capacità, normalmente espressa in *bit/sec*; da qui è possibile calcolare il **Tempo di trasmissione** $T_t = M/B$ dove M è la grandezza del messaggio.

- **Latenza di trasmissione** (L_t)**:** il nodo mittente ha bisogno di un certo tempo per completare le procedure di trasmissione del messaggio sul canale.

- **Latenza di ricezione** (L_r)**:** dualmente, anche il nodo destinazione deve completare determinate procedure prima di rendere disponibile il messaggio al destinatario.

- **Routing:** in generale, un messaggio deve passare più host per arrivare a destinazione; supponiamo che ci siano n passaggi intermedi, ogni canale ha una banda diversa, il tempo di trasmissione allora assume la forma:

Eq 1: $$T_t = \sum_{i=1}^{n} \left(Lt_i + Lr_i + \frac{M}{B_i} \right)$$

In funzione di questi parametri, il **Costo di un messaggio** (T_m) è dato dalla seguente relazione:

Eq 2: $$T_m = L_t + L_r + T_t$$

La latenza, in generale dipende dalla grandezza del messaggio con una legge strettamente dipendente dal Software di gestione delle risorse di rete. SE ipotizziamo questa dipendenza di tipo lineare, possiamo affermare anche che la latenza sia costante per piccole variazioni della grandezza del messaggio.

Se consideriamo un passaggio senza intermediari, ipotizzando cioè la connessione diretta tra mittente e destinatario, il calcolo di T_m necessita di 3 parametri: L_t, L_r, B.

Il calcolo della latenza è possibile attraverso la spedizione di un messaggio campione sul nodo stesso, in questo modo la Eq.2 diventa:

Eq 3: $$T_m' = L_t + L_r$$

Se per ipotesi $L_t = L_r = L$ la Eq.3 diventa:

Eq 4: $$T_m' = 2 \cdot L \Rightarrow L = \frac{T_m'}{2}$$

La misura di B si può ottenere con un messaggio campione tra due nodi che già conoscono la propria latenza (L_1 e L_2):

Eq 5: $$T_m'' = 2 \cdot (L_1 + \frac{M}{B} + L_2) \Rightarrow B = \frac{M}{\frac{T_m''}{2} - L_1 - L_2}$$

Il fattore 2 è dovuto alla necessità di misurare il tempo sempre sul nodo mittente poiché l'eventuale costo di sincronizzazione degli orologi delle due macchine sarebbe molto elevato.

La misura di B può dare l'indicazione del traffico che passa sul canale, la misura della latenza invece della quantità di dati in ingresso/uscita dal nodo. Per quanto riguarda la Banda, l'ipotesi di connessione punto-punto è molto restrittiva: ci sono molte situazioni topologiche che permettono la condivisione dello stesso canale fisico tra più di due nodi. In una LAN ad esempio, tutte le macchine sono generalmente connesse sullo stesso canale e quindi condividono la stessa banda. In queste condizioni, la velocità di trasferimento tra due punti è influenzata dal traffico degli altri nodi. Bisogna anche tenere presente che solitamente le connessioni di tipo locale presentano bande molto ampie, tanto da considerare il tempo di trasmissione trascurabile rispetto alla latenza (ovviamente in condizioni di carico non troppo elevate e per messaggi non troppo grandi). Si tratta quindi di valutare di volta in volta quali sono i limiti per cui è lecita l'approssimazione di banda infinita e quando invece è necessario tenerne conto.

2.5.4 L'Agente

Nel modello ad Agenti, una delle necessità fondamentali è l'interazione; gli agenti devono essere quindi in grado di percepire qualitativamente l'esistenza degli altri. Si tratta di realizzare quindi una semplice sonda che verifichi l'esistenza di uno specifico agente e la sua eventuale disponibilità a comunicare. Si tratta essenzialmente di un monitor di livello applicativo che può essere logicamente integrato nel servizio di gestione delle informazioni generale senza bisogno di particolari sonde.

3 Il Linguaggio Java

Java è stato realizzato dalla Sun Microsystems [Java], con l'obbiettivo di ottenere un linguaggio di alto livello orientato alla programmazione di rete ed in grado di superare lo scoglio dell'eterogeneità. Per permetterne l'uso ad ogni piattaforma, la scelta è caduta sull'uso di una Macchina Virtuale **(Java Virtual Machine JVM)** le cui specifiche sono pubbliche affinché chiunque possa realizzarne una versione sulla propria architettura.

È un linguaggio ad Oggetti, ispirato a C++ e SmallTalk, multi-threaded e fornito di un set di librerie, detti *package*, in grado di offrire qualsiasi tipo di servizio per la programmazione, cosicché un programma scritto in puro Java sia sempre eseguibile su qualsiasi piattaforma che implementa una JVM senza nessun tipo di modifica.

3.1 Code Mobility e Java

Le specifiche del linguaggio, definiscono la granularità del binding del codice eseguibile a livello di **Classe**. Un programma Java è caratterizzato da un insieme di Classi che interagiscono tra loro solo dinamicamente quando vengono istanziate: gli **Oggetti**.

In uno scenario simile, la mobilità del codice è implicitamente fornita attraverso due strumenti:

- **Programmabilità del *Class Loader:*** è una Classe di Sistema che si occupa del caricamento del codice delle nuove classi che vengono di volta in volta istanziate; a default, carica da disco, cercando nelle aree specificate da una variabile di ambiente (**CLASSPATH**); è però possibile l'overloading dei suoi metodi per realizzare un caricamento da eventuali dispositivi remoti.

- **Astrazione *Object Stream* e possibilità di gestione delle chiamate a metodo *Runtime:*** *ObjectStream* è un filtro sovrapponibile ad un qualsiasi tipo di connessione, fornisce trasparentemente le fasi di *Serialize* e *Deserialize* necessarie al trasferimento di un Oggetto passivo. È possibile quindi prendere un oggetto sul nodo A, trasferirlo sul nodo B e lì invocare dinamicamente uno qualsiasi dei suoi metodi.

Con questi meccanismi, secondo la classificazione precedentemente data, la mobilità è di tipo **Weak** ed il modello di programmazione naturale è sicuramente **REV** (sia di tipo *PUSH* che *PULL*).

I *Threads* Java non possono essere trasferiti nello stesso modo, non sono implementati come *Oggetti Attivi*, contesti a cui possono essere associati uno o più thread strettamente dipendenti, ma sono in realtà degli Oggetti Passivi allocati su un processo leggero della JVM. Quando si trasferisce un Oggetto Thread viene trasferita la parte passiva, mentre lo stato attivo rimane nascosto nella macchina virtuale.

3.1.1 Realizzazione del Modello ad Agenti Mobili

Con il meccanismo debole di Java, è possibile simulare il comportamento di un Agente; quando si trasferisce un Thread da un contesto ad un altro si perde il suo stato di esecuzione, ma si mantengono del sue variabile interne. Perdendo le informazioni degli Stack ed i Program Counter, il Thread sarà costretto a ripartire dal principio o per lo meno dall'inizio di un suo metodo.

Simulare Strong Mobility, in questo caso, significa imporre un comportamento ripetitivo all'Agente ogni qualvolta cambia contesto, avendo come unico ricordo della *vita passata* le sue variabili di stato. L'implementazione degli algoritmi degli Agenti dovranno tenere conto di questo limite, utilizzando eventualmente *salti condizionati*, per eseguire diverse parti di codice (figura 12).

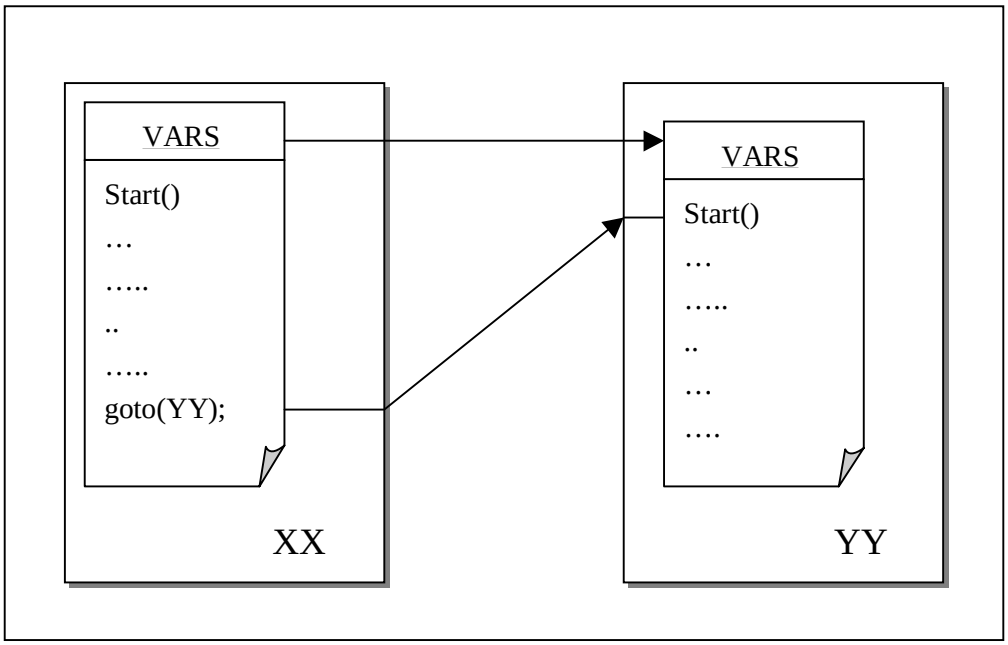

Figura 12: Quando un thread viene trasferito in un nuovo nodo, mantiene solamente lo stato delle sue variabili interne ed è costretto a ripartire dal metodo dall'inizio.

Si può rendere il meccanismo un poco più elastico rispetto all'implementazione di figura 12 caratteristica degli ambienti Mole e Aglets. La possibilità di chiamare dinamicamente i metodi di un Oggetto permette che l'Agente stesso decida, all'atto della richiesta di muoversi, quale dei suoi metodi debba essere invocato sul nodo destinazione (Figura 13).

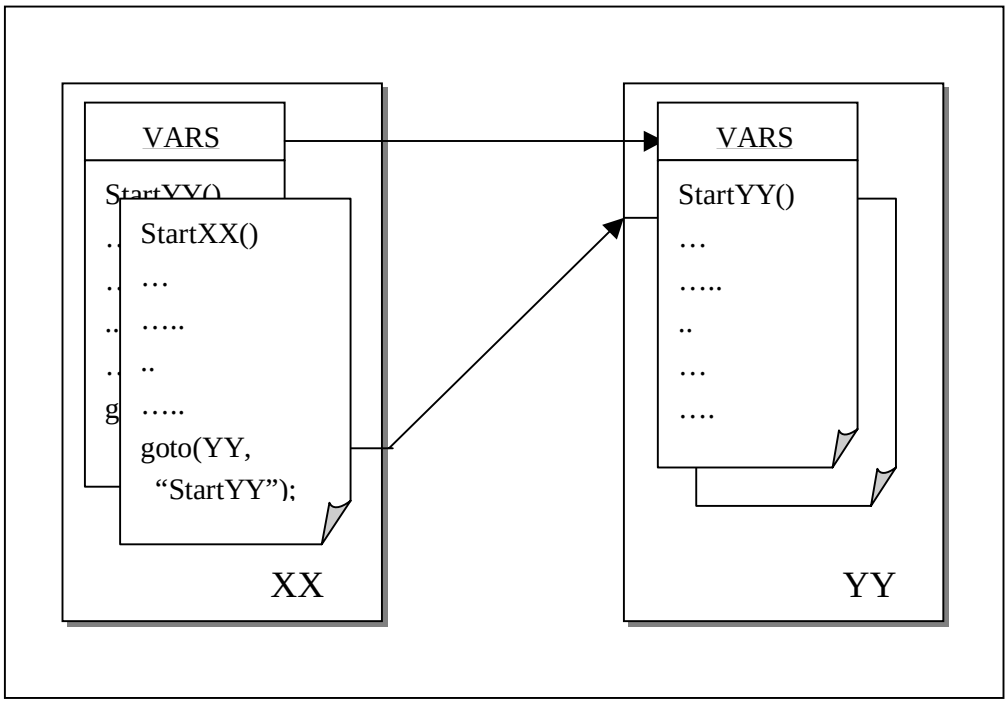

Figura 13: La possibilità di scegliere dinamicamente il metodo da eseguire sulla macchina destinazione rende più agevole la programmazione degli Agenti.

La chiamata potrebbe essere del tipo:

goto(Place, Method, < Args[ ] >);

neanche in questo caso si ha realmente Strong Mobility, ci si svincola però dalla necessità di gestione esplicita del codice da eseguire in funzione dello stato. Come si vede nell'esempio, gli argomenti della chiamata **goto()** obbligatori sono la destinazione *(Place)* ed il metodo da eseguire *(Method)*,

gli argomenti per la invocazione del metodo *(Args[ ])* sono invece facoltativi. La semantica di accesso agli argomenti potrebbe essere sia per riferimento che per copia; un passaggio per riferimento però implica una gestione trasparente della locazione degli oggetti (attraverso RMI) contraria al modello ad Agenti. Il passaggio degli argomenti dovrà quindi essere sicuramente per copia.

È possibile realizzare il meccanismo in entrambi i modi: permettendo il passaggio di parametri si facilità la programmazione, negandolo si semplifica il meccanismo di chiamata. La scelta su l'una o l'altra soluzione dipende strettamente dal tipo di sistema che si intende realizzare, se si desidera un supporto sperimentale per sviluppare la tecnologia ad Agenti oppure si elabora un prodotto commerciale.

Un'altra tecnica per simulare la Strong Mobility consiste nello sfruttare un meccanismo ad eventi: in Aglets ad esempio, un modo per decidere quale parte di codice eseguire una volta arrivati a destinazione consiste nell'associare al movimento un messaggio, questo, all'arrivo dell'agente nel sito di destinazione, provoca un evento che può essere raccolto mediante overloading del metodo **handleMessage()** e quindi decidere cosa fare in base al messaggio ottenuto.

3.2 Tecniche per ottenere Strong Mobility

Se si desidera ottenere un meccanismo di mobilità Strong reale, è necessario cercare di ottenere lo stato interno della JVM; ci sono due diverse strade per ottenerlo:

- **Checkpointing**
- **Modifica della Macchina Virtuale**

La prima tecnica è esattamente quella già esposta nel primo capitolo, si tratta di monitorare completamente il codice per collezionare i dati necessari alla migrazione del Thread. Quando l'Agente si muove, si trasferisce lo stato raccolto e lo si ripristina. Per rigenerare lo stack è necessario ridichiarare e settare le variabili locali, per ottenere lo stesso program counter è necessario un salto al checkpoint. Questa tecnica è fortemente intrusiva: è necessario realizzare un post-compilatore in grado di inserire all'interno del bytecode delle Classi interessate tutte quelle chiamate necessarie alla fase di Store e Restore.

La seconda soluzione permette di ottenere i massimi risultati; a livello di macchina virtuale, infatti, sono già presenti tutti i dati necessari alla migrazione. Effettuare delle modifiche al di fuori delle specifiche standard significa però assumersi completamente la responsabilità dello sviluppo futuro del sistema, portabilità compresa. Siccome le specifiche standard non tengono conto delle modifiche apportate, eventuali nuove release di Java potranno essere completamente incompatibili con le soluzioni adottate.

3.2.1 Consistenza e gestione dei riferimenti

La consistenza dei riferimenti remoti all'atto della migrazione deve essere esplicitamente gestita in ogni caso. I riferimenti agli *Oggetti membro* (gli Oggetti pubblici o privati che caratterizzano lo stato di un Agente), gli unici che possono essere considerati locali in senso stretto, sono automaticamente gestiti nella migrazione dell'Oggetto passivo; bisogna però tenere presente che all'atto della migrazione avviene un passaggio *per copia* e quindi la semantica per riferimento di Java viene violata. Questo tipo di

anomalia non crea nessun problema agli Oggetti privati, questi infatti non sono accessibili dall'esterno; anche se l'Oggetto viene copiato la consistenza non viene a mancare. Per quanto riguarda gli Oggetti Condivisi, cioè quegli Oggetti pubblici a cui possono accedere più thread in concorrenza, il passaggio per copia porta ad una replicazione non gestita ed quindi ad una conseguente inconsistenza. In questo caso è quindi necessaria una gestione esplicita dei riferimenti, la creazione di un meccanismo ad hoc a cui è naturalmente possibile associare eventuali politiche di cashing e replicazione. Il package *Java.rmi* permette già una gestione di questo tipo, ha però il problema di essere fortemente incentrato sul paradigma Client-Server come evoluzione in ambiente Java delle RPC di Sun. La forma d'uso di questo package risulta troppo statica per integrarla in un meccanismo di questo tipo, è necessario infatti generare delle **stub** per ogni cliente e servitore e gli Oggetti remoti devono essere progettati ad hoc.

Esistono però gli strumenti sufficienti per superare questo tipo di impostazione e realizzare una chiamata a metodo remoto completamente dinamica e generale. La possibilità di utilizzare tecniche di tipo REV e di invocare dinamicamente un metodo permettono di creare un Oggetto *RemoteInvocation* che semplicemente porta la chiamata con i suoi argomenti verso l'Oggetto di interesse e poi riporta indietro la risposta (figura 14). Per completare questo meccanismo, è necessaria una gestione globale dei nomi degli Oggetti e la realizzazione di un supporto che mantenga *vivi* (o gli attivi se non lo sono) gli oggetti remoti (*Object Manager*); entrambi comunque devono essere in ogni caso realizzati in un supporto ad Agenti.

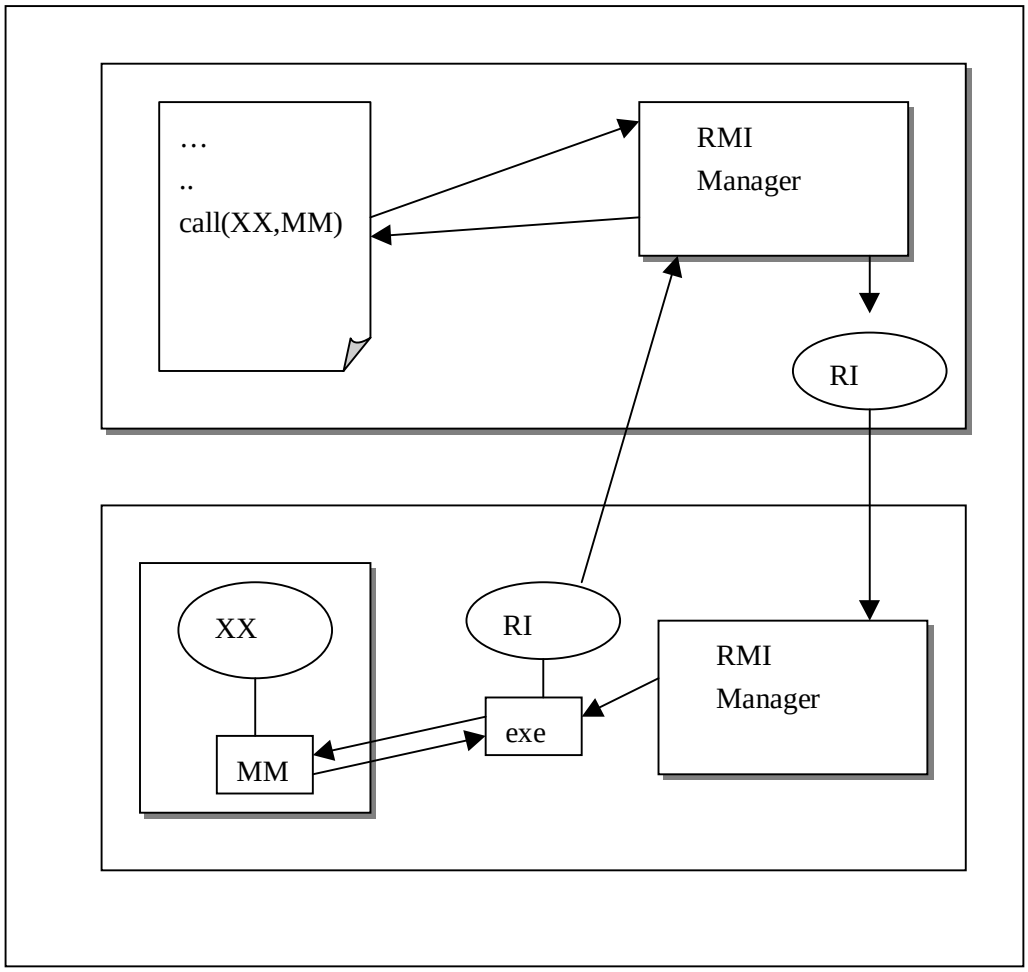

Figura 14: La chiamata a metodo remoto può essere realizzata sfruttando la mobilità del codice. All'atto della chiamata, il supporto (*RMI Manager*) crea un Oggetto *RemoteInvocation* (*RI*) con incapsulati gli argomenti per la chiamata; questo oggetto viene spedito dove risiede l'oggetto remoto, lì viene eseguito un suo metodo fissato (*exe*) che esegue realmente l'invocazione al metodo. Terminata l'esecuzione l'Oggetto RI viene rispedito indietro contenendo la risposta.

3.3 La Comunicazione

Dal punto di vista della comunicazione Java non pone nessun tipo di limite; nella sua natura, il paradigma di interazione naturale è *Shared Memory* attraverso l'uso di *Oggetti Sincronizzati*. Il *Message Passing* è comunque implementabile, quindi ci sono le basi sufficienti per realizzare qualsiasi tipo di modello di astrazione di comunicazione.

La verifica sperimentale sul costo di una invocazione a metodo locale e remota (Tab. VII), mette in evidenza come ci siano circa 3 ordini di grandezza di differenza già in una rete locale. Questo implica la necessità di una attenta scelta di quali meccanismi di comunicazione mettere a disposizione, sia in remoto che in locale e soprattutto evidenzia come non sia logico, in un modello ad Agenti, fornire meccanismi di *Interazione Stretta* trasparenti alla rete. Per quanto riguarda le comunicazioni di tipo *Lasco*, deve essere messo a disposizione almeno un metodo di comunicazione trasparente alla locazione, per evitare antieconomici movimenti degli Agenti.

	Locale	Remota	Rapporto R/L
Costo chiamata	4 μs	12 ms	3000

Tabella VII: Il costo di una chiamata a metodo differisce di 3 ordini di grandezza tra un caso locale ed uno remoto, la chiamata remota è stata effettuata con la tecnica descritta in figura 13.

3.4 I Sistemi ad Agenti in Java

Come già esposto nei precedenti capitoli, esistono vari sistemi che utilizzano Java sia per la progettazione del supporto che per l'implementazione degli Agenti. Tra i sistemi precedentemente analizzati Aglets e Mole sono scritti in Java puro, simulando la mobilità Strong al di sopra dei meccanismi Weak di Java, mentre in Sumatra è stata modificata la macchina virtuale per la realizzazione dei meccanismi Strong e la gestione degli Eventi [Ach97].

3.4.1 Sumatra e la modifica della macchina virtuale

Come si è visto, una delle tecniche per ottenere Strong Mobility in Java consiste nella modifica della macchina virtuale per poter estrarre e quindi trasferire lo stato interno di un thread. In Sumatra sono stati affrontate tutte le problematiche di gestione dei riferimenti attraverso il concetto di **proxy reference**: l'interprete mantiene una copia separata degli stack parallelamente a quelli di esecuzione; in questa copia, mantiene tutti i tipi ed i valori delle variabili per le fasi di serializzazione e deserializzazione. All'atto della migrazione, vengono trasportati sul nodo destinazione tutti gli oggetti privati, mentre i riferimenti agli oggetti che rimangono indietro vengono trasformati in proxy reference: riferimenti ad oggetti di sistema con il compito di gestire **trasparentemente** la chiamata a metodo remoto.

Una gestione di questo tipo è molto efficiente, il costo di gestione dei riferimenti e dei cambi di contesto è limitato al mantenimento del doppio stack. Il problema principale di questo tipo di approccio consiste nell'oggettiva impossibilità di usufruire di piattaforme Java standard; la modifica della JVM ha reso il sistema proprietario si è quindi rinunciato ad uno dei maggiori vantaggi nell'uso di Java. La peculiarità della modifica, in particolare per quanto riguarda la gestione degli stack rende il sistema potenzialmente incompatibile con eventuali nuove specifiche standard nelle future versioni della JVM.

3.4.2 La programmazione ad Agenti in Java Standard: Aglets

Per evidenziare più in dettaglio le caratteristiche d'uso del modello ad Agenti in Java puro, senza cioè modifiche di sistema ma solo con la semplice mobilità Weak, sono stati analizzati due esempi di semplici applicazioni scritte per il supporto di IBM.

Il primo esempio è il semplicissimo programma *Hello World:* un agente si muove verso un sito fornito dall'utente dove apre una finestra in cui saluta e poi torna al mittente per comunicare il successo. In questo esempio è molto evidente lo stile di programmazione ad eventi: non è infatti presente né il costruttore né il metodo ***run()*** ma ci sono diversi metodi che vengono invocati dal supporto in diverse occasioni.

```
public class HelloAglet extends Aglet {
transient Frame my_dialog = null;
String message = "Hello World";
String home = null;
SimpleItinerary itinerary = null;

public void onCreation(Object init) {
        itinerary = new SimpleItinerary(this);
        home = getAgletContext().getHostingURL().toString();
        }
public void go(Message msg) {
        URL dest = (URL) msg.getArg();
        try {
                itinerary.go(dest.toString(),"sayHello");
        } catch (Exception ex) {
                ex.printStackTrace();
        }
}
public syncronized void goDestination(String destination) {
        < serve il comando di dispatch proveniente dall'esterno (sistema
oppure altri agenti) esegue esattamente gli stessi passi di go() >

}
```

```
public boolean handleMessage(Message msg) {
        if(msg.sameKind(“atHome”)) {
                atHome(msg);
        } else if(msg.sameKind(“sayHello”)) {
                sayHello(msg);
        } else if(msg.sameKind(“dialog”)) {
                dialog(msg);
        } else
                return false;
        return true;
}
public void atHome(Message msg) {
        setText(“I’m back.”);
        waitMessage(2*1000);
        dispose();
}
public void sayHello(Message msg) {
        setText(message);
        waitMessage(5*1000);
        try {
                setText(“I’ll go back to.. “ + home);
                waitMessage(1000);
                itinerary.go(home,”atHome”);
        } catch (Exception ex) {
                ex.printStackTrace();
        }
}
public void dialog(Message msg) {
< serve per controllare l’evento di richiesta di apertura di una finestra di
dialogo da parte dell’utente >
}
```

Questo esempio risulta scomposto in diversi metodi senza nessuna apparente correlazione; il comportamento dell'agente infatti è determinato solo ed esclusivamente da una serie di eventi che ne caratterizzano l'esistenza:

- ***onCreation(Object init):*** questo metodo viene invocato dal costruttore della superclasse **Aglet**; sostituisce di fatto il costruttore di classe.

- ***go(Message msg):*** viene invocato dall'oggetto *my_dialog* che gestisce la finestra di interazione con l'utente; questo metodo invoca la funzione di migrazione ***itinerary.go(dest, msg)*** dove *dest* è la stringa che contiene l'URL di destinazione mentre *msg* contiene un messaggio che deve essere recapitato all'arrivo all'agente stesso. Questo messaggio serve per la discriminazione di cosa eseguire una volta arrivati a destinazione poiché, come già esposto, l'agente altrimenti dovrebbe ripartire sempre da zero.

- ***handleMessage(Message msg):*** questo metodo viene invocato dal sistema ogni qualvolta arriva un messaggio, in particolare quindi quando l'agente ha completato una migrazione. Attraverso il contenuto del messaggio riconosce dove si trova e quindi cosa fare.

Lo stile di programmazione ad eventi non è in questo caso molto immediato, rispecchia un modo di ragionare tipico della programmazione di GUI ma crea grossi problemi di leggibilità soprattutto se si possiede poca familiarità con le specifiche delle API. La gestione delle risorse è esattamente quella di Java, il *Frame my_dialog* è realizzato con le normali chiamate Java, è dichiarato *transient* perché non deve fare parte dell'oggetto all'atto della migrazione poiché ciò comporterebbe un eccezione visto che un Frame non è serializzabile.

L'agente non è in grado di conoscere dove può muoversi e se ne ha il permesso: tutto è a carico dell'utente che deve fornire la destinazione ed assicurarsi che sia un movimento permesso.

Il secondo esempio tratta l'interazione tra gli agenti: un agente padre (*Stationary*) genera due agenti figli (*Traveller*) che manda, al comando dell'utente, su diverse destinazioni; i figli tornano spontaneamente e cercano una interazione con il padre.

```
public class Stationary extends Aglet implements MobilityListener,
                                                  PersistencyListener {
private Meeting _meeting = null;

public void  onCreation(Object o) {
        setText("Don't try to move me!!");
        addMobilityListener(this);
}

pubilc void onActivation(PersistencyEvent event) {
        private Hashtable aglets =new Hashtable();
}

public void run() {
        Agletproxy p = null;
        try {
                aglets.put(getAgletID(),"Stationary C");
                Vector c = new Vector();
                c.addElement(getAgletID());
                _meeting = new Meeting(getAgletContext().getHostingURL(),c);
                _meeting.setAttachedInfo(aglets);

                syncronized (_mmeting) {
                        p= getAgletContext().createAglet(null,"Traveller",_meeting);
                        aglet.put(p.getAgletID(),"Traveller A");
                        p= getAgletContext().createAglet(null,"Traveller",_meeting);
                        aglet.put(p.getAgletID(),"Traveller B");
                }
        } catch (Exception ex) {
                e.printStackTrace();
        }
}
```

```
public class Traveller extends Aglet implements MobilityListener,
                                                 PersistencyListener {
private static final int AT_HOME=0;
private static final int OUT=1;
private static final int BACK_HOME=2;

private URL _origin = null;
private Meeting _meeting = null;
private int stat = AT_HOME;
private Hashtable aglets = null;

private int count = 0;

public void onCreation(Object o) {
        _meeting =(Meeting) o;
        aglets = (Hashtable) _meeting.getAttachedInfo();
        setText("Dispatch me somewhere.. I will return");
        addMobilityListener(this);
}

public void onDispatching(MobilityEvent ev) {
        _state = ((_state==OUT) ? BACk_HOME : OUT);
}

public void run() {
        try {
                switch(_state) {
                case OUT:
                        dispatch(_meeting.getPLace());
                        break;
                case BACK_HOME:
                        setText("I'm back");
                        Enumeration e = _meeting.ready(this);
                        Vector v = toVector(e);
                        setText(" I met with "+ v);
```

```
                    break;
                }
        } catch (Exception e) {
                e.printStackTrace();
        }
        return false;
}

public boolean handleMessage(Message msg) {
        try {
                if(_meeting.getID().equals(msg.kind)) {
                        setText(" I met with "+(String) (aglets.get( (AgletID)
                                                                (mgs.getArg()))));
                        msg.sendReply(getIdentifier());
                        return true;
                }
        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

L'agente stazionario si iscrive come *Stationary C* in una tabella, apre un meeting registrandosi ed associando la tabella generata. Successivamente crea due agenti, iscrivendoli nella tabella come *Traveller A e Traveller B* e poi si ferma in attesa degli eventi.

L'agente viaggiatore riceve come argomento all'atto della creazione l'oggetto Meeting, copia la tabella contenuta e poi si mette a disposizione per essere trasferito. Nel metodo **run()** è evidente la gestione esplicita della scelta di cosa fare dopo la migrazione: quando arriva in un nodo remoto (OUT) ritorna subito nella sede del meeting, una volta tornato lo notifica al meeting e stampa quali altri agenti sono pronti.

In questo esempio è evidente la complessità di coordinamento anche tra agenti che sono legati da un rapporto di parentela. Questo è dovuto al tentativo di non mantenere nessun tipo di stato nel sistema sugli agenti generati per semplificare la gestione del supporto. In questo modo però, gli

agenti sono completamente ciechi a tutto quello che non è stato precedentemente pensato in sede di progettazione dell'applicazione: le applicazioni possono essere considerate in un certo senso statiche, incapaci di adattarsi dinamicamente all'evolversi dell'ambiente in cui opera. Questo limite, in un sistema aperto, è molto forte, come evidenziato in Agent Tcl ed in Sumatra; la capacità di interagire con l'ambiente fornisce al modello ad agenti potenzialità decisamente superiori.
Questo tipo di funzionalità non sono per nulla legate al concetto di Strong Mobility, sono servizi di supporto che possono essere realizzati anche in assenza di mobilità e possono quindi essere forniti anche in Java puro.

3.4.3 Realizzazione di un Servizio di Naming in Aglets

Aglets fornisce tutti i meccanismi di basso livello necessari per la realizzazione di un supporto ad Agenti, ma allo stato attuale è difficile considerare Aglet stesso un ambiente ad Agenti, in quanto sembra piuttosto un insieme di tool di libreria atti alla creazione, a livello applicativo, di tutti i servizi necessari. Per questo motivo si è pensato alla realizzazione di un Servizio di Naming che possa permettere agli agenti di muoversi conoscendo solamente il nome del place di destinazione invece dell'URL completo come di default.

Questo servizio, per semplicità, è stato pensato completamente replicato su tutti i nodi gestiti: in ogni sito è presente un agente stazionario che ha il compito di mantenere la copia locale del DataBase. Un agente cliente, può richiedere due tipi di servizio:

- *Registrazione:* può chiedere la registrazione di un nuovo place; indipendentemente da dove si effettua la registrazione, tutte le copie replicate devono essere rese consistenti, bisogna quindi realizzare un meccanismo di coordinamento tra servitori.

- *Query:* il servizio principale è ovviamente la risoluzione dei nomi; per semplicità il servitore restituisce l'URL del place indicato se esiste nel suo DataBase, *null* in caso contrario. In questo caso infatti si ipotizza che tutti i servitori presenti siano sempre consistenti.

Non abbiamo tenuto conto delle problematiche legate alla sicurezza nell'uso del servizio; sebbene siano fondamentali soprattutto per quanto riguarda la registrazione, possono essere trattate a posteriori aggiungendo un livello superiore.

Affinché i clienti possano interagire con il servitore, deve essere disponibile una qualche forma di comunicazione anonima, Aglet fornisce l'astrazione di Whiteboard in maniera indiretta: in ogni sito, esistono uno o più contesti (concettualmente simili alla definizione generale di place). All'interno di questi contesti, è possibile *registrare delle proprietà* associate ad una *keyword*.

L'agente servitore registra il suo identificatore unico (*AgletID*) associandolo alla keyword *NameServer*:

```
AgletContext Place = getAgletContext();
Place.setProperty("NameServer",this.getAgletID());
```

Dall'altra parte, il cliente ha necessità di ottenere il *proxy* necessario per la comunicazione: richiede quindi prima la proprietà associata alla keyword NameServer al sistema e poi ottiene *AgletProxy* dall'identificatore del servitore:

```
AgletID NSId = (AgletID) Place.getProperty("NameServer");
AgletProxy NameServer = Place.getAgletProxy(NSId);
```

La comunicazione tra cliente e servitore avviene quindi per scambio di messaggi il cui tipo (in Aglet ai messaggi viene associata una keyword che ne identifica il tipo) determina quale servizio si richiede.

Il servitore così realizzato è stazionario, ha una sua Base di Dati locale e non ha nessuna visione globale. È possibile quindi creare un agente servitore su ogni nodo e poi realizzare dei clienti speciali che vengono creati dinamicamente dal servitore dopo ogni nuova registrazione ed hanno il compito di muoversi verso gli altri servitori e notificare la nuova registrazione mantenendo quindi consistenti tra loro le basi di dati locali.

4 Progetto di un Ambiente ad Agenti

Nei capitoli precedenti sono state evidenziate le problematiche più significative nello sviluppo di un ambiente per la programmazione secondo il modello ad Agenti. In particolare si è considerato il supporto per la programmazione ad Agenti, valutandone i problemi principali e le possibili soluzioni. In questo capitolo verrà descritta la realizzazione e le caratteristiche principali di un sistema ad Agenti sviluppato interamente in Java, progettato rispettando le specifiche evidenziate della sezione 1.7 e sviluppato con le tecniche descritte nel capitolo 3.

4.1 Organizzazione logica

La struttura topologica della rete non deve essere nascosta alle applicazioni, deve però essere rappresentata in modo uniforme, fornendo una rappresentazione che trascenda i particolari e ne permetta un uso semplice e standardizzato.

4.1.1 I Place ed i Domini

La topologia delle reti di grandi dimensioni è spesso gerarchica, come lo è anche l'organizzazione di Internet; ci sono diversi domini all'interno dei quali vengono distinti sotto-domini fino a giungere al semplice nodo.

Per meglio rappresentare il *mondo* agli Agenti è importante che la topologia da loro vista rispecchi, con una certa approssimazione, questo tipo di struttura. Come è già stato evidenziato nei capitoli precedenti, è interessante astrarre il concetto di nodo attraverso contesti di esecuzione che non siano strettamente legati alla macchina fisica; in ogni nodo, possono quindi esistere uno o più **Place**, contesti in cui gli agenti eseguono ed interagiscono con i servizi messi a disposizione. Più nodi, contenenti i loro place, sono raggruppati in un **Dominio**, che rappresenta una località nella rete e può rispecchiare un località reale (come una LAN) oppure solo una caratteristica logica (all'interno della stessa sub-net possono essere distinti diversi domini per necessità puramente logiche)(Figura 15). Un Agente ha consapevolezza del Dominio in cui si trova, può conoscere tutti i nodi (ed i place) che compongono il dominio e cercare in maniera semplice qualsiasi servizio all'interno di esso. Nel caso in cui invece si desiderasse qualcosa di appartenente a un altro Dominio, l'Agente deve esplicitamente spostarsi nel nuovo contesto oppure demandare il compito ad un agente di servizio specificando in quale dominio muoversi.

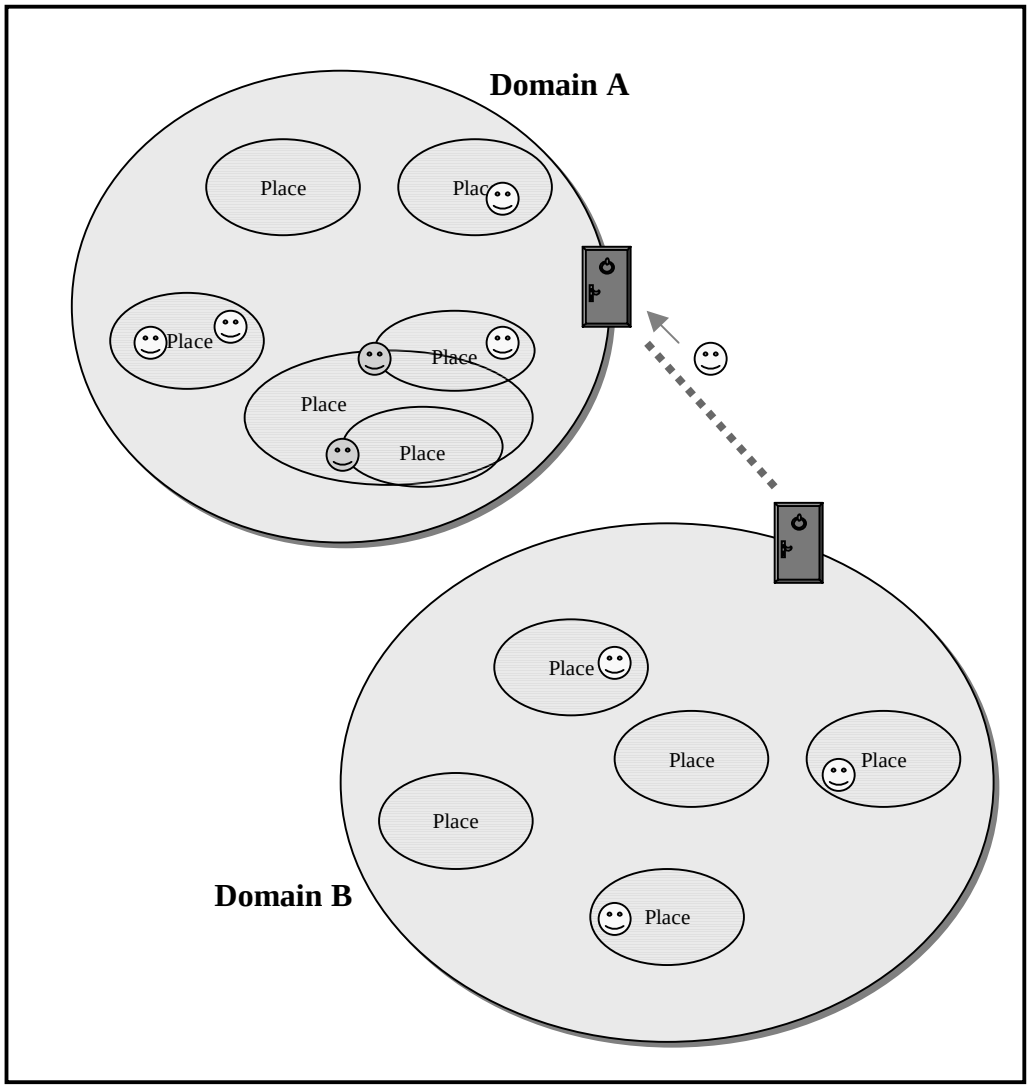

Figura 15: La rete è *vista* dagli Agenti come un insieme Place raggruppati in Domini.

4.1.2 Il Contesto di Esecuzione degli Agenti

Il nodo fisico viene rappresentato agli agenti come un insieme di place; esiste un contesto iniziale (*Default Place*) il cui nome caratterizza il nodo

stesso. All'interno di questo place sono disponibili una serie di servizi standard che forniscono le funzionalità necessarie (Figura 16).

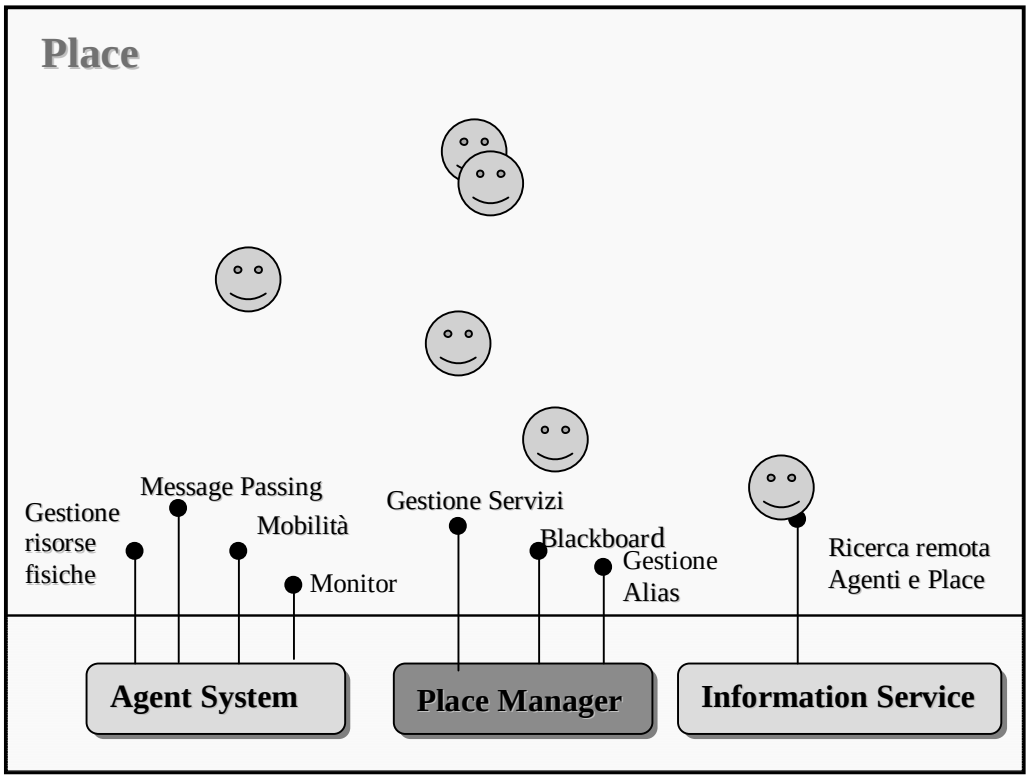

Figura 16: Il contesto in cui eseguono gli agenti. Esiste la possibilità di creare un proprio contesto privato che condivide il sistema ed il motore di ricerca mentre ha un proprio *place manager* distinto.

Ci sono tre moduli che interagiscono con gli Agenti:

- **L'Agent System,** che fornisce tutte le funzionalità di base: la *mobilità*, lo *scambio di messaggi* la *gestione delle risorse fisiche* ed il *monitoraggio.*
- **Il Place Manager,** che fornisce l'uso di una *Blackboard***,** gestisce un *servizio di naming* locale per gli Agenti Servitori e da la possibilità ad un

generico agente di *registrarsi con un alias* presso il Place; quest'ultima funzionalità è molto utile perché permette una interazione anonima come i *badge* di Mole (par 1.5.3) in modo molto semplice.

- **L'Information Service,** che permette la *ricerca remota di informazioni*, quali la posizione di un Agente, la risoluzione di un Alias in un Place remoto oppure la posizione di un determinato Place nel Dominio. La ricerca, viene limitata sempre a un Dominio, a default viene effettuata nel Dominio di residenza ma è possibile effettuare una ricerca altrove specificandone il nome.

Come è stato già accennato, un Agente può creare un suo contesto di esecuzione locale al proprio Place di appartenenza; questo nuovo Place eredita sia il modulo Agent System che il l'Information Service, crea invece un nuovo Place Manager permettendo quindi di creare sotto-contesti ristretti sia come servizi che come Blackboard. Un agente *vede* sempre il Place Manager del contesto in cui si trova e quando arriva in un nodo entra sempre nel Place di Default, ma ha la possibilità di entrare successivamente nei Place locali di suo interesse.

4.1.3 La Comunicazione

La comunicazione tra Agenti può avvenire mediante tre distinti meccanismi corrispondenti alle esigenze di interazione lasca, stretta ed anonima:

- **Scambio di messaggi**: ogni Agente possiede una Mailbox, ha la possibilità, in qualsiasi momento, di spedire un semplice messaggio ad un qualsiasi altro Agente di cui conosce l'identificatore; la spedizione del messaggio è *asincrona bloccante,* il Sistema considera completata la spedizione una volta acquisito il messaggio senza aspettare che sia correttamente recapitato. La richiesta di ricezione del messaggio è anch'essa *bloccante,* in quanto l'Agente che chiede il messaggio rimane in attesa finché non viene recapitato. Esiste comunque la possibilità di verificare se la Mailbox è piena prima di bloccarsi nell'attesa di un messaggio che non è ancora arrivato. Come meccanismo di interazione lasca, lo scambio di messaggi è l'unica funzionalità fornita in maniera

trasparente alla locazione: due Agenti possono comunicare con questo meccanismo anche se sono situati in due diversi Domini.

- **Blackboard:** permette l'interazione anonima ed asincrona; un Agente può lasciare in un place un messaggio sulla Blackboard, caratterizzato da una Stringa che lo identifica ed un Oggetto. Chiunque può leggere il messaggio (se conosce la Stringa di identificazione) ed eventualmente eliminarlo dallo spazio comune.

- **Condivisione Oggetti:** è il meccanismo di comunicazione naturale in Java e permette una interazione molto stretta tra un numero indeterminato di partecipanti. Il Sistema gestisce un contenitore in cui possono essere depositati gli Oggetti destinati alla condivisione e fornisce un identificatore unico attraverso cui è possibile accedervi. I partecipanti devono solo scambiarsi l'identificatore (mediante un messaggio nella Blackboard ad esempio) e possono quindi ottenere un riferimento all'oggetto. Il servizio è strettamente locale, non è prevista RMI per accedere ad oggetti remoti nel rispetto dell'ipotesi di località per tutti i meccanismi di interazione stretta.

4.1.4 La gestione delle risorse fisiche

Il Sistema fornisce la possibilità di accedere a tutte le risorse messe a disposizione della macchina virtuale Java senza per questo motivo rinunciare alla mobilità. Come è stato più volte evidenziato, un agente perde la possibilità di essere *serializzato* (e quindi di muoversi) ogni qualvolta accede a risorse il cui lo stato sia dipendente dal sistema locale. Esistono infatti moltissime risorse il cui lo stato è strettamente dipendente dal nodo fisico di esecuzione, l'accesso ad un file oppure all'interfaccia grafica non possono essere trasferiti all'atto della migrazione. La proprietà di *serializzabilità* è quindi in generale transitoria ed è necessario acquisirla per poter migrare in un nuovo sito. Per questo motivo esiste la possibilità di depositare gli oggetti legati al nodo in un contenitore gestito dal sistema ed ottenere in cambio un identificatore, il quale ovviamente non ha nessun legame con l'oggetto e

quindi permette all'Agente di muoversi mantenendo comunque una traccia della risorsa in uso.

Un Agente che intende usare una risorsa, può utilizzare quindi tutte le API standard fornite da Java, se poi ha bisogno di muoversi senza perdere lo stato della risorsa, deve registrarla presso l'Agent System e poi muoversi mantenendo l'identificatore della stessa per un eventuale uso futuro. Gli identificatori sono unici nel nodo, possono essere scambiati e condivisi e rappresentano l'unica via per ottenere l'accesso agli oggetti depositati.

4.1.5 La Mobilità

La possibilità di muoversi da parte di un Agente è realizzata attraverso la chiamata **go()** (in Appendice A sono descritte tutte le chiamate a disposizione degli agenti), la destinazione può essere specificata in diversi modi, a seconda delle necessità dell'Agente. La chiamata accetta come argomento destinazione i seguenti Tipi:

- **NodeName:** il nome fisico del nodo destinazione; se il nodo non appartiene al Dominio è necessario specificare anche il Dominio di appartenenza.

- **PlaceName:** il nome del Place di Default di un nodo all'interno del Dominio; se si desidera entrare in un Place locale al nodo è possibile entrarci una volta arrivati a destinazione.

- **DomainName:** è possibile muoversi verso un nuovo Dominio senza conoscere nulla di esso; il Sistema porterà l'Agente in un nodo di default nel Dominio richiesto.

Un secondo argomento obbligatorio è *StartMethod,* è necessario infatti specificare il metodo che verrà eseguito una volta arrivati a destinazione; non è possibile passare dei parametri ma le variabili locali all'Agente vengono mantenute nel trasferimento quindi non è una funzionalità indispensabile (Sez. 3.1).

Quando la migrazione non è possibile, per sopravvenuti guasti oppure perché alcuni nodi non sono permanentemente connessi, viene lanciata una eccezione: **CantGoException.** L'Agente ha così modo di comportarsi di

conseguenza all'interno del flusso di esecuzione che ha invocato la **go;** ha la possibilità di realizzare eventuali politiche di tolleranza ai guasti oppure gestire sistemi con connessioni temporanee.

Non esiste la possibilità di muoversi *inseguendo* un altro Agente, è una funzionalità che può però essere fornita come servizio realizzato completamente a livello applicativo.

4.2 Architettura del Supporto nel Nodo

Ogni nodo è caratterizzato da un supporto che ha il compito di realizzare tutti i servizi necessari alla gestione dei Place (Figura 17).

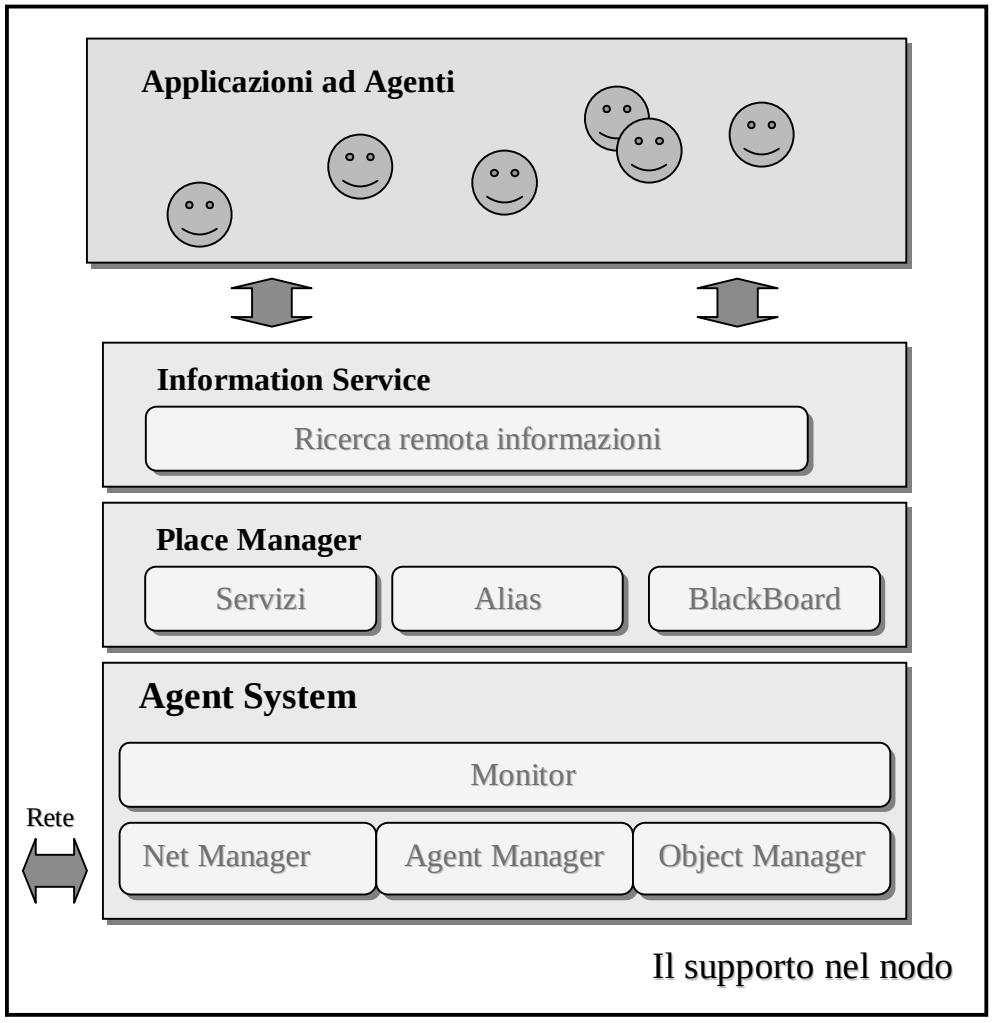

Figura 17: La struttura del supporto di gestione e controllo del nodo.

In Figura 17, viene evidenziata la struttura del gestore di nodo; si possono distinguere i diversi moduli che compongono il sistema a partire dal *core* del sistema detto **AgentSystem** fino all'**Information Service**, che fornisce la possibilità di acquisire informazioni remote ed è esso stesso realizzato ad Agenti.

4.2.1 Realizzazione dell'astrazione di Agente

Dal punto di vista implementativo, un qualsiasi Agente è realizzato come Classe derivata dalla Super Classe astratta ***Agent:*** contrariamente a quanto ci si potrebbe aspettare, questa classe non è derivata dalla Classe *Java.lang.Thread,* ma è un semplice oggetto passivo *serializzabile*. Quando un Agente nasce, viene affidato ad un thread (***Worker***) (Figura 18) che ha come unico compito quello di associare un flusso di esecuzione all'agente; all'atto della sua creazione, il Worker riceve come parametro un Agente, dopodiché, quando viene messo in esecuzione, invoca il metodo specificato. Quando il metodo esce, indipendentemente dal motivo del ritorno, il Worker notifica la sua terminazione al sistema e chiude.

Quando un Agente viene creato, gli viene assegnato un identificatore globale e viene creata una *Mailbox* associata alla sua variabile privata **Mail.** Attraverso la Mailbox l'agente è in grado di ricevere messaggi e di spedirli a chiunque conosca mediante l'identificatore (App. A). Esiste la possibilità di disabilitare la gestione dello scambio di messaggi rendendo così l'agente più agile e quindi più efficiente se ha forti necessità di movimento. Disabilitando la Mailbox però viene interrotto anche il meccanismo di verifica della posizione dell'agente (che verrà analizzato in seguito); questo rende di fatto l'agente non rintracciabile, con ovvi problemi di gestione della sicurezza. Per questo motivo questa possibilità dovrà essere in futuro permessa solo ad agenti fidati per forti necessità di sistema.

4.2.2 Il modulo Agent System

Il *core* dell'ambiente deve gestire tutti i meccanismi necessari al corretto funzionamento dei servizi di base (Figura 18). È stato progettato in

diversi moduli concettualmente e implementativamente distinti, in base alle diverse funzionalità fornite.

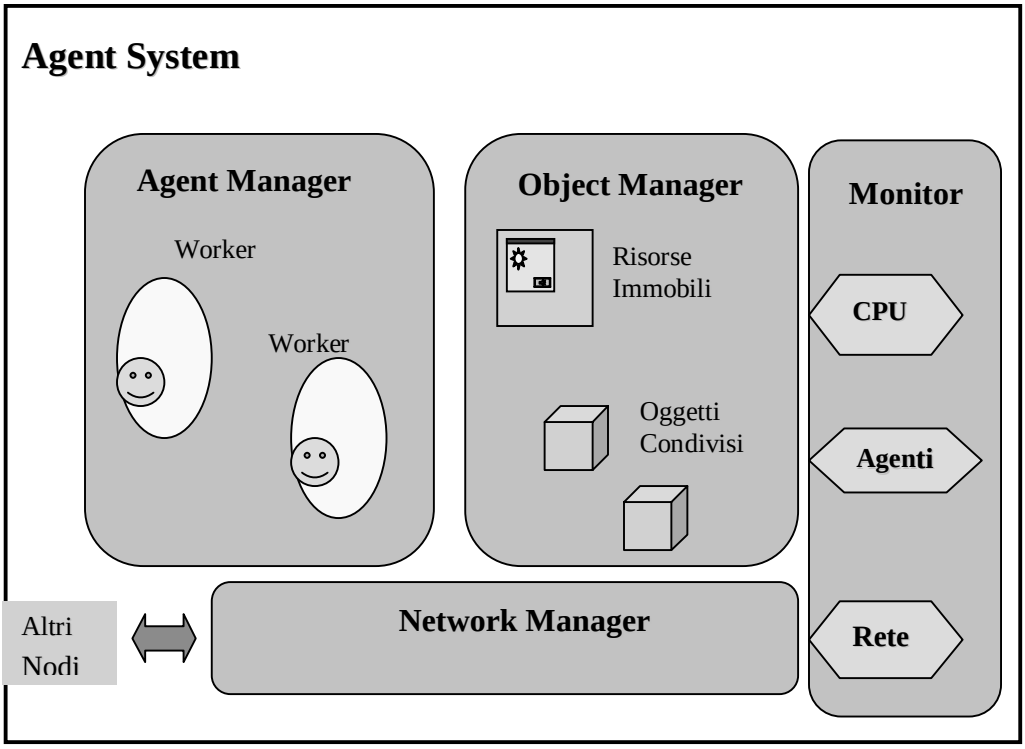

Figura 18: Architettura del modulo Agent System, con i moduli di gestione della Rete, degli Agenti, degli Oggetti e del Monitor.

In Figura 18 sono evidenziati quattro moduli:

- **Agent Manager:** ha il compito di gestire gli Agenti, in particolare si occupa di permettere l'esecuzione, l'ingresso e l'uscita dal nodo.
- **Object Manager:** gestisce l'insieme degli Oggetti che devono essere condivisi e di quelli che non possono essere mossi perché legati al nodo; sebbene siano concettualmente due cose diverse in realtà si tratta, per entrambe le necessità, di fornire un contenitore che mantenga disponibili gli Oggetti ai proprietari quando risiedono nel nodo.

- **Monitor:** fornisce dati sullo stato del nodo e della rete, permette inoltre di verificare lo stato di esecuzione degli agenti attualmente residenti nel nodo.

- **Network Manager:** gestisce l'interazione con gli altri nodi, mantiene delle connessioni *stream* con gli altri membri del dominio e demanda tutto il traffico destinato all'esterno del Dominio ad un server dedicato detto **Gateway.**

L'Agent Manager mantiene un DataBase dei Worker attualmente in esecuzione e quando arriva un Agente da remoto crea un nuovo Worker e lo aggiunge. Il DataBase è realizzato come una semplice lista dinamica a ricerca sequenziale che può essere poco efficiente nel caso ci siano molti Agenti presenti nel nodo.

Ogni Agente, all'atto della creazione, viene caratterizzato da un identificatore globale unico (**AgentID**), formato dai valori ***Dominio : nodo di creazione : numero progressivo***. Questo identificatore è necessario per qualsiasi interazione con l'Agente. Il nodo responsabile della creazione di un Agente ha anche il compito di mantenere traccia della sua posizione, la presenza dei due campi *dominio e nodo di creazione* nell'identificatore sono molto utili in questo senso per la gestione del protocollo di consistenza di questo dato e per l'eventuale necessità di rintracciare l'Agente nella rete. Quando un agente modifica la sua posizione, l'Agent Manager locale invia un *comando di aggiornamento* all'Agent Manager del sito di origine dell'agente (in seguito verranno analizzati in dettaglio i meccanismi di coordinamento tra nodi e domini). In qualsiasi istante sia necessario conoscere la posizione di un agente (ad esempio perché deve essere recapitato un messaggio) è sufficiente chiederla al gestore nel nodo di creazione che mantiene sempre aggiornata l'informazione.

Il modulo **Object Manager** è in realtà molto semplice, ha il compito di mantenere i riferimenti agli Oggetti depositati per motivi di condivisione o di persistenza. Non ha quindi bisogno di realizzare particolari funzioni, necessità solo di un semplice DataBase (realizzato con una lista dinamica) in

cui mantiene gli Oggetti che vengono depositati. Per ogni oggetto, l'Object Manager crea un identificatore con validità locale (un semplice numero progressivo) attraverso cui l'Agente può riottenere il riferimento.

Chiunque possieda l'identificatore può accedere agli Oggetti depositati, non c'è neppure verifica che l'identificatore sia corretto oppure appartenga ad un altro contesto; questo crea seri problemi di sicurezza sulla protezione e la privacy dei dati di un Agente, uno sviluppo futuro necessario dovrà essere la creazione di una interfaccia *sicura* a questo gestore.

Il **Monitor** ha il compito di fornire la percezione del mondo agli Agenti, sono necessarie rilevazioni sia sullo stato del nodo e della rete che degli agenti. Tra le tre rilevazioni, le prime due sono fondamentali solamente per la realizzazione di applicazioni adattative mentre la rilevazione dello stato degli agenti può essere necessario in applicazioni più generali. Per *stato degli Agenti* in particolare si può intendere:

- **Esistenza, Posizione, Funzionamento**
- **Identità e Permessi**
- **Storia passata**

Il primo gruppo di dati è rilevabile con estrema semplicità essendo informazioni direttamente gestite dall'Agent Manager per le necessità di esecuzione. In Appendice A sono evidenziate le diverse chiamate che permettono di verificare queste caratteristiche conoscendo l'identificatore dell'Agente monitorato.

L'Identità ed i Permessi di un Agente sono informazioni strettamente legate alla gestione della sicurezza, dovranno quindi essere realizzate in un secondo tempo in base al modello di protezione che verrà applicato.

La rilevazione e registrazione della storia di un Agente può essere molto interessante sia per necessità di management che di debugging delle applicazioni; la granularità della traccia dell'agente è determinante per l'efficienza del servizio, per questo motivo si è pensato ad una semplice registrazione dei diversi siti visitati dall'agente e dei messaggi trasferiti. Una analisi più particolareggiata infatti, necessità la creazione di sonde che

prevedono l'instrumentazione del bytecode degli agenti e quindi non è stata affrontata. In ogni caso, questo servizio è da considerarsi *on-demand*, attivabile quindi in un certo istante e per un determinato periodo; sicuramente sarebbe troppo pesante registrare la vita di tutti gli Agenti senza effettiva necessità.

Questi servizi ora analizzati sono orientati sull'analisi del singolo agente; indici globali (all'interno di un place o di un nodo) possono rivestire particolare importanza soprattutto per applicazioni di System Management e di Sicurezza. Come evidenziato in Appendice A, sono disponibili i seguenti rilievi sul nodo e sui Place:

- **Numero Place presenti in un Nodo (e nome di essi)**
- **Numero Agenti presenti in un Nodo (ed in un Place)**
- **Quanti e quali oggetti sono depositati presso l'Object Manager**

È importante sottolineare l'importanza del monitoraggio, sicuramente una evoluzione di questo modulo in diverse direzioni è fondamentale per la qualità dell'intero supporto.

Il **Network Manager** ha il compito di gestire le connessioni con gli altri nodi partecipanti per tutte le necessità remote. Inizialmente legge da un file di configurazione le informazioni che caratterizzano il Dominio e quindi il Nodo:

- **Indirizzo Nodo : Nome Default Place** di tutti i nodi che appartengono al nodo, il Network Manager deve avere completa conoscenza di chi è ammesso a partecipare al Dominio indipendentemente dal fatto che in quell'istante sia presente o meno.
- **Nome Dominio : Indirizzo GateWay : Porta di accesso**. Con queste informazioni il gestore è in grado di comunicare, se necessario, con il GateWay di Dominio e quindi eventualmente permettere la migrazione e lo scambio di messaggi al di fuori del contesto locale.

Dopo aver registrato queste informazioni, il gestore verifica la presenza dei nodi partecipanti al dominio tentando di aprire una connessione. I nodi che

rispondono vengono marcati come attivi e viene creato un Thread che ha il compito di bloccarsi in ricezione sulla connessione pronto ad esaudire le richieste del nodo remoto. In partecipanti che non rispondono vengono invece marcati come inattivi, in qualsiasi momento provenisse una richiesta di connessione da parte di uno di questi siti il gestore attiva la connessione e genera il Thread di gestione.

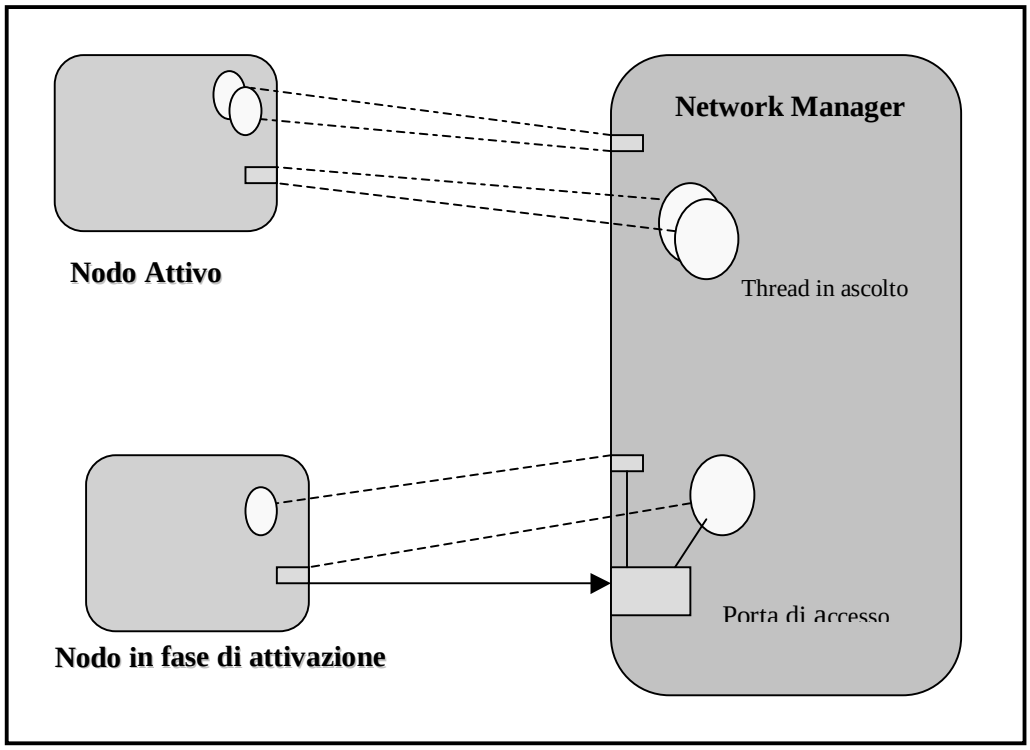

Figura 19: Il Network Manager è sempre in attesa su una porta fissata (caratteristica del Dominio); quando arriva la richiesta di attivazione da parte di un nodo partecipante crea dinamicamente due thread che *ascoltano* le richieste sulle connessioni.

Come evidenziato in Figura 19, le connessioni tra due nodi sono in realtà due; vengono infatti distinti due canali bidirezionali per il passaggio separato di *Agenti* e *Comandi* (il motivo di questa separazione verrà evidenziato in seguito); esistono quindi due Thread in ascolto, uno per ogni connessione.

Il Network Manager mette quindi a disposizione una serie di servizi all'Agent System per la spedizione di richieste sia ai nodi partecipanti attivi che al Gateway per l'esterno. La connessione con il Gateway non è fissa, il gestore, solo in caso di necessità, chiede un accesso e apre una connessione solo per la interazione desiderata. Questa realizzazione è stata scelta per semplificare lo stato del Gateway che non necessita di tenere traccia di tutte le sue connessioni ma semplicemente, ogni volta che arriva la richiesta di un passaggio, esegue il servizio.

4.2.3 Il Place Manager

Questo modulo ha il compito di creare e gestire i diversi contesti all'interno dei quali eseguono gli Agenti (Figura 20).

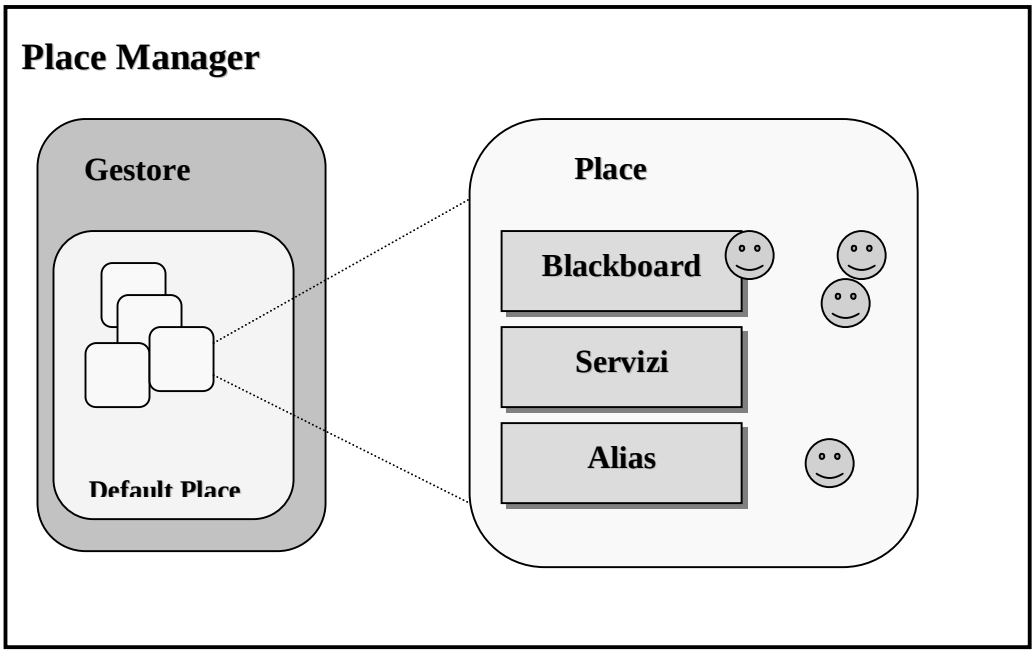

Figura 20: Struttura del Place Manager; ogni Place generato dinamicamente possiede una propria Blackboard, una lista dei Servizi ed una lista di Alias.

Un contesto di esecuzione (semplicemente detto Place) è caratterizzato da tre servizi:

- **Blackboard:** un Agente ha la possibilità di lasciare un messaggio depositato. Il messaggio è caratterizzato da una *keyword*, chiunque può accedere sia conoscendo la keyword che scorrendo dall'inizio la lista dei messaggi.

- **Servizi:** ogni Place possiede una lista dei servizi disponibili, devono essere gli Agenti servitori stessi che si registrano nel Place per fornire la coppia **Nome Servizio : Identificatore Servitore**. Se un Agente desidera uno specifico servizio è sufficiente ne conosca il nome per ottenere l'identificatore del servitore di interesse.

- **Alias:** simile al concetto di badge in Mole, un Alias è un nome astratto che un Agente può assumere dinamicamente in un Place, in qualsiasi istante l'Agente può registrarsi oppure ritirarsi; i nomi devono essere unici all'interno del place, la registrazione può quindi fallire. Un Agente registrato deposita il proprio identificatore associato al nome, non ha nessun vincolo in quel Place e non perde la sua mobilità.

Chiunque può entrare in qualsiasi Place, non esistono attualmente restrizioni di nessun tipo ma potrebbe essere importante rendere possibile la regolamentazione degli accessi da parte del creatore del Place, l'Agente che si è assunto il compito di creare un contesto dovrebbe avere questa possibilità.

4.2.4 Information Manager

Questo modulo ha il compito di gestire la ricerca delle informazioni all'interno ed all'esterno del Dominio. È costruito completamente al di sopra degli altri due moduli, per questo motivo è stato possibile una realizzazione completa ad Agenti. Il gestore del servizio lancia degli agenti servitori che hanno il compito di muoversi nel dominio (o tra i domini) per ottenere le informazioni desiderate. Sono disponibili due diversi tipi di ricerca (App. A):

- **Ricerca di un Agente:** è possibile chiedere la posizione di un Agente attraverso il suo identificatore oppure un alias (sia un servizio che un

vero e proprio alias). In entrambi i casi il gestore crea un agente di servizio (sprovvisto di Mailbox per efficienza) che si muove tra i nodi del Dominio per cercare l'Agente. Una volta trovato torna al mittente per comunicare la locazione del bersaglio. Nel caso della ricerca mediante alias, il servitore percorre tutti i Place di tutti i nodi e torna quando trova il primo alias uguale; ovviamente è possibile che più Agenti siano registrati con lo stesso nome in place diversi, questo può essere interessante per realizzare servizi replicati (sarebbero quindi necessarie diverse politiche di ricerca).

- **Ricerca di un Place:** permette ad un Agente di trovare uno specifico Place all'interno di un dominio. Anche in questo caso il gestore crea un agente che naviga il dominio e torna una volta acquisita l'informazione.

Questo modulo in realtà può essere considerato come una applicazione, è stato incapsulato nel supporto perché concettualmente la possibilità di ottenere informazioni remote è una funzionalità fondamentale di un ambiente ad Agenti. La non unicità globale (anche all'interno di un dominio) dei nomi dei servizi, degli alias e dei place evidenzia l'importanza del servizio di informazioni come gestore del Dominio. L'esperienza delle Yellow Pages di AgentTcl ha evidenziato come il servizio informativo, attraverso specifiche politiche di ricerca, può fornire un importante supporto per la gestione di risorse distribuite e/o replicate.

4.3 Il Coordinamento dei Nodi

All'interno di un Dominio, i gestori presenti su ogni nodo si coordinano per realizzare le loro funzionalità. Durante l'esecuzione, i nodi attivi sono completamente connessi tra loro, ogni connessione è realizzata attraverso due canali bidirezionali ***ObjectStream***: come evidenziato nel Capitolo 3, Java è un linguaggio naturalmente orientato alla programmazione secondo il modello REV (Sez. 1.2), per questo motivo l'interazione tra nodi viene effettuata mediante lo scambio di *Comandi:* sono oggetti che vengono eseguiti dal supporto destinazione e realizzano direttamente ciò che il supporto mittente desiderava. Ci sono quindi due canali, un canale per i Comandi ed uno per gli Agenti. Quando riceve un Comando, il supporto invoca il metodo **exe()**, crea un Worker invece per ogni oggetto che proviene dal canale Agenti (Figura 21).

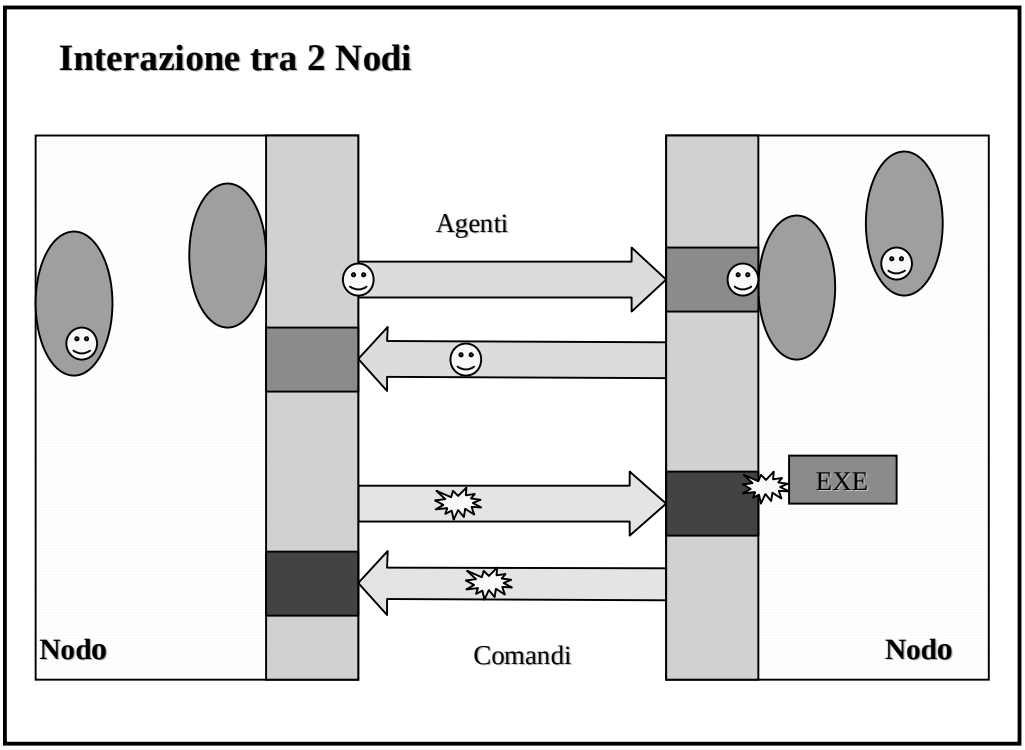

Figura 21: L'interazione tra due nodi avviene attraverso due canali bidirezionali.

L'interazione tra i nodi è completamente asincrona e sfrutta la *reliability* dello *Stream* come garanzia del corretto trasferimento degli Oggetti. Questo rende l'interazione estremamente flessibile, è possibile un qualsiasi tipo di coordinamento tra i nodi semplicemente realizzando l'oggetto Comando ad hoc.

4.3.1 La posizione degli Agenti

Conoscere in ogni istante la posizione degli Agenti è una necessità fondamentale; la possibilità di recapitare dei messaggi indipendentemente dalla posizione dell'Agente e la necessità di poterlo sempre individuare sia per ragioni di sicurezza che per ragioni applicative obbligano il sistema a tenere traccia della locazione di un Agente che si muove nella rete.

L'informazione viene mantenuta unicamente dall'Agent System del nodo dove è stato creato l'Agente, questa soluzione ha due vantaggi essenziali:

- Ipotizzando una fondamentale equiprobabilità tra i nodi per quanto riguarda la nascita di nuovi Agenti, ogni nodo ha pari responsabilità di gestione, in questo modo si ottiene una distribuzione uniforme del servizio e quindi del carico necessario per il mantenimento.

- L'identificatore dell'Agente contiene, per garantirne l'unicità, sia il nome del Dominio che del nodo di nascita; è sempre possibile quindi contattare il nodo di nascita per chiedere l'attuale posizione oppure per comunicare un avvenuto movimento.

Ogni nodo quindi mantiene una tabella con la posizione degli agenti che sono stati creati localmente, non c'è alcun legame tra il creatore dell'Agente ed il nodo. Questo può comportare problemi di sicurezza, risolvibili solamente regolamentando la capacità di creare Agenti figli in un determinato nodo.

4.4 I Gateway e la Gestione del Dominio

Il Dominio rappresenta una località logica, può però anche rispecchiare una località fisica. Per questo motivo il sistema deve ipotizzare che non ci sia nessun legame fisico tra i Domini se non quelli esplicitamente indicati in fase di installazione. Deve infatti essere indicato un nodo che assuma il ruolo di **Gateway,** di porta con l'esterno: un servitore dedicato all'interazione con i domini esterni.

Il Gateway è un Server completamente indipendente dai nodi di esecuzione, può condividere la stessa macchina fisica con uno di essi ma non esiste nessun vincolo a riguardo. In fase di inizializzazione, legge da un file di configurazione quali sono i Domini noti a cui gli Agenti possono accedere. Questa impostazione è fortemente statica, un approccio dinamico al problema richiede un livello gerarchico superiore al Dominio, l'esistenza di un Super Dominio che abbia conoscenza di tutte le località presenti. La realizzazione di Super Domini è un passo fondamentale per la futura evoluzione del sistema in ipotesi di utilizzo su reti di grandi dimensioni.

Il Gateway non ha stato, non mantiene connessioni attive o tabelle di gestione, rimane semplicemente in attesa sulle porte designate e aspetta una richiesta di trasmissione dall'esterno verso l'interno o viceversa.

Quando un nodo ha bisogno di trasferire un Oggetto al di fuori del Dominio lo incapsula in un Comando della Classe *TransCommand* all'interno del quale viene registrato anche la destinazione. Il nodo quindi passa il comando al Gateway del Dominio, questo lo trasmette al Gateway del Dominio di destinazione (se lo conosce) dove il comando viene eseguito (viene invocato il metodo exe()). Il comando quindi spedisce l'oggetto che trasporta verso il nodo destinazione (Figura 22); se il nodo destinazione non è specificato, chiede al gestore quale è il *Nodo di Default* per il Dominio.

Se i Domini si trovano in due reti collegate solamente attraverso una macchina bridge oppure attraverso un FireWall, il Gateway dirige il comando verso un **Proxy**, un semplice demone che risiede sulla macchina bridge ed ha il compito di passare l'oggetto tra le due località. La presenza di un proxy per l'interazione tra due Domini deve essere specificata nel file di configurazione. La verifica sperimentale ha evidenziato un peggioramento di circa il 70% nel tempo di trasferimento nel caso di uso del proxy tra diverse reti locali; in caso

di reti geografiche il costo dovrebbe essere pressoché trascurabile a causa della banda limitata.

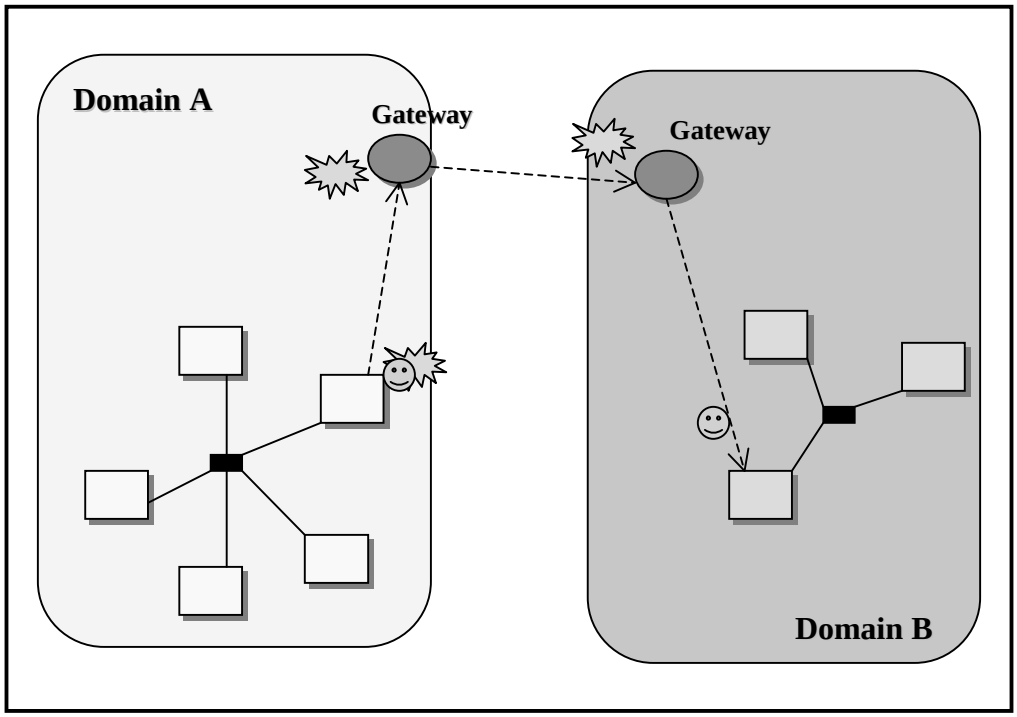

Figura 22: Migrazione di un Agente in un altro Dominio; l'Agente viene incapsulato in un Oggetto di tipo *TransCommand*, viene spedito al Gateway di Dominio, da qui a quello dell'altro Dominio dove l'Agente viene estratto e

Il Gateway rappresenta quindi a tutti gli effetti l'unico gestore centralizzato del Dominio, permettendo l'ingresso e l'uscita degli Agenti, ha la possibilità di verificare ed eventualmente negare l'accesso ad agenti non accreditati, in questo senso è quindi il punto più importante per la gestione della sicurezza del Dominio.

4.5 Applicazioni per il Monitoraggio e l'Amministrazione

Il Sistema ad Agenti realizzato è stato utilizzato per creare un **Sistema di Amministrazione e Monitoraggio:** la gestione di reti locali di grandi dimensioni è spesso molto complessa, ripetitiva e richiede assidui controlli da parte di uno o più tecnici anche solo per semplici controlli di routine, verifiche di corretto funzionamento o aggiornamenti del software installato.

Il monitoraggio distribuito permette di controllare lo stato dei nodi della rete da una unica stazione di controllo e permette interventi tempestivi in caso di malfunzionamenti.

Sono stati inoltre creati un insieme di tool di Amministrazione per eseguire diversi compiti in modo automatizzato, senza cioè l'intervento dell'operatore su ogni nodo.

4.5.1 Il Monitoraggio secondo il Modello ad Agenti

Il monitoraggio distribuito per fini di Management è tradizionalmente realizzato secondo un modello Master/Slave [Fink97]: un coordinatore centrale (Master) ha il compito di interrogare e controllare dei demoni dislocati sui nodi del sistema il cui compito è verificare localmente lo stato della risorsa da monitorare (Figura 23). Attraverso queste informazioni, l'amministratore del sistema può verificare da una unica postazione lo stato generale ed eventualmente intervenire per mantenerne l'efficienza.

Questo tipo di approccio, sebbene molto efficiente per semplici query (come lo stato della CPU), risulta in pratica molto poco flessibile e non permette una facile espansione delle funzionalità.

La stessa fase di creazione e coordinamento degli Slave su tutti i nodi presenta grosse difficoltà ed il Master è un punto critico per quanto riguarda la tolleranza e la gestione dei guasti.

Un sistema di rilevazione basato sulla mobilità non è legato ad uno schema centralizzato, gli Agenti Analizzatori si muovono per il dominio a raccogliere dati seguendo diverse politiche dinamiche e sono il grado di eseguire operazioni particolari localmente.

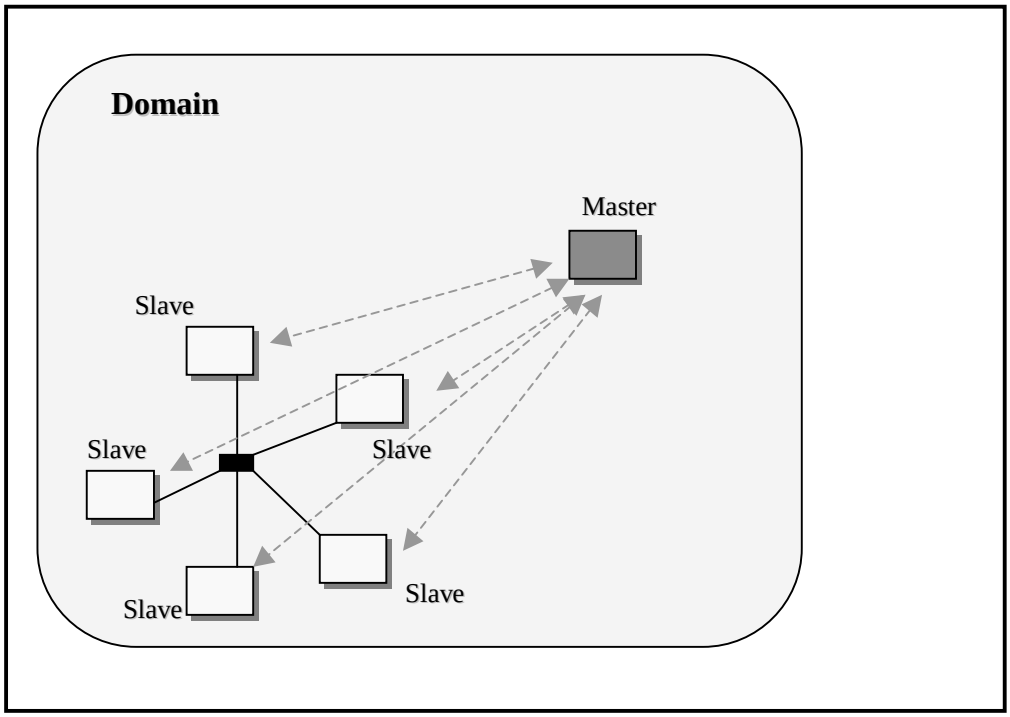

Figura 23: Un sistema di monitoraggio nel modello Master/Slave prevede l'esistenza di uno Slave in ogni nodo in grado di rispondere solo a semplici query prefissate. Dallo schema risulta evidente il collo di bottiglia nella comunicazione con il Master.

Lo schema più semplice ed immediato consiste nell'esplorazione di tutto il Dominio da parte di un solo Agente: inizialmente verifica quali nodi sono attivi, poi si sposta nei diversi siti per l'analisi (Figura 24).

Quest'ultima soluzione non è sicuramente veloce a causa del lungo percorso che l'agente deve compiere in caso di Domini molto grandi; esiste però il vantaggio di avere una intrusione bassissima rispetto a qualsiasi altro modello. Per migliorare le prestazioni è possibile aumentare il parallelismo della rilevazione creando un Agente Controllore in grado di lanciare un certo numero di Agenti Analizzatori con il compito di analizzare una determinata lista di siti.

Negli approcci tradizionali basati sul modello Master/Slave, lo Slave è in grado di eseguire solo delle semplici rilevazioni predefinite, mentre un

Agente può ottenere dinamicamente più informazioni, raccogliere dati statistici ed elaborare localmente eventuali grosse moli di dati ed eseguire operazioni specifiche in loco. Intuitivamente il modello ad Agenti risulta più efficiente in quanto distribuisce meglio il carico tra i nodi e necessita di una banda inferiore. Sono stati effettuati dei confronti sperimentali tra i due modelli i cui risultati sono evidenziati in Figura 25.

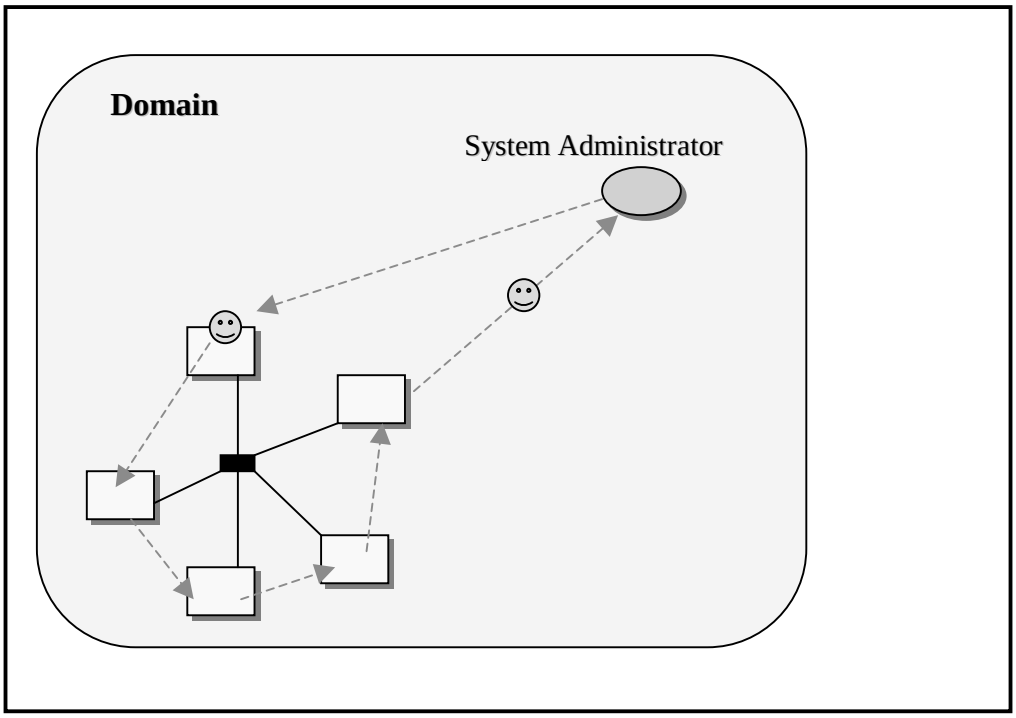

Figura 24: La analisi effettuata da un Agente da un singolo Agente può essere più lenta perché sequenziale, permette però molta flessibilità ed l'efficienza in molte condizioni.

4.5.2 Tipo di Monitoraggio

Lo stato di un Nodo può essere visto sotto diversi aspetti:

- **Livello Applicativo:** è possibile rilevare diversi dati sullo stato dei Place e delle Applicazioni che eseguono sui diversi nodi. Tipicamente può essere interessante conoscere quanti Agenti sono presenti, quante e quali risorse utilizzano e quali servizi ci sono a disposizione. In Tabella VIII sono evidenziate le informazioni che caratterizzano lo stato del Nodo.

Informazioni disponibili
n° Agenti di Sistema Attivi (Agenti non Rintracciabili)
n° Agenti Utente Attivi (Agenti Rintracciabili)
n° Place attivati
Quali Servizi sono disponibili nei diversi Place

Tabella VIII: Dati rilevabili a Livello Utente. In Appendice A sono evidenziate le chiamate che necessarie per ottenere le informazioni dal supporto.

Nell'applicazione tradizionale, lo Slave è in grado di rispondere ad una query per volta con la conseguenza che il Master deve eseguire più query per ottenere lo stato del Nodo.

L'applicazione ad Agenti ha parallelismo variabile, i risultati in Figura 25 si riferiscono ad una rete di 9 nodi di Sun SPARC4, SPARC5 ed ULTRA1; come si può vedere nel grafico, un solo Agente Rilevatore è mediamente più lento di una rilevazione M/S, mentre già due Agenti riescono ad ottenere lo stato di tutti i nodi in un tempo inferiore e con un minore uso di risorse.

- **Livello di Sistema:** la rilevazione dello stato del Nodo inteso come insieme di risorse fisiche disponibili, richiede una realizzazione dipendente dalla piattaforma ed è stata per questo affrontata la realizzazione limitata al mondo UNIX. In Tabella IX sono evidenziate le sonde disponibili che corrispondono a specifici comandi UNIX; le sonde così create sono state logicamente raggruppate nella Classe **Monitor** che è stata inglobata nel supporto di base e quindi disponibile anche agli altri Agenti (App. A).

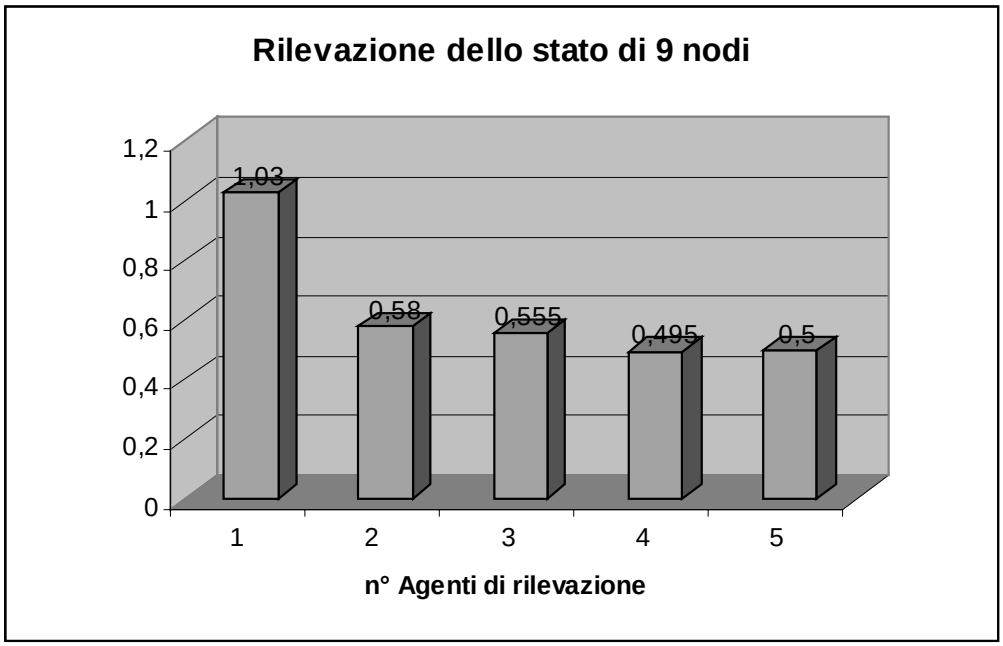

Figura 25: Tempo necessario alla rilevazione dello stato di 9 nodi con un numero variabile di Agenti Rilevatori. I tempi sono stati normalizzati al tempo necessario ad un supporto Master/Slave.

Informazioni sul Nodo
Carico CPU
Spazio disponibile su disco
Swap disponibile
Informazioni sulla Rete
Connettività altri Host
Collision Rate
Network Latency

Tabella IX: Sonde disponibili per la determinazione dello stato del nodo e della rete. Sono tutte realizzate attraverso l'esecuzione di comandi di Sistema UNIX come ad esempio **ping** e **df**.

4.5.3 L'Amministrazione

Per l'amministrazione e la gestione di grandi reti locali esistono software specifici, ci sono per lo più diversi tool realizzati ad hoc per supplire a specifiche necessità. Questo è dovuto alla specificità delle esigenze, che necessitano una estrema flessibilità nel software che intende fornire un servizio generale.

Un approccio ad Agenti, nell'ipotesi che questi possano eseguire con diritti di Super Utente, si risolve nella realizzazione di una famiglia di diversi Agenti da cui è eventualmente possibile evolverne di nuovi più specializzati.

Nel tentativo di evidenziare quali siano le caratteristiche principali che devono possedere questi Agenti sono stati realizzati diversi tool per l'amministrazione nel mondo UNIX: un *Installer per applicazioni*, un *Gestore per il Backup distribuito* ed un *Esecutore Remoto.*

L'Installazione Remota. Un tipico problema di gestione è il mantenimento della consistenza dei pacchetti software installati sulle diverse macchine. Un approccio ad Agenti per questo tipo di problemi, si risolve in generale nella realizzazione di un Agente ad hoc che ha il compito dapprima, di verificare nei vari nodi l'eventuale necessità di aggiornamento, e poi di eseguire e controllare localmente i programmi di installazione propri del prodotto.

Le applicazioni ad Agenti realizzabili con questo Sistema, ad esempio, necessitano la presenza dei loro *class file* (il bytecode delle classi che le compongono [Java]) disponibili su tutti i nodi, questo affinché il *Class Loader* di Java sia sempre in grado di caricare dinamicamente le classi. Per poter installare un applicativo ad Agenti è quindi necessario rendere disponibile una copia consistente dei class file su tutti i nodi su cui si intende eseguire. La propagazione consistente di questi pacchetti viene realizzata da un Agente che segue i seguenti passi:

- Verifica l'esistenza e la versione di un file (o di un insieme di file) e genera una lista di update.

- Preleva l'insieme dei file aggiornati da una locazione prefissata: l'Agente crea un *tar file* (un formato standard per gli archivi) e lo carica in memoria sotto forma di un *array di byte* nel suo stato interno.

- Si muove nei nodi da aggiornare, dove scarica il tar file e ricrea i direttori dell'applicazione.

Nell'ipotesi che il pacchetto sia molto grande, l'agente verifica prima quali siano i nodi che hanno bisogno di aggiornamento, e solo a questo punto si carica del pacchetto per muoversi solo dove è necessario. Il *tar file* non è un formato *compresso*, i file nell'archivio non vengono processati per ridurne la grandezza dovuta a ridondanza, è però possibile utilizzare qualsiasi filtro di compressione per ridurre così la quantità di dati che devono essere trasferiti e quindi la grandezza dell'agente in fase di installazione (comprimere il tar file ha ovviamente senso se il costo di compressione è minore del costo di trasmissione del file nella forma originale).

La gestione del Backup distribuito. Uno dei principali problemi di amministrazione di un File System distribuito è il periodico Backup realizzato per motivi di protezione dei dati. Spesso infatti il File System non ha visibilità unica, ma è partizionato ed organizzato in modo da fornire apparenze personalizzate a diversi utenti in diverse postazioni. Il salvataggio dei dati in queste situazioni è lungo e complesso, e spesso richiede agli utenti di sospendere il lavoro, in modo che l'amministratore possa accedere alla macchina fisica.

Per risolvere questo problema è stato realizzato un Agente che ha il compito di muoversi in ogni nodo, verificare quali utenti hanno i loro dati localmente, leggere una loro lista di Backup dove specificano cosa desiderano venga copiato e poi scaricare questi dati in un sito di Backup specifico (eventualmente provvisto di unità nastro). Questo approccio permette una gestione del backup dei dati indipendentemente dalla organizzazione del sistema e dalla sua grandezza. È possibile anche trasformare il servizio da periodico in *on-demand;* l'utente stesso può invocare il Backup quando la situazione lo richiede.

L'Esecuzione Remota di Comandi. Questo servizio viene già fornito nel mondo UNIX dal comando **rsh**; è stato fornito anche in un'applicazione ad Agenti sia per renderlo disponibile al di fuori del mondo UNIX, sia perché rappresenta la necessità fondamentale per risolvere problemi specifici di amministrazione. L'Agente può essere istruito a visitare un insieme di nodi ed eseguire diversi comandi dipendentemente dal contesto; la programmazione è a un livello sicuramente più elevato di quello degli script normalmente utilizzati con rsh, per questo è più semplice creare tool anche complessi per le necessità più diverse.

Un esempio tipico di uso di questo tool è la realizzazione una sequenza di *shutdown* coordinato di un gruppo di macchine; un Agente si visita i nodi specificati ed invoca localmente il comando **shutdown.**

Per eseguire queste funzioni, gli Agenti devono avere la possibilità di eseguire con diritti di Super Utente. La garanzia dal punto di vista della sicurezza viene in questo caso fornita dal fatto che solo il Super Utente è autorizzato a creare Agenti.

4.5.4 L'installazione di un Pacchetto Software

Per verificare il tool ad Agenti di installazione remota è stato confrontato con un semplice script in shell Unix che fa uso dei comandi **rcp** ed **rsh**. Questo script esegue i seguenti passi:

- verifica del pacchetto sul nodo remoto;
- crea il tar file del pacchetto da installare;
- copia il file sulla macchina destinazione mediante il comando **rcp.**
- esegue sulla macchina destinazione l'estrazione del pacchetto dal tar file (**rsh nomehost tar cf …**);
- attiva il programma installato (ad esempio lancia un demone).

Gli ultimi tre passi vengono ripetuti per tutti i nodi che necessitano dell'installazione. Si tratta di un esempio molto semplice di soluzione ad hoc

per risolvere uno specifico problema; la sequenza è molto inefficiente perché viene letto da disco il tar file dal comando rcp per ogni copia in ogni nodo.

Lo stesso compito eseguito ad Agenti può essere realizzato da un numero variabile di Agenti Installatori che leggono il pacchetto, creano il tar file e poi lo propagano nei nodi loro destinati. In Figura 26 sono espressi i risultati del confronto tra le due applicazioni; i risultati sono stati normalizzati rispetto al costo dell'applicazione ottimizzata, che consiste nella sequenza dello script, ipotizzando però che il file non venga riletto da disco per effettuare la copia su ogni nodo.

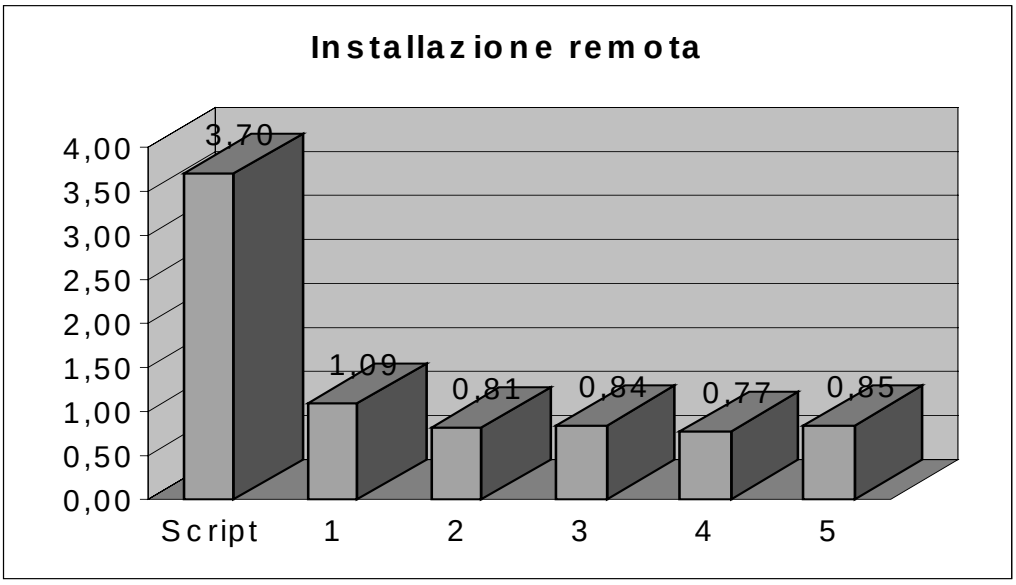

Figura 26: Il costo di installazione di un pacchetto di circa 40 Kbyte per uno script Unix e per un numero variabile di Agenti Installatori (da 1 a 5).

La simulazione non evidenzia un particolare miglioramento al crescere del numero di Agenti Installatori; probabilmente questo risultato è l'effetto del costo di accesso ai dischi superiore ai costi di migrazione e coordinamento degli Agenti. L'esperienza ha comunque evidenziato la validità dell'uso del modello ad Agenti non solo sotto il profilo della flessibilità ma anche dell'efficienza nell'uso delle risorse.

5 Conclusione

L’esperienza acquisita nella programmazione ad Agenti e le evidenze sperimentali hanno sottolineato la validità del modello. Il tempo di sviluppo di una applicazione di rete è drasticamente ridotto poiché essenzialmente diminuiscono le difficoltà di coordinamento e progettazione.

L'uso di Java ha favorito molti aspetti, primo fra tutti l'indipendenza dalla piattaforma d'esecuzione; il fatto che Java fornisca solo una mobilità di tipo Weak però vincola la realizzazione degli Agenti a spezzettare in diversi metodi eseguiti nei diversi contesti, ottenendo un codice poco strutturato. La possibilità di utilizzare ogni risorsa di Java all'interno degli Agenti, rende comunque il supporto potenzialmente valido per lo sviluppo di applicazioni di una certa complessità in tempi piuttosto brevi. Un programmatore di Agenti non deve conoscere i particolari tipici della programmazione di rete: è sufficiente che conosca i principi della programmazione concorrente e può ragionare sempre a livello applicativo senza occuparsi della gestione della mobilità.

Gli strumenti di comunicazione introdotti sono sufficienti per ottenere qualsiasi tipo di interazione; è possibile però creare astrazioni di più alto livello (come lo spazio delle tuple) che permettano una più semplice e potente interazione; ciò ha un duplice effetto: semplificare la programmazione degli Agenti e conseguentemente ridurre la grandezza fisica dell'agente stesso che quindi è più agile in caso di migrazioni.

La suddivisione logica della rete in Place e Domini ha permesso una agevole gestione di diverse situazioni topologiche; in particolare è stata verificata la validità del modello in presenza di clustering e quindi nell'accesso a reti locali protette. I Domini possono rappresentare una località

dal punto di vista della sicurezza, permettono la visione della rete come un insieme di isole indipendenti con le proprie politiche di protezione e la propria autonomia.

Durante lo sviluppo del supporto e delle applicazioni si è notata l'importanza di fornire diverse astrazioni nella mobilità come ad esempio la possibilità di muoversi in un altro Dominio senza conoscerne i Place. Queste funzionalità, spesso molto semplici e poco costose, permettono all'Agente di demandare eventuali decisioni a posteriori, di *avvicinarsi* prima alle informazioni (entrando nel Dominio di destinazione) e poi prendere la decisione di migrazione opportuna (Place specifico), riducendo il costo per l'acquisizione delle informazioni.

La progettazione di una applicazione di Network Management ad Agenti e nel modello Master/Slave ha evidenziato come, non solo, gli Agenti siano in generale più efficienti ed utilizzino le risorse in modo più equilibrato, ma ha anche messo in luce la flessibilità di una applicazione ad Agenti soprattutto nello sviluppo di nuove funzionalità, estensioni estremamente complesse in applicazioni tradizionali.

L'esperienza acquisita non può quindi che confermare l'ottimismo di molti autori sulla validità del Modello ad Agenti Mobili: le potenzialità sono tali da permettere un salto notevole nella programmazione distribuita. Il più grande problema sta proprio nelle ampie potenzialità: è infatti necessario un sistema di protezione e sicurezza molto completo ed affidabile affinché sia possibile utilizzare questo modello in applicazioni per Internet. Tale sistema deve essere sufficientemente forte da prevenire attacchi e violazioni ma anche deve permettere un uso efficiente delle risorse. Queste due necessità sono spesso in contrasto: il sistema di sicurezza deve quindi cercare di integrarsi in modo completo con l'organizzazione del supporto ad Agenti per sfruttarne le caratteristiche.

6 App.A: Le chiamate disponibili agli Agenti

Nei seguenti paragrafi verranno elencate e descritte le chiamate a disposizione dell'Agente per usufruire delle funzionalità del Sistema; questi documenti sono stati generati automaticamente mediante il tool di Java ***JavaDoc*** [Java].

6.1 Il Package AgentSystem

Questo package contiene il modulo **Agent System** descritto nel Capitolo 4.

6.1.1 La Classe AgentSystem

```
java.lang.Object
   |
   +----AgentSystem.AgentSystem
```

---

public class **AgentSystem**

extends Object

Classe di interazione tra sistema ed agenti. Sono resi disponibili tutti i servizi di base necessari all'agente, in particolare provvede alla interfaccia per la mobilita'.

Constructor Index

- **AgentSystem**()

Method Index

- **createAgent**(String, Object)

  Crea nuovo agente dalla classe cl ad associa args obj Restituisce AgentID dell'Agente

- **createAgent**(String, Object, boolean)

  Crea nuovo agente dalla classe cl ad associa args obj Restituisce AgentID dell'Agente potendo specificare se e' o no Searchable.

- **getAllDomain**()

  Ritorna la lista di tutti i siti del dominio ed il nome dei place di default.

- **getCurrentLocation**()

  Ritorna la attuale posizione dell'Agente.

- **getNotSearchableAgentsNumber**()

- **getPersistentObject**(ObjectID)

  Cerca nel DB oggetto persistente

- **getPlaceNumber**()

- **getSearchableAgentsNumber**()

- **go**(AgentID, DomainName, NodeName, String)

  Muove Agente identificato da AgentID aid.

- **go**(AgentID, DomainName, String)

  Muove Agente identificato da AgentID aid.

- **go**(AgentID, NodeName, String)

  Muove Agente identificato da AgentID aid.

- **go**(AgentID, PlaceName, String)

- **go**(AgentID, String, String)

  Muove Agente identificato da AgentID aid.

- **isActive**(NodeName)

  Verifica se una specifico nodo e' attivo.

- **kill**(AgentID)

  Uccide Agente caratterizzato da AgentID

- **removePersistentObject**(ObjectID)

  Rimuove oggetto persistente dal DB

- **storePersistentObject**(Object)

  Salva nel DB oggetto che deve rimanere persistente

Constructors

AgentSystem

```
public AgentSystem()
```

Methods

go

```
public static synchronized void go(AgentID agid,
                                   NodeName n,
                                   String metodo) throws
CantGoException
```

Muove Agente identificato da AgentID aid. La specifica della destinazione e' il nodo in forma di NodeName. String metodo e' il metodo che verra' eseguito una volta arrivato a destinazione

go

```
public static synchronized void go(AgentID agid,
                                   String nodo,
                                   String metodo) throws
CantGoException
```

Muove Agente identificato da AgentID aid. La specifica della destinazione e' il nodo in forma di String. String metodo e' il metodo che verra' eseguito na volta arrivato a destinazione

go

```
public static synchronized void go(AgentID agid,
                                   PlaceName p,
                                   String metodo) throws
CantGoException
```

go

```
public static synchronized void go(AgentID agid,
                                   DomainName d,
                                   String metodo) throws
CantGoException
```

Muove Agente identificato da AgentID aid. La specifica della destinazione e' il Dominio DomainName d (il nodo viene preso quello di Default). String metodo e' il metodo che verra' eseguito na volta arrivato a destinazione

go

```
public static synchronized void go(AgentID agid,
                                   DomainName d,
                                   NodeName n,
                                   String metodo) throws
CantGoException
```

Muove Agente identificato da AgentID aid. La specifica della destinazione e' il Dominio DomainName d ed il nodo e' NodeName n. String metodo e' il metodo che verra' eseguito na volta arrivato a destinazione

kill

```
public static synchronized void kill(AgentID id)
```

Uccide Agente caratterizzato da AgentID

createAgent

```
public static synchronized AgentID createAgent(String cl,
                                                Object obj)
```

Crea nuovo agente dalla classe cl ad associa args obj Restituisce AgentID dell'Agente

createAgent

```
public static synchronized AgentID createAgent(String cl,
                                                Object obj,
                                                boolean s)
```

Crea nuovo agente dalla classe cl ad associa args obj Restituisce AgentID dell'Agente potendo specificare se e' o no Searchable.

storePersistentObject

```
public static ObjectID storePersistentObject(Object o)
```

Salva nel DB oggetto che deve rimanere persistente

getPersistentObject

```
public static Object getPersistentObject(ObjectID oid)
```

Cerca nel DB oggetto persistente

removePersistentObject

```
public static void removePersistentObject(ObjectID oid)
```

Rimuove oggetto persistente dal DB

getAllDomain

```
public static synchronized Location[] getAllDomain()
```

Ritorna la lista di tutti i siti del dominio ed il nome dei place di default.

getCurrentLocation

```
public static Location getCurrentLocation()
```

Ritorna la attuale posizione dell'Agente.

isActive

```
public static synchronized boolean isActive(NodeName n)
```

Verifica se una specifico nodo e' attivo.

getNotSearchableAgentsNumber

```
public static int getNotSearchableAgentsNumber()
```

getSearchableAgentsNumber

```
public static int getSearchableAgentsNumber()
```

getPlaceNumber

```
public static int getPlaceNumber()
```

6.1.2 La Classe Monitor

```
java.lang.Object
   |
   +----AgentSystem.Monitor
```

---

public class **Monitor**

extends Object

Questa classe viene istanziata dalla classe statica AgentSystem; ha il compito di rilevare lo stato del sistema, l'attuale implementazione e' vincoltata ai sistemi UNIX poiche' le sonde sono realizzate con degli exec di comandi. La tipologia dei dati rilevati e' essenzialmente per fini di Managment [Fink97].

Constructor Index

Monitor()

Method Index

- **getCollisionRate**()

  Calcola Collision Rate per la rete locale ("netstat").

- **getCpuLoad**()

  Risponde il carico della CPU fornito dal comando "uptime".

- **getFreeSpace**()

  Risponde lo spazio libero su disco in blocchi (comando "df").

- **getFreeSwap**()

  Risponde lo swap ancora disponibile (programma C "getswap").

- **getNetworkLatency**()

  Calcola la latenza nella rete locale ("ping").

- **verifyHostConnection**(String)

  Verifica se uno specifico Host e' attivo ("ping").

Constructors

Monitor

```
public Monitor()
```

Methods

getCpuLoad

```
public int getCpuLoad()
```

Risponde il carico della CPU fornito dal comando "uptime".

- **getFreeSpace**

```
public int getFreeSpace()
```

Risponde lo spazio libero su disco in blocchi (comando "df").

- **getFreeSwap**

```
public int getFreeSwap()
```

Risponde lo swap ancora disponibile (programma C "getswap").

- **verifyHostConnection**

```
public boolean verifyHostConnection(String Host)
```

Verifica se uno specifico Host e' attivo ("ping").

- **getCollisionRate**

```
public int getCollisionRate()
```

Calcola Collision Rate per la rete locale ("netstat").

- **getNetworkLatency**

```
public int getNetworkLatency()
```

Calcola la latenza nella rete locale ("ping").

6.1.3 La Classe Agent

```
java.lang.Object
   |
   +----AgentSystem.Agent
```

---

public abstract class **Agent**

extends Object

implements Serializable

Field Index

- **Mail**
- **Traceable**
- **Start**

Constructor Index

- **Agent**()

Method Index

- **getID**()
- **putArg**(Object)
- **run**()
- **setID**(AgentID)
- **setTraceable**(boolean)
- **setStart**(String)

Fields

Start

```
public String Start
```

Traceable

```
public boolean Traceable
```

Mail

```
public Mailbox Mail
```

Constructors

Agent

```
public Agent()
```

Methods

setTraceable

```
public void setTraceable(boolean mm)
```

setID

```
public void setID(AgentID aid)
```

getID

```
public AgentID getID()
```

putArg

```
public abstract void putArg(Object obj)
```

run

```
public abstract void run()
```

setStart

```
public void setStart(String met)
```

6.1.4 La Classe Mailbox

```
java.lang.Object
   |
   +----AgentSystem.Mailbox
```

---

public class **Mailbox**

extends Object

implements Serializable

Classe di gestione dello scambio dei messaggi. La Mailbox e' serializzabile e viene creata implicitamente alla creazione dell'agente (variabile privata Mail)

Method Index

- **getMessage**()

  Restituisce il primo messaggio in mailbox.

- **isMessage**()

  Verifica se sono presenti messaggi in Mailbox.

- **sendMessage**(Message)

  Spedisce un Messaggio.

- **storeMessage**(Message)

  Metodo implicitamente chiamato dal sistema quando arriva un messaggio

Methods

storeMessage

```
public synchronized void storeMessage(Message msg)
```

Metodo implicitamente chiamato dal sistema quando arriva un messaggio

getMessage

```
public synchronized Message getMessage()
```

Restituisce il primo messaggio in mailbox. La chiamata e' sospensiva ma esiste la possibilita' di verificare se la Maibox e' piena

sendMessage

```
public synchronized void sendMessage(Message msg)
```

Spedisce un Messaggio. La chiamata e' sincrona, il controllo viene reso al programma solo dopo che il messaggio e' stato spedito con successo. Non

e' bloccante nel senso che non aspetta che il messaggio sia correttamente recapitato.

isMessage

```
public synchronized boolean isMessage()
```

Verifica se sono presenti messaggi in Mailbox.

6.2 Il Package PlaceManager

Questo package contiene il modulo di gestione dei Place, è caratterizzato essenzialmente da una classe statica di interfaccia al Place di Default (*PlaceManager*) ed una dinamica (*Place*) che mantiene i contesti locali dei place.

6.2.1 La Classe PlaceManager

```
java.lang.Object
   |
   +----PlaceManager.PlaceManager
```

---

public class **PlaceManager**

extends Object

Classe statica di gestione del contesto di esecuzione di default. fornisce il servizio di mantenimento degli alias locali e l'astrazione di spazio delle Tuple (oppure della blackboard dipende da successive evoluzioni

---

Constructor Index

- **PlaceManager**()

Method Index

- **createNewPlace**(String)

  Crea un nuovo place con il nome specificato.

- **enterInPlace**(String)

  Ritorna il place locale con il nome specificato.

- **findAgent**(String)

  Cerca Agente mediante un suo alias.

- **init**(String)

- **isAlias**(String)

  verifica esistenza di un particolare nome (per verifica info).

- **isPlace**(String)

  Verifica se esiste il place locale specificato (serve per verifica info).

- **registerAlias**(AgentID, String)

  Registra un Agente nel Place sotto l'alias specificato.

- **removeAlias**(AgentID, String)

  Elimina un alias per l'agente corrispondente.

Constructors

PlaceManager

```
public PlaceManager()
```

Methods

init

```
public static void init(String nomesito)
```

createNewPlace

```
public static Place createNewPlace(String nome)
```

Crea un nuovo place con il nome specificato.

enterInPlace

```
public static Place enterInPlace(String nome)
```

Ritorna il place locale con il nome specificato.

isPlace

```
public static boolean isPlace(String nome)
```

Verifica se esiste il place locale specificato (serve per verifica info).

registerAlias

```
public static boolean registerAlias(AgentID agent,
                                    String nome)
```

Registra un Agente nel Place sotto l'alias specificato.

findAgent

```
public static AgentID findAgent(String nome)
```

Cerca Agente mediante un suo alias.

removeAlias

```
public static void removeAlias(AgentID aid,
                               String nome)
```

Elimina un alias per l'agente corrispondente.

isAlias

```
public static boolean isAlias(String nome)
```

verifica esistenza di un particolare nome (per verifica info).

6.2.2 La Classe Place

```
java.lang.Object
   |
   +----PlaceManager.Place
```

---

public class **Place**

extends Object

Classe di gestione del contesto di esecuzione. fornisce il servizio di mantenimento degli alias locali e l'astrazione di spazio delle Tuple (oppure della blackboard dipende da successive evoluzioni

---

Field Index

Blackboard

Constructor Index

Place(String)

Method Index

- **findAgent**(String)

  Cerca Agente mediante un suo alias

- **findService**(String)

  Cerca Agente Srvitore mediante il nome del servizio

- **getAllServices**()

  Ritorna la lista di tutti i servizi registrati.

- **getBlackboard**()

  Ritorna riferimento alla Blackboard.

- **isAlias**(String)

  Verifica esistenza di un particolare nome (per verifica info)

- **isService**(String)

  verifica esistenza di un particolare nome (per verifica info)

- **registerAlias**(AgentID, String)

  Registra un nuovo alias per l'agente AgentID.

- **registerService**(AgentID, String)

  Registra un nuovo Servizio per l'agente AgentID.

- **removeAlias**(AgentID, String)

  Elimina un alias per l'agente corrispondente

- **removeService**(AgentID, String)

  Elimina un Servizio dalla lista

Fields

Blackboard

```
public Blackboard Blackboard
```

Constructors

Place

```
public Place(String nome)
```

Methods

registerAlias

```
public boolean registerAlias(AgentID agent,
                             String nome)
```

Registra un nuovo alias per l'agente AgentID. un agente puo' essere registrato con piu' nomi ma non ci possono essere due agenti con lo stesso nome.

findAgent

```
public AgentID findAgent(String nome)
```

Cerca Agente mediante un suo alias

removeAlias

```
public void removeAlias(AgentID aid,
                        String nome)
```

Elimina un alias per l'agente corrispondente

isAlias

```
public boolean isAlias(String nome)
```

Verifica esistenza di un particolare nome (per verifica info)

- **registerService**

```
public boolean registerService(AgentID agent,
                               String nomeserv)
```

Registra un nuovo Servizio per l'agente AgentID. un agente puo' essere registrato con piu' nomi ma non ci possono essere due agenti con lo stesso nome.

- **findService**

```
public AgentID findService(String nome)
```

Cerca Agente Srvitore mediante il nome del servizio

- **getAllServices**

```
public String[] getAllServices()
```

Ritorna la lista di tutti i servizi registrati.

- **removeService**

```
public void removeService(AgentID aid,
                          String nome)
```

Elimina un Servizio dalla lista

- **isService**

```
public boolean isService(String nome)
```

verifica esistenza di un particolare nome (per verifica info)

- **getBlackboard**

```
public Blackboard getBlackboard()
```

Ritorna riferimento alla Blackboard.

6.2.3 La Classe Blackboard

```
java.lang.Object
   |
   +----PlaceManager.Blackboard
```

---

public class **Blackboard**

extends Object

Realizza la Blackboard. E' una lista di messaggi BMessage che vengono identificati da una keyword (una Stringa)

Constructor Index

- **Blackboard**()

Method Index

- **add**(BMessage)

  Aggiunge un Messaggio in lista.

- **delete**(String)

  Elimina un Messaggio dalla Blackboard (il primo con keyword data)

- **get**(String)

  Prende un messaggio caratterizzato da String keyword.

Constructors

Blackboard

```
public Blackboard()
```

Methods

- **add**

```
public boolean add(BMessage entry)
```

Aggiunge un Messaggio in lista.

- **get**

```
public BMessage get(String keyword)
```

Prende un messaggio caratterizzato da String keyword. Non c'e' garanzia di unicita' delle chiavi, questa funzione risponde la prima che trova.

- **delete**

```
public void delete(String keyword)
```

Elimina un Messaggio dalla Blackboard (il primo con keyword data)

6.3 Il Package InfoManager

Questo package incapsula le funzioni di ricerca remota delle informazioni, esiste una sola classe statica di interfaccia a tutte le funzioni.

6.3.1 La Classe InfoManager

```
java.lang.Object
   |
   +----InfoManager.InfoManager
```

---

public class **InfoManager**

extends Object

Questa classe fornisce il servizio di ricerca ed (eventualmente) analisi delle informazioni. Si distingue tra ricerche in dominio (default) e fuori dominio

tranne per quanto riguarda la ricerca di un Agente conoscendo gia l'AgentID che e' trasperente alla posizione. E' realizzato ad Agenti di che si muovono alla ricerca delle informazioni.

Constructor Index

- **InfoManager**()

Method Index

- **findAgent**(AgentID)

  Cerca l'Agente specificato spedendo un agente di servizio sul nodo di origine dell'Agente e chiedendone la posizione.

- **findAgent**(String)

  Cerca Agente attraverso il suo alias.

Constructors

InfoManager

```
public InfoManager()
```

Methods

findAgent

```
public static Location findAgent(AgentID aid)
```

Cerca l'Agente specificato spedendo un agente di servizio sul nodo di origine dell'Agente e chiedendone la posizione. Funziona solo per Agenti Searchable.

findAgent

```
public static Location findAgent(String Alias)
```

Cerca Agente attraverso il suo alias. Crea un agente di servizio che si muove in tutto il dominio alla ricerca nel PLACE di DEFAULT della registrazione e poi torna fornendo la locazione di questa registrazione. Non verifica se l'Agente e' ancora presente nel sito dove si e' registrato.

Riferimenti Bibliografici

[Ach97] A.Acharya et al.: *"Sumatra: A Language for Resource-aware Mobile Programs"*, in *Mobile Object Systems: Towards the Programmable Internet*, Lecture Note on Computer Science, vol.1222, pp.111-130, April 1997.

[Aglets96] D.B.Lange, D.T.Chang: *"IBM Aglets Workbench: a White Paper"*, IBM Corporation, September 1996, http://www.trl.ibm.co.jp/aglets.

[ArF89] Y.Artsy, R.Finkel: *"Designing a Process Migration Facility: The Charlotte Experience"*, IEEE Computer pp 47-56, September 1989.

[ATP97] D.B.Lange, Y.Aridor: *"Agent Tranfer Protocol –ATP/0.1"*, IBM Tokyo Research Laboratory http://www.trl.ibm.co.jp/aglets/atp/atp.html.

[BaGP97] 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, Springer, April 1997.

[BaKT92] H.E.Bal, M.F.Kaashoek, A.S.Tanenbaum: *"ORCA: A Language for Parallel Programming of Distributed Systems"*, IEEE Transaction on Software Engineering, vol. 18, n.3, pp.190-205, March 1992.

[Baum97] J.Baumann et al.: *"Communication Concepts for Mobile Agent Systems"*, Mobile Agents, Proc. of the 1st International Workshop, MA'97, Springer 1997.

[CaGe89] N.Carriero, D.Gelernter: *"Linda in Context"*, Comunication of ACM, vol.32, n.4, April 1989.

[CaPV96] A.Carzaniga, G.P.Picco, G.Vigna: *"Designing Distributed Applications with Mobile Code Paradigms"*, Proc. of 19th International Conference on Software Engineering ICSE'97, pp.22-32, May 1997.

[CoLZ96] A.Corradi, L.Leonardi, F.Zambonelli: *"High Level Management of Allocation in a Parallel Objects Environment"*, Tech Report n.DEIS-LIA-96-003 Università di Bologna, June 1996.

[CoSt97] A.Corradi, C.Stefanelli: *"HOLMES: a Tool for Monitoring Heterogeneous Architectures"*, IEEE HiPC'97 High Performance Computing, Bangalore, India, December 1997.

[Daup92] P.Dauphin et al.: *"ZM4/SIMPLE : a general approach to performance measurement and evaluation of distributed systems*"**,** Advances in Distribuited Computing Concept and Design, IEEE Computer Society Press, 1992.

[DoOu91] F.Douglis, J.Ousterbout: *"Transparent Process Migration: Design Alternatives and the Sprite Implementation"*, Software Practice and Experience vol. 21, n.8, pp.757-785, August 1991.

[EW94] O.Etzioni, D.Weld: *"A soft-based interface to the Internet"*, Communication of the ACM, vol.37, n.7, pp. 72-76, 1994.

[Fink97] R.A.Finkel: *"Pulsar: An Extensible Tool for Monitoring Large Unix Sites"*, Software Practice and Experience, vol. 27 n.10, pp. 1163-1176, October 1997.

[GuVS94] W.Gu, J.Vetter, K.Schwan, *"An Annotated Bibliography of Interactive Program Steering"***,** ACM SIGPlan Notices, vol.29, n.9, September 1994.

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

[Java] K.Arnold, J.Gosling: *"The Java Programming Language"*, Addison-Wesley 1996, http://java.sun.com.

[KiSc91] C.E.Kilpatrick, K.Schwan, *"ChaosMON - application-specific monitoring and display of performance information for parallel and distributed systems"*, Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging, May 1991.

[Kok96] P.Kok Keong Loh et al.: *"How Network Topology Affect Dynamic Load Balancing"*, IEEE Parallel & Distributed Technology, pp. 25-35, Fall 1996.

[LaKG92] F.Lange, R.Kroeger, M.Gerleit: *"Jewel: Design and Implementation of a Distributed Measurement System"*, IEEE Transactions on Parallel and Distributed Systems, vol.3, n.6, November 1992.

[LDD95] A.Lingnan, O.Drobnik, P.Doemel: *"An HTTP-based structure for Mobile Agents"*, Proc. of the 4th International WWW Conference, December 1995.

[LeMC87] T.J.LeBlanc J.M.Mellor-Crummey: *"Debugging Parallel Programs with Instant Replay"*, IEEE Transactions on Parallel and Distributed Systems, vol. C-36, n. 7, July 1987.

[OMG94] *Common Object Services Specification*, vol. 1, OMG Document n.94-1-1, March 1994.

[Ous94] J.Ousterhout: *"Tcl and the Tk Toolkit"*, Addison-Wesley, 1994.

[Pein97] H.Peine: *"An Introduction to Mobile Agent Programming and the ARA System"*, ZRI-Report, Dept. of Computer Science, University of Kaiserlautern, January 1997.

[PoMi83] M.L.Powell, B.P.Miller: *"Process migration in DEMOS/MP"*, Proc. of the 6th Symposium on Operating System Principles, November 1983.

[Rang97] M.Ranganathan et al.: *"Network-Aware Mobile Programs"*, Proc. of the USENIX'97 Annual Technical Conference, Anaheim, California, January 1997.

[RuGK97] D.Rus, R.Gray, D.Kotz: *"Transportable Information Agents"*, International Conference on Autonomous Agents, February 1997.

[Sno82] R.Snodgrass*: "Monitoring Distributed Systems: a Relational Approach"*, PhD Thesis, Carnegie-Mellon University, December 1982.

[Sno87] R.Snodgrass: *"The temporal query language Tquel"*, ACM Transactions on Database Systems, vol.12, n.2, June 1987.

[SteJu95] B.Steensgard, E.Jul: *"Object and Native Code Thread Mobility among Heterogeneous Computer"*, Proc. of the 1st ECOOP Workshop of Mobile Object System, August 1995.

[StrBH96] M.Straβer, J.Baumann, F.Hohl: *"Mole- A Java Based Mobile Agent System"*, Proc. of the 2nd ECOOP Workshop on Mobile Object System, University of Linz Austria, July 1996.

[Ven97] B.Venners: *"Under the Hood: The architecture of aglets"*, Java World, June 1997.