Introduzione

Grazie all'enorme evoluzione tecnologica dell'hardware, e in particolare con la nascita dei sistemi multimediali, Internet ha cambiato il suo aspetto diventando sempre più interessante per il grande pubblico e di conseguenza anche per le aziende che cercano di essere presenti mentre nuove aree di mercato stanno nascendo. Tutto ciò ha portato ad una grande espansione della rete delle reti che negli ultimi anni ha collegato tra loro un grande numero di calcolatori e utenti.

I modelli classici di progettazione di applicazioni distribuite si trovano un po' in difficoltà ad affrontare le elevate dimensioni che Internet ha raggiunto. In questo scenario hanno assunto importanza le tecnologie basate sulla mobilità di codice e sugli agenti mobili come mezzo per superare i limiti degli approcci tradizionali.

Gli agenti mobili sono oggetti attivi in grado di spostarsi sulla rete al fine di svolgere il proprio compito nella maniera più efficiente possibile. Le operazioni che richiedono ripetute sessioni di interazione con una risorsa remota possono avvenire localmente ad essa semplicemente spostandosi. In questo modo è possibile abbattere gli alti costi di latenza di una rete delle dimensioni di Internet, ed eliminare la necessità di trasmissione dei messaggi necessari all'interazione. Come risultato globale si ottiene un aumento dell'efficienza dell'applicazione ed una diminuzione del traffico di rete.

Il grande interesse di aziende e accademie per questo modello di programmazione ha portato alla nascita di una molteplicità di sistemi ad agenti mobili eterogenei. Per consentire lo sviluppo di applicazioni ad agenti su larga scala è necessario che i differenti prodotti siano in grado di interagire. Per ottenere un risultato tale è necessario rendere standard alcuni degli aspetti della tecnologia ad agenti mobili. Definendo le interfacce necessarie per l'interoperabilità tra i sistemi ad agenti, si rende possibile l'interazione lasciando la libertà di implementazione ai

progettisti, e promuovendo in questo modo anche lo sviluppo e la proliferazione di nuovi sistemi.

Nel primo capitolo viene esposto CORBA, uno standard in fase di accettazione definito nel 1989 da un consorzio di aziende, l'OMG (Object Management Group), basato sui concetti di sistema di elaborazione aperto e distribuito, e di progettazione per componenti software riutilizzabili. CORBA permette la costruzione di applicazioni distribuite in ambiente eterogeneo.

Nel secondo capitolo si introducono i concetti di mobilità di codice ed agenti mobili. Viene esaminata l'influenza che ha la sicurezza e l'eterogeneità dell'ambiente su questo paradigma, ed infine vengono esposte le possibili applicazioni e alcuni esempi di sistemi ad agenti mobili.

Nel terzo capitolo si passa ad esaminare il problema dell'interoperabilità nei sistemi ad agenti mobili e viene data una soluzione di carattere generale basata sull'utilizzo di standard. Si passano poi in rassegna i principali standard attualmente proposti nel campo degli agenti mobili, con una particolare attenzione al binomio CORBA, MASIF.

Infine il quarto e il quinto capitolo descrivono l'estensione del supporto SOMA (Secure and Open Mobile Agent) realizzato presso il dipartimento del DEIS dell'Università di Bologna, illustrandone l'uno le scelte di progetto, l'altro gli aspetti di carattere implementativo. Tale estensione, ideata al fine di rendere SOMA un sistema aperto capace di interagire con altri ambienti ad agenti, si suddivide in due parti: una prima adozione dello standard CORBA per consentire l'interazione con tutti i sistemi che già lo adottano; l'implementazione dello standard MASIF in annessione a CORBA per consentire un ulteriore grado di apertura ed in particolare la capacità di rendere disponibili meccanismi di migrazione capaci di superare le barriere poste dalla diversità dei sistemi ad agenti. Al fine di testare le capacità del supporto fornito sono state effettuate misure di prestazioni e prove di interazione con il sistema ad agenti Grasshopper

quali l’accettazione di un agente Grasshopper all’interno del sistema SOMA.

CAP 1.

LO STANDARD CORBA

Lo standard CORBA fonde il concetto di progettazione per componenti software riutilizzabili, che è alla base di tutti i modelli software Object Oriented, con la programmazione di rete in sistemi di calcolo costituiti da entità eterogenee, giungendo alla formulazione di una architettura di supporto alla computazione distribuita orientata agli oggetti. Grazie a CORBA disponiamo di uno strato software (middleware) che consente di considerare la rete come un unico bus al quale i componenti software (gli oggetti CORBA) sono collegati. Molti sono i vantaggi che otteniamo con l’utilizzo di CORBA. Mediante l’astrazione di bus si ha la trasparenza alla locazione dei componenti, i quali possono essere situati su macchine con differenti architetture hardware, differenti sistemi operativi e differenti linguaggi di programmazione. CORBA è infatti un supporto in grado di vincere i problemi dovuti alla diversità degli host della rete, e dei linguaggi di programmazione utilizzati per la scrittura dei componenti software.

Se immaginiamo, ad esempio, di scrivere una semplice applicazione per lo scambio di messaggi in formato testo tra due host di una rete di

calcolatori, ci rendiamo subito conto di quali possono essere le problematiche che si incontrano quando non si può ipotizzare l'omogeneità delle macchine, dei sistemi operativi, oppure del linguaggio utilizzato per scrivere le due parti dell'applicazione (mittente e destinatario del messaggio). In un contesto di questo tipo, lo scambio diretto di caratteri via rete può portare ad una decodifica erronea delle informazioni, dovuta alle differenze di rappresentazione dei dati nei differenti hardware, sistemi operativi e linguaggi. In pratica quello che succede è che il testo inviato dal mittente verrà ricevuto ed interpretato in modo sbagliato dal destinatario. Tutto questo sottolinea come, quando si vogliono scambiare informazioni tra sistemi eterogenei, sia necessario stabilire una codifica delle informazioni uguale per tutti, un formato esterno del quale tutti siano a conoscenza e siano in grado di tradurre verso la propria rappresentazione.

CORBA IDL (Interface Definition Language) insieme a CDR (Common Data Representation), di cui diremo ai paragrafi successivi, [CDR95] sono gli strumenti standard che CORBA definisce per vincere le barriere dovute all'eterogeneità dei sistemi. In questo modo il progettista di applicazioni distribuite è libero di scegliere il linguaggio di codifica che più ama ed è in grado di riutilizzare servizi già esistenti. CORBA IDL consente il riutilizzo di sistemi sviluppati con tecnologie non recenti, o non orientate agli oggetti, rendendoli oggetti CORBA standard, semplicemente fornendo loro di un interfaccia IDL.

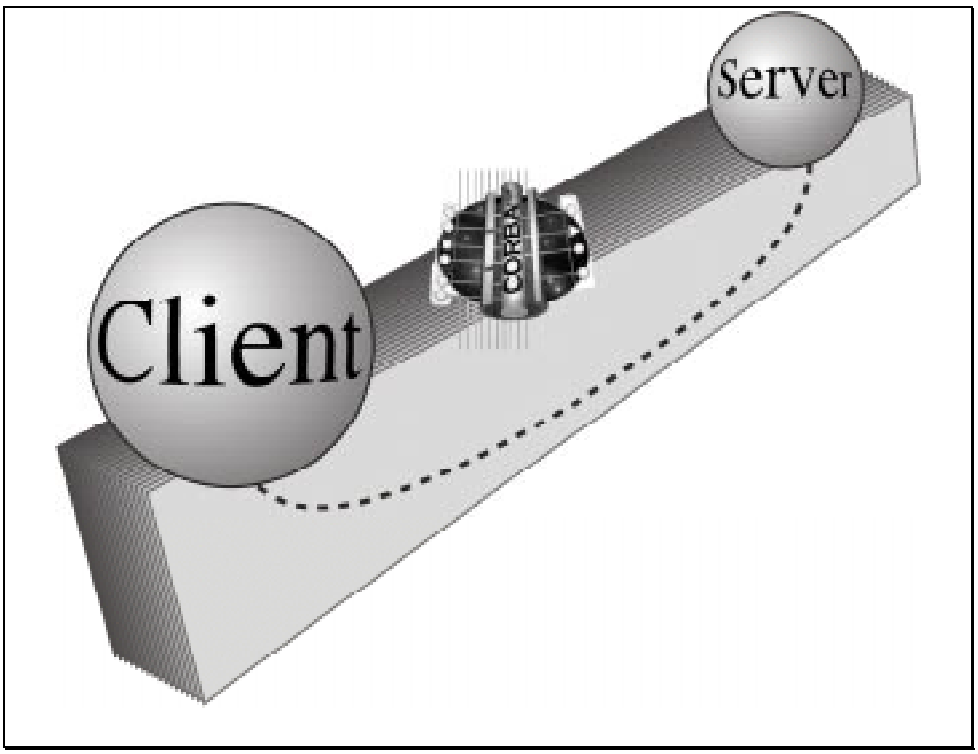

Figura 1. Invocazione client-server CORBA.

Oltre alla trasparenza e alla possibilità di lavorare in ambienti eterogenei, CORBA ci fornisce la possibilità di fare invocazioni su oggetti di cui non si conosce a priori l'interfaccia, dimostrando così anche la sua predisposizione all'apertura. Questa caratteristica è resa possibile dalle funzionalità di invocazione dinamica e di introspezione di CORBA, che è in grado di autodescriversi per mezzo del Repository di interfacce.

Prima di passare all'analisi dei componenti che costituiscono CORBA e che consentono di ottenere i risultati appena descritti, illustriamo un primo breve scenario di cosa succeda durante una invocazione client-server (figura 1). Quando un client vuole effettuare delle richieste di servizio ad un componente CORBA, la prima cosa che deve fare è ottenere un riferimento ad esso. Una volta in possesso di un riferimento ad un oggetto il client può invocare tutti i metodi che l'oggetto implementa. CORBA provvederà al ritrovamento dell'oggetto a partire dal riferimento, alla conversione dei parametri in un formato trasportabile via rete, al

trasferimento effettivo delle informazioni, all'attivazione del servizio; ed una volta ottenuti i risultati provvederà al processo di trasporto inverso.

1.1. I componenti di CORBA

CORBA è un'architettura object-oriented per il supporto alla computazione distribuita. Può essere rappresentato come un bus comune al quale sono collegati tutti gli oggetti del sistema, i quali possono diventare, a seconda delle necessità, clienti o servitori gli uni degli altri. Per usufruire dei servizi di un oggetto è sufficiente possederne un riferimento; non è necessario conoscere la sua locazione sulla rete, anzi questa è in generale trasparente, in quanto il bus provvede alla sua individuazione a partire dal riferimento.

Gli oggetti collegati al bus possono essere suddivisi in tre categorie principali.

- **Common Object Service Specification (COSS).** Le COSS [Orf98] costituiscono i servizi CORBA, ossia una collezione di servizi a livello di sistema impacchettati con le loro specifiche IDL. Tra essi troviamo servizi di naming, tempo di vita, controllo di concorrenza e sicurezza.

- **Common Facilities (CF).** Le CF [Orf98] possono essere suddivise in facilities orizzontali e verticali, orizzontali nel senso di servizi di utilità generale e verticali intese come oggetti dedicati per particolari funzioni. Le proposte facility riguardano: agenti mobili (MASIF), scambio dati (data interchange), ambienti per descrivere concetti di carattere economico (business object frameworks) e firewalls.

- **Oggetti Applicativi**. Gli oggetti applicativi costituiscono tutte le applicazioni che andranno ad aggiungersi alle altre due categorie e che ne faranno uso. Su questa categoria OMG non impone standardizzazione.

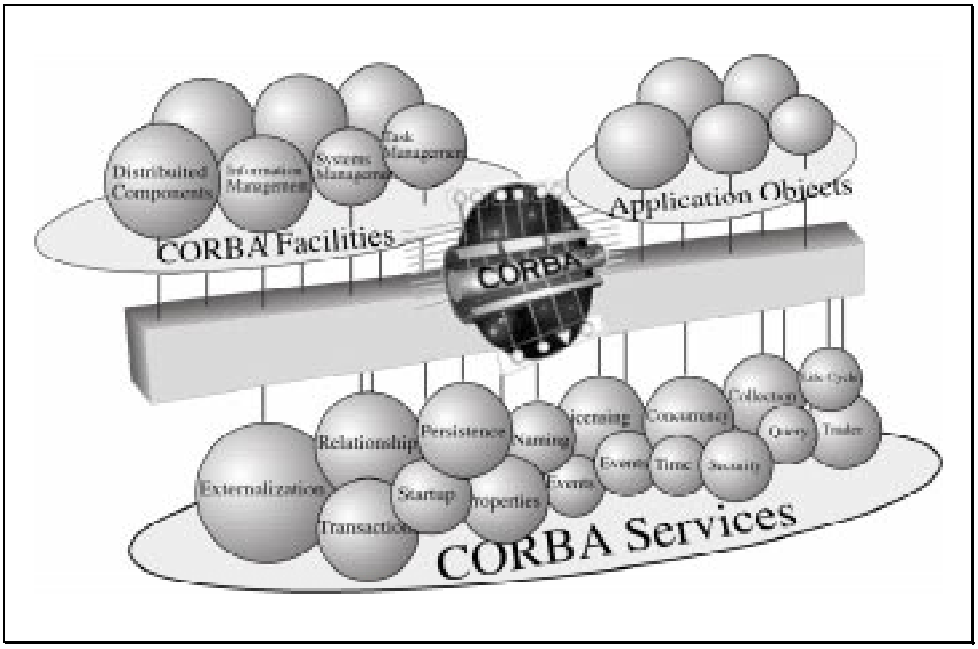

Figura 2. Il bus CORBA mette in collegamento tutti gli oggetti ad esso collegati siano essi servizi, facility o applicazioni.

Passiamo ora ad analizzare l'anatomia vera e propria di CORBA ossia la sua struttura e tutte le componenti che ne fanno parte. Gli elementi che costituiscono CORBA sono riassunti nel seguente elenco:

- CORBA IDL
- Object Request Broker (ORB) Core.
- Invocazione statica (SII): stub e skeleton.
- Invocazione dinamica: DII e DSI.
- L'adattatore degli oggetti (Object Adapter).

- Interface e Implementation Repository.

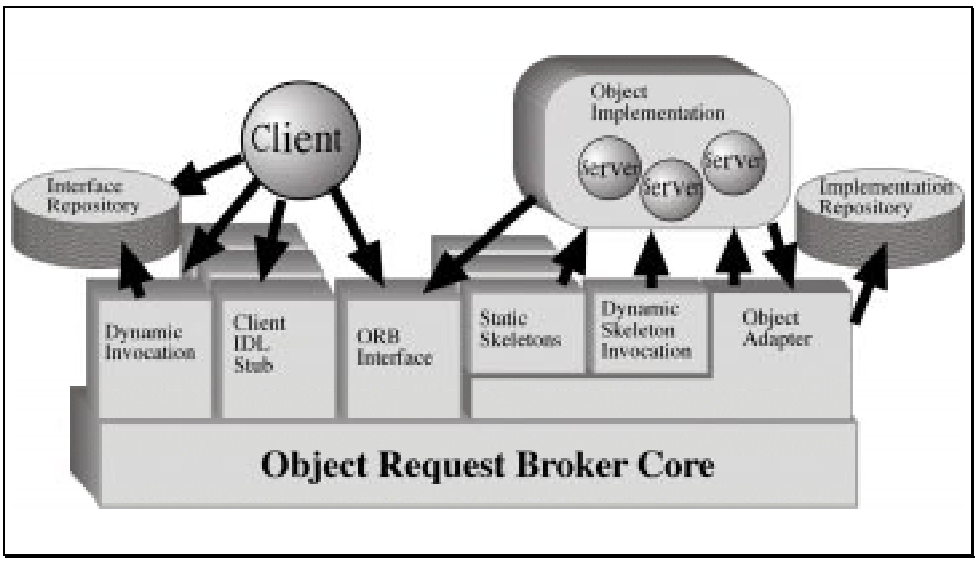

Figura 3. I componenti di CORBA.

1.1.1. CORBA IDL

CORBA IDL è un linguaggio object-oriented di tipo dichiarativo che prende la sintassi dal C++ e che consente di specificare i confini di un oggetto indicandone la sua interfaccia. La sua natura puramente dichiarativa gli consente di fornire ad ogni oggetto CORBA un'interfaccia indipendente dal sistema operativo e dal linguaggio di implementazione. In questo modo CORBA rende possibile l'interazione tra oggetti codificati con linguaggi di programmazione differenti.

```
module Counter{
interface Count{
          attribute long sum;
          long increment();
          void reset();
          void set(in long value);
        };
      };
```

L'interfaccia IDL di un contatore che ha un attributo per mantenere lo stato, e tre metodi per effettuare incremento, azzeramento e impostazione dello stato.

1.1.1.1. Legame tra CORBA IDL e linguaggio di programmazione.

Ogni qual volta si trasferiscano informazioni tra due ambienti di programmazione differenti è necessario un accordo sulla semantica data ai nomi dei tipi, ed un accordo sulle rappresentazioni concrete utilizzate. Il legame tra CORBA IDL e linguaggio di programmazione mette in relazione i tipi IDL con i tipi specifici del linguaggio. In questo modo ad ogni scambio di informazioni è possibile la traduzione del tipo utilizzato nell'ambiente di programmazione mittente verso il tipo IDL corrispondente. L'ambiente ricevente essendo in grado di effettuare la traduzione inversa potrà interpretare correttamente l'informazione ricevuta.

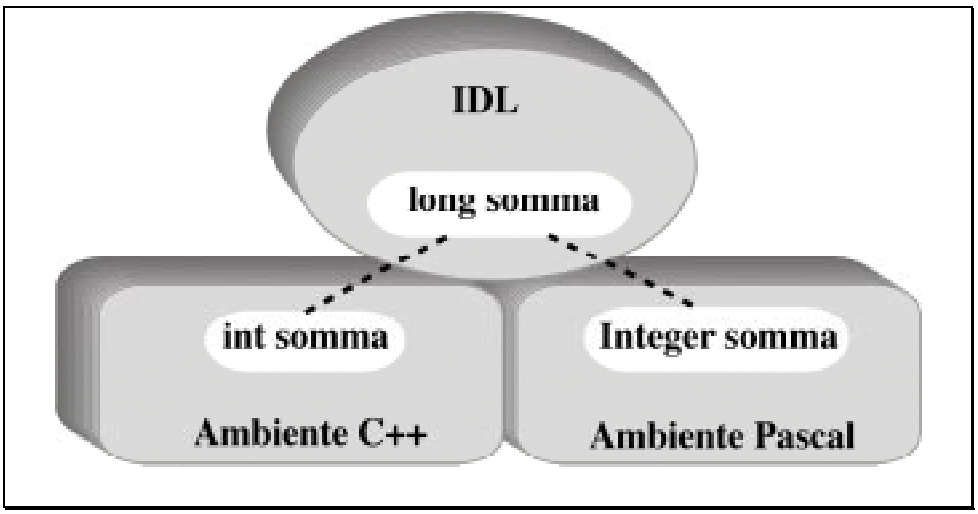

Figura 4. Mettendo in relazione un dato di tipo int C++ con un long IDL, e mettendo in relazione un tipo integer Pascal con un long IDL, è possibile trasferire l'intero somma C++ nell'ambiente Pascal. Ogni ambiente deve anche mettere a disposizione le funzioni di traduzione della rappresentazione concreta da e verso IDL. Naturalmente se non si fa la traduzione diretta da C++ a Pascal è perché quando i linguaggi di programmazione diventano più di due, sarebbe necessaria la gestione delle funzioni di traduzione da e verso tutti i linguaggi.

La traduzione da CORBA IDL ad un linguaggio specifico viene chiamata mapping e viene resa automatica da compilatori che prendendo un interfaccia IDL sono in grado di generare la corrispondente interfaccia scritta nel linguaggio prescelto. Ad esempio il compilatore *idl2java* partendo dalla definizione IDL di un oggetto, genera l'interfaccia e la struttura dell'oggetto Java comprese le *signature* dei vari metodi (la signature è l'interfaccia del metodo ed è costituita dal nome dello stesso e dall'elenco del tipo dei parametri). Ma la generazione dell'interfaccia scritta nel linguaggio scelto non è l'unica operazione che deve fare il compilatore che fa il mapping, infatti esso deve generare le funzioni che provvederanno alla conversione del formato di rappresentazione dei dati durante il trasferimento degli stessi. In pratica possiamo pensare il mapping suddiviso in due livelli, il primo nel quale si decide la

corrispondenza tra i tipi IDL e del linguaggio, e il secondo in cui si effettua la conversione della rappresentazione concreta del dato in un formato esterno (il formato esterno usato da CORBA è CDR). CDR è il formato di rappresentazione concreta dei dati che viene specificato da CORBA e che definisce per ogni tipo la modalità di codifica. Ad esempio al tipo unsigned long di IDL viene fatto corrispondere il tipo int di Java (prima fase del mapping); durante l'invocazione di un metodo di un oggetto scritto in Java che accetta un int tra i suoi parametri è necessaria la traduzione dalla rappresentazione CDR di un unsigned long alla rappresentazione Java di un int (per intenderci a livello di bit). Questa seconda fase viene detta *marshalling* (o più precisamente *unmarhalling* nel caso descritto qui) e oltre alla conversione in un formato esterno (o da esterno a interno) deve essere in grado di impacchettare e ricostruire strutture dati complesse.

Attualmente il legame tra IDL e linguaggio è stato implementato per C, C++, Ada, Smalltalk, COBOL e Java, e comunque a livello concettuale non ci sono problemi per altri linguaggi che magari devono ancora nascere. Naturalmente più il linguaggio è simile a IDL e meno saranno le problematiche del mapping. Ad esempio il concetto di eccezione può essere mappato nello stesso concetto se il linguaggio lo supporta, e in un tipo di dato astratto (magari denominato exception) se il linguaggio non ha la nozione di eccezione. In ogni caso il mapping è possibile pur se con un diverso livello di difficoltà e di aderenza al concetto.

1.1.1.2. I fondamenti di CORBA IDL

IDL consente la specifica dell'interfaccia per un oggetto CORBA tramite il costrutto *interface*. Un'interfaccia specifica la signature (la signature è l'interfaccia del metodo ed è costituita dal nome dello stesso e dall'elenco del tipo dei parametri) dei metodi dell'oggetto definendo il tipo dei parametri, il valore di ritorno e le eventuali eccezioni che il metodo può lanciare. Esiste la possibilità di definire attributi per i quali saranno automaticamente generate le funzioni get e put che consentono di ottenere

e impostare il valore dell'attributo. È consentita l'ereditarietà multipla delle interfacce.

Ogni interfaccia può essere definita all'interno di un modulo (*module*) che fornisce un contesto di nomi comuni (come può essere un package Java). Oltre alle interfacce, all'interno di un modulo possono essere definiti dei tipi di dato astratto che estendono i tipi base IDL. La differenza fondamentale tra la definizione di un interfaccia e di un tipo di dato astratto sta nel fatto che l'interfaccia rappresenta un oggetto CORBA che potrà fornire un servizio, mentre un tipo di dato è utilizzato solo come parametro o valore di ritorno di un metodo (anche perché costituito solo da una parte dati). A questo proposito vale la pena fare una distinzione tra come vengono passati durante una chiamata gli oggetti CORBA e i dati, siano essi tipi base o tipi complessi. La semantica del passaggio di oggetti CORBA nelle invocazioni è per riferimento, mentre per i dati si ha una semantica per valore. Si noti che il trasferimento di oggetti CORBA per valore comporterebbe necessariamente il trasferimento della classe dell'oggetto e quindi mobilità di codice. Quali problemi introduca la mobilità di codice in ambito di sistemi eterogenei verrà trattato in tutta la discussione, a partire dal capitolo due con l'introduzione dei problemi, nel capitolo tre per le soluzioni proposte dagli standard e nel capitolo quattro per le soluzioni adottate nel tentativo di rendere aperto un sistema ad agenti. In questo contesto si vuole semplicemente sottolineare che quando si trasferiscono degli oggetti CORBA come parametri di una invocazione si vengono a creare dei riferimenti remoti.

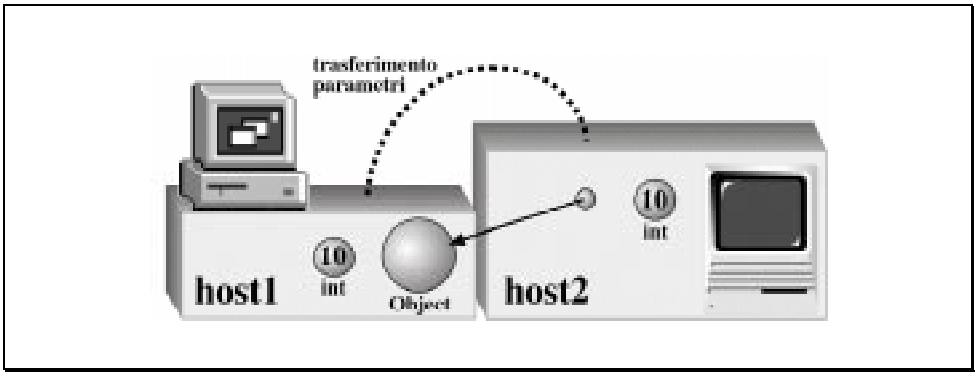

Figura 5. Il trasferimento dei due parametri copia nel sito di destinazione l'intero, mentre crea un riferimento remoto per l'oggetto CORBA.

Tra i tipi IDL merita di essere menzionato il tipo org.omg.CORBA.Any (di cui si parlerà anche nel paragrafo 1.1.4.) che è molto usato in situazioni in cui è richiesto un certo grado di dinamicità, esso infatti consente di passare qualunque tipo di informazione, anche tipi non definiti a tempo di compilazione.

Per una specifica dettagliata su sintassi e semantica del linguaggio CORBA IDL si veda [CORBA95].

1.1.2. Object Request Broker (ORB) Core

L'ORB è il motore di CORBA e costituisce il nucleo del sistema. Le funzioni principali che svolge l'ORB Core sono quelle di veicolare le invocazioni dei metodi dall'agente client verso quello server (e viceversa) e di nascondere al client l'allocazione dei processi server sulla macchina fisica.

Quando avviene l'invocazione di un metodo di un certo oggetto di cui si possiede il riferimento, l'ORB provvede ad individuare l'implementazione associata al riferimento del chiamante, preparare il servitore a ricevere la richiesta (con l'eventuale creazione o attivazione dell'oggetto), trasferire la richiesta dal cliente all'oggetto servitore e restituire la risposta al chiamante.

È evidente che l'ORB deve svolgere una funzione di name server: deve infatti ritrovare l'implementazione dell'oggetto a partire dal riferimento dato, ed in pratica deve tradurre il riferimento nella locazione fisica dell'oggetto. Come questo avvenga dipende dall'implementazione dell'ORB e non è per noi di grande interesse; è importante invece capire che cosa possa succedere se siamo in presenza di diverse implementazioni dell'ORB che interagiscono tra loro. I riferimenti vengono naturalmente generati dall'ORB ogni qual volta un nuovo oggetto si registra (come un client possa ottenere un riferimento verrà discusso nel paragrafo 1.2).

Nella prima specifica di CORBA non veniva descritto il formato che dovevano avere i riferimenti creati dall'ORB. In questo modo l'interoperabilità tra diverse implementazioni del bus CORBA non era possibile. Nella versione 2.0 di CORBA è stato specificato un formato per i riferimenti denominato Interoperable Object Reference (IOR), rendendo possibile l'utilizzo dei riferimenti indipendentemente dall'ORB che gli ha generati. Del problema dell'interoperabilità tra le differenti implementazioni di CORBA si discuterà nel paragrafo 1.3.

1.1.3. Invocazione statica (SII): stub e skeleton

L'invocazione statica presuppone la conoscenza a tempo di compilazione dell'interfaccia dell'oggetto sul quale si vuole effettuare la chiamata. In questo modo è possibile la generazione di tutte le routine di *marshalling* e *unmarshalling* dei parametri (ossia di tutte le funzioni che consentono la conversione dei dati dal formato del linguaggio a quello standard esterno e viceversa). Durante l'invocazione sarà possibile convertire le rappresentazioni interne dei parametri in un formato esterno, facile da spedire via rete, e comprensibile a prescindere dal tipo di sistema operativo o architettura hardware dei calcolatori coinvolti nell'invocazione.

Lo scopo di stub e skeleton è quello di nascondere tutte le operazioni che sono necessarie quando deve avvenire un'invocazione di un metodo su un oggetto remoto. Dal lato del client lo stub sarà un proxy (intermediario) locale che viene invocato al posto dell'oggetto vero e proprio: in questo modo il client potrà effettuare invocazioni su oggetti remoti come se fossero locali. Sarà lo stub ad eseguire le operazioni di marshalling, spedizione della richiesta all'ORB, attesa e unmarshalling del risultato, e ritorno dello stesso al chiamante. Quale sia la modalità con cui la richiesta viene spedita all'ORB dipende dall'implementazione dell'ORB stesso; è importante sottolineare che, in presenza di implementazioni diverse di CORBA, si dovrà avere a disposizione uno stub differente per ognuna di esse.

Lo skeleton è del tutto analogo allo stub, ma dalla parte del server. Il suo scopo è quello di fare l'unmarshalling dei parametri e di eseguire una up-call al metodo e all'oggetto indicati nella richiesta. Una volta eseguita l'operazione, il risultato che l'implementazione ritornerà allo skeleton, sarà convertito per essere rispedito allo stub del cliente che aveva fatto l'invocazione.

La semantica standard delle invocazioni è del tutto sincrona, il cliente attende il risultato (anche se il metodo non presuppone un valore di ritorno) prima di proseguire nella sua esecuzione. Esiste comunque la possibilità con l'utilizzo di meccanismi di più basso livello, quali l'invocazione dinamica, di gestire in maniera diversa il sincronismo. È possibile inviare richieste e non attendere che esse siano eseguite. Il controllo ritorna non appena la richiesta è stata spedita realizzando così una semantica asincrona bloccante. Con le funzioni appropriate sarà poi possibile capire se una richiesta è stata soddisfatta e ottenere il risultato.

1.1.4. Invocazione dinamica: Dynamic Invocation Interface (DII) e Dynamic Skeleton Interface (DSI)

L'invocazione dinamica di CORBA fornisce un meccanismo che dal lato client rende possibile l'effettuazione di chiamate remote su oggetti dei quali non si conosceva l'interfaccia a tempo di compilazione, e dal lato server rende possibile la costruzione dinamica di servizi la cui interfaccia può essere decisa a run-time. Quando le esigenze di programmazione non consentono di avere a disposizione le interfacce degli oggetti a tempo di compilazione, l'utilizzo di stub e skeleton statici diventa impossibile. Ecco allora che con le interfacce DII (per il lato client), e DSI (per il lato server), si dispone di tutte le funzioni necessarie per la costruzione di richieste e la gestione delle invocazioni.

Parlando di invocazione dinamica non si può evitare di far entrare nel discorso l'Interface Repository (verrà discusso dettagliatamente in seguito). Questo componente di CORBA consente di reperire l'interfaccia degli oggetti a run-time, e rende in questo modo possibile, mediante

l’utilizzo degli strumenti di invocazione dinamica, la costruzione delle richieste per le invocazioni remote. Senza conoscere i nomi delle operazioni che un oggetto mette a disposizione e la lista dei tipi dei parametri che ciascuna operazione richiede, non sarebbe di certo possibile la costruzione dinamica delle richieste.

Un problema che si incontra quando si vuole costruire un meccanismo dinamico per scambiare dati attraverso la rete senza avere definito staticamente i tipi, riguarda l’accordo tra mittente e destinatario. Bisogna accordarsi sui tipi di dato esistenti, ma anche sulla modalità di scambio. In generale l’informazione scambiata dovrà comprendere, oltre alla rappresentazione concreta delle informazioni, anche una parte in grado di descrivere il tipo di dato in questione (meta-dati o meta-informazioni). In caso contrario non sarebbe possibile per chi riceve il messaggio decodificare in maniera appropriata i bit ricevuti. CORBA usa un altro tipo di soluzione evitando così lo spreco di banda dovuto all’inserimento delle meta-informazioni. In pratica l’accordo, che nell’invocazione statica viene fatto già a tempo di compilazione, con l’invocazione dinamica viene ritardato a run-time, ma sempre in anticipo rispetto alla trasmissione e allo scambio di dati vero e proprio. Anche questo fa parte delle funzionalità offerte dall’Interface Repository. Fornendo al client le interfacce delle implementazioni, si forniscono anche quelle informazioni che gli consentiranno di costruire una richiesta che contenga una lista dei parametri ben formata. Lo stesso server potrà ottenere ancora una volta dall’Interface Repository le informazioni che gli serviranno per decodificare in maniera corretta il messaggio ricevuto. L’utilizzo dell’Interface Repository non è necessario se si presuppone un accordo implicito tra cliente e servitore. In generale il client avrà la necessità di ottenere l’interfaccia dell’implementazione, ma quest’ultima potrà implicitamente dedurre il tipo dei parametri ottenendo dalla richiesta il nome del metodo che deve essere invocato. Naturalmente essendo il server colui che soddisfa il servizio è possibile ipotizzare che abbia la conoscenza dei parametri dei metodi che mette a disposizione.

Quando si fa uso dell'invocazione dinamica le operazioni di marshalling e unmarshalling non possono più essere gestite in maniera totalmente automatica. Se con l'utilizzo dell'invocazione statica si potevano nascondere dietro stub e skeleton tutte le procedure di appiattimento delle strutture dati, ora questo non è più possibile. CORBA mette a disposizione un certo numero di funzioni che provvedono alla preparazione della lista dei parametri e che provvedono anche alla conversione di tipi in un formato esterno, ma tutto il carico di conversione di array, liste e altre strutture in un formato piatto cade sul programmatore. I parametri che CORBA richiede per l'invocazione dinamica sono una lista di elementi di tipo org.omg.CORBA.Any. Questo tipo di dato consente di inglobare valori di qualsiasi genere. Per i tipi primitivi sono a disposizione funzioni di inserimento ed estrazione specifiche, mentre per altri tipi è possibile la creazione di *stream*, ossia di una sequenza di byte (o come denomina CORBA octet), nella quale inserire tutte le informazioni necessarie. Tale stream sarà poi associato al tipo Any che si occuperà di tenere traccia del tipo contenuto.

In conclusione vale la pena evidenziare che CORBA, non inserendo informazioni sui tipi dei dati trasferiti nelle operazioni di trasmissione, ipotizza che i clienti producano richieste sempre ben formate. Nel caso di invocazione statica il problema non esiste in quanto gli stub, essendo costruiti in base alle interfacce degli oggetti, sono sicuri e producono sempre richieste contenenti i parametri nel giusto formato. Per l'invocazione dinamica questo non è garantito, in quanto il compito di costruire la lista dei parametri è lasciato al programmatore che potrebbe inserire i tipi non adeguati. Questo porterebbe ad una interpretazione sbagliata dal lato servitore, ma non necessariamente ad una rilevazione degli errori. Comunque CORBA mette a disposizione un meccanismo dal lato del cliente che attraverso il ritrovamento dell'interfaccia dell'implementazione permette un certo grado di sicurezza. Una volta che il client possiede l'interfaccia può chiedere la costruzione di una lista dei parametri aderente al metodo che deve essere invocato. Questa lista conterrà il giusto numero di elementi Any con il tipo settato

correttamente; il solo valore del parametro andrà inserito ad opera del programmatore.

Rimane a questo punto da evidenziare il fatto che implementazione dinamica e statica possono essere utilizzate insieme. Dal lato client non si ha la percezione di quale sia il meccanismo utilizzato per servire la richiesta. Dal lato server non si può dedurre se l'invocazione sia stata generata in modo dinamico, oppure in modo statico attraverso stub precompilati.

1.1.5. L'adattatore degli oggetti (Object Adapter)

L'Object Adapter è il punto di accesso per i servizi forniti dall'ORB. Attraverso questo elemento è possibile la generazione e l'interpretazione dei riferimenti, l'attivazione e la registrazione delle implementazioni e l'autenticazione delle richieste.

Quando si effettua un'invocazione è l'ORB che si preoccupa di trovare dove risiede l'implementazione, ma è l'Object Adapter che provvede all'eventuale attivazione dell'implementazione e all'invocazione del metodo richiesto che può avvenire attraverso lo skeleton (statico o dinamico) oppure in maniera diretta.

Per soddisfare differenti esigenze che possono insorgere nello sviluppo di applicazioni CORBA, non è necessario che tutti gli Object Adapter abbiano la stessa interfaccia o offrano le stesse funzionalità. Se ad esempio un database Object Oriented ha la necessità di registrare una grande quantità di oggetti all'ORB, si potrebbe pensare ad un Object Adapter in grado di consentire la registrazione di più oggetti attraverso una unica invocazione.

Il BOA, Basic Object Adapter, è l'adattatore agli oggetti che mette a disposizione le funzioni base; è presente in tutte le implementazioni di CORBA, anche se con interfacce differenti. La diversità dei vari BOA, causata dalla mancanza di una specifica standard, ha un forte impatto sulla portabilità delle implementazioni dei servizi che dipendono direttamente da esso. Per questo nella versione 3.0 di CORBA è stato data una

specifica completa per un object adapter chiamato POA [POA98], Portable Object Adapter. Con esso si vuole costituire un punto standard di accesso ai servizi necessari alla registrazione e attivazione delle implementazioni, in modo tale da assicurare la portabilità tra ORB realizzati da differenti software house.

Attraverso l'Object Adapter è possibile impostare diverse modalità di attivazione delle implementazioni (figura 6), esiste la possibilità di scelta sulla durata di vita di un oggetto che può essere registrato come persistente, ovvero disponibile anche dopo la scomparsa del processo che lo ha registrato, o non persistente in caso contrario. Si può anche specificare quale debba essere la modalità di attivazione dei thread che servono le richieste. Tra le scelte possibili troviamo:

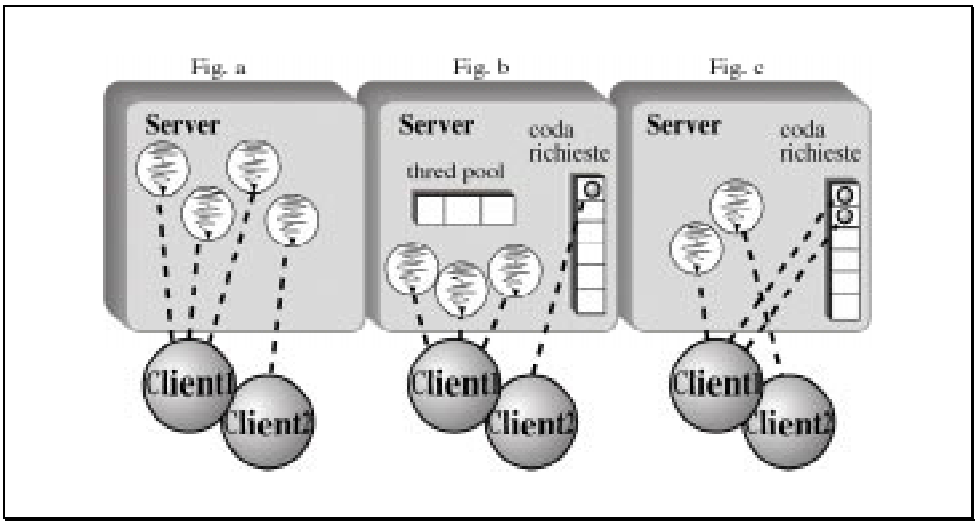

Figura 6. Il client1 fa tre richieste e il client2 fa una sola richiesta. Nel primo caso viene generato un thread diverso per ogni richiesta: in questo modo si ha un elevato grado di parallelizzazione e nessuna richiesta attende in coda, ma si paga il costo di generazione di un thread a fronte di ogni invocazione. Nel secondo caso si utilizza un pool di tre thread preattivati: avendo il client1 impegnato tutto il pool obbliga la richiesta del client2 ad attendere in coda. Si ha comunque un buon livello di parallelizzazione che

dipende dalle dimensioni del pool di thread e non si paga il costo di generazione di un nuovo flusso di esecuzione ad ogni nuova invocazione. Il terzo caso attiva un thread per ogni client (per ogni sessione di interazione): in questo modo una richiesta per cliente è in esecuzione e il tempo di attesa in coda non si ripercuote su un solo utente.

- generazione di un thread per richiesta. Impone il costo di generazione di un flusso di esecuzione per ogni invocazione. Consente un alto livello di parallelizzazione e non ha bisogno della gestione di una coda d'attesa;

- utilizzo di un pool di thread generati in anticipo. Non si paga il costo di generazione di un thread, ma limita il grado di parallelizzazione alla dimensioni del pool. In caso nessun thread sia disponibile si deve inserire in coda la richiesta oppure generare un nuovo thread;

- utilizzo di un unico flusso di esecuzione per sessione. Ad ogni sessione di interazione con un cliente si genera un thread, se più richieste provengono dallo stesso client si provvede a metterle in attesa. Il parallelismo è limitato per ogni cliente.

Per ulteriori dettagli sulle politiche di attivazione degli oggetti si veda [Orf97].

1.1.6. Interface e Implementation Repository

Lo scopo del Repository è quello di mantenere un database in cui vengono memorizzate le informazioni riguardanti gli oggetti CORBA e le loro interfacce. Le informazioni relative all'oggetto, come ad esempio la locazione, vengono memorizzate nell'Implementation Repository, mentre le informazioni relative all'interfaccia che l'oggetto implementa sono registrate nell'Interface Repository.

Quando un oggetto CORBA si registra all'ORB viene aggiornato l'Implementation Repository con le informazioni che serviranno all'ORB

per garantire la trasparenza alla locazione. Ogni qual volta si ha la necessità di attivare un oggetto o di inviargli delle informazioni si può scoprire la sua posizione consultando l'Implementation Repository.

L'Interface Repository serve come gestore delle interfacce degli oggetti. Si è già discusso nel paragrafo sull'invocazione dinamica della necessità di scoprire a run-time quali operazioni implementi un oggetto. Le modalità con cui si possono ricavare le interfacce dei servitori sono diverse. Si può partire da un riferimento all'oggetto stesso, oppure effettuare una ricerca per nome direttamente accedendo all'Interface Repository. Naturalmente l'interfaccia dell'implementazione che si ottiene è essa stessa un oggetto e da esso è possibile ricavare tutte le caratteristiche dell'implementazione quali gli attributi, i parametri, le operazioni e i risultati di quest'ultime e tutto quello che serve per descrivere in maniera completa l'oggetto.

Analizzando più in dettaglio la struttura dell'Interface Repository, si trova che essa contiene una gerarchia di classi che cerca di trasporre la struttura di ogni possibile implementazione. Se consideriamo la struttura dell'oggetto che mantiene tutte le informazioni di un'implementazione troveremo una classe per ognuna di esse. Ci sarà la classe per la descrizione dei parametri, quella per gli attributi, per le eccezioni e così per ogni elemento che possa esistere dentro un'implementazione.

Non si è ancora detto come vengano inserite le interfacce nel Repository. Mentre per le implementazioni si ha una registrazione esplicita all'ORB, che provvede anche alla memorizzazione di tutte le informazioni sulla locazione dell'oggetto nel Repository, per l'interfaccia le cose vanno diversamente. Quando si realizza un oggetto CORBA non è detto sia necessario inserire l'interfaccia nel Repository, in quanto se si fa utilizzo della sola invocazione statica non esiste la necessità di ricerche di interfacce. Se però si prevede la necessità di avere a disposizione l'interfaccia dell'oggetto nel Repository, allora sarà necessaria la registrazione. CORBA consente attraverso una serie di funzioni la costruzione e l'inserimento di interfacce nel Repository. L'utilizzo di tali funzioni non è semplice ed è giustificato solo nel caso di una gestione

altamente dinamica del sistema. In pratica risulta più semplice scrivere direttamente il codice IDL di un'interfaccia piuttosto che costruirla attraverso i metodi CORBA. Ecco allora che le varie implementazioni di CORBA mettono a disposizione tool per l'inserimento di codice IDL nel Repository. Si deve evidenziare che questi tool sono realizzati per caricare le interfacce a tempo di compilazione dato che devono avere il codice scritto in IDL, mentre le funzioni di CORBA hanno una capacità espressiva più elevata in quanto consentono la generazione delle interfacce a run-time.

1.2. CORBA Services

CORBA definisce un insieme di oggetti di servizio standard che accrescono e complementano le funzionalità di base fornite dall'ORB. Per questa trattazione risulta importante il servizio di nomi che verrà discusso nel prossimo paragrafo, è comunque interessante dare un piccolo sguardo anche agli altri oggetti di servizio.

- Il servizio di ciclo di vita denominato *Life Cycle Service*, definisce le operazioni per la creazione, la copia, lo spostamento e la cancellazione di componenti collegati al bus.

- Il servizio di persistenza, denominato *Persistence Service*, fornisce un'interfaccia unica per la registrazione di componenti persistenti su una varietà di differenti mezzi di memorizzazione come database ad oggetti, database relazionali o semplicemente file.

- Gli eventi, il cui corrispondente servizio è denominato *Event Service*, consente ai componenti sul bus di registrarsi per l'attesa di un evento. Il servizio fornisce un oggetto chiamato *event channel* che colleziona e distribuisce gli eventi a componenti che non hanno la nozione gli uni degli altri.

- Il servizio di controllo della concorrenza, *Concurrency Control Service*, fornisce un gestore di lock (semafori).

- Le transazioni con il servizio *Transaction Service*, sono basate sul protocollo two-phase commit [TwoPhase93].

- *Relationship Service* fornisce la possibilità di creare associazioni (o link) tra differenti componenti che non sono a conoscenza gli uni degli altri. Viene anche fornito il meccanismo per attraversare i link che raggruppano un certo numero di componenti.

- Il servizio di esternalizzazione, *Externalization Service*, fornisce un modo standard per fare entrare o uscire dati da un componente, attraverso un meccanismo basato sull'utilizzo di stream.

- *Query Service* fornisce un insieme di operazioni di interrogazione per gli oggetti basato sulla specifica Object Query Language (OQL) di Object Database Management Group's (ODMG).

- *Licensing Service* fornisce le operazioni necessarie per la misura dell'uso dei servizi al fine di attribuire il giusto compenso per il loro utilizzo.

- *Properties Service* consente l'attribuzione a ciascun componente di una lista di coppie (le proprietà) costituite da una parte nome a cui corrisponde una parte valore. Lo si può utilizzare ad esempio per dotare i componenti di un titolo o un di una data.

- *Time Service* fornisce all'utente la possibilità di ottenere il tempo corrente e l'errore stimato associato, in un contesto di gestione distribuita. *Timer Event Service* è un altro servizio di gestione del tempo che consente l'ordinamento degli eventi.

- Il servizio di sicurezza definito da CORBA, *Security Service* [CORBASec95], fornisce un ambiente completo per la gestione della sicurezza in ambiente distribuito. Sono previsti servizi di autenticazione, controllo di accesso alle risorse (access control list), segretezza e nonché la delegazione delle credenziali attraverso gli oggetti.

  I meccanismi di sicurezza che consentono la segretezza e l'autenticazione delle invocazioni possono mettere in contatto diverse implementazioni dell'ORB. Come succede per il passaggio dei riferimenti generati da differenti implementazioni di CORBA, anche per l'utilizzo di meccanismi di sicurezza viene definito uno standard per consentire la possibilità, a ORB differenti dal punto di vista implementativo, di interagire. Tale standard prende il nome di CSI (Common Secure Interoperability Specification) e definisce tre livelli di sicurezza (CSI level 0,1, o 2). Ad ogni livello sono associate differenti possibilità e servizi di sicurezza. Non si vuole entrare nei dettagli di quali siano le capacità di ciascun livello, ma la cosa che si vuole mettere in evidenza è che con un livello di sicurezza CSI 2 è possibile il duplice passaggio di credenziali nelle invocazioni. Questo consente, nell'ambito dei sistemi ad agenti mobili (di cui si parlerà in seguito), l'autenticazione sia del sistema da cui proviene l'invocazione che dell'agente entro il sistema.

- *Trader Service* è un servizio di pagine gialle che consente ai servitori di pubblicizzare le proprie funzionalità.

- *Collection Service* specifica come trattare e gestire collezioni di oggetti come liste, insiemi, alberi, code e stack.

1.2.1. Il servizio di nomi CORBA: CosNaming

La prima cosa che si deve fare quando si vuole invocare un servizio è ottenere un riferimento all'oggetto che lo realizza. Per fare questo si può

accedere al servizio di nomi specificato da CORBA, che prendendo in ingresso il nome di un oggetto ne restituisce il riferimento.

CosNaming come tutti i servizi CORBA si presenta come un oggetto collegato all'ORB ed è possibile ottenere un riferimento ad esso invocando un metodo direttamente sull'ORB (resolve_initial_references). Una volta ottenuto il riferimento è possibile effettuare operazioni di ricerca, registrazione e cancellazione di nomi semplicemente invocando sull'oggetto di cui si possiede il riferimento i metodi resolve, bind e unbind.

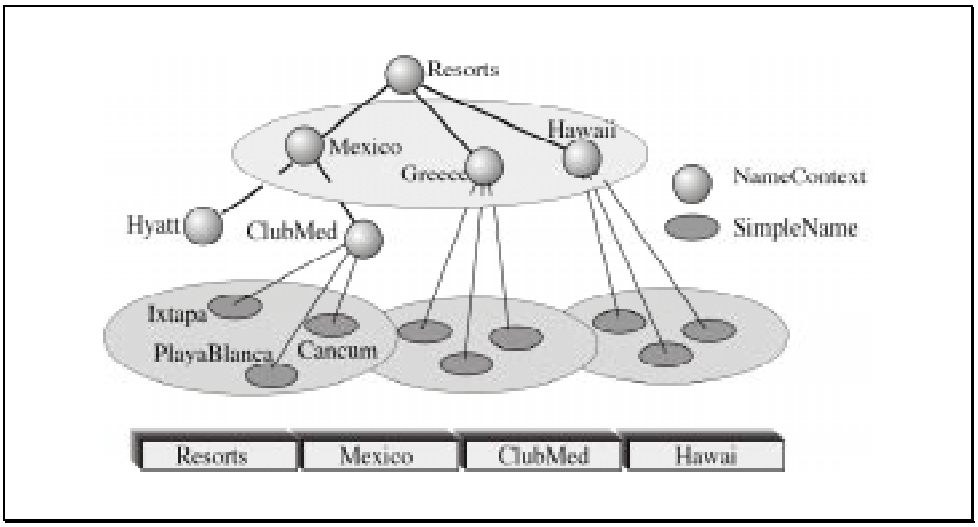

Figura 7. I nomi sono organizzati in una struttura ad albero in cui le foglie definiscono il nome semplice che è unico all'interno del contesto (ossia tra tutte le foglie che hanno lo stesso padre). Un nome completo è definito da una sequenza di contesti terminata da un nome semplice. Nell'esempio si mostra il nome Resorts, Mexico, ClubMed, PlayaBlanca.

I nomi sono organizzati in una struttura ad albero (figura 7) nella quale i nodi vengono chiamati contesti e le foglie nomi semplici. Un nome completo è costituito da una sequenza di contesti terminata da un nome semplice.

Anche se un unico servitore di nomi può mantenere nomi composti da più contesti, esiste anche la possibilità di far gestire ciascun contesto ad un server di nomi differente, in modo da ripartire il carico di lavoro. Per fare questo è necessario lanciare esplicitamente un servitore per ogni contesto. La ricerca dei nomi attraverso i vari contesti, quando siano essi gestiti da più servitori, non è automatica. Se, ad esempio, si sta cercando un nome costituito da due contesti più il nome semplice, avendo a disposizione l'oggetto che gestisce il primo contesto considerato (il contesto radice ottenuto mediante l'inizializzazione dell'ORB), si dovrà interrogare tale servitore per ottenere il secondo contesto. Di seguito si accederà al secondo contesto, fornendo il nome semplice, per ottenere il riferimento all'oggetto. Eventualmente è possibile, in fase di inizializzazione, ottenere oltre alla radice, il contesto che gestisce l'eventuale località in cui si è situati. In questo modo si può attuare una politica che eviti di ricercare nelle radice per la risoluzione dei nomi appartenenti alla località. Tutto deve comunque essere gestito manualmente.

Esiste un secondo servizio di nomi basato su Internet che è in grado di associare un nome URL a oggetti CORBA. Questo servizio denominato URLNaming non è però definito nello standard CORBA, anche se lo si può trovare in diverse realizzazioni dell'ORB. CosNaming è un servizio indipendente dal protocollo di rete, è quindi di utilizzo più generale, URLNaming si appoggia su un protocollo TCP/IP ed è quindi più adatto nello scenario di Internet.

A volte può essere necessario ritrovare un riferimento all'oggetto che implementa una certa interfaccia, in quanto si può conoscere il tipo di servizio di cui si ha bisogno, ma non il nome dell'oggetto che lo fornisce. In questo, il servizio di nomi CosNaming non aiuta, in quanto l'accesso avviene sempre per nome. CORBA, almeno nella versione 2.0, pur prevedendo di ritrovare oggetti partendo da interfacce, non specifica un servizio di questo tipo. È comunque possibile trovare nelle varie implementazioni di CORBA oggetti di servizio, che pur non essendo standard, consentano la ricerca di oggetti a partire da una interfaccia.

1.3. CORBA e l’interoperabilità

Nel paragrafo precedente si è messo in evidenza quali siano le possibili modalità per ottenere un riferimento ad un oggetto. L’ORB a fronte di una richiesta di invocazione deve essere in grado di interpretare il riferimento e risalire da esso alla locazione dell’oggetto al quale inviare la richiesta. Nella versione 1.1 di CORBA il formato dei riferimenti non venne specificato. In questo modo i riferimenti generati da una particolare implementazione dell’ORB erano interpretabili soltanto da ORB dello stesso tipo. Per superare questo limite, ed avvicinarsi sempre più alla costruzione di un unico bus “globale” (eventualmente appoggiato ad Internet), nella versione 2.0 della specifica CORBA è stata aggiunta la definizione di un protocollo standard per la generazione dei riferimenti, e in generale per tutte le esigenze di comunicazione tra ORB.

GIOP (General Inter-ORB Protocol) è il primo elemento dello standard e si preoccupa di specificare il formato di un insieme di messaggi e rappresentazioni comuni dei dati per la comunicazione tra ORB. CDR è il formato di rappresentazione concreta dei dati del quale si è già parlato, la formulazione della sua specifica avviene in questo contesto. All’interno della specifica GIOP è possibile trovare la definizione di IOR Interoperable Object Reference che determina come i riferimenti agli oggetti debbano essere strutturati. Uno IOR può essere passato attraverso differenti implementazioni dell’ORB in quanto standard.

La specifica del protocollo con cui l'ORB scambia i messaggi GIOP non é vincolata ad un particolare protocollo di rete. Nella versione 2.0 di CORBA viene specificata una mappatura su TCP/IP denominata IIOP (Internet Inter-ORB Protocol). L'aderenza a questo standard è necessaria per essere CORBA-compliant (compatibili con CORBA). In alternativa ad essa è consentito utilizzare un livello di trasporto differente; si deve però fornire un ponte che consenta la traduzione di richieste da e verso IIOP. IIOP diventa così lo standard di trasporto basato su Internet che

consente alle differenti implementazioni di CORBA di interagire, ma allo stesso tempo non si vincola lo sviluppo di ORB basati su protocolli di trasporto differenti, promuovendo la diversità e con essa la flessibilità delle soluzioni che si possono ottenere. IIOP aggiunge al normale livello di trasporto TCP/IP servizi come security service e transaction service.

Accanto GIOP e IIOP esiste la specifica di una parte standard opzionale per l'interoperabilità su reti specifiche. Con ESIOP (Environment-Specific Inter-ORB Protocol) si intende in generale la definizione di uno standard basato su uno specifico livello di trasporto. ESIOP si pone allo stesso livello di GIOP, ma a differenza di esso è dipendente dal livello di trasporto sottostante. Questo in molti casi consente una maggiore efficienza e la possibilità di sfruttare pienamente i servizi forniti dal livello di trasporto. Il primo tra gli ESIOP proposti è DCE/ESIOP, basato su DCE RPC. DCE/ESIOP fornisce servizi di sicurezza avanzati come Kerberos, servizi di distributed time, e RPC con autenticazione. Con DCE è possibile la trasmissione di una grande quantità di informazioni usando sia protocolli connection-oriented che protocolli connection-less.

Anche se definire uno standard di interoperabilità dipendente dal protocollo di trasporto potrebbe sembrare un controsenso, in realtà la struttura Inter-ORB che CORBA fornisce risulta essere ben concepita. Accanto ad un protocollo generale come GIOP, che consente l'interoperabilità in senso globale, viene sviluppato uno scenario di sottoreti che fanno uso di protocolli specifici, che sono però raggruppabili in base al tipo di trasporto supportato. Ecco allora che tutte le implementazioni di CORBA supportate da DCE saranno direttamente compatibili in quanto basate su DCE/ESIOP, mentre quelle che utilizzano differenti livelli di trasporto saranno compatibili mediante un ponte di collegamento verso IIOP.

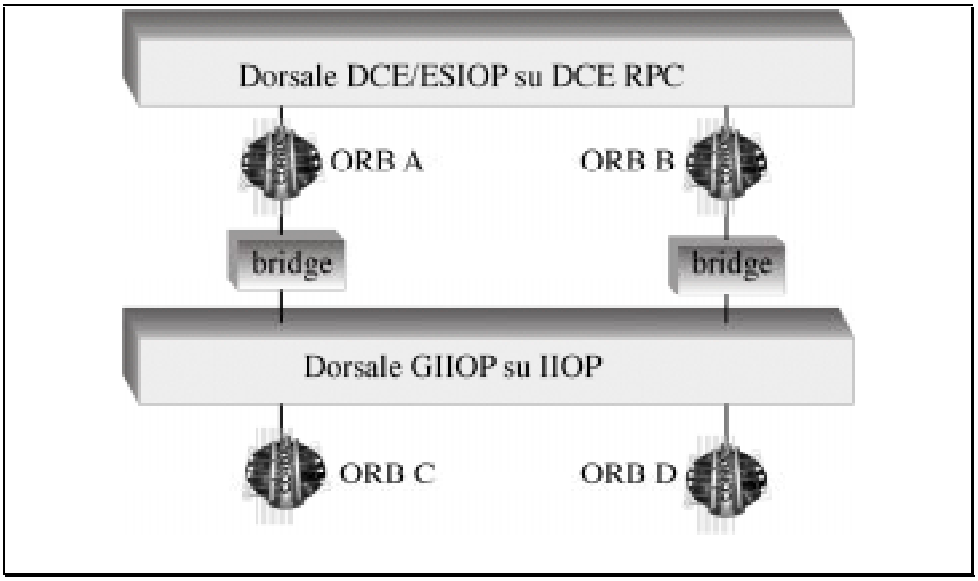

Figura 8. Architettura Inter-ORB. Gli ORB C e D possono comunicare direttamente tra loro grazie a GIOP e IIOP come anche gli ORB A e B mediante DCE/ESIOP supportato da DCE RPC. L'interoperabilità tra A e C può avvenire attraverso il ponte che ogni implementazione CORBA-compliant che non utilizza direttamente IIOP deve mettere a disposizione.

CAP 2.

MOBILITÀ DI CODICE E AGENTI MOBILI

Seppur non esista una definizione universalmente riconosciuta di cosa sia la *mobilità di codice*, possiamo inizialmente definirla come la capacità di trasferire risorse aventi capacità computazionali, da un sito all'altro di una rete di calcolatori. Le risorse che, oltre ad essere in grado di effettuare computazioni, hanno la capacità di muoversi, vengono in generale chiamate *agenti mobili*.

Visto il grande interesse in ambito accademico ed aziendale per mobilità di codice e agenti mobili, questo paradigma di programmazione distribuita potrebbe apparire come una novità. In realtà la possibilità di trasferire codice via rete è nata già da qualche anno. Nella specifica OSI a livello di presentazione [PilaOSI91] viene definito il linguaggio astratto ASN1 (abstract sintax notation) che ha lo scopo di consentire la descrizione dei dati che vengono trasferiti in rete. ASN1 prevede oltre alla possibilità di scambiare dati (anche non previsti inizialmente), lo scambio di codice.

Probabilmente però solo adesso le tecnologie sono mature per mettere in atto la *code mobility* [FugPV98] ed è forse per questo che accademie ed aziende si stanno interessando all'utilizzo di questi modelli. Anche la grande diffusione della macchina virtuale Java (la Java Virtual Machine) ha incentivato l'implementazione dei paradigmi di progettazione basati sulla mobilità di codice e sugli agenti mobili. La JVM, essendo realizzata ormai per diversi sistemi operativi, vince tutte le barriere di eterogeneità degli ambienti di computazione, consentendo la grande diffusione delle applicazioni a codice mobile che possono quindi essere realizzate per reti WAN di grandi dimensioni quali Internet.

Sin dai primi esperimenti, la mobilità di codice ha mostrato nuove capacità in grado di risolvere alcuni problemi e di superare alcuni limiti del tradizionale approccio ai sistemi distribuiti.

- Se il nodo su cui si risiede ha un carico computazionale elevato, mentre altri nodi limitrofi sono scarsamente utilizzati, la mobilità di codice consente di scegliere di proseguire l'esecuzione nell'ambiente meno carico, raggiungendo in minor tempo la conclusione del processo.

- Se è necessario interagire con una risorsa remota, è possibile spostare il frammento di codice interessato sul nodo in cui risiede la risorsa, per aumentare le prestazioni attraverso il beneficio di una interazione locale, meno costosa di quella remota.

- Spostare l'esecuzione sulla macchina in cui risiede la risorsa da utilizzare, permette di ridurre anche il traffico di rete, evitando interazioni remote che sono la causa delle congestioni della rete.

- Sfruttando la mobilità del codice, lo svolgimento di operazioni critiche non è più minacciato da tratti di connessioni inaffidabili: è sufficiente spostare l'intera esecuzione nelle porzioni di rete in cui i collegamenti

sono più stabili ed eventualmente restituire il risultato dell'operazione svolta in remoto.

- Non solo i collegamenti, ma anche una macchina può non fornire caratteristiche di stabilità: la mobilità consente di spostare dinamicamente l'esecuzione sul nodo che offre caratteristiche migliori in termini di stabilità.

- Grazie alla code mobility può nascere il concetto di *mobile computing*: i nodi della rete non sono più vincolati ad essere situati in una locazione fisica fissa, e gli utenti possono muoversi insieme alle loro macchine su differenti posizioni, rimanendo connessi alla rete attraverso collegamenti senza fili (*wireless*).

Dopo avere meglio chiarito cosa si intende per mobilità e quali siano i problemi di sicurezza introdotti da essa, verranno esposti i modelli di programmazione basati su questo concetto soffermandosi in particolare sul paradigma Mobile Agent (MA) e sulle sue possibili applicazioni. In fine saranno descritti alcuni sistemi ad agenti mobili.

2.1. Mobilità di codice: un modello astratto

Scopo di questo paragrafo è dare un modello [CugGPV97], seppure informale, delle astrazioni che è possibile ritrovare in tutti i linguaggi che consentono mobilità.

Nei convenzionali linguaggi sequenziali, un programma è costituito in generale da un segmento di codice che fornisce la descrizione statica del comportamento del programma stesso, e da uno stato che indica il punto di esecuzione raggiunto. Lo stato può essere a sua volta suddiviso in stato di esecuzione e area dati. Lo stato di esecuzione contiene informazioni di controllo quali il'istruction pointer (ossia il puntatore alla prossima istruzione da eseguire), e il punto di ritorno per tutte le funzioni invocate; l'area dati costituisce l'insieme di tutte le risorse accessibili dalle funzioni

del programma e raccoglie le informazioni iniziali e i risultati parziali della computazione avvenuta fino ad un certo istante.

Il segmento di codice e lo stato di esecuzione costituiscono tutte le informazioni di controllo che consentono al programma di procedere nella sua esecuzione ed insieme definiscono di una unità di esecuzione *EU* (*Execution Unit*). Una EU costituisce un concetto più generale degli agenti di cui si parlerà in seguito, ed è in grado di modellare, oltre ad essi, anche entità di computazione presenti in modelli basati sulla *code mobility*, differenti da quello ad agenti mobili. L'area dati del programma viene usata per definire le *risorse*. Le risorse sono entità passive che rappresentano dati, come ad esempio file o aree di memoria centrale come lo Heap.

Per consentire l'analisi delle caratteristiche dei linguaggi a codice mobile bisogna introdurre il nuovo concetto di ambiente di computazione *CE* (*Computational Environment*). Una CE è un contenitore di componenti. Si tratta di una astrazione che non deve essere necessariamente collegata ad un host, nel quale in generale si possono trovare più di una CE. I componenti all'interno di un ambiente di computazione possono essere EU oppure risorse. I linguaggi convenzionali non hanno bisogno dell'introduzione delle CE in quanto ogni EU è legata al suo ambiente di esecuzione per tutta la sua vita. I linguaggi che consentono la mobilità devono invece consentire alla proprie entità di esecuzione di cambiare CE durante la loro vita. È possibile distinguere i *Mobile Code Language* (MCL) in base agli elementi di una EU che è consentito spostare tra ambienti di esecuzione:

- **Strong Mobility** (mobilità forte) quando è consentito lo spostamento del segmento di codice insieme allo stato di esecuzione;

- **Weak Mobility** (mobilità debole) quando è possibile spostare solamente il segmento di codice.

Quando una EU cambia ambiente di computazione le risorse a cui faceva riferimento devono essere in un qualche modo gestite. Sono possibili due differenti strategie:

- **replicazione**: questa strategia che prevede di ridirigere il riferimento alle risorse del CE di partenza verso una copia nel CE di arrivo. Può essere ulteriormente suddivisa in replicazione *statica* o *dinamica*. La replicazione statica prevede di avere risorse replicate in tutti gli ambienti computazionali. Risulta applicabile per le librerie e le variabili di sistema in quanto risultano essere in generale risorse non modificabili. La replicazione dinamica prevede di fare una copia delle risorse a tempo di migrazione. È possibile decidere se lasciare anche l'originale (*dynamic replication by copy*) oppure eliminarlo (*dynamic replication by move*);

- **condivisione**: il vecchio riferimento viene sostituito da uno nuovo di tipo inter-CE. La risorsa indicata rimane la stessa, in questo modo i riferimenti possono indicare anche risorse remote.

I linguaggi a mobilità di codice possono utilizzare anche differenti strategie di gestione delle risorse. La replicazione statica può essere usata per risorse senza stato; quella dinamica con copia per avere disponibilità in entrambi gli ambienti di computazione coinvolti nella migrazione; quella dinamica con spostamento può essere usata quando EU differenti non condividono gli stessi dati, in modo da evitare riferimenti non validi (dangling reference); infine la condivisione di risorse è utile nel caso si vogliano avere dati condivisibili tra differenti CE.

2.2. I problemi di sicurezza in presenza di mobilità

Se da un lato la mobilità di codice mette a disposizione strumenti molto flessibili, in grado di superare molti limiti degli approcci classici alla progettazione di applicazioni distribuite, dall'altro, consentire lo

spostamento di entità capaci di effettuare computazioni, introduce nuovi problemi legati alla sicurezza. Quando un CE riceve una nuova EU deve potersi difendere dalle azioni che l'entità di esecuzione vuole effettuare. Non solo, anche alle EU deve essere garantita la protezione dagli eventuali attacchi di CE malintenzionati. Questo tipo di problemi non erano di certo presenti nel modello client/server, e nascono solo quando siamo in presenza di mobilità di codice.

Naturalmente i problemi di sicurezza tipici di tutti i sistemi distribuiti, quali la sicurezza dei canali di comunicazione che collega i vari ambienti di computazione, devono essere gestiti anche nell'ambito di sistemi ad agenti mobili. Le EU e le risorse, devono potersi muovere tra vari CE lungo strade sicure, in grado di garantire segretezza e integrità del codice e dei dati. Questi risultati si possono ottenere con tecniche classiche di crittografia ed autenticazione.

In aggiunta bisogna gestire i problemi di sicurezza specifici dei linguaggi a codice mobile. Per presentare le nuove esigenze di sicurezza facciamo riferimento al modello presentato nel paragrafo precedente. I meccanismi di protezione devono agire sui confini [JanMD97] tra i vari componenti del sistema. Controllando i confini di una entità è possibile proteggerla dagli attacchi provenienti dall'esterno. Considerando le risorse come facenti parte dei CE è possibile individuare i seguenti confini:

- **protezione di CE da EU malintenzionati**: le EU in ingresso in un sistema devono essere autenticate per stabilirne la provenienza e conseguentemente il grado di fiducia che può essere loro associato, la classe di risorse cui hanno accesso, a chi deve essere addebitato l'uso delle risorse locali [Tenti98] (*autorizzazione*); l'accesso da parte delle EU alle risorse locali deve essere controllato per proteggere le risorse stesse (*access control*);

- **protezione di EU da CE malintenzionati**: anche le EU devono essere protette dall'ambiente in cui eseguono. Poche soluzioni sono state

proposte a questo proposito in quanto proteggere una entità dall'ambiente in cui esegue è un problema complesso. Per l'ambiente di computazione è infatti sempre possibile effettuare operazioni non lecite (come ad esempio la clonazione ripetuta del flusso di esecuzione) sulla EU che ospita. Forse si dovrà fare uso di supporti hardware garantiti dei quali ci si può fidare. In questo modo un qualunque programmatore male intenzionato non potrà di certo modificare il supporto di esecuzione (il CE) delle EU;

- **protezione tra EU**: le entità che eseguono sullo stesso CE devono essere isolati fra loro per impedire qualunque tipo di attacco reciproco.

In conclusione oltre alla normale sicurezza che bisogna garantire durante il trasporto via rete di una qualunque entità, è necessaria l'introduzione di meccanismi per la protezione, questa volta locale, dei componenti che è possibile trovare all'interno di un ambiente di computazione.

2.3. Eterogeneità dell'hardware e mobilità di codice

Le forti differenze delle architetture hardware esistenti (si pensi alle differenze tra sistemi Unix, Windows e MAC) rendono impossibile la portabilità delle applicazioni a livello di codice binario. Questo problema è sempre esistito, ma si trasforma con la mobilità di codice in un problema che limita la capacità di interazione tra i sistemi. L'impossibilità di spostare il codice binario di un'applicazione tra sistemi con architetture hardware differenti può essere affrontata fondamentalmente in due modi:

- **definizione di una macchina virtuale** in grado di interpretare un codice binario standard. Definire una macchina virtuale significa specificare un codice binario ed un interprete in grado di eseguirlo. Implementando l'interprete per le differenti architetture e utilizzando compilatori in grado di generare il codice binario specificato dalla

macchina virtuale, è possibile spostare il codice delle applicazioni sulle differenti architetture hardware;

- **spostamento del codice sorgente**. In questo caso è necessaria distinguere tra linguaggi compilati ed interpretati. Un codice interpretato può supportare più facilmente la mobilità su un ambiente eterogeneo, perché è sufficiente fornire un interprete per ogni tipo di piattaforma su cui il programma dovrà eseguire. Nel caso invece di codice compilato, la macchina che riceve deve essere in grado, una volta ottenuto il codice sorgente, di avviare un processo di compilazione per produrre il codice binario.

Si noti che l'utilizzo di una macchina virtuale oppure l'utilizzo di un linguaggio interpretato implica sempre l'esistenza di un interprete disponibile su ogni host. La differenza sostanziale sta nel fatto che la macchina virtuale, definendo un formato binario, fornisce la possibilità a un qualunque tipo di linguaggio di essere eseguito (dopo una fase di compilazione), mentre l'interprete che lavora a livello di codice sorgente consente l'esecuzione ad applicazioni scritte in un solo linguaggio.

L'utilizzo di linguaggi compilati è in generale più efficiente rispetto all'utilizzo di interpreti di qualunque tipo (sia a livello di codice sorgente che di macchina virtuale), presenta però lo svantaggio di una più difficile gestione dei trasferimenti di codice in quanto richiede la ricompilazione sull'host di arrivo dove deve essere presente un compilatore per ogni linguaggio esistente. I linguaggi interpreti sono in generale più lenti e presentano il problema della necessità di un interprete per ogni linguaggio. L'utilizzo di una macchina virtuale sembra essere un buon compromesso tra efficienza, potabilità e aperture in quanto l'utilizzo di una sola macchina virtuale non solo consente l'esecuzione di un qualsiasi linguaggio, ma permette di ottenere livelli di prestazioni migliori rispetto all'utilizzo di interpreti a livello di codice sorgente.

I problemi di eterogeneità fino ad ora riportati riguardano fondamentalmente le differenze dell'hardware. La mobilità di codice

risente anche dei problemi legati all'eterogeneità dei CE. Le differenti realizzazioni degli ambienti di computazione limitano le possibilità di movimento ed interazione delle EU. Di questo si parlerà nel capitolo tre in cui si tratterà il problema in maniera estesa.

2.4. I Modelli di programmazione basati sulla mobilità di codice

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]:

- **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 client ha la possibilità di fornire il codice necessario per ottenere il servizio ad un generico servitore. In pratica il cliente fornisce la logica e il servitore la capacità di computazione. Per questo tipo di modello è sufficiente che il linguaggio supporti mobilità di tipo *Weak.*

- **Code on Demand (COD):** il client può chiedere il codice ad un servitore per completare un dato compito. 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 risorsa remota si muove localmente ad essa per utilizzarla. Questo modello è estremamente flessibile ed è in realtà la visibilità e l'uso diretto del meccanismo di migrazione da parte del programmatore.

Confrontando il classico modello CS con gli altri tre dotati di mobilità si nota che REV e COD sono in realtà un'evoluzione di CS verso una maggiore dinamicità. Esistono attualmente varie applicazioni basate su questi modelli: gli *Applet* di Java ad esempio rispecchiano un modello di tipo COD (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.

MA invece è qualcosa di completamente nuovo. La realizzazione di una applicazione, utilizzando il modello MA, risulta completamente svincolata da ogni problematica di programmazione di rete; indipendentemente dalla sua struttura, il sistema viene visto dall'agente come un insieme di contesti (i CE) 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 con le più svariate tecniche di comunicazione.

2.5. I sistemi ad Agenti Mobili

La terminologia adottata normalmente nei sistemi ad agenti mobili si differenzia da quella usata nel modello astratto presentato nel paragrafo 2.1. Le EU hanno il loro corrispondente negli Agenti Mobili, mentre i CE vengono di solito chiamati place. Quest'ultimo termine è però in parte ricoperto e non adottato in tutti i sistemi con lo stesso significato. A volte si intende effettivamente per place l'ambiente di computazione degli agenti; in altri casi il termine è utilizzato per indicare un raggruppamento di servizi localizzati all'interno di un CE. In alcuni sistemi con place si intendono ambedue le cose; in altri si differenzia il nome dell'ambiente di esecuzione da quello usato per i gruppi di servizi (Grasshopper utilizza il termine agency come sinonimo di CE e place come luogo in cui ritrovare servizi con caratteristiche comuni); in altri ancora l'astrazione di CE non appare (come in SOMA, si veda paragrafo 4.2.3.). In seguito utilizzeremo il termine place con il significato di ambiente di esecuzione (CE) e, nei

casi in cui sarà necessario trattare i gruppi di servizi, i termini verranno dovutamente specificati.

Le caratteristiche concettuali del modello MA sono state esposte in precedenza. Ora si vogliono descrivere gli aspetti di carattere secondario che arricchiscono i sistemi ad Agenti Mobili, rendendoli più articolati ed interessanti.

Una prima importante caratteristica dei sistemi ad agenti è la *non* trasparenza alla locazione. Un agente si trova immerso in un mondo costituito da place, conosce la sua posizione e può decidere di spostarsi scegliendo il place che preferisce (entro i limiti che la sicurezza impone). Per consentire sviluppo di applicazioni su larga scala è importante non limitare il numero di place che si possono aggiungere al sistema nel suo complesso. Ottenere un risultato di questo tipo è possibile solo se le cose vengono strutturate in maniera da rendere scalabile il sistema, quindi senza incorrere in cali di efficienza dovuti alla presenza di un numero elevato di place. Il fine di allargare le applicazioni ad un contesto globale porta a rendere standard gli elementi che consentono lo scambio di agenti e le interfacce di interazione tra agenti e risorse (di questo si parlerà in tutto il capitolo tre). Solo in questo modo è possibile far interagire differenti implementazioni di sistemi ad agenti e costruire un sistema realmente aperto.

Gli agenti nascono in un place, creati da un utente, che assumerà il ruolo di principal (o authority; è il responsabile al quale possono essere, ad esempio, attribuiti i costi dei servizi utilizzati dall'agente) a vantaggio del quale gli agenti producono i loro risultati. In generale non è detto che un agente debba essere rintracciabile (gli agenti rintracciabili o *traceable* sono in generale quelli per cui è possibile la gestione remota, o con i quali è possibile mettersi in contatto anche quando sono distanti) dal suo principal o da altri agenti. In pratica molti sistemi ad agenti mettono a disposizione servizi che consentono di ritrovare gli agenti. Sarà poi il programmatore a scegliere se il suo agente dovrà essere rintracciabile oppure no. I servizi che consentono il ritrovamento di un agente possono basarsi su diversi modi di agire [MASIF97]:

- **ricerca esaustiva**: si cerca in tutti i place l'agente. Si tratta di un modo di procedere che si può usare solo in ambito di località dato il suo alto costo di ricerca;

- **tecnica di logging**: ogni volta che l'agent migra viene lasciato un riferimento verso il place in cui si è spostato. Si deve conoscere quindi il place dove l'agente è nato, partendo da li' si seguono le tracce fino al ritrovamento dell'agente. Quando l'agente muore bisogna utilizzare una tecnica di garbage per eliminare tutti i riferimenti lasciati.

- **registrazione ad ogni migrazione**: si fa utilizzo di un servitore di nomi. Un agente ad ogni migrazione dice al servitore quale è il place in cui è arrivato, quando qualcuno ha bisogno di ritrovare un agente interroga il name server. Questo tipo si soluzione contribuisce ad aumentare il costo di ogni migrazione;

- **autopubblicazione**: un agente si registra al name server solamente quando vuole poter essere trovato. Per trovare un agente che non si è mai pubblicizzato è necessario un algoritmo di ricerca esaustivo.

In un sistema ad agenti, è necessario definire il modo in cui gli agenti possono interagire tra loro per raggiungere il loro obiettivo. Non è necessario avere particolari strumenti studiati appositamente per il modello ad agenti, in quanto qualunque meccanismo di comunicazione può essere utilizzato dagli agenti per scambiare informazioni tra loro e con le risorse. Scambio di messaggi, memoria condivisa, invocazioni remote di qualunque tipo (RMI Java [RMI98], o anche tramite l'uso di CORBA [CORBA95]) e spazi delle tuple [CaGe89] sono tutti modelli che possono andare bene per la comunicazione tra Agenti. L'unica considerazione che può essere fatta riguarda una caratteristica chiave che gli agenti hanno: la mobilità. Evidentemente è bene evitare l'utilizzo eccessivo di un qualunque mezzo di comunicazione di carattere remoto e

privilegiare lo scambio locale di informazioni tra agenti o anche tra agenti e servizi. Se ad esempi un agente ha bisogno di richiedere una molteplicità di servizi ad una risorsa (o ad un altro agente), è bene che migri nel place in cui la risorsa risiede e utilizzi invocazioni locali piuttosto che rimanere fermo ed utilizzare invocazioni remote.

In un contesto di questo tipo è possibile suddividere i meccanismi di comunicazione di carattere remoto e di carattere locale. Tutti i meccanismi che vanno bene per comunicare tra agenti situati in place differenti possono essere utilizzati anche per la comunicazione intra-place. Se però si ha a disposizione l'ipotesi di località dei partecipanti alla comunicazione, è possibile cercare di ottimizzare le scambio di informazioni. Ecco allora che l'utilizzo di memoria condivisa, o in un contesto ad oggetti, l'utilizzo di shared object, diventa un modo efficiente e di facile implementazione con cui gli agenti possono scambiarsi dati quando si trovano all'interno dello stesso place.

2.6. Applicazioni nel modello ad Agenti

Il modello di programmazione ad agenti mobili è un modello di computazione completo e quindi utilizzabile per la realizzazione di un qualsiasi tipo di software. I suoi vantaggi emergono maggiormente nei campi applicativi in cui i costi di comunicazione di rete sono particolarmente critici. I modelli che consentono mobilità di codice fanno spesso uso di linguaggi portabili, di solito interpretati, che sono spesso meno efficienti di linguaggi compilati appositamente per il tipo di architettura su cui devono eseguire. Ecco allora che per le applicazioni in cui il bisogno di computazione è inferiore a quello di comunicazione, il modello ad agenti mobili mostra tutte le sue potenzialità.

2.6.1. Network Management

Il controllo e l'amministrazione di reti aperte eterogenee e di grandi dimensioni necessita lo sviluppo di strumenti che permettano di eseguire in remoto le procedure di normale manutenzione dei nodi. I sistemi esistenti e gli standard proposti [BaGP97] per il Network Management, 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.

2.6.2. Information Retrieval

Forse la ricerca ed il recupero di informazioni distribuite è il campo in cui il modello ad agenti mostra i suoi vantaggi in maniera più eclatante. Questo tipo di applicazione richiede in generale l'esplorazione di un grande numero di siti e il controllo di grandi quantità di informazioni. Piuttosto che trasferire grandi moli di dati per poi eliminarne una grande percentuale, forse è più conveniente decidere già, localmente alla base di dati quali siano quelli veramente interessanti. Strutturando l'applicazione ad Agenti, non solo si riduce enormemente la quantità di informazioni trasferite via rete, ma è possibile ottimizzare e specializzare la ricerca oltre gli strumenti normalmente forniti dal server.

Quando si utilizzano connessioni non permanenti è anche possibile ridurre i costi di esercizio effettuando una ricerca *off-line*. Una volta spedito l'agente alla ricerca delle informazioni ci si può scollegare dalla

rete. Al successivo collegamento si potrà accogliere l'agente che riporterà i risultati della sua ricerca.

2.6.3. Mobile Computing

Lo sviluppo delle tecnologie senza fili, liberano i nodi del sistema dal vincolo di rimanere collocati in una locazione fissa, stabilita senza dinamicità al momento dell'attivazione, e consentono l'introduzione di un nuovo concetto: il *mobile computing*. In questa nuova situazione, l'utente mobile può decidere di spostarsi, affiancato dalla sua macchina (si pensi per esempio ai computer portatili) in locazioni fisiche differenti, in cui è ancora capace di connettersi alla rete, attraverso collegamenti senza fili. In questo contesto gli agenti mobili consentono un modo efficiente di interagire con la rete. I tempi in cui si rimane scollegati dalla rete diventano ugualmente momenti di interazione grazie all'indipendenza degli agenti dall'utente che gli ha creati.[FugPV98].

2.6.4. 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.

2.6.5. Specializzazione di protocolli e costruzione dinamica di servitori

Gli agenti mobili consentono ai servitori di utilizzare protocolli specifici per la comunicazione con il cliente. Naturalmente è necessario che cliente e servitore siano in grado di scambiare tra loro agenti. Una volta che l'agente ha raggiunto il sito server, potrà iniziare una sessione di comunicazione con il client che risiede nella sua *home*, utilizzando il protocollo specifico che il client ha inserito nella logica dell'agente. Ecco allora costruito dinamicamente un nuovo server pronto per soddisfare tutte le esigenze del client.

In questo contesto si colloca la tecnologia *active network* che si occupa della costruzione dinamica di protocolli di routing. In generale quando un messaggio viene spedito via rete, vengono inserite al suo interno le informazioni che serviranno ai router per farlo giungere a destinazione. La tecnologia active network prevede l'inserimento di codice come informazioni utili per recapitare il messaggio. I router che ricevono il pacchetto, eseguendo il codice specificato, sceglieranno la strada più appropriata per far proseguire il messaggio verso la destinazione.

2.6.6. 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, da parte di più utenti contemporaneamente in azione sugli stessi lavori, è essenzialmente legata ad un problema di sincronizzazione per il mantenimento della consistenza dei dati condivisi. Un coordinamento gestito ad Agenti 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 più 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.

2.7. Alcune implementazioni di sistemi ad agenti mobili

In questo paragrafo saranno illustrati alcuni sistemi ad agenti per fornire uno scenario delle attuali implementazioni del paradigma MA. In alcuni casi si accennerà al problema dell’interoperabilità e a come i vari sistemi lo hanno affrontato. Per una definizione ed una trattazione completa di interoperabilità si rimanda al capitolo tre.

2.7.1. SOMA

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

2.7.1.1. L’architettura di SOMA

SOMA offre un facile modo di rispondere alle richieste di diverse configurazioni di sistemi distribuiti aperti, non fidati e globali, a partire da

semplici architetture LAN ad architetture composte da molte LAN interconnesse mediante bridge, router, gateway e firewall.

SOMA fornisce una gerarchia di astrazioni di località adatta per descrivere ogni tipo di scenario di connessione: ogni nodo ha almeno un place per l'esecuzione di agenti; diversi place sono raggruppati nell'astrazione di un dominio; ogni dominio introduce un default place per comportarsi omogeneamente con i place che racchiude; i domini possono essere interconnessi usando gateway.

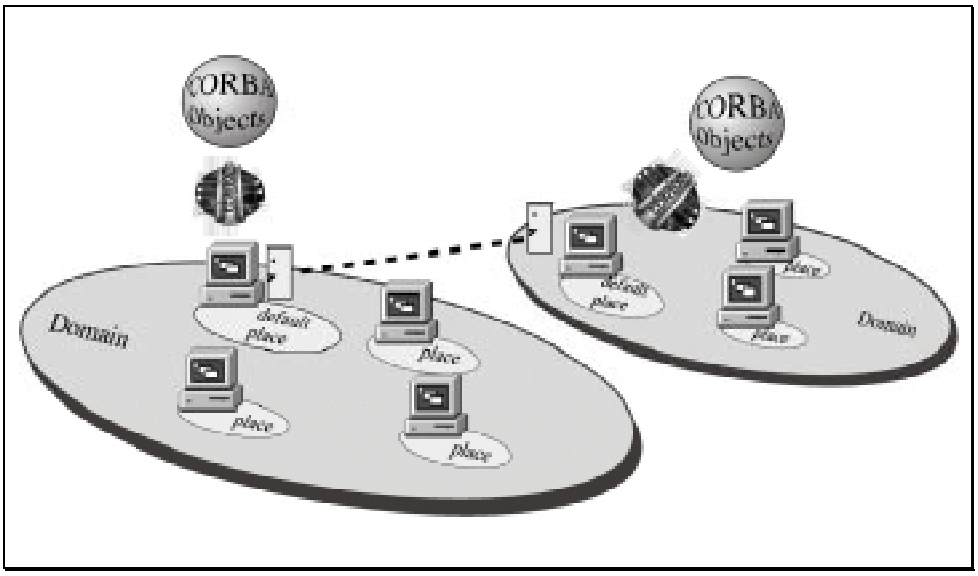

Figura 9. SOMA fornisce una gerarchia di astrazioni di località basata su place e domini, e consente attraverso CORBA l'interazione con il mondo esterno.

Le astrazioni di località di SOMA hanno un diretto mappaggio sulle risorse fisiche: gli agenti eseguono in place che rappresentano nodi fisici (anche se più place possono essere attivati sullo stesso host, consentendo l'esistenza di distinte località per la protezione di risorse ed il management); i place sono raggruppati in domini che modellano LAN. La migrazione fra domini avviene passando attraverso i loro place di default.

La definizione di differenti astrazioni di località ha anche un impatto positivo sulla politica di sicurezza: aiuta nel progetto di località protette, i domini di responsabilità di SOMA, che rappresentano lo scopo di controllo dei principal per i quali gli agenti agiscono. La politica di sicurezza può essere suddivisa e le azioni possono essere controllate sia a livello di place che a livello di dominio.

A riguardo della comunicazione e coordinazione, gli agenti all'interno di un place possono interagire condividendo risorse comuni. Ogni qualvolta un agente ha bisogno di condividere una risorsa con un altro agente che risiede in un place remoto, viene forzato a migrare in tale place. Fuori dallo scope di un place gli agenti possono interagire solo tramite scambio di messaggi. I messaggi sono eventualmente consegnati agli agenti anche in caso di migrazione.

Il place è l'ambiente di esecuzione degli agenti, è composto da diversi moduli:

- l'Agent System offre le funzionalità di base per la mobilità degli agenti e fornisce lo stile di comunicazione message-passing per gli agenti che risiedono in differenti place e anche differenti domini;

- il Manager delle Risorse Locali (Place Manager) è l'interfaccia per le risorse di nodo alle quali gli agenti possono accedere. Agenti nello stesso place interagiscono per mezzo di oggetti condivisi come blackboards o spazi di tuple. Questo modulo controlla le autorizzazioni degli agenti e assicura la politica di sicurezza del place;

- il Servizio di Informazioni Distribuito è responsabile del reperimento di informazioni sugli agenti e i place remoti. Esso fornisce le funzionalità di un DNS e di un Directory Service. Questo modulo è un servizio implementato da un set di agenti dedicati; si ha l'intenzione di realizzare le stessa funzionalità in termini di agenti che interagiscono con tradizionali Directory Services e Internet DNS;

2.7.1.2. Sicurezza in SOMA

SOMA considera tutte le esigenze di sicurezza che emergono nel contesto di reali applicazioni, basate su agenti, che eseguono su Internet: SOMA fornisce protezione sia per gli host che per gli agenti, e assicura integrità e segretezza per la migrazione di agenti e per la comunicazione.

La definizione di differenti astrazioni di località consente di assicurare politiche di sicurezza nelle quali le azioni sono controllate sia a livello di place che a livello di dominio. Il dominio definisce una politica di sicurezza globale che impone autorizzazioni e proibizioni generali; ogni place può solo applicare restrizioni ai permessi consentiti a livello di dominio.

Anche autenticazione ed autorizzazione sono controllate a livello di place e di dominio i quali possono accettare o meno agenti dipendentemente dalla politica impostata: gli agenti entranti in un dominio sono autenticati sulla base di una serie di informazioni protette (il nome del place e del dominio di origine, i nomi delle classi che implementano gli agenti, e il nome degli utenti a vantaggio dei quali gli agenti lavorano).

Una volta autenticati, gli agenti sono autorizzati ad interagire con le risorse del place in accordo alla politica di sicurezza; ogni risorsa ha una sua specifica access control list per tutti i principal che potenzialmente possono accedere (es. Unix proprietario, gruppo, altri). Inoltre i place sono protetti grazie al fatto che gli agenti possono accedere alle risorse di un nodo solo per mezzo di interfacce.

La sicurezza può essere raggiunta al solo danno delle prestazioni, SOMA fornisce però un ampio spettro di meccanismi di sicurezza (ben distinti dalle politiche) che consentono la scelta di un adatto trade-off tra la sicurezza richiesta e le prestazioni desiderate, a seconda delle circostanze: ad esempio, i parametri di connessione SSL [FKK96] possono essere negoziati tra due endpoint al fine di non avere nessuna/corta/lunga chiave di protezione, in accordo con la natura dell’applicazione e al livello di fiducia che essi hanno l’uno nei confronti

dell'altro; e anche agenti dall'interno di domini fidati possono accedere direttamente al controllo di autorizzazione, mentre agenti da domini non fidati devono oltrepassare tutti i passi di segretezza, integrità, autenticazione ed autorizzazione.

La grande flessibilità della gestione della sicurezza in SOMA è consentita grazie alla chiara distinzione tra meccanismi e politiche. Le politiche di sicurezza devono poter essere liberamente scelte in base alle esigenze che emergono nello sviluppo a livello applicativo. Per poter consentire questo grado di flessibilità è necessario che i meccanismi non siano sviluppati in relazione ad una politica prescelta, ma siano liberi da preconcetti e da scelte effettuate erroneamente in anticipo.

2.7.1.3. Interoperabilità in SOMA

SOMA affronta il problema dell'interoperabilità (si veda capitolo tre) adottando CORBA e MASIF come standard. Lo scenario SOMA è costituito da agenti che utilizzano funzioni proprietarie per l'interazione con altre entità SOMA e funzioni standard CORBA per l'interazione con il mondo esterno. L'interoperabilità fornita da SOMA può essere suddivisa in tre classi fondamentali:

- agenti SOMA che si registrano a CORBA per rendere disponibili servizi;

- agenti SOMA che diventano clienti CORBA per usufruire di servizi;

- agenti SOMA che utilizzano MASIF per interagire con altri sistemi ad agenti.

In questo contesto è possibile ritrovare le caratteristiche analizzate a proposito di CORBA e MASIF. CORBA offre il mezzo di comunicazione standard con tutto ciò che è CORBA-oriented, e MASIF consente ad

agent system differenti di scambiare tra loro agenti (CORBA-based o meno).

Le caratteristiche più specifiche di SOMA a riguardo dell'interoperabilità verranno prese in considerazione nel prossimo capitolo nel quale verrà discussa l'estensione di SOMA mediante l'implementazione di MASIF.

2.7.1.4. Comunicazione

La comunicazione tra agenti avviene mediante l'utilizzo di tre modalità: condivisione di risorse, message passing, e comunicazione anonima.

- *Condivisione di risorse*: questo tipo di comunicazione prevede un'interazione stretta tra gli agenti partecipanti, imponendo ad essi di essere presenti contemporaneamente sullo stesso ambiente, ma non impone limiti sul numero di partecipanti.

- *Scambio di messaggi*: questo tipo di comunicazione prevede un'interazione lasca tra gli agenti partecipanti. Ogni agente possiede una *mailbox* che può utilizzare per ricevere e spedire messaggi. La semantica di comunicazione a default è asincrona bloccante. In ricezione esiste la possibilità di realizzare una semantica non bloccante grazie ad un meccanismo che consente di verificare se nella mailbox sono presenti messaggi.

- *Comunicazione anonima*: la blackboard. Gli agenti possono depositare un messaggio nella blackboard e ad esso viene associata una stringa: la conoscenza di questa stringa consente di leggere il messaggio, senza nessuna conoscenza di che lo ha inserito.

2.7.1.5. Mobilità

SOMA, essendo basato sulla macchina virtuale Java (JVM) [Java98] standard, non consente un meccanismo di strong mobility, ma cerca di avvicinarsi ad esso mediante la suddivisone del codice da eseguire in diversi metodi. Quando un agente migra oltre a specificare la destinazione, deve dire quale sarà il suo metodo di partenza una volta giunto nel nuovo place. In questo modo è possibile in un certo qual modo tenere traccia del punto di esecuzione a cui si era giunti e non dover ogni volta ricominciare da zero.

Il metodo che un agente deve invocare per migrare è denominato go e accetta come primo parametro il nome del metodo di cui sopra, e come secondo la destinazione. La destinazione può essere il nome di un place se si vuole migrare senza uscire dal dominio e il nome del dominio nel caso contrario. Quando un agente cambia dominio viene posto nel place di default, da qui potrà poi muoversi a suo piacimento.

2.7.2. Grasshopper

Grasshopper [GRAS98] prodotto da IKV++ GmbH è un sistema ad agenti che pur non consentendo un meccanismo di mobilità di tipo Strong, cerca di avvicinarsi ad esso mediante la partizione del codice da eseguire su place differenti, un meccanismo del tutto analogo a quello adottato da SOMA.

Grasshopper distingue tra *agenti stazionari*, che non hanno la capacità di migrare e sono associati ad una locazione, ed *agenti mobili*. Entrambi devono la loro capacità di esecuzione alle *agency*, ossia al supporto di sistema che deve essere presente in ogni host in almeno una unità. Ogni agency contiene sempre un *place* di default chiamato “InformationDesk”, ma possono essere creati più place, ciascuno allo scopo di contenere un gruppo logico di servizi. Potrebbe esistere un place che fornisce funzionalità sofisticate di comunicazione ed un place per il reperimento di informazioni. Le agency sono raggruppate in *region* all’interno delle quali è disponibile un servizio chiamato *region registry* che consente la ricerca di agency, place e agenti all’interno della regione stessa.

Gli agenti Grasshopper possono essere creati in modo locale o remoto, e alla nascita viene associato loro un identificatore unico immutabile. Il ciclo di vita di un agente è suddiviso in tre stati: sospeso, attivo e disattivo (cioè sospeso e memorizzato in un database persistente).

Grasshopper fornisce anche un servizio di persistenza (ossia di memorizzazione su file system) per agenti e place. Viene distinta la persistenza implicita da quella esplicita. La persistenza implicita viene gestita dalle agency che quando vengono disattivate sono in grado di salvare tutti i place e gli agenti ospitati. Alla riattivazione della agency vengono ripristinati tutti i place e gli agenti in maniera trasparente. La persistenza esplicita può essere utilizzata in modo diretto dagli agenti e dai creatori degli agenti. Esistono due differenti possibilità:

- Persistenza *flush*. Gli agenti vengono rimossi dal sistema e le risorse da loro utilizzate rilasciate. Su richiesta è possibile la riattivazione.

- Persistenza *save*. Gli agenti vengono salvati nel database, ma loro esecuzione continua. In caso di guasto si può così ripartire dal punto in cui l'agente era stato salvato.

La comunicazione inter-agente è basata sull'invocazione remota di metodi. Per comunicare con un agente è necessario possedere il suo identificatore (chiamato GUID) e se si fa utilizzo di un meccanismo di invocazione statica è necessario anche la classe proxy (ossia lo stub che deve essere generato in fase di compilazione dallo stub-generator di Grasshopper), altrimenti è possibile fare utilizzo di una interfaccia di invocazione dinamica (meccanismo del tutto analogo a CORBA).

Un proxy utilizzato per la comunicazione con un agente rimane valido anche in caso di migrazione dell'agente o del proxy stesso. Se l'agente riferito da un proxy si è spostato il proxy viene aggiornato alla nuova posizione accedendo al region registry.

La sicurezza è basata sui meccanismi forniti da Java e protegge le agency da accessi di agenti non autorizzati. Ogni agente contiene

l'informazione immutabile di quale sia l'utente che lo ha creato. In base a questa informazione vengono definite le politiche di accesso alle risorse. La sicurezza della comunicazione e delle autenticazione è garantita con l'utilizzo di SSL [FKK96] e X.509 [KauPS95].

In quanto all'interoperabilità Grasshopper è il primo sistema che è MASIF-compliant e ha l'intenzione di aderire anche allo standard FIPA (si veda capitolo tre).

2.7.3. Aglets

Aglets Workbench**,** un prodotto di IBM [Aglets96], è un ambiente scritto interamente in Java per Agenti Java e 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 *message digest*.

2.7.4. Voyager

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

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

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

2.7.5. April

April (Agent PRocess Interaction Language) è un linguaggio di programmazione orientato ai processi e progettato per l'implementazione di applicazioni di rete intelligenti. Sviluppato all'Imperial College è attualmente mantenuto dai laboratori Fujitsu. April è scritto per sistemi UNIX in C e utilizza un protocollo TCP/IP per il trasporto di agenti e messaggi.

Il compilatore April, il servitore per la comunicazione e l'ambiente runtime di April costituiscono il sistema. Su ogni host deve essere presente un comunication server che gestisce i messaggi entranti

recapitandoli agli agenti locali. L'ambiente runtime è una macchina virtuale in grado di eseguire agenti compilati con April compiler.

Un agente è un processo dotato di una coda di messaggi e di uno stato privato. Tipicamente un processo April fa uso di operatori pattern matching per prelevare le richieste dalla coda dei messaggi. Gli agenti vengono creati dalla linea di comando e si registrano alla loro nascita al comunication server in modo tale che esso possa recapitare loro i messaggi. Un agente può poi creare lui stesso altri agenti mediante l'utilizzo di un comando di *fork*. Non esistono concetti di place o regioni, ogni processo April è un possibile supporto runtime per l'esecuzione di un agente.

April tratta funzioni e procedure come oggetti di prima classe e realizza la mobilità con la spedizione di questi elementi a processi remoti. Esistono due alternative di mobilità*: for methods e for agents*. Nel primo caso le chiusure di codice trasferite vengono ricevute da un agente che le ingloba rendendole sue; nel secondo caso il processo ricevente esegue un comando di fork per la creazione di un nuovo agente che sarà costituito da funzioni e procedure spedite. Un agente che si vuole trasferire deve quindi spedire la chiusura di codice costituita dall'intero processo e realizzare una migrazione for agents. La sua esecuzione riprenderà poi da un entry point che avrà specificato, in quanto April non realizza un meccanismo di strong mobility. Il processo che ha inviato la sua chiusura per effettuare la migrazione potrà poi decidere di continuare la sua esecuzione, uscire oppure diventare un proxy per recapitare i messaggi al nuovo agente.

La comunicazione avviene con scambio di messaggi i quali possono contenere tipi di dato primitivi, strutture complesse e agenti ed è del tutto asincrona. I messaggi sono trasparenti alla locazione e possono essere memorizzati dal comunication server nel caso l'agente non sia presente al loro arrivo. Saranno poi eventualmente scaricati dall'agente al momento opportuno. Per spedire messaggi ad un agente bisogna avere un *handle* che corrisponde al suo identificatore (costituito dall'IP della macchina in cui è stato creato, più un identificatore unico localmente che a default è automaticamente generato, ma che può essere anche specificato). Non

essendo presente l'esistenza di un servizio di nomi, l'unico modo per ottenere un handle e mediante lo scambio di messaggi e la generazione di agenti figli tramite fork. In alcuni casi si può supporre la conoscenza pregressa di un handle considerato pubblico il quale potrebbe essere il mezzo per comunicare con un agente che fornisce un servizio di nomi (in questi casi è utile specificare la parte locale dell'identificatore per ottenere un nome user-friendly). La trasparenza alla rete nello scambio di messaggi è ottenuta mediante l'utilizzo di agenti proxy (risultato di migrazioni nelle quali il processo che ha iniziato l'operazione non termina). April garantisce la paternità dei messaggi.

Linee comuni e contrastanti tra i sistemi ad agenti esistenti

Nei sistemi ad agenti presentati, e comunque in generale nella maggior parte delle implementazioni del paradigma ad agenti mobili che sono state sviluppate, si riscontra la tendenza all'utilizzo di Java come linguaggio di codifica. Se da un lato con Java non è possibile il trasferimento dello stato di esecuzione, e quindi non si può attuare un meccanismo di strong mobility, dall'altro l'utilizzo di una macchina virtuale consente di superare molti problemi imposti dalla mobilità di codice in ambiente eterogeneo.

I meccanismi di comunicazione adottati sono abbastanza differenti. Esiste sempre la possibilità da parte degli agenti di fare utilizzo di strumenti di comunicazione remota, anche se le caratteristiche del paradigma ad agenti mobili conducono verso l'utilizzo di meccanismi di comunicazione locali. In alcuni sistemi (come SOMA ed APRIL) si tende a fare uso di un modello di interazione lasca e si utilizza lo scambio di messaggi. In altri sistemi (come Grasshopper e Voyager) si è orientati all'utilizzo di meccanismi più di alto livello come l'invocazione remota di metodi.

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, il controllo di accesso alle risorse e la segretezza della comunicazione.

Pur essendo Grasshopper l'unico sistema commerciale che sia già conforme allo standard MASIF, anche gli altri sistemi sono intenzionati a diventarlo il più velocemente possibile (in seguito si discuterà proprio del'estensione di SOMA per ottenere la compatibilità con MASIF).

CAP 3.

AGENTI MOBILI E INTEROPERABILITÀ

Il grande interesse mostrato negli ultimi tempi da accademie e aziende per gli agenti mobili, ha portato alla nascita di un considerevole numero di differenti sistemi ad agenti. Pur potendo ricondurre ciascuno di essi al modello MA presentato nel capitolo due, ogni implementazione ha dato una diversa interpretazione di sistema ad agenti mobili. La diversità delle caratteristiche individuali di ciascuna implementazione rende difficile l'interazione tra gli agenti appartenenti a sistemi differenti. L'aspetto dell'interazione, o interoperabilità, tra differenti implementazioni del paradigma MA, diventa di importanza fondamentale se consideriamo lo scenario di rete costituito da Internet. Data l'importanza che la grande rete ha assunto negli ultimi anni, è d'obbligo considerare quale sia il rapporto tra essa e il modello MA. Alcune delle caratteristiche degli agenti mobili rendono vantaggioso il loro utilizzo nell'ambito di Internet. Ecco ad esempio che la possibilità di limitare lo spreco di banda diventa importante in una rete globale che deve fornire servizi a milioni di utenti. Per poter sviluppare una tecnologia ad agenti mobili in un ambito così

esteso, è però necessario rendere possibile l'interazione tra le differenti implementazioni di sistemi ad agenti. Non è accettabile imporre ad un numero elevato di utenti con differenti necessità l'utilizzo di un solo tipo di sistema. Per potere costruire applicazioni ad agenti che si estendano a tutta la rete, è quindi necessario lasciare la libertà di scelta del tipo di sistema da utilizzare, e rendere possibile l'interazione tra le diverse implementazioni. Il solo modo per promuovere sia l'interoperabilità, che la diversità dei sistemi, è standardizzare alcuni aspetti di questa tecnologia.

3.1. Barriere all'interoperabilità

L'interoperabilità è la capacità che diverse entità hanno di interagire e lavorare insieme per svolgere determinati compiti o funzioni. Fornire interoperabilità in un contesto costituito da agenti mobili vuole certamente dire rendere possibile la cooperazione tra gli agenti, ma vuole anche dire fornire la possibilità ad un agente di spostarsi in sistemi differenti l'uno dall'altro. Ecco allora che diventa necessaria la capacità dell'agente di interagire con il sistema sottostante anche quando quest'ultimo abbia caratteristiche di volta in volta differenti. Le entità che devono essere capaci di interazione non sono quindi solamente gli agenti tra loro, ma in generale tutte quelle presenti nel paradigma MA.

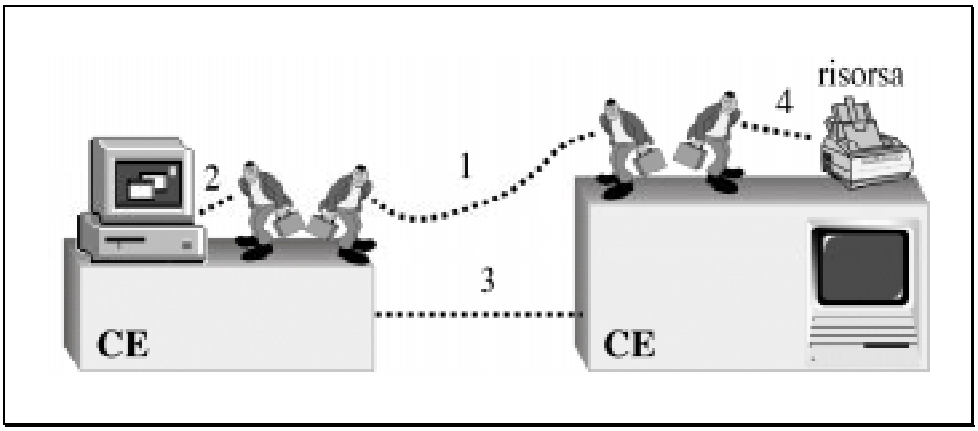

Figura 10. I confini tra le entità che costituiscono i sistemi ad agenti mobili interoperabilità e le barriere da abbattere per consentire un buon livello d'interazione.

Facendo riferimento al modello presentato nel capitolo due, possiamo dire che per ottenere l'interoperabilità nei sistemi ad agenti mobili, dobbiamo fare in modo che le EU (ossia gli agenti), le CE (spesso chiamate place) e le risorse di differenti sistemi, siano in generale capaci di interagire. Per ottenere questo risultato sarà quindi necessario individuare i confini che separano ciascuna di queste entità, ed intervenire per eliminare le eventuali barriere che limitano l'interazione. La cosa che in generale limita l'interoperabilità tra entità, è la diversità. I confini non sono necessariamente delle barrire all'interoperabilità, lo diventano quando la diversità delle entità contrapposte è tale da impedire qualsiasi scambio di informazioni, o in generale la cooperazione. I confini individuabili in ambito di sistemi ad agenti mobili, e le relative barriere che possono insorgere sono così classificabili:

- **confine tra EU ed EU**. Questo confine può diventare una barriera all'interazione quando agenti di diverso tipo non siano in grado di scambiare informazioni. Si tratta in pratica di un problema di comunicazione in ambiente eterogeneo. Non è necessario che gli agenti abbiano una particolare struttura interna, l'unica cosa richiesta per evitare il formarsi di limiti all'interazione su questo confine è la capacità di comunicazione tra agenti indipendentemente dal sistema di appartenenza. Si deve gestire quindi l'eventuale diversità dei protocolli utilizzati dagli agenti per comunicare;

- **confine tra EU e CE**. Questo confine è molto importante, ed è spesso la prima causa che produce limiti di interazione nei sistemi ad agenti. La migrazione (punto chiave del paradigma ad agenti mobili) in un contesto di agenti eterogenei è possibile solo se un supporto (CE) è in grado di fornire risorse di computazione a qualunque tipo di agente (EU). Ogni CE è in generale in grado di gestire un certo tipo di agenti

con caratteristiche strutturali ben precise, soprattutto a riguardo del linguaggio di codifica usato per scrivere l'agente stesso, e più difficilmente è in grado di essere una risorsa di computazione indipendente dal linguaggio. Anche l'interfaccia ai servizi proposti agli agenti dal CE assume un ruolo importante. Per sfruttare le capacità proposte dal CE (tra cui le stesse funzioni di migrazione), gli agenti giunti nel nuovo sistema, devono sapere come accedervi;

- **confine tra CE e CE**. Si tratta ancora una volta di un problema di comunicazione. In molti casi diversi supporti hanno la necessità di scambiare informazioni. Le stesse migrazioni comportano un passaggio tra due CE dello stato dell'agente e comportano quindi un esigenza di comunicazione. Tutte le operazioni di autenticazione sono legate a questo confine. Un sistema che vuole autenticare un agente si deve rivolgere al CE da cui l'agente è arrivato e deve quindi comunicare con esso;

- **confine tra EU e risorse**. Il termine risorsa è molto generale e comprende spesso uno svariato insieme di servizi con caratteristiche anche molto differenti tra loro. Proprio per questo a volte risulta difficile ottenere in modo generale la capacita di interazione tra agenti e risorse. La diversità insita nelle risorse risulta spesso difficile da gestire e a volte ci si può accontentare di ottenere l'interazione in modo specifico caso per caso. In questo contesto bisogna inserire anche l'interazione tra gli agenti e i sistemi non basati sul paradigma MA i quali possono essere raggruppati sotto la categoria generale risorse.

3.2. L'utilizzo di standard per ottenere l'interoperabilità

Si già messo in evidenza come il vero scoglio dell'interoperabilità sia la diversità. Il modo più radicale per ottenere interoperabilità è quindi quello di eliminare le diversità e creare degli standard. In un mondo

standard dove tutto è uguale per tutti, dove esiste una sola lingua, un solo tipo di architettura di calcolatori, una sola codifica delle informazioni, non esiste il problema di come ottenere l'interoperabilità, in quanto i confini tra le entità che costituiscono il sistema risultano totalmente privi di barriere. Rendere tutto assolutamente standard non è però possibile e neanche utile. Quello che è utile rendere standard non sono i dettagli di implementazione, bensì le specifiche di alto livello. Al fine di raggiungere l'interoperabilità non interessa come le cose vengano implementate, ma quale sia il comportamento del sistema visto dall'esterno. Inoltre lasciare la libertà di implementazione consente di ottenere un alto livello di sviluppo e ricerca di nuove tecnologie e soluzioni.

Se da un lato l'utilizzo di standard è la soluzione ai problemi che nascono quando si vuole ottenere un certo grado di interoperabilità tra entità diverse, o nel nostro caso tra agenti, dall'altro bisogna porre attenzione per limitare il numero e la tipologia dei vincoli da imporre, in modo tale da evitare il rischio di ottenere come risultato, un sistema chiuso o con scarsa capacità espressiva e limitate potenzialità di sviluppo.

3.3. Gli elementi da rendere standard

Per eliminare le barriere che sorgono sui confini tra le entità che costituiscono i sistemi ad agenti mobili, è necessario definire uno standard per ciascun confine esistente. Quanto debba essere vincolante lo standard di ogni confine, è quello che si vuole discutere in questo paragrafo. Per evitare di imporre inutili vincoli bisogna cercare di capire cosa sia veramente necessario rendere standard e cosa possa invece essere lasciato libero.

Considerando il confine situato tra EU ed EU (confine 1 nella figura 10) e quello situato tra CE e CE (3 nella figura), si rileva una unica necessità di comunicazione. Per consentire lo scambio di informazioni tra entità eterogenee l'unica cosa che bisogna rendere standard è il meccanismo di comunicazione, non ha importanza quale sia la struttura delle EU o dei CE. È necessario un meccanismo completo che consenta

lo scambio di informazioni in qualsiasi forma e codifica, di basso livello o meno, che comunque non limiti la capacità espressiva. Eventualmente si potrà decidere di disporre di protocolli di alto livello in grado di gestire anche aspetti legati alla sicurezza, ma la cosa importante è che il protocollo utilizzato sia conosciuto da tutti.

Prendendo in esame l'interazione tra agenti e supporti (tra EU e CE) il primo aspetto che emerge riguarda il linguaggio di codifica degli agenti. Lasciare libero il linguaggio di codifica comporta notevoli difficoltà da superare per raggiungere l'interoperabilità ogni qual volta ci si trovi in presenza di mobilità di codice, ma non costituisce uno scoglio concettuale, e per questo, anche se in alcuni casi le difficoltà possono risultare tali da impedire il raggiungimento dell'interazione cercata (che in questo caso si traduce in mobilità), non è giusto imporre dei vincoli a questo livello.

Decidere a priori un linguaggio implicherebbe una chiusura del sistema e una limitazione nella capacità espressiva nella codifica degli agenti. Se ad esempio volessimo scrivere agenti con un comportamento intelligente, probabilmente la scelta del linguaggio da utilizzare per la loro codifica ricadrebbe sul PROLOG, ma se volessimo implementare agenti in grado di trasportare ed elaborare informazioni di carattere multimediale probabilmente la scelta migliore sarebbe Java che consente un rapido sviluppo di applicazioni dotate di interfacce grafiche e capacità multimediali.

Per consentire la libertà di scelta del linguaggio e allo stesso tempo favorire l'interazione, si può fare utilizzo di un formato astratto standard di codifica adibito a rappresentazione esterna dei linguaggi (un po' come si fa per lo scambio di informazioni in reti costituite da sistemi che hanno una rappresentazione dei dati differente l'uno rispetto agli altri). Lo scambio di codice tra sistemi che adottano differenti linguaggi può avvenire tramite il passaggio attraverso la rappresentazione esterna standard. Definire le funzioni di traduzione dai linguaggi alla rappresentazione esterna, e viceversa, è un operazione molto complessa e in alcuni casi addirittura impraticabile se le caratteristiche dei linguaggi

sono troppo differenti (si pensi che esistono linguaggi funzionali, imperativi e logici che sono differenti in maniera radicale).

In alternativa a questa soluzione si può pensare di utilizzare una macchina virtuale (si veda paragrafo 2.3.) che renda uniforme il sistema nel suo complesso. In questo modo il problema viene risolto alla radice e i CE costruiti sulla macchina virtuale hanno una visione omogenea degli agenti che, anche se codificati in linguaggi differenti, vengono poi compilati per la macchina virtuale nello stesso byte code.

Un secondo aspetto che riguarda l'interazione tra agenti e CE è relativo all'interfaccia attraverso cui un agente può accedere ai servizi di sistema. Se ad esempio un agente vuole cambiare sistema dovrà in generale effettuare l'invocazione di una funzione del CE in cui si trova. Sarà poi il CE stesso che provvederà ad effettuare tutto ciò che è necessario per il trasferimento dell'agente. La definizione di uno standard dell'interfaccia del supporto è necessaria per consentire l'interazione tra un agente e qualunque implementazione del supporto stesso. Le restrizioni imposte da questo vincolo sono praticamente nulle in quanto si tratta di un accordo tra i diversi produttori di software sui nomi da dare alle funzioni di sistema (l'implementazione è completamente libera da vincoli). Le difficoltà di accettazione degli standard sono le uniche che possono limitare la diffusione di un CE con interfaccia ben definita.

Rimane da considerare il confine tra agenti e risorse. L'imposizione di uno standard a questo livello può voler dire dover incapsulare tutti i possibili servizi e risorse in un insieme standard di interfacce. Questo è un vincolo troppo elevato soprattutto considerando la forte diversità dei servizi che possono esistere. Una soluzione che invece può essere applicata è l'incapsulamento delle risorse all'interno di agenti gestori. L'interazione con la risorsa così incapsulata può avvenire tramite l'utilizzo del meccanismo di comunicazione scelto per fare interagire gli agenti. In alternativa o in aggiunta a questo è possibile ipotizzare la conoscenza specifica delle risorse con cui un agente ha bisogno di interagire. Se volessimo ad esempio ottenere un set di dieci pagine web che parlano di un determinato argomento potremmo pensare di mandare

un agente mobile in giro per la rete a consultare server Web, diventando ogni volta un normale cliente che usa http come protocollo di richiesta di una pagina. Evidentemente l'ipotesi di conoscenza pregressa risulta limitata ad un certo numero di risorse. Una vastità di server che usano protocolli diversi imporrebbero all'agente la conoscenza di tutti i protocolli, portando ad una gestione non accettabile.

3.4. Gli standard proposti

Ora è chiaro che lo standard è la soluzione ai nostri problemi e sappiamo anche cosa dobbiamo sottoporre a vincoli, ma è ancora interamente da percorrere la strada che porta alla definizione vera e propria dello standard. Per fortuna qualcuno ha già pensato a tutto ciò e diverse proposte di standard sono nate, alcune basate su meccanismi già noti e già in fase di accettazione come CORBA, e altre nate dal nulla ma capaci di attirare l'attenzione degli esperti del settore.

3.4.1. Knowledge Sharing Effort (KSE)

KSE [KSE94] è uno standard per le regole di sviluppo, la condivisione di facility e il riutilizzo di basi di conoscenza e sistemi basati sulla conoscenza, sviluppata da ARPA. KSE si muove nel settore dell'intelligenza artificiale.

KSE specifica Knowledge Querying and Manipulation Language (KQML) [KQML93], [Labrou94] e Knowledge Interchange Format (KIF). KQML è la specifica di un linguaggio per la comunicazione inter-agente. Il suo scopo è quello di fornire le basi di un linguaggio comune per la comunicazione tra agenti. *Speech act Theory* è la teoria su cui KQML appoggia le sue fondamenta, secondo la quale la comunicazione tra due individui può essere ricondotta allo scambio di un piccolo numero di messaggi primitivi (*speech act*) che danno forma ai significati base della comunicazione. KQML cerca di coprire i bisogni della

comunicazione inter-agente definendo un nucleo di messaggi primitivi (KQML usa il termine *performative* al posto di *speech act*) cercando di bilanciare le necessità espressive con il desiderio di limitare il numero dei *performative*.

The Agent Society fu fondata per assistere lo sviluppo e la diffusione delle tecnologie ad agenti. Sul Web si trovano riferimenti a riguardo di una piattaforma comune per gli agenti e per il loro trasferimento, e riferimenti a MASIF (si veda in seguito)[AgSoc97].

3.4.2. FIPA 97

La Foundation for Intelligent Physical Agents è un'associazione senza scopi di lucro per la promozione delle applicazioni basate su agenti intelligenti. La prima specifica definita dall'associazione è FIPA 97. Si tratta di un documento suddiviso in sette parti, tre delle quali riguardano le normative sulla tecnologia dei sistemi ad agenti e le restanti presentano le possibili applicazioni nel modello ad agenti. Esamineremo in seguito le tre parti riguardanti le tecnologie, trascurando tutto quello che riguarda le applicazioni delle quali si è già parlato nel capitolo due.

3.4.2.1 Agent Comunication Language (ACL)

ACL è il linguaggio di comunicazione che gli agenti dovrebbero utilizzare per scambiare informazioni. Come KQML anche ACL è basato sull'idea di fornire un insieme di messaggi primitivi che consentano lo scambio di informazioni in maniera completa. Come lo scambio dei messaggi possa avvenire non è specificato da FIPA che ipotizza l'utilizzo di uno strato di trasporto (ad esempio TCP/IP) per questa esigenza.

3.4.2.2 Agent Management

Questa parte di FIPA descrive gli Agent Platform Services, ovvero specifica i servizi necessari per costruire una società multi agente. Tra i servizi vi sono funzioni per costruire una comunità di agenti, entrare ed uscire da essa, incontrare agenti all'interno di una comunità e comunicare con agenti remoti, cioè residenti in altre comunità. Vengono specificati in questa parte anche un servizio di nomi (Agent Name Server) e di pagine gialle (Directory Facility).

In questo contesto viene anche definita l'interfaccia di CE (denominato da FIPA AgentPlatform). Una AgentPlatform può corrispondere o meno ad un singolo host e contiene sempre almeno un servizio di nomi (ASN) e un servizio di pagine gialle (DF).

Infine viene specificato Agent Comunication Channel (ACC), ossia il meccanismo di default che è alla base della costruzione dei canali di comunicazione tra agenti. Gli agenti, sfruttando ACC, possono scambiare messaggi in formato ACL. Come meccanismo di default alla base della costruzione del canale di comunicazione viene posto CORBA IIOP In questo modo FIPA non si deve occupare di una parte di problemi relativi alla computazione distribuita già affrontati da CORBA.

3.4.2.3. Agent/Software integration

In questo modulo FIPA cerca di raccogliere le soluzioni ad hoc che sono state utilizzate in passato per mettere in relazione sistemi ad agenti con sistemi software diversi. In base alla classificazione fatta nel paragrafo 3.1. ci troviamo sul confine tra agenti e risorse. La soluzione proposta è costituita da un *wrapper agent* che incapsula la risorsa rendendola un agente e permettendo così una visione uniforme anche di risorse che non sono orientate agli agenti, e di un Agent Request Broker (ARB) che è un directory service che raccoglie le risorse incapsulate. Un agente si deve quindi rivolgere all'ARB per raggiungere l'agente che gestisce la risorsa per cui è interessato. Dopo di che l'agente cliente potrà chiedere i servizi di cui necessita al *wrapper agent*. Ogni interazione tra cliente e ARB, e tra cliente e *wrapper agent*, avviene attraverso l'utilizzo di ACL. In

questo modo qualunque sia la risorsa si può interagire con essa con l'utilizzo di un meccanismo di comunicazione standard. Il collegamento tra *wrapper agent* e risorsa non è reso standard. In base alla risorsa si dovrà realizzare un wrapper agent differente in grado di comunicare con essa sfruttando il protocollo specifico della risorsa.

3.4.2.4. Aspetti non specificati da FIPA 97

FIPA 97 [FIPA97] non fornisce uno standard completo in tutti gli aspetti che vengono coinvolti quando si parla di agenti, ma lascia per ora non definiti alcuni punti:

- Sicurezza. Per quello che concerne la sicurezza FIPA deve ancora affrontare i problemi di autenticazione, privacy della comunicazione, paternità dei messaggi (es. mandare un messaggio e poi negare di averlo fatto), e comunque tutti i problemi di sicurezza che comporta un mondo di agenti mobili.

- Descrizione e condivisione dell'ontologia. Un'ontologia da il significato ai simboli usati in un messaggio. In FIPA 97 l'ontologia è specificata informalmente.

- Semantica formale per ACL. Non esiste ancora un meccanismo che verifichi formalmente se un agente è conforme alla specifica ACL perché la semantica di ACL non è definita formalmente, ma è implicitamente specificata dallo stato mentale di chi lo usa.

- Supporto per la mobilità. In FIPA 97 non è specificata la mobilità. Quello che bisogna definire è come gli agenti possano cambiare piattaforma senza dire come la struttura interna dei differenti supporti o degli agenti debba essere organizzata. Nella terza Call for Proposal di FIPA [FIPACFP97] si inizia a trattare il problema della mobilità, in quanto anche secondo FIPA questo aspetto sta assumendo un ruolo

sempre più importante. Comunque per ora il limite fondamentale di FIPA riguarda proprio l'assenza di normative riguardo la migrazione. Probabilmente questo è dovuto al fatto che la visione iniziale dell'associazione, era quella di fornire gli elementi chiave per un ambiente di computazione distribuito costituito da agenti intelligenti, avvicinandosi più ad un modello analogo a quello di CORBA, piuttosto che ad un modello ad agenti mobili vero e proprio.

3.4.3. CORBA

In questo contesto CORBA può diventare lo strumento di comunicazione standard necessario per lo scambio di informazioni tra agenti e tra sistemi. CORBA fornisce uno standard per la comunicazione tra i suoi oggetti basato sull'invocazione remota di metodi, se quindi identifichiamo negli oggetti CORBA gli agenti, abbiamo già uno standard di comunicazione che fornisce un meccanismo completo e semplice da utilizzare per scambiare informazioni complesse in un ambiente eterogeneo. L'invocazione remota di CORBA può naturalmente essere utilizzata anche dagli agent system (i CE) per scambiare informazioni.

CORBA ci fornisce anche la soluzione al problema legato all'eterogeneità dei linguaggi di codifica degli agenti. Come spiegato nel capitolo uno è possibile porre un'interfaccia IDL a servitori scritti in un qualunque linguaggio, ed ottenere così la possibilità di utilizzare servizi indipendentemente dalla loro implementazione.

A questo punto il nostro mondo di agenti è un mondo di oggetti CORBA in grado di comunicare in maniera indipendente dal linguaggio che essi utilizzano. Lo standard CORBA potrebbe quindi non solo essere la soluzione per fornire il mezzo di comunicazione tra gli agenti, e tra i sistemi, ma potrebbe divenire nel suo complesso il sistema ad agenti. Purtroppo gli oggetti CORBA sono privi di capacità di migrazione, se CORBA riesce a risolvere in maniera così brillante il problema del

linguaggio di codifica, è proprio perché lo fa in un ambito dove la mobilità non è consentita.

Nonostante ciò CORBA rimane un importante strumento in grado di fornire uno standard di comunicazione di alto livello accolto già da molte aziende e accademie. Può quindi diventare il punto di partenza su cui basare le fondamenta dell'interoperabilità tra sistemi ad agenti, integrando su esso le funzioni che stanno alla base del modello ad agenti mobili iniziando dall'integrazione dei meccanismi di mobilità.

CORBA non fornisce soltanto comunicazione, bensì mette a disposizione molti servizi che possono essere direttamente utilizzati dalle tecnologie ad agenti mobili. Ecco ad esempio che i servizi di sicurezza di CORBA [CSI95] sono immediatamente disponibili per le esigenze dei sistemi ad agenti mobili, o ancora il servizio di nomi COSNaming (si veda paragrafo 1.2.) diventa utile per mantenere la locazione degli agenti stazionari. Quale siano le vere potenzialità di questi servizi in ambito di agenti mobili verrà evidenziato nel prossimo paragrafo, quando si tratterà MASIF ossia l'estensione di CORBA per soddisfare le esigenze del modello MA.

CORBA è un supporto standard utile anche per quello che riguarda l'interazione tra agenti e risorse. Per quello che concerne l'integrazione tra agenti e sistemi diversi otteniamo infatti in maniera semplice il risultato di incapsulamento delle risorse entro agenti. CORBA permette infatti di trasformare applicazioni di qualunque genere in oggetti CORBA, semplicemente aggiungendo un'interfaccia IDL, senza modifiche del codice.

Non solo comunicazione e interazione agenti risorse, CORBA è anche uno standard per l'interazione tra agenti e CE. Per le operazioni di management e in generale di interazione con il supporto degli agenti (che in questo caso coinciderebbe con l'ORB) l'interoperabilità è garantita se utilizziamo le interfacce dell'ORB e quella del POA (si veda capitolo uno) in quanto soggette a standard e uguali per tutte le implementazioni di CORBA. Nel caso di utilizzo del BOA gli agenti dovranno essere in grado di distinguere l'implementazione di CORBA su cui risiedono e utilizzare

le funzioni appropriate, ma se questo può essere forse accettabile in un contesto in cui gli agenti non si muovono, in quanto è necessaria la conoscenza di una sola interfaccia, la situazione diventa difficile da gestire se un agente si muove e trova come supporto dopo ogni spostamento, un implementazione diversa di CORBA.

Nel prossimo paragrafo sarà considerato come tutte le possibilità offerte da CORBA, possano effettivamente essere riutilizzate per avvicinare il modello di computazione distribuita che è CORBA, ad un modello di computazione basato sulla mobilità di codice. Accanto ai servizi CORBA utili in questo processo, verranno specificati i nuovi meccanismi necessari che CORBA non è in grado di fornire.

3.4.4. MASIF, Mobile Agent System Interoperability Facility

Il limite fondamentale di CORBA si è visto essere l'assenza di mobilità, per questo l'OMG ha proposto lo standard MASIF che, affiancandosi a CORBA, consente di ottenere, oltre a tutti i vantaggi presentati nel paragrafo precedente, la mobilità di codice.

Lo standard MASIF va collocato sul confine tra CE e CE, in quanto propone un interfaccia standard che specifica la comunicazione tra agent system differenti. Si tratta di un ulteriore livello standard che cerca di inquadrare i meccanismi di comunicazione CORBA, per fornire agli agent system uno strumento dedicato in grado di soddisfare in maniera più appropriata le necessità di comunicazione inter-CE.

MASIF, cercando di creare un ponte standard tra le diverse implementazioni di sistemi ad agenti, affronta le seguenti problematiche:

- **Agent Management**, nell'interfaccia proposta sono presenti le funzioni di management che consentono di creare, sospendere, riattivare e terminare agenti;

- **Agent Transfer**, probabilmente il punto più delicato. Vengono specificate le funzioni per il trasferimento di un agente da un sistema ad un altro di tipo diverso;

- **Agent and Agent System Names**, viene specificato come debba essere costruito l'identificatore di un agente o di un agent system;

- **Agent System Type and Location Syntax**, vengono classificati gli agent system esistenti in modo da capire se un certo agent system può supportare un agente nato in un altro agent system. La sintassi della posizione di un agente viene specificata, in modo che tutti gli agent system che possiedono una stringa che rappresenta tale posizione siano poi in grado di decifrarla per ritrovare l'agente.

Se utilizzando esclusivamente lo standard CORBA gli agenti del sistema dovevano necessariamente essere oggetti CORBA, con MASIF si vuole lasciare libere da vincoli le entità di esecuzione (EU). È per questo che tra i problemi affrontati da MASIF troviamo anche il management degli agenti e l'identificazione. Non potendo ipotizzare nulla sulla struttura degli agenti non è possibile in generale utilizzare i servizi di nomi e di management CORBA.

Lo standard MASIF specifica l'interfaccia di comunicazione tra sistemi ad agenti, dando la definizione di un oggetto CORBA chiamato MAFAgentSystem. Ogni agent system deve realizzare un'implementazione dell'interfaccia MAFAgentSystem. In questo modo, un primo sistem

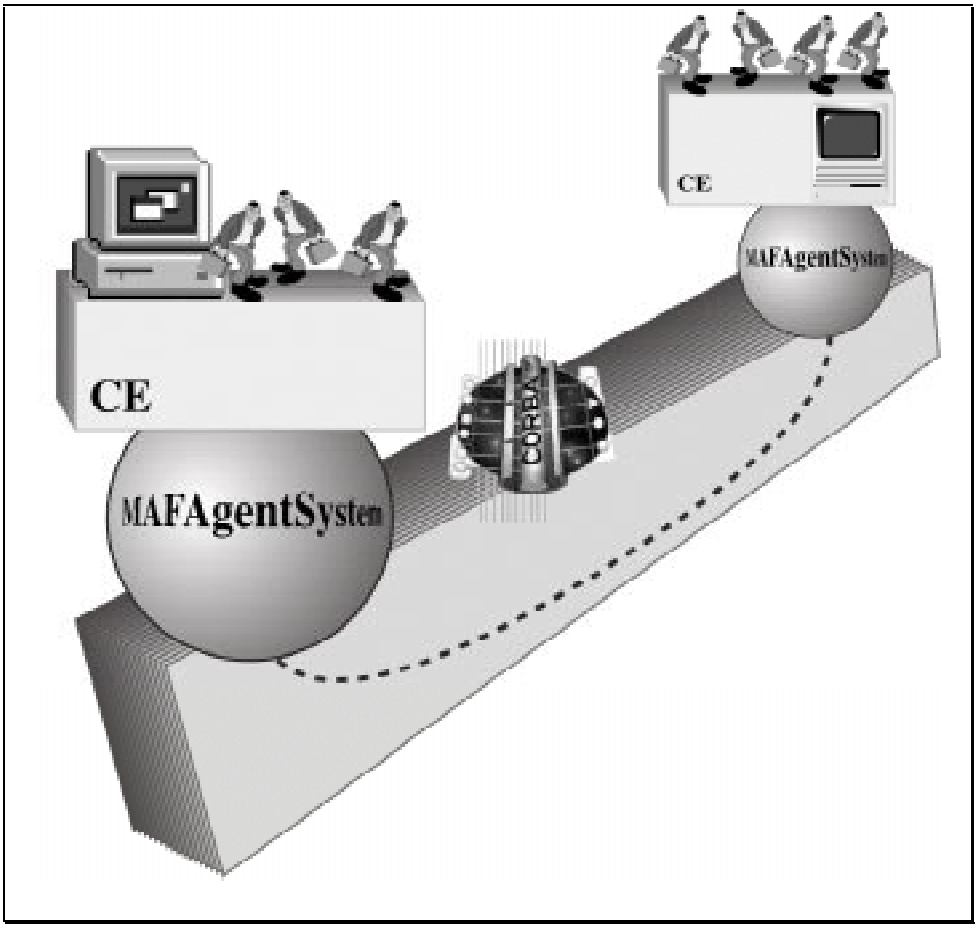

Figura 11. Interazione attraverso MASIF.

MASIF propone un'interfaccia standard tra diversi agent system, ma non si occupa di definire l'interfaccia tra un agente e il supporto sottostante. Considera quest'ultimo aspetto meno rilevante dal punto di vista dell'impatto sull'interoperabilità, e ne rimanda quindi la specifica.

A questo proposito vale la pena osservare che fino ad ora si è trattato il confine tra EU e CE in ambito totalmente locale, intendendo con esso l'interazione tra un agente e l'ambiente di esecuzione sottostante. Ogni forma di interazione tra un agente ed un sistema remoto deve quindi avvenire attraverso l'interazione con il sistema locale. In questo modo si mantiene distinto tutto ciò che è a livello di sistema da tutto ciò che è a

livello applicativo. L'introduzione di uno standard come MASIF, ed in particolare del MAFAgentSystem, rende possibile in maniera molto semplice la comunicazione tra agenti e CE remoti. Anche se questo tipo di scambio di informazioni tende a confondere il livello di sistema con il livello applicativo, si può, in alcuni casi, fare uso di esso per sopperire alla mancanza attuale di uno standard di interazione tra agente e sistema. Se ad esempio un agente vuole migrare da CE1 a CE2 può farlo anche attraverso l'utilizzo diretto delle funzioni che il MAFAgentSystem di CE2 mette a disposizione per questa operazione. In questo modo non è necessaria la conoscenza da parte dell'agente dell'interfaccia che CE1 propone ai sui agenti.

Soffermiamoci a considerare se l'utilizzo esclusivo di CORBA avrebbe potuto soddisfare le esigenze di comunicazione inter-CE. Effettivamente grazie alla dinamicità di CORBA sarebbe stato possibile fare a meno della specifica dettagliata della interfaccia MASIF. In un scenario privato del MAFAgentSystem un agente che vuole migrare deve chiedere al proprio agent system X di voler migrare sull'agent system Y (un'implementazione diversa). Per soddisfare la richiesta di un proprio agente, X potrebbe interrogare Y chiedendo all'Interface Repository quale sia l'interfaccia messa a disposizione da Y per consentire la migrazione, e cercare all'interno dell'interfaccia il metodo da invocare (l'introspezione di CORBA permette tutto ciò). In questo modo dopo la serializzazione dell'agente e la costruzione dei parametri necessari, la chiamata al metodo trovato completerebbe la migrazione. Tutto ciò è certamente possibile, ma presenta forti limiti legati alla determinazione della semantica associata ai nomi delle interfacce, dei metodi e dei parametri. Comprendere quali siano i significati dati ad un insieme di parametri di cui si possiede nome e tipo non è cosa in generale semplice tanto più che automatizzabile. Se in alcuni casi la semantica associata ad un nome è banale (per una funzione che si chiama somma, che ritorna un intero, ed accetta come parametri due interi, si può presupporre banalmente che esegua la somma dei numeri passati), in altri casi può risultare complessa fino al punto che neanche l'intervento dell'uomo può comprenderla se non ottenendo

informazioni ulteriori. Per questo MASIF è necessario. Il suo scopo è quello di definire, in modo standard, quali siano i nomi delle funzioni necessarie per la comunicazione inter-CE, e associare a ciascuno di essi la giusta semantica.

Dopo questa breve digressione possiamo fare il punto della situazione di tutto ciò che fino ad ora abbiamo specificato. La comunicazione per la quale si utilizzerà l'invocazione remota di CORBA, l'integrazione con sistemi non ad agenti (grazie all'incapsulamento che IDL consente), e l'interazione tra agent system attraverso la definizione dell'oggetto CORBA MAFAgentSystem, sono gli standard che vengono addottati quando si sceglie di seguire MASIF.

Rimane da trattare come MASIF affronta il problema legato ai differenti linguaggi di codifica degli agenti. Per la codifica di oggetti CORBA il problema non sussiste in quanto, seppure con la privazione della mobilità, la soluzione è già fornita da CORBA IDL (paragrafo 1.1.1.). Per gli agenti non stazionari i problemi introdotti da questo grado di libertà sono molto più complessi e MASIF non fornisce una risposta per la loro risoluzione, in quanto ritiene le tecnologie non ancora pronte, propone comunque le strutture necessarie a classificare i linguaggi esistenti. In questo modo è sempre possibile capire quale linguaggio sia stato utilizzato per codificare l'agente e gestire la situazione di conseguenza.

3.4.4.1. Concetti base e terminologia

Prima di entrare nei dettagli di MASIF è bene fare una carrellata di quali siano i concetti di base e la corrispondente terminologia utilizzata nella specifica MASIF.

MASIF introduce i termini *stato*, *authority* e *nome* di un agente. Per *stato* si intende, come si fa di solito, lo stato di esecuzione dell'agente; l'*authority* identifica la persona per la quale l'agente lavora (in altri contesti si usa dire *principal*); mentre il *nome* è il termine che usa MASIF

per intendere l’identificatore unico dell’agente, di cui viene specifica anche la struttura.

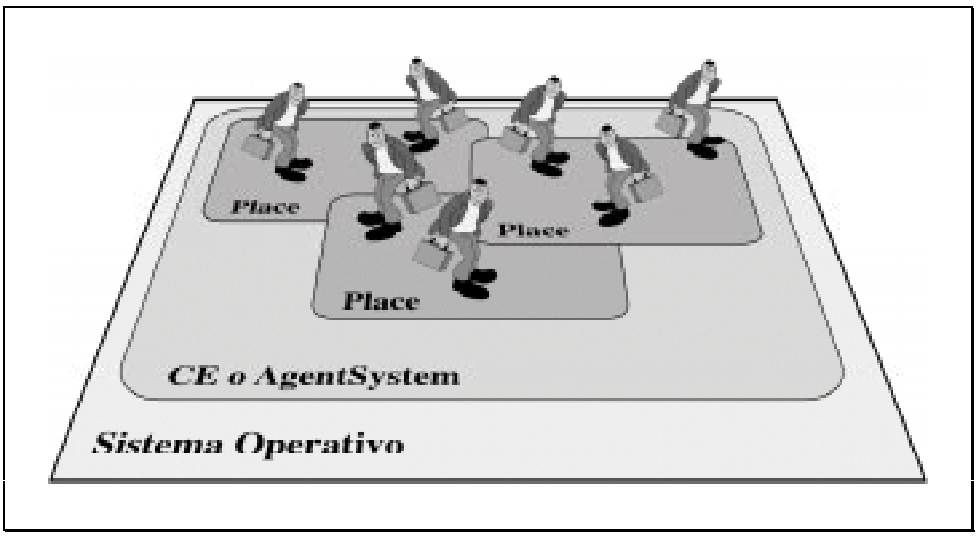

Figura 12. Lo scenario di MASIF è costituito da AgentSystem, Place e Agenti.

Se si legge la specifica MASIF si troverà largo uso del termine *agent system*, con questo si intende una piattaforma che sia in grado di eseguire, trasferire ed in generale gestire agenti. Il termine agent system o sistema ad agenti o supporto o CE (*Computational Environment)* è stato utilizzato fino ad ora con questo stesso significato. Con *agent system type* si intende il tipo di sistema ad agenti. MASIF vuole classificare i diversi sistemi ad agenti associando a ciascuno un *agent system type* differente. Oltre alla distinzione tra gli agent system, MASIF introduce una classificazione dei linguaggi e dei formati di serializzazione. Ogni agente può così essere corredato di un insieme di informazioni che definiscono l'agent system di appartenenza, la serializzazione utilizzata per i trasferimenti e il linguaggio di codifica dell'agente stesso. In questo modo gli agent system possono stabilire la propria capacità di ricevere un agente analizzando le informazioni di corredo.

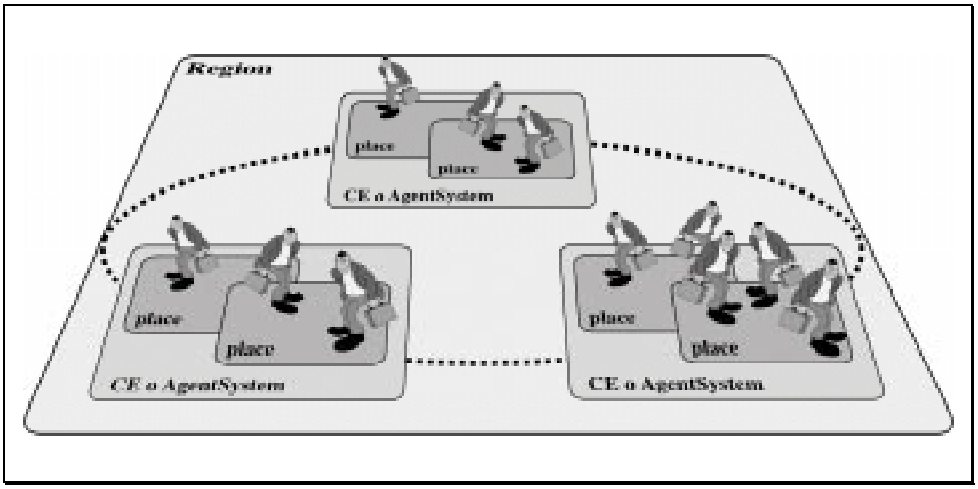

Figura 13. Una regione è costituita da un insieme di più AgentSystem che hanno in comune l'authority (o principal).

Infine, con le *region* e i *place* si dà una struttura al mondo degli agenti: i place sono un contesto di esecuzione entro un agent system dove un agente può eseguire, mentre le regioni sono un insieme di agent system che hanno come caratteristica comune la stessa authority e possono differire per tipo. Nell'ambito MASIF un place non è quindi un CE, ma va considerato un raggruppamento di servizi che fornisce un contesto per le necessità di esecuzione degli agenti.

3.4.4.2. MAFAgentSystem e MAFFinder

A questo punto è possibile spiegare anche nei dettagli quale sia lo scenario proposto da MASIF, si può finalmente chiarire come sia fatta quella interfaccia che è posta tra due agent system.

I sistemi ad agenti che vogliono aderire a questo standard devono implementare due oggetti CORBA MAFAgentSystem e MAFFinder. Sono proprio questi che costituiscono la parte standard di ogni agent system e che devono essere utilizzati come normalmente si fa con gli oggetti CORBA. Se ad esempio un agent system X vuole ottenere

l'informazione di quale sia l'agent system type di Y, non deve fare altro che ottenere un riferimento al MAFAgentSystem di Y e invocare su di esso il metodo get_agent_system_info.

Il MAFAgentSystem mette a disposizione tutte le operazioni che consentono la migrazione e il management degli agenti. Sono presenti anche funzioni che permettono di ottenere informazioni sull'agent system (es. get_agent_system_info) e sugli agenti che vi risiedono.

Il MAFFinder mette a disposizione un servizio di nomi necessario per consentire la localizzazione di agenti. I servizi di nomi CORBA, quali COSNaming, non sono in generale adatti per le esigenze di ripetute registrazioni che gli agenti mobili mostrano. Ecco allora che MASIF definisce il MAFFinder e propone con esso le modalità utilizzabili per l'individuazione degli agenti.

3.4.4.3. Migrazione e creazione remota di agenti

Considerando che la mobilità di codice è la caratteristica più importante del paradigma MA, si inizierà ad analizzare MASIF da questo punto di vista. La migrazione e la creazione remota hanno in comune l'esigenza di trasferire del codice da un ambiente di computazione a un altro. Per fornire un meccanismo elastico adatto alle diverse esigenze, MASIF non lega il trasferimento di codice direttamente alle funzioni di migrazione e creazione, ma specifica per esso una funzione (fetch_class) a parte. In questo modo, in base alle esigenze, si potrà adottare la giusta politica di richiesta del codice:

- **Trasferimento automatico delle classi.** Durante creazione o migrazione l'agent system sender provvede a spedire all'agent system di destinazione tutte le classi necessarie per l'esecuzione futura dell'agente, oltre che per la creazione immediata dell'istanza. In questo modo l'agent system mittente può anche scomparire (si pensi a reti mobili e computer portatili che rimangono collegati per il tempo

indispensabile la creazione dell'agente che poi lavorerà a loro vantaggio).

- **Trasferimento automatico delle classi dell'agente, e trasferimento on demand delle altre**. Vengono inviate tutte le classi necessarie ad istanziare l'agente. Le eventuali classi che dovessero servire all'agente durante le sua esecuzione vengono richieste dinamicamente (on demand). In questo modo si evita che vengano inviate tutte le classi, anche quelle che potrebbero non essere necessarie, dato il non determinismo a priori delle operazioni che svolgerà l'agente.

- **Variazione dei precedenti**. Utilizzando il trasferimento automatico alla creazione e on demand per le migrazioni, otteniamo i vantaggi di ambe due le soluzioni, ossia la possibilità di scollegarsi dopo la creazione e di evitare la trasmissione inutile di classi nelle migrazioni successive.

- **Altra variazione**. Si spedisce una lista di tutte le classi di cui avrà bisogno l'agente, in modo che l'agent system di destinazione possa ottenere tutte le classi come nel caso di trasferimento automatico, ma evitando di richiedere quelle che già possiede.

I metodi del MAFAgentSystem che sono coinvolti nel processo di migrazione di un agente sono receive_agent e fetch_class. MASIF prevede che a fronte di una richiesta di migrazione fatta da un agente al suo agent system X per migrare in un agent system Y, X provveda a sospendere l'agente, a serializzarlo, ad autenticarsi con Y attraverso i meccanismi forniti da CORBA e ad invocare la receive_agent di Y, che dovrà provvedere a deserializzare l'agente, a ricostruire il suo stato e a metterlo in esecuzione.

Il metodo fetch_class entra in gioco per recuperare le classi di cui ha bisogno l'agente (essendo MASIF orientato agli oggetti si parla sempre di classe piuttosto che di codice), se una classe è necessaria all'agent system

che ha ricevuto l'agente, il metodo fetch_class del mittente deve essere invocato, per ottenere la classe. Quando questo debba avvenire dipende dalla politica scelta.

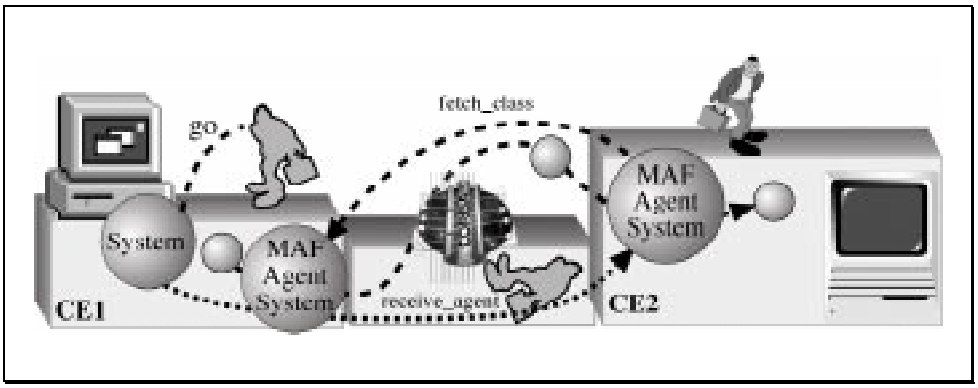

Figura 14. La migrazione con una politica on demand. L'agente chiede a CE1 di andare su CE2. CE1 prepara l'agente per il trasferimento ed invoca il metodo receive_agent di CE2. Lo stato dell'agente viene così trasferito su CE2. Non essendo presente nell'ambiente di arrivo la classe necessaria per ripristinare lo stato dell'agente, CE2 invoca il metodo fetch_class di CE1 e ottiene la classe. Eventuali altre classi che si rivelassero necessarie durante l'evoluzione futura dell'agente (ad esempio per la necessità dell'agente di creare nuovi oggetti), verranno richieste sempre attraverso l'invocazione di fetch_class.

La creazione risulta essere molto simile alla migrazione in quanto come già detto anche essa implica il trasferimento di classi e quindi mobilità di codice. Le differenze fondamentali sono l'assenza di uno stato da ripristinare, dato che l'agente viene creato per la prima volta, e la necessità di creare un nuovo nome che consenta l'identificazione univoca dell'agente creato. Per la creazione MASIF specifica il metodo create_agent.

3.4.4.4. Sicurezza in MASIF

Il ruolo della sicurezza nelle applicazioni di rete è di importanza fondamentale. Nel capitolo due si è visto anche quali nuovi problemi nascano quando siamo in presenza di mobilità di codice e agenti mobili. MASIF non affronta in maniera diretta tali problematiche, ma rinvia a CORBA il quale specifica già una serie di servizi che possono essere utilizzati per soddisfare le esigenze di sicurezza dei sistemi ad agenti mobili. Senza ripercorrere quali siano tali esigenze, vediamo ora come CORBA fa per soddisfarle (per informazioni più dettagliate [CSI95]).

- **Autenticazione di un client che vuole creare un agente**. Prima di invocare la create_agent, il client invoca l'operazione di autenticazione sull'oggetto CORBA PrincipalAuthenticator per stabilire le sue credenziali. Una volta ottenute le credenziali può renderle disponibili all'agent system remoto, effettuando una invocazione sicura, il quale potrà così sottoporre alla giusta politica di sicurezza il nuovo agente. L'agent system che deve creare l'agente può ottenere le credenziali mediante l'utilizzo dell'interfaccia SecureCurrent che CORBA mette a disposizione proprio per questo scopo.

- **Autenticazione reciproca di agent system**. L'autenticazione reciproca può essere utile nel caso in cui un agent system sender vuole essere sicuro di chi sia veramente l'agent system su cui sta spedendo un suo agente. CORBA la consente settando le giuste opzioni (EstabilishTrustInClinet, EstabilishTrustInTarget).

- **Autenticazione e delegazione.** Quando siamo in presenza di migrazioni multiple di un agente diventano critiche le problematiche di autenticazione. Un agente non può autenticarsi in quanto non può avere con se, quando si sposta, la sua chiave privata (per evidenti motivi di sicurezza). Il problema dell'autenticazione durante migrazioni multiple (multi-hop authenticator) non è affrontato da MASIF, che però specifica un modello di autenticazione e delegazione in cui, in un certo qual modo, ci si fida dell'ultimo agent system da cui l'agente proviene.

Anche tale modello sfrutta i servizi CORBA. La sua realizzazione è possibile solo se si lavora con implementazioni dell'ORB che sono conformi a CSI level 2 [CSI95], il livello 2 di sicurezza secondo la specifica di CORBA (si veda paragrafo 1.2.). Tale modello ha bisogno del trasferimento di entrambe le credenziali dell'agente e dell'agent system su cui risiede e solo CSI level 2 supporta il trasferimento duplice di credenziali. Avendo a disposizione entrambe le credenziali è possibile autenticare l'agent system in cui l'agente risiede e di conseguenza fidarsi delle credenziali dell'agente (che vengono spedite, sempre tramite CORBA, dall'agent system).

- **Politiche di sicurezza di agenti e agent system**. CORBA mette a disposizione strumenti che consentono di definire politiche di sicurezza basate sulle credenziali del cliente. In pratica è possibile per un oggetto decidere di rifiutare il proprio servizio se le credenziali del client non lo soddisfano.

- **Sicurezza nella comunicazione**. CORBA consente di specificare il livello di sicurezza che si vuole avere durante la comunicazione. È possibile richiedere integrità, segretezza, autenticazione e rilevazione di richieste duplicate (qualcuno potrebbe intercettare un messaggio di richiesta e ripeterlo più volte per provocare danni al sistema).

In questo caso i servizi di sicurezza di CORBA possono essere utilizzati in maniera soddisfacente per trattare le esigenze di sicurezza del paradigma MA. MASIF non ha bisogno quindi di introdurre altre specifiche a questo proposito.

3.4.4.5. MASIF: nomi, identificatori e servizio di nomi

Gli identificatori MASIF per agenti ed agent system.
MASIF specifica quale debba essere la struttura degli identificatori degli agenti ed agent system. Un nome MASIF è costituito da tre elementi:

authority, identificatore e tipo di agent system. L'authority rappresenta la persona o l'organizzazione a vantaggio della quale l'agente (o l'agent system) lavora. L'authority inserita nell'identificare dell'agente deve essere la stessa che viene associata mediante l'uso delle credenziali quando si usano i meccanismi di sicurezza. L'identificatore è la parte del nome che risulta unico all'interno di un agent system di un certo tipo. Il tipo di agent system serve per differenziare sistemi diversi che generino per caso coppie authority identity uguali.

Lo scenario proposto da MASIF per la ricerca di un agente.
I servizi di nomi CORBA possono essere utilizzati anche nei sistemi ad agenti mobili (quando gli agenti siano oggetti CORBA), facendo però attenzione al fatto che la parte riferimento nella coppia nome-riferimento che i servizi di nomi mantengono, non è trasparente alla locazione dell'oggetto. Se quindi un oggetto registrato ad un certo name server si sposta, il riferimento perde di validità. Questo problema può essere risolto in diversi modi:

- **Gestione automatica da parte dell'ORB**. L'ORB mantiene valido il riferimento iniziale utilizzando una tabella che mantiene le corrispondenze tra i riferimenti iniziali e attuali. Il tutto viene gestito dall'ORB che mantiene un oggetto proxy per la gestione della tabella. Naturalmente in questa visione l'ORB deve sempre sapere quando un oggetto si muove e siamo quindi in un ambito in cui la mobilità è messa a disposizione dallo stesso ORB. Per questo si dovrà però probabilmente attendere CORBA 3.0;

- **Aggiornamento del name server**. Ad ogni migrazione un agente (o agent system) registra il nuovo riferimento al name server. In questo modo è possibile ritrovare l'oggetto anche dopo migrazioni successive. Bisogna naturalmente fare attenzione ai riferimenti che si possiedono perché potrebbero scadere di validità. Si può pensare che se l'oggetto di cui si possiede il riferimento non viene trovato dall'ORB venga

sollevata un eccezione, in seguito alla quale sarà necessario richiedere il nuovo riferimento al name server. Questa soluzione comporta un aumento dell'overhead di migrazione, in quanto comporta una registrazione al servitore di nomi ad ogni spostamento, ed inoltre tende a far diventare il name server un collo di bottiglia;

- **Proxy all'host di origine**. Si può pensare di lasciare un riferimento all'attuale locazione nell'host in cui si è nati. Si evita con questo tipo di soluzione di fare scadere i riferimenti che già si possiedono e si distribuisce il carico delle registrazioni, che devono ancora essere effettuate ad ogni migrazione, su diversi siti, evitando che uno diventi un collo di bottiglia. Esiste però lo svantaggio che se il nodo di origine cade si perde la possibilità di ritrovare l'oggetto. Si noti che in quest'ultima soluzione è l'ORB che a fronte di una richiesta deve ottenere il riferimento alla locazione attuale che è stato lasciato dall'agente sul nodo di origine. Questo implica un accordo tra l'ORB e l'agente sul formato da utilizzare per lasciare il riferimento proxy.

Se siamo in presenza di agenti che non siano oggetti CORBA tutto quello che si è appena discusso non vale più, in quanto i servizi di nomi di CORBA mantengono sempre una coppia nome-riferimento, dove la parte riferimento deve puntare ad un CORBA-object. Proprio per questo MASIF introduce il MAFFinder: un servizio di nomi supplementare utilizzabile da agenti che non siano CORBA-object e più adatto per affrontare le registrazioni di entità che si muovono.

MAFFinder è un oggetto CORBA che si propone di mantenere le posizioni di agenti, agent system e place. Il MAFFinder è un servitore di nomi abbastanza standard, mette a disposizione i classici servizi di registrazione e cancellazione di un *binding* (legame) nome-posizione. Oltre al normale meccanismo di ricerca di per nome, è possibile effettuare ricerche di tipo *pattern-matching* con le quali si può ottenere una lista di posizioni degli agenti (o agent system), che possiedono determinate caratteristiche. Se con i servitori di nomi CORBA ottenevamo sempre un

riferimento ad un oggetto, con il MAFFinder ci muoviamo ad un livello più alto. Le posizioni che si ottengono quando vengono effettuate delle ricerche al MAFFinder si riferiscono sempre all'ubicazione di agent system, anche nel caso si stia cercando un agente o un place. In questi due ultimi casi la posizione che si ottiene va interpretata come quella dell'agent system in cui l'agente o il place sono situati.

Una volta ottenuta la posizione di un agent system (che è rappresentata da una stringa e che MASIF chiama *location*) è necessario, per effettuare una qualsiasi operazione (migrazione, creazione o management), ottenere il riferimento all'oggetto CORBA MAFAgentSystem che gestisce il sistema di cui si conosce la posizione. Per ottenere il riferimento voluto bisogna risolvere la location. MASIF propone due possibili tipi di location che possono essere risolte in due differenti modi. Una stringa che rappresenta una posizione può contenere un nome CORBA CosNaming oppure un indirizzo Internet.

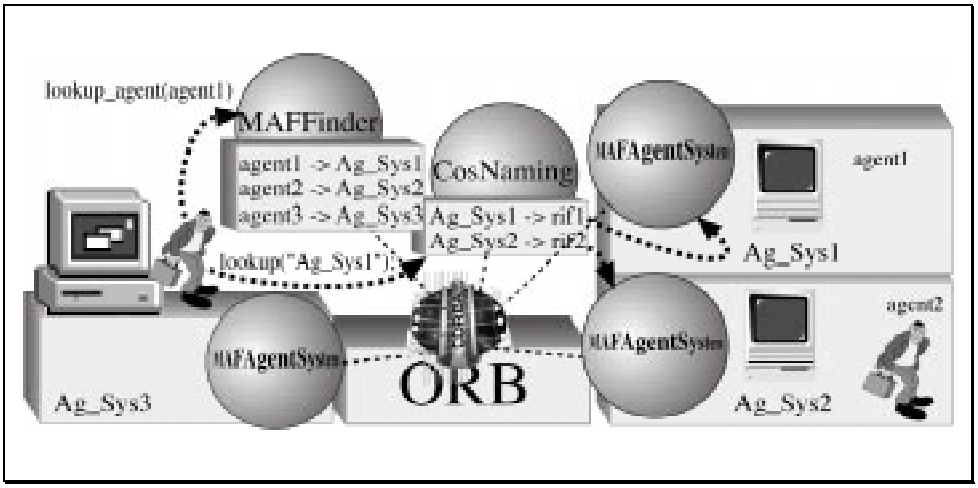

Figura 15. Un agente che cerca un suo simile. Interrogazione del MAFFinder per ottenere la location. La location in questo caso contiene un nome CORBA, e come si dirà in seguito, per ottenere il riferimento all'oggetto MAFAgentSystem corrispondente, è necessario accedere al servizio di nomi CosNaming (si veda paragrafo 1.2). Una volta ottenuto il riferimento si potrà interagire con l'agent system Ag_Sys1.

Prima di entrare nei dettagli della sintassi della location e di quali siano le modalità per risolverla, è bene concludere lo scenario. Adesso dovrebbe essere chiaro come sia possibile ritrovare un agente: ci si rivolge al MAFFinder per procurarsi la posizione con la quale si può risalire al riferimento al MAFAgentSystem. A questo punto è possibile interagire con l'agente richiedendo, ad esempio, la sua sospensione, la sua terminazione, o quale sia il suo stato di esecuzione attuale. Non abbiamo ancora detto come si comporta il MAFFinder se l'agente migra. In presenza di spostamenti il MAFFinder mantiene l'ultima location che era stata registrata. MASIF, dando la specifica del MAFFinder, non vuole definire una tecnica di ricerca, ma mettere a disposizione i metodi per la gestione di un database di agenti, agent system e place e le corrispondenti location. La tecnica che consente ad un cliente di ritrovare un agente non viene specificata nello standard dove invece è possibile trovare i possibili modi di agire. Le possibili modalità di ricerca sono già state trattate nel capitolo due, verranno ora riconsiderate nell'ottica MASIF:

- **ricerca esaustiva**: si chiede a tutti gli agent system se l'agente che si sta cercando risieda lì. Il MAFAgentSystem mette a disposizione un metodo denominato list_all_agents che può essere utilizzato per ottenere la lista dei nomi di tutti gli agenti presenti nell'agent system. Il MAFFinder è necessario per ottenere le location di tutti gli agent system che si vogliono ispezionare, ma può non essere utilizzato dall'agente. In alternativa al metodo list_all_agents si può utilizzare il metodo get_agent_status che fornisce lo stato di un agente dato il suo nome. Nel caso l'agente non sia presente viene lanciata una eccezione (di tipo NameInvalid). In questo modo, è possibile capire se un agente si trova all'interno di un agent system evitando il trasferimento della lista di nomi (e con essa la ricerca per verificare se il nome è presente nella lista), e ottenendo, nel caso di successo lo stato corrente dell'agente. Come già detto nel capitolo due questo algoritmo è molto costoso e può essere utilizzato esclusivamente in ambiti in cui la località di ricerca è limitata;

- **tecnica di logging**: ogni volta che l'agent migra viene lasciato un riferimento all'agent system in cui si è spostato. Si parte quindi dal MAFFinder il quale fornisce il primo agent system su cui l'agente è andato, e poi si seguono le tracce lasciate fino all'agente. Quando l'agente muore bisogna utilizzare una tecnica di garbage per eliminare tutti i riferimenti inutili. Per ottenere i riferimenti lasciati dall'agente bisogna fare utilizzo di meccanismi proprietari non standard. Non è possibile interrogare un agent system, facendo utilizzo dell'interfaccia MASIF, per sapere dove un certo agente sia andato. Nel MAFAgentSystem non esiste alcuna funzione che lo consenta;

- **registrazioni ad ogni migrazione**: come si poteva fare con i name server CORBA, l'agente ad ogni migrazione deve aggiornare il MAFFinder alla nuova locazione, oppure è possibile aggiornare l'agent system di origine e lasciare il MAFFinder diretto a questa locazione. In ogni caso si paga un costo aggiuntivo ad ogni migrazione, nella seconda situazione si evita però che il MAFFinder diventi un collo di bottiglia. Se si registrano gli spostamenti all'agent system di origine piuttosto che al MAFFinder, si deve gestire l'indirettezza introdotta in maniera non trasparente. Dovendo infatti utilizzare esplicitamente la location che si riceve dal MAFFinder, per ottenere il riferimento ad un agent system, non è possibile nascondere l'indirettezza.

- **autopubblicazione**: un agente si registra al MAFFinder solamente quando vuole poter essere trovato. Per trovare un agente che non si è mai pubblicizzato è necessario un algoritmo di ricerca esaustivo.

La scelta di un metodo di ricerca piuttosto che un altro deve essere effettuata in base alle esigenze e alle situazioni che si presentano. Non si può di certo pensare di utilizzare la ricerca esaustiva per ritrovare un agente in un ambito quale la rete Internet, però si può ipotizzare il suo utilizzo in un ambito di località come può essere una regione. La

registrazione ad ogni migrazione consente di ritrovare l'agente in ogni momento, comporta però l'aggiunta di un overhead ad ogni spostamento dell'agente. Si potrebbe pensare ad una tecnica mista che registra le migrazioni che avvengono tra differenti regioni, ma non quelle intra-regione. In questo modo la ricerca di un agente può avvenire consultando un name server per stabilire la regione corrente in cui l'agente risiede, e di seguito con un metodo di ricerca esaustiva all'interno della regione.

In realtà prima di stabilire il giusto modo con cui sia possibile ritrovare un agente bisogna capire se tale esigenza sia veramente necessaria. In alcuni casi applicativi come il recupero di informazioni (Information Retrieval) l'agente può girare alla ricerca dei dati necessari e decidere di tornare a casa una volta che li ha ottenuti. Non è mai necessario che il creatore dell'agente sappia la locazione attuale del suo rappresentante

In altri casi è invece indispensabile che un agente sia ritrovato dal suo creatore o da altri agenti. In questi casi si dovrà stabilire la modalità di ricerca adeguata al problema che si sta affrontando. Se ad esempio si necessità di cooperazione continua tra un gruppo di agenti al fine di portare a termine un certo lavoro, si dovrà rendere gli agenti rintracciabili in modo facile ed efficiente, magari al prezzo di registrazioni ripetute ad ogni migrazione. L'alta necessità di coordinamento richiede sempre l'attuazione del principio di località, per evitare che i costi di interazione delle entità in gioco inseriscano un overhead troppo elevato all'applicazione. Per questo motivo chi sviluppa applicazioni con agenti fortemente coordinati dovrà restringere il campo di mobilità a reti di piccole dimensioni, e potrà quindi effettuare anche molte registrazioni in maniera efficiente potendo usufruire di un servitore di nomi che gestisce un ristretto ambito di clienti.

L'utilizzo di tecniche di logging consente il ritrovamento degli agenti anche in reti di grandi dimensioni e senza la necessità di ripetute registrazioni ad ogni spostamento dell'agente. Questo tipo di soluzione implica però la gestione dei riferimenti che devono essere lasciati di volta in volta, ed eliminati quando non più necessari.

L’autopubblicazione sta a ribadire il concetto che le modalità di ricerca di un agente devono essere di volta in volta adeguate al problema. Il fatto che l’agente si renda rintracciabile solamente quando ne manifesta la volontà indica proprio le differenti esigenze che possono insorgere di volta in volta.

MASIF specifica il MAFFinder come meccanismo in grado di gestire un database di informazioni riguardanti le locazioni di agenti, agent system e place, ma non definisce la politica di ricerca che deve essere scelta in base alle esigenze applicative.

Location: nome CORBA CosNaming e indirizzo Internet

La location che si ottiene dal MAFFinder è una stringa che può contenere un nome CORBA o un indirizzo Internet. MASIF specifica dettagliatamente per le due situazioni quale sia la sintassi da utilizzare per la costruzione della location, e come si possa risalire da essa al riferimento CORBA corrispondente:

- **nome CORBA CosNaming**. Nella specifica MASIF si trova la denominazione *CosNaming Location String Format* per indicare la sintassi della location. In questo caso per ottenere il riferimento al MAFAgentSystem corrispondente bisogna trasformare la posizione (che è in formato stringa) in un formato CosNaming.Name ed accedere al servizio di nomi CosNaming (si veda paragrafa 1.2). Un CosNaming.Name è fondamentalmente un vettore di coppie, chiamate NameComponent, e costituite da un identificatore ed un tipo. La parte identificatore di ogni coppia è utilizzata dal name server per costruire il nome finale dell’oggetto che si sta cercando. La parte tipo ha uno scopo di carattere puramente descrittivo. MASIF definisce la sintassi della location utilizzando un formato URI (si veda RFC 1630, “Universal Resource Identifiers in WWW” per i dettagli del formato URI) e mettendo in corrispondenza diretta le informazioni necessarie per costruire un CosNaming.Name con il formato della stringa. Nel riquadro un esempio per mostrare quale sia il formato della location e

la corrispondenza con CosNaming.Name (per ulteriori dettagli sulla grammatica e le regole di produzione si rimanda a [MASIF97]).

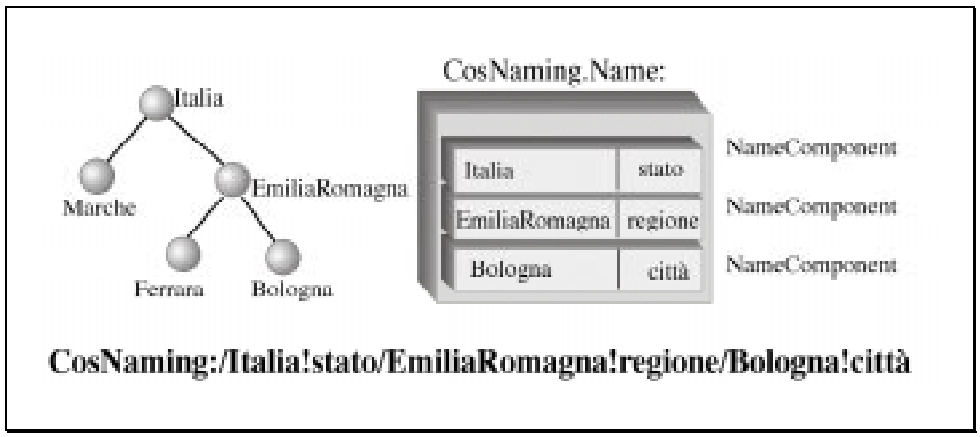

Figura 16. Per identificare Bologna si deve specificare interamente il nome a partire dalla radice della gerarchia, ottenendo così il nome composto da Italia, EmiliaRomagna e Bologna. Un nome di questo tipo viene mappato nel CosNaming.Name presente nel disegno, nel quale è possibile individuare oltre alla parte identificativa una parte puramente descrittiva (es. Italia viene descritta con la parola stato). In basso è posto il nome corrispondente in formato URI.

- **indirizzo Internet.** Quando la location contiene un indirizzo Internet è possibile costruire il riferimento all'oggetto direttamente dalle informazioni che si trovano nella stringa. Per costruire uno IOR IIOP (si veda paragrafo 1.3.), ossia un riferimento ad un oggetto basato sul protocollo Internet Inter-ORB, è necessario avere come informazioni il nome dell'host, la porta e il nome unico dell'oggetto (una sequenza di byte). MASIF specifica che la locazione sia costruita seguendo un formato URL (si veda RFC1738, "Uniform Resources Location", per la definizione dettagliata di URL) nel quale possono essere inserite in maniera agevole le informazioni necessarie. Anche in questo caso si rimanda a [MASIF97] per una specifica dettagliata della grammatica necessaria per costruire un URL e si fornisce semplicemente un esempio.

mafiiop://www.deis.unibo.it:1500/Bologna

Figura 17. Indirizzo Internet dal quale possono essere reperite le informazioni per costruire uno IIOP IOR. In pratica si ottiene il riferimento ad un oggetto CORBA localizzato sull'host www.deis.unib.it alla porta 1500 e di nome Bologna.

L'utilizzo di location di tipo Internet non esclude l'utilizzo di location CosNaming e viceversa. Dal MAFFinder è possibile ottenere location di ambo i tipi, è compito del client che sta effettuando la ricerca capire dal formato della stringa quale sia il giusto modo per risolvere la location. Per capire quale sia il tipo di informazione contenuta nella location, MASIF propone di scandire la stringa ottenuta fino ad ottenere il primo carattere ':' e controllare se tutto quello che precede è CosNaming o mafiiop. In base a questo si deciderà come trattare la location per ottenere il riferimento.

Per location di tipo Internet bisogna fare una considerazione. La costruzione di un riferimento partendo da un indirizzo Internet è un'operazione che va posta a livello di ORB. Le informazioni necessarie per la costruzione di uno IOR (host, port, e key dell'oggetto CORBA di cui si vuole ottenere il riferimento) sono in generale nascoste dall'ORB per dare una visione trasparente della locazione dei componenti del sistema.

Allocazione dei MAFFinder.

La situazione ideale sarebbe avere un unico MAFFinder al quale tutti si registrano e tutti possono ritrovare tutti. Questo non è possibile per motivi di efficienza e scalabilità. Come sempre nelle applicazioni di rete quando si vuole fornire un servizio efficiente ad un numero non determinato di utenti, e quando si vuole che l'efficienza di tale servizio non degradi all'aumentare degli utenti, bisogna trovare una soluzione che suddivida il

carico computazionale su più siti e che non introduca forti costi di coordinamento. Uno scenario che propone MASIF è di allocare un MAFAgentSystem in ogni regione, in questo modo, con le opportune ipotesi di località, si ha uno spettro limitato di agent system da servire. Per la ricerca di agenti che siano al di fuori della regione in cui il client (agente o meno) risiede è possibile adottare diverse tecniche. Si possono attuare ricerche esaustive in tutte le regioni (anche questa soluzione non certo scalabile), o richiedere semplicemente al MAFFinder che gestisce la regione del client, ipotizzando che gli agenti situati altrove si registrino al nostro MAFFinder se si vogliono fare trovare.

Quale sia il giusto modo di disporre le cose dipende da quello che si vuole ottenere, MASIF non si occupa di questo aspetto, che però e stato accennato per dare una visione più ampia dello scenario che si sta configurando.

Nomi delle classi.

Leggendo la specifica MASIF si trova messo in rilievo il problema dell'unicità dei nomi delle classi, ma la ricerca della soluzione viene lasciata agli implementatori. Quando un agente si sposta da un agent system ad un altro porta con se le classi che sono necessarie per permettere di creare la sua istanza. Se in un certo agent system arrivano due agenti che hanno nel loro bagaglio di classi due classi con lo stesso nome è necessario gestire le cose in modo che ognuno venga istanziato utilizzando le classi appropriate (in pratica non è possibile mettere insieme tutte le classi che arrivano dall'esterno proprio perché ci potrebbe essere collisione di nomi).

CAP 4.

PROGETTO DELL'INTEROPERABILITÀ IN SOMA

In questo capitolo verranno discussi gli aspetti e le problematiche di progetto incontrate nel tentativo di rendere SOMA un sistema aperto. CORBA e MASIF saranno gli standard che consentiranno il raggiungimento dell'interoperabilità. Nel capitolo tre è stato analizzato in che modo CORBA possa consentire l'interazione tra sistemi ad agenti differenti. Nel seguito verrà descritta la relazione che esiste tra SOMA e CORBA e nel capitolo cinque si dirà come questa relazione sia stata rafforzata. Dopo questo primo passo entrerà nel discorso anche MASIF. La sua implementazione, che consentirà di superare alcuni limiti di CORBA (tra cui l'assenza di mobilità di codice), doterà SOMA di un ulteriore grado di apertura.

4.1. L'interoperabilità tramite CORBA

Il paradigma di comunicazione di CORBA è di tipo client server, si tratta quindi di un modello asimmetrico. I soggetti della comunicazione possono quindi essere classificati in clienti e in servitori. Per essere un server CORBA è necessario essere un oggetto CORBA registrato all'ORB; per essere un client è necessario disporre di un riferimento ad un oggetto CORBA e di uno strato software che consenta le invocazioni sull'oggetto.
Per capire la relazione che lega CORBA e SOMA è necessario comprendere come gli agenti SOMA possano essere clienti o servitori di CORBA. Una volta ottenuta questa possibilità si potrà considerare resa standard la comunicazione tra agenti SOMA e una qualsiasi altra entità CORBA. Sarà quindi possibile l'interazione con agenti e risorse di qualunque tipo purché aderenti allo standard.

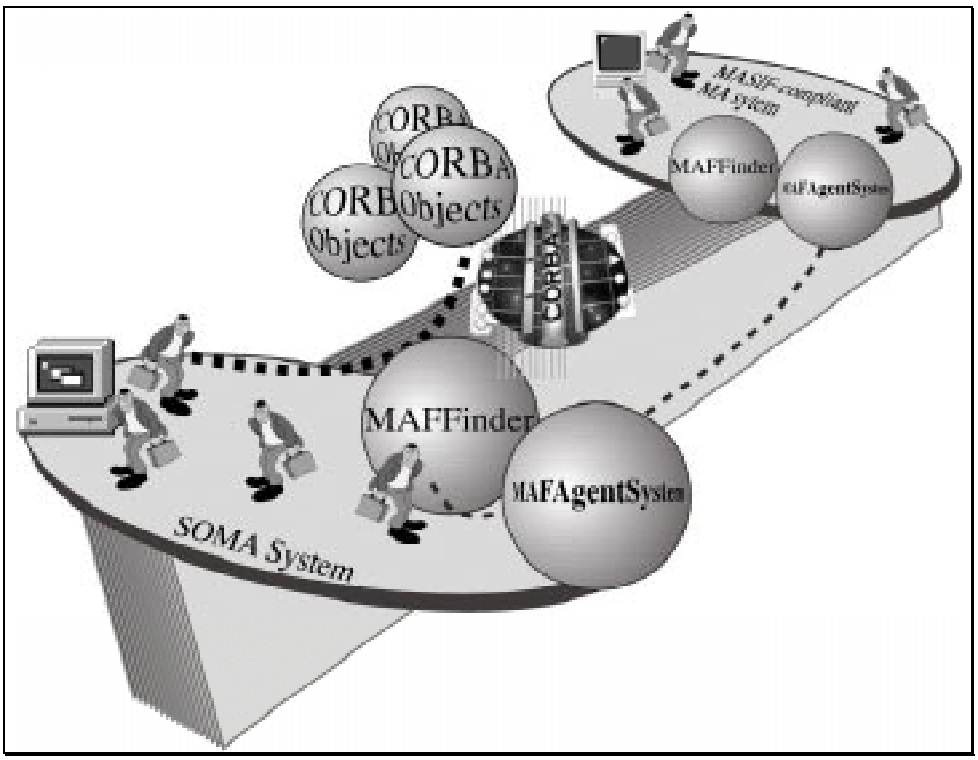

Figura 18. L'interoperabilità in SOMA si raggiunge attraverso CORBA e MASIF.

4.1.1. Agenti servitori di CORBA

Per capire se un agente SOMA può essere un server CORBA bisogna comprendere cosa sia un oggetto CORBA, solo questi ultimi possono infatti essere registrati all'ORB e diventare servitori.

Come spiegato nel capitolo uno, CORBA consente due modalità con cui le invocazioni possono avvenire: modalità statica e modalità dinamica. In base a questa distinzione è possibile individuare due tipologie fondamentali di oggetti CORBA, quelli che si appoggiano allo skeleton statico e quelli sostenuti dallo skeleton dinamico. A seconda del tipo di server che si vuole realizzare si dovrà procedere seguendo differenti processi di sviluppo, ma si otterrà comunque il risultato che la classe scritta sarà una discendente di org.omg.CORBA.Object.

A questo punto è lecito chiedere se un agente SOMA possa essere un discendente di questa classe. Un agente SOMA in generale deve essere discendente della classe AgentSystem.Agent e quindi (non essendo consentita l'ereditarietà multipla in Java) non può essere un oggetto CORBA.

A questo punto sembrerebbero chiuse tutte le strade, esistono però diversi modi per superare questo problema. La prima soluzione deriva dalla considerazione che in realtà non è strettamente necessario che l'agente sia direttamente registrato all'ORB. Si potrebbe ad esempio pensare di racchiudere la parte di codice per la quale è prevista la pubblicazione su CORBA, in un oggetto CORBA indipendente, e fare diventare l'agente il mezzo di trasporto per questo oggetto. In questo modo l'agente potrà muoversi e decidere, una volta arrivato in un certo sito, di istanziare e registrare l'oggetto che diventerà il server CORBA al suo posto. In sostanza si progettano server CORBA secondo la normale metodologia (statica o dinamica) i quali verranno poi istanziati e registrati dagli agenti SOMA.

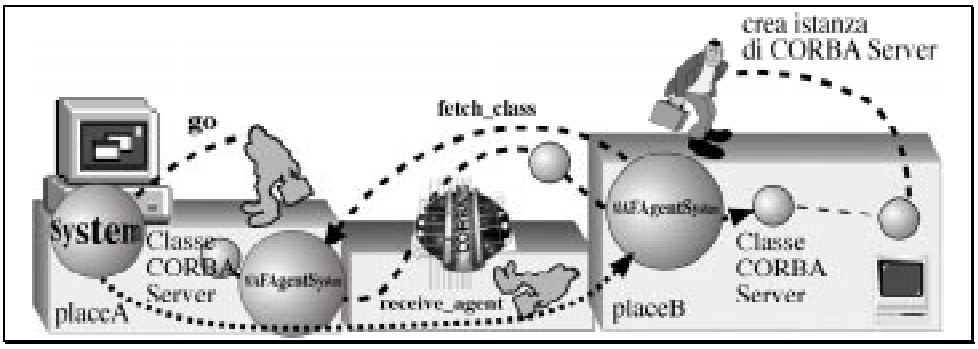

Figura 19. L'agente migra, il meccanismo di caricamento remoto delle classi provvede a reperire il codice dell'oggetto CORBA e l'agente può creare l'istanza e registrare all'ORB.

Gli oggetti CORBA non sono serializzabili, non è quindi possibile migrare dopo averne creato un'istanza. Eventualmente è possibile abbandonarli dopo averli istanziati (bisogna gettare il riferimento ottenuto dalla creazione). Nel caso siano stati creati come oggetti CORBA

persistenti (si veda paragrafo 1.1.5) continueranno il loro lavoro, altrimenti termineranno.

Pur avendo mostrato che l'interazione degli agenti SOMA con CORBA è possibile, è bene continuare l'esplorazione di ulteriori alternative che potrebbero risultare più adeguate a particolari esigenze di programmazione.

Analizzando l'implementazione di CORBA utilizzata [Visibroker98] si scopre la possibilità di fare uso di un meccanismo che consente la registrazione a CORBA, di oggetti la cui classe non sia discendente di org.omg.CORBA.Object. In realtà si tratta di un trucco che fa uso di un oggetto intermedio il quale viene registrato all'ORB al posto dell'oggetto che dovrebbe diventare il server CORBA. L'intermediario riceve le richieste provenienti dall'ORB e le invia all'implementazione del servizio che, non essendo registrata direttamente, può non essere un oggetto CORBA. Con questa tecnica è come se l'agente fosse direttamente collegato all'ORB; il codice eseguito durante le invocazioni è quello racchiuso all'interno dell'agente stesso.

Figura 20. L'intermediario è il vero oggetto CORBA, l'agente riceve le richieste come invocazioni locale fatte dall'intermediario.

Il processo di sviluppo da seguire per ottenere questo risultato non è differente da quello normale, l'unica variazione è costituita dalla registrazione all'ORB. Normalmente una volta scritto il codice di un oggetto CORBA è necessaria la stesura di una parte di codice che provveda a crearne un istanza e a registrarla all'ORB. In questo caso ci si

dovrà preoccupare anche di creare l'istanza dell'oggetto intermedio e fornire ad esso il riferimento all'agente.

Un altro modo per ottenere la registrazione di agenti SOMA all'ORB è far si che essi non debbano estendere la classe AgentSystem.Agent. Di questo si parlerà in seguito quando si tratterà dell'implementazione di MASIF. La possibilità di consentire ad agenti di non essere legati ad una particolare classe servirà per accettare agenti provenienti da altri sistemi. Questa nuova caratteristica aggiunta a SOMA rende possibile la costruzione di agenti a partire da oggetti qualunque e quindi anche da oggetti CORBA. Potenzialmente gli agenti costruiti in questo modo possono ancora effettuare migrazioni (cambia solamente la modalità di invocazione della go), a patto che siano però costruiti a partire da oggetti serializzabili. Gli oggetti CORBA non sono serializzabili, di conseguenza procedendo in questo modo è possibile ottenere agenti che si possono registrare direttamente all'ORB, ma che devono essere stazionari. Potranno quindi usufruire di tutti i servizi di place e di comunicazione che sono offerti normalmente ad un agente e potranno in più decidere di pubblicarsi nel mondo CORBA, ma non potranno mai migrare.

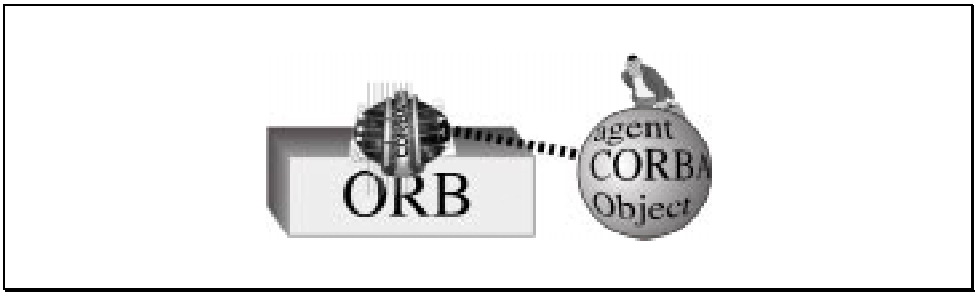

Figura 21. L'agente può registrarsi all'ORB ma non può migrare perché essendo un oggetto CORBA non è serializzabile.

4.1.2 Agenti clienti CORBA

Per essere clienti CORBA non occorre nessuna caratteristica particolare, un qualunque oggetto può essere in grado di effettuare delle invocazioni a partire da un riferimento ad un server CORBA. Il client potrà essere un agente, un server CORBA, un applet Java, l'unica cosa che bisogna essere in grado di fare è ottenere un riferimento all'oggetto che si intende utilizzare nell'invocazione, dopo di che in base alla modalità di invocazione scelte (statica o dinamica) si dovranno svolgere le adeguate operazioni.

Figura 22. Un agente SOMA che invoca un servizio su un server CORBA.

4.2. MASIF e SOMA: scelte di progetto

Accanto all'integrazione tra CORBA e SOMA, va posto MASIF come mezzo per raggiungere un ulteriore livello di interoperabilità. Lo standard MASIF è stato ampiamente discusso nel capitolo tre. In seguito verranno descritte le ipotesi aggiuntive imposte ed esso e le scelte di progetto effettuate durante la sua implementazione.

4.2.1. Linguaggio e tipo di serializzazione

MASIF vuole essere uno standard aperto e lo dimostra nella sua specifica sotto diversi aspetti. Tra questi quello che pone le più grosse difficoltà, è la libertà lasciata nella scelta del linguaggio di programmazione di codifica degli agenti.

Le grandi problematiche introdotte da questo grado di libertà (si vedano paragrafi 2.3. e 3.3.) vengono per ora eliminate scegliendo di accettare agenti esclusivamente codificati in Java. Questo vincolo è necessario in quanto è il primo lavoro di apertura del sistema, in un secondo passo sarà possibile riprendere in esame la possibilità di gestione di linguaggi diversi. Pur essendo la limitazione imposta molto restrittiva in generale, risulta un po' più accettabile nel nostro caso, essendo Java il linguaggio utilizzato. Costruendo un sistema basato sulla JVM si ottiene la possibilità di gestire agenti codificati in linguaggi diversi da Java semplicemente utilizzando compilatori in grado di generare byte code Java a partire dai linguaggi utilizzati dagli agenti. Non sono necessarie modifiche al sistema e si ottiene così un sistema aperto in grado di accogliere agenti codificati in linguaggi non ancora esistenti (a patto che esista la possibilità di compilare il linguaggio per ottenere il byte code Java).

Un secondo aspetto che MASIF lascia non specificato è il modo in cui lo stato di esecuzione degli agenti debba essere memorizzato prima di essere trasferito, in particolare è previsto che lo stato sia memorizzato in un vettore di byte, ma nulla è detto su come questi byte siano strutturati. Utilizzando Java risulta comodo scegliere come formato di memorizzazione la serializzazione che il linguaggio stesso mette a disposizione. In questo modo ci si chiude ad altre forme di serializzazione, ma si ritiene che avendo posto già il vincolo di accettare agenti scritti in Java, questi possano sempre utilizzare la serializzazione proposta per trasferirsi.

Dato che il sistema ad agenti da estendere considera lo stato di esecuzione di un agente composto dalle variabili, più un metodo da cui riprendere l'esecuzione, si pone come ulteriore vincolo che il vettore di byte contenga ambedue le informazioni. Il metodo da cui riprendere l'esecuzione sarà rappresentato da una stringa (che sarà poi serializzata), le variabili dell'agente saranno automaticamente salvate serializzando l'agente stesso. Quello che ci si aspetta al momento della deserializzazione è di ottenere una stringa ed un oggetto contenente il

metodo specificato dalla stringa. In questo modo non si richiede nulla sulla classe di appartenenza dell'agente, in quanto ci si aspetta la cosa più generica che in Java si possa ottenere: un oggetto.

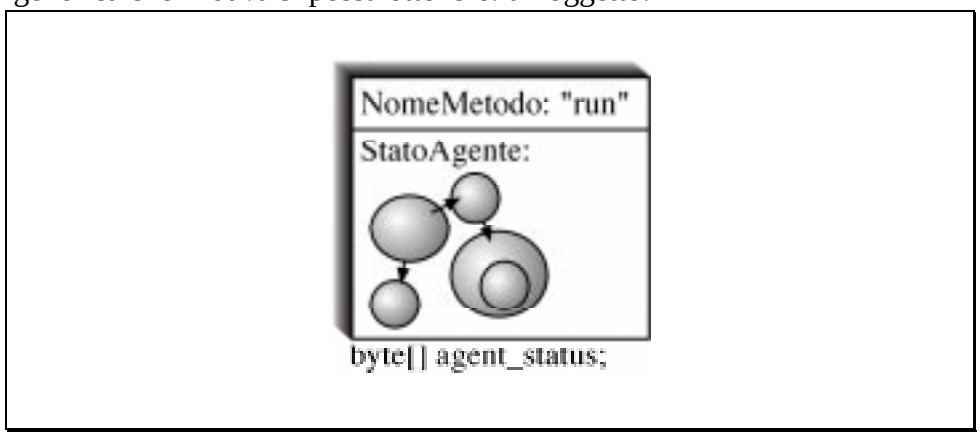

Figura 25. Lo stato dell'agente è costituito da una parte dati e da uno stato di esecuzione. La parte dati è costituita dal grafo degli oggetti collegati all'agente che viene appiattito mediante la serializzazione Java. Lo stato di esecuzione si riduce al nome del metodo da eseguire una volta giunti nel sito di destinazione.

Bisogna dire qualche cosa anche a proposito della creazione che deve potere essere utilizzata anche da clienti che non siano agent system. Quando si crea un nuovo agente viene fatta l'ipotesi che il creatore specifichi nei parametri della create_agent il nome del metodo da cui l'agente deve iniziare, questo indipendentemente dal fatto che il client sia o meno un agent system. Se poi il cliente non è un agent system, e quindi non può rendere disponibile un metodo di fetch class, ci si aspetta di trovare nei parametri della creazione la definizione di tutte le classi necessarie per l'esecuzione del nuovo agente. Queste informazioni (nome del metodo iniziale e definizione delle classi) non sono esplicitamente previste tra i parametri della create_agent, ma possono essere inserite in uno di essi, che è stato appositamente pensato proprio per aggiungere eventuali parametri necessari. Tale parametro per essere il più generico possibile consiste di un vettore di byte.

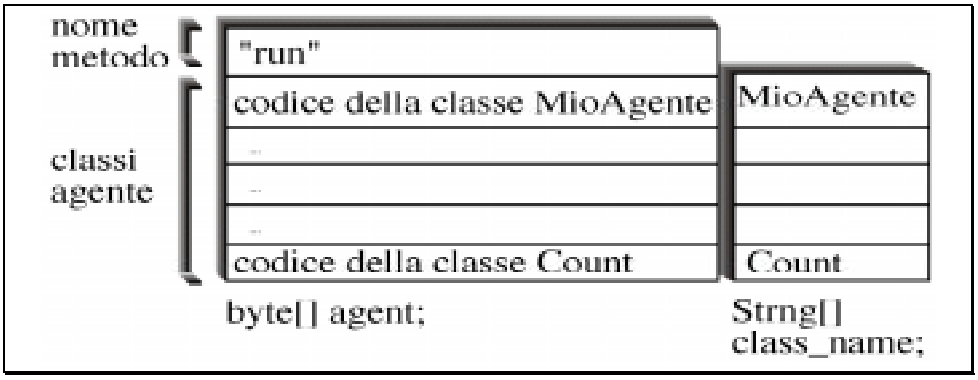

Figura 26. Il codice delle classi necessarie alla creazione di un agente sono inserite in un vettore di byte una di seguito all'altra.

Utilizzando la deserializzazione di Java per ottenere gli elementi inseriti nel vettore di byte, ci si aspetta di ottenere una stringa come primo oggetto e una serie di vettori di byte, tanti quanti i nomi specificati nella lista di classi (parametro della creazione), e ognuno costituente il bytecode di una classe (ordinati come nella lista dei nomi).

4.2.2. Supporto di esecuzione per gli agenti

MASIF si occupa esclusivamente di porre un'interfaccia standard tra i diversi agent system, ma non fornisce uno standard per l'interfaccia che il supporto deve mostrare agli agenti. In attesa di un ulteriore livello di specifica di quali siano le operazioni che un agent system deve mettere a disposizione degli agenti, ma volendo cercare di realizzare un'implementazione che sia utilizzabile già da ora anche in un contesto di eterogeneità, si ipotizza l'utilizzo diretto delle interfacce MASIF da parte degli agenti (si era accennato di questa possibilità già nel paragrafo 3.5.4). Essendo il MAFFinder e il MAFAgentSystem già standard si è deciso di utilizzarli anche come interfacce di supporto agli agenti per migrare o effettuare altre operazioni in sistemi a loro sconosciuti. Per fornire flessibilità all'implementazione saranno consentite differenti modalità di

migrazione (ed in generale di interazione col sistema) in base al grado di conoscenza di SOMA che gli agenti possiedono:

- un agente anche se estraneo a SOMA può **essere a conoscenza** dell'interfaccia proprietaria che SOMA stesso mette a disposizione ai suoi agenti. Un eventuale standard futuro potrà specificare un interfaccia standard del supporto dei sistemi ad agenti. In questo modo si potrà supporre che tutti gli agenti siano a conoscenza di qualsiasi supporto;

- un agente estraneo a SOMA può **non essere a conoscenza** di quali siano i metodi del supporto non standard SOMA. Questi agenti avranno la possibilità di interagire col sistema utilizzando l'interfaccia MASIF;

Il secondo punto si basa sull'utilizzo del MAFAgentSystem come interfaccia standard conosciuta da tutti gli agenti, per consentire la migrazione o altre funzionalità. Se ad esempio un agente vuole migrare, nel modello proposto da MASIF deve invocare un metodo di sistema, sarà poi il sistema stesso a provvedere a tutte le operazioni necessarie. Tra queste operazioni troveremo sicuramente l'invocazione della receive_agent del MAFAgentSystem corrispondente all'agent system di destinazione. Se si rende un agente in grado di fare tutto ciò che il supporto faceva per lui a fronte di una richiesta di migrazione in un sistema straniero, non è più necessaria l'invocazione al metodo di sistema. In pratica si ottiene il risultato che un agente è in grado di migrare autospedendosi da un agent system ad un altro, utilizzando solo la receive_agent come chiamata di sistema (la quale risulta standard).

Con un comportamento di questo tipo un agente è in grado di richiedere all'agent system in cui è giunto tutte quelle informazioni che possono essere ottenute invocando i metodi dell'interfaccia MAFAgentSystem. MASIF specifica, oltre alle funzioni per il management e la migrazione di agenti, un metodo per ottenere le

informazioni relative ad un agent system. Invocando get_agent_system_info si ricava una struttura che contiene il tipo del sistema ad agenti, il linguaggio e il tipo di serializzazione utilizzato da esso, informazioni relative alla sicurezza ed anche una lista di proprietà i cui elementi sono costituiti da coppie nome-valore. Sfruttando queste proprietà è possibile ipotizzare anche un grado di conoscenza degli agenti intermedio tra i due elencati. Un agente può non sapere quali siano i nomi delle funzioni che permettono l'interazione con l'agent system, ma potrebbe essere a conoscenza di un insieme di nomi di proprietà alle quali ogni sistema è libero di associare un valore diverso. Ottenendo i valori corrispondenti al nome che interessa è possibile risalire a metodi specifici del sistema in cui si è giunti ed in questo modo utilizzare anche funzioni di tipo nativo che sono di norma le più efficienti.

Per decidere i nomi delle proprietà si dovrebbe dare uno standard che associ a ciascuna di esse la giusta semantica. Se ci fosse stato uno standard completo di questo tipo lo si sarebbe potuto utilizzare direttamente per l'interfaccia del supporto. L'utilizzo di un insieme di nomi ai quali è associato un valore, diventa utile nel caso si voglia far avvenire l'interazione, magari temporanea, tra due o più sistemi, senza avere bisogno di definire uno standard completo, e senza dovere conoscere i nomi dei meccanismi proprietari di tutti i sistemi nei quali ci si vuole spostare (basta sapere semantica e nomi delle proprietà).

4.2.3. Individuazione degli agent system in SOMA

La struttura gerarchica costituita da place e domini di SOMA è già stata introdotta nel paragrafo 2.7.1, ora si vuole entrare un po' più nel dettaglio di cosa sia un place e di quale sia la sua corrispondenza con il concetto di place che MASIF propone.

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. Ci sono tre moduli che

interagiscono con gli Agenti (e che sono stati descritti nel paragrafo 2.7.1.1):

- l'Agent System;
- il Place Manager;
- l'Information Service.

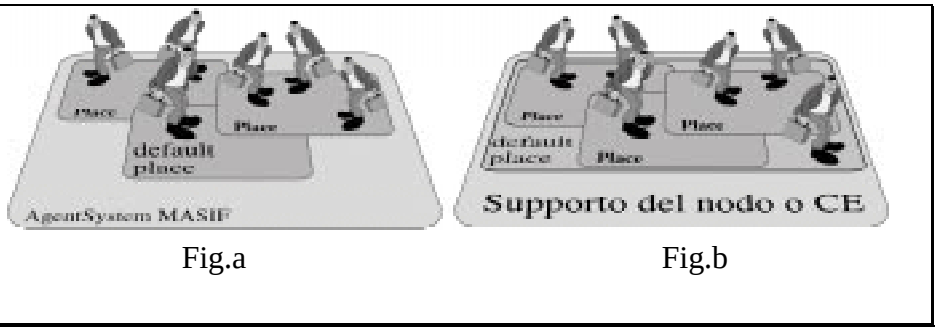

Fig.a Fig.b

Figura 27. a) Un agent system MASIF è un CE (o anche un supporto del nodo) che al suo interno contiene delle località chiamate place dove è possibile trovare differenti tipi di servizi (esiste sempre almeno un place chiamato di default). b) SOMA non fornisce l'astrazione corrispondente all'agent system MASIF, ma fornisce uno scenario costituito semplicemente da place che possono contenere a loro volta altri place.

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.

L'agent system MASIF trova la corrispondenza nel supporto di nodo che SOMA tende a far coincidere con il place di default che esso contiene. Si noti che i place locali costruiti all'interno del place di default non si possono mettere in relazione diretta con i place MASIF in quanto

l'accesso ad uno di essi deve avvenire sempre passando prima per il default place. In pratica se si fa coincidere il place di default SOMA con l'agent system MASIF (si noti che questo significa identificare il place di default con il supporto di nodo, cosa non scontata), non è comunque possibile, durante le migrazioni, ottenere l'accesso diretto ad un place locale. MASIF specifica invece come destinazioni delle migrazioni i place ed eventualmente propone l'utilizzo di un place di default soltanto se non esiste l'astrazione di place nel sistema in cui si è giunti.

In questo contesto sono possibili due modalità di identificazione degli agent system MASIF nel sistema SOMA. Un primo modo di vedere le cose associa l'agent system con il place di default (e i place MASIF non trovano una diretta corrispondenza nei place locali). Un secondo scenario è quello che vede associati i place SOMA con i place MASIF, e correla gruppi di place SOMA con un agent system. Seppure il primo scenario sembri essere il più aderente all'idea di agent system che MASIF propone, la seconda situazione si rivelerà essere più flessibile ed in grado di soddisfare un numero maggiore di esigenze.

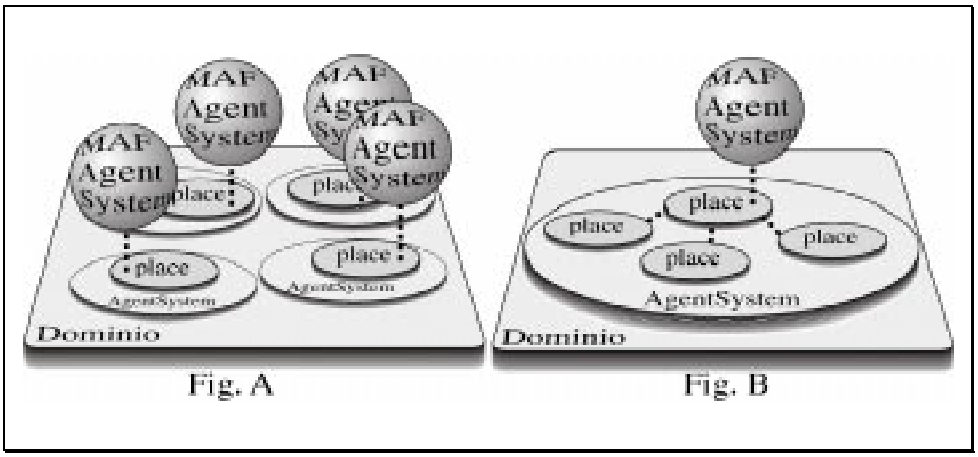

Figura 28. a). Ogni place SOMA diventa un agent system. b).Un agent system MASIF racchiude differenti place SOMA e consente così la gestione anche di un intero dominio. Il place accanto al quale risiede il MAFAgentSystem diventa il place gestore dell'agent system.

Nel caso si utilizzi l'astrazione di agent system per raggruppare un insieme di place (fig. b) si ottengono diversi aspetti positivi tra cui: la possibilità di non appesantire tutti i place del dominio con l'implementazione di MASIF; la possibilità di nascondere un intero dominio (mappato probabilmente in una rete LAN) entro un unico agent system; la possibilità di controllare l'accesso degli agenti stranieri in un unico punto, con vantaggi per la sicurezza del sistema; ed inoltre otteniamo una soluzione che è certamente una generalizzazione della prima in quanto facendo coincidere i gruppi di place con un solo place si ricade al caso particolare di associazione tra agent system e place.

Per questi motivi d'ora in avanti quando si parlerà di agent system si intenderà un gruppo di place di default gestiti da un unico place coordinatore o gestore.

Da notare che alcuni dei vantaggi elencati (quali l'accentramento dell'ingresso di agenti stranieri in un solo host e la possibilità di non appesantire tutti i place del dominio) sarebbe stato possibile ottenerli anche associando place di default ad agent system, installando un solo sistema MASIF per dominio. In questo caso però la gestione attraverso il MAFAgentSystem sarebbe stata limitata agli agenti di un solo place, inoltre si sarebbe persa la capacità di far migrare gli agenti tra i place del dominio attraverso l'utilizzo dei meccanismi standard MASIF (in sostanza, gli agenti una volta entrati avrebbero dovute migrare esclusivamente in modo nativo).

4.2.4. Tipologia degli agenti che SOMA è in grado di accogliere

Una volta mappato l'agent system di MASIF in un insieme di place SOMA uno dei quali sia il gestore, è possibile chiedersi come si possa estendere il supporto per consentire la gestione di agenti provenienti da altri sistemi, potenzialmente diversi dagli agenti SOMA.

Prima però di entrare nei dettagli dei meccanismi che esistono e che vanno estesi, bisogna analizzare la situazione di cosa ci si debba aspettare

provenire dall'esterno. MASIF non specifica cosa debba essere un agente. Noi abbiamo già limitato il campo ad agenti scritti in Java, e nel tentativo di essere il più generici possibili all'interno di questo linguaggio abbiamo definito un agente come un oggetto Java. In questo modo noi siamo aperti verso l'esterno ad un qualunque tipo di agente scritto in Java.

Da un altro lato dobbiamo anche essere pronti ad accettare agenti SOMA che provengono da fuori. Potrebbero essere nostri agenti che ritornano, o agenti di altri sistemi SOMA che sono però al di fuori della nostra configurazione di place e domini e che usano il canale MASIF come ponte di trasmissione, o ancora (in futuro) potrebbero essere gli agenti frutto di una ulteriore specifica di MASIF.

Sembrerà strano che ci si chieda di essere pronti alla ricezione di agenti SOMA quando si presuppone di essere aperti a qualunque oggetto Java È naturale che un agente SOMA sia un oggetto Java dato che il sistema ad agenti è interamente scritto in questo linguaggio. Se si è in grado di ricevere un oggetto Java non dovrebbero esistere altri problemi nel ricevere un agente SOMA. Effettivamente una volta che si è esteso il sistema ad agenti per ricevere oggetti Java, è sicuramente possibile trattare gli agenti SOMA provenienti dall'esterno, nello stesso modo in cui si trattano gli agenti di qualunque altro tipo. Anche se però è possibile trattare un agente SOMA come se fosse un qualsiasi oggetto Java, è preferibile riconoscerlo per consentire ad esso l'utilizzo più semplice e più efficiente dei meccanismi di sistema. Non è ancora stato specificato come SOMA farà per adattare gli agenti di altri sistemi ad eseguire nel suo ambiente, ma penso sia intuibile che un agente che esegue nel suo sistema debba avere in generale un più ampio spettro di possibilità (o comunque di semplicità di utilizzo dei meccanismi di base) rispetto ad un agente che esegue in un sistema diverso dal suo.

4.2.5 Come trattare agenti non SOMA

L'estensione da apportare vuole essere un vero ampliamento del sistema ad agenti, e vuole soprattutto consentire il massimo riutilizzo di tutti i

meccanismi già sviluppati. Un agente esterno (o straniero) deve avere, in accordo con la politica specificata, pari opportunità di sfruttamento delle risorse potenzialmente utilizzabili (evidentemente le risorse con interfaccia non standard non potranno venire usate a meno non si ipotizzi un certo grado di conoscenza dei meccanismi proprietari). Essendo nel sistema ad agenti già presenti tutti i meccanismi di sicurezza si vuole poterli sfruttare e non doverli riscrivere.

Per ottenere un tale risultato è necessario che l'agente straniero sia riconosciuto a tutti gli effetti come un agente del sistema stesso. Gli agenti che vengono trattati in tale modo (ossia quelli che fino ad ora sono stati chiamati agenti SOMA) sono quelli che appartengono alla classe Agent, specificata dal sistema. Non volendo richiedere ad un agente straniero di appartenere a questa classe (si ricorda che si richiede solamente sia un oggetto Java), si è deciso di utilizzare un agente di trasporto (classe TransportAgent). L'agente di trasporto è un agente SOMA e può quindi essere gestito dal sistema (per i dettagli della struttura di questo agente si veda il paragrafo 5.7.1.).

4.2.6. Agenti non SOMA e migrazioni

In base alla distinzione che è stata fatta in precedenza a proposito delle conoscenze che un agente può avere sui meccanismi SOMA, sono state messe a disposizione diverse modalità di migrazione tra un place all'altro dello stesso agent system per gli agenti stranieri.

Se gli agenti stranieri sono SOMA-based possono certamente utilizzare le funzionalità di tipo nativo che sono direttamente disponibili sotto forma di metodi che fanno parte dell'agente stesso (è evidente che anche questi agenti possono utilizzare le altre modalità che verranno esposte di seguito, ma non ne hanno il motivo, dato che sono a conoscenza dei metodi nativi che consentono migrazioni più semplici ed efficienti).

Nel caso in cui un agente non sia SOMA si possono distinguere due possibili modi di migrare, in base al grado di conoscenza di SOMA:

- migrazione nativa;

- migrazione tramite CORBA e il MAFAgentSystem.

4.2.6.1. Migrazione nativa

La migrazione nativa è utilizzabile nei casi in cui l'agente straniero sia a conoscenza diretta del suo utilizzo o nel caso in cui sia in grado di acquisire la conoscenza facendo uso delle proprietà di sistema (come spiegato nel paragrafo 4.2.2.). Essendo gli agenti stranieri non SOMA incapsulati in un agente di trasporto, per realizzare la migrazione nativa si usa un meccanismo che, a fronte di una richiesta dell'agente straniero, induce la migrazione dell'agente di trasporto, e di conseguenza dell'oggetto trasportato. Se un agente straniero vuole usare la migrazione nativa per muoversi deve istanziare un oggetto di classe NativeSystemUtilities. Questa classe può essere istanziata da un agente straniero per utilizzare metodi nativi del sistema ad agenti. Due le modalità: conoscenza pregressa del nome della classe oppure invocazione del metodo get_agent_system_info() dello standard MASIF, e utilizzo della lista delle proprietà di sistema comprese nei parametri di ritorno (lista costituita da coppie nome valore). In tale lista sarà inserito il nome di questa classe. La lista delle proprietà non è utilizzata in nessuna funzione di MASIF, ma è prevista per evoluzioni future. In questa sede si propone una piccolo standard che fa corrispondere al nome NativeUtility il valore NativeSystemUtilities, che dovrà essere interpretato come il nome della classe da istanziare per potere utilizzare metodi nativi del sistema ad agenti. Una volta ottenuta la classe è possibile procurarsi attraverso Java i nomi dei metodi che la classe stessa propone (oppure ottenerli mediante l'utilizzo di altre proprietà da definire). La migrazione può quindi avvenire mediante l'invocazione del metodo go sull'oggetto istanziato a partire dalla classe NativeSystemUtilities.

Un’altra proprietà che si propone è NativeUtilityDoc, al quale si vuole fare corrispondere il nome di un metodo da invocare sulla classe che fornisce le utility native per ottenere ulteriori informazioni (in più di quelle che si possono ottenere con l’introspezione di Java). Associando ad ogni nome di proprietà una semantica è possibile ottenere un piccolo nucleo standard facilmente ridefinibile che può essere utilizzato per l’interazione con agenti provenienti dai sistemi con i quali si sono concordate le proprietà.

4.2.6.2. Migrazione tramite MAFAgentSystem

A riguardo della migrazione attraverso l’utilizzo diretto dell’oggetto MAFAgentSystem bisogna tenere in considerazione che un agent system MASIF è composto da un insieme di place SOMA, e che l’invocazione di un metodo del MAFAgentSystem giunge sempre al place che fa da gestore per quell’agent system. Un agente che voglia cambiare place ma non agent system deve rivolgere le sue richieste di migrazione sempre allo stesso MAFAgentSystem.

Supponiamo che esistano tre place A, B e C (ognuno su un diverso host) all’interno dello stesso dominio, che A sia il gestore, e che in B sia presente un agente straniero che vuole migrare in C. L’agente invocherà il metodo receive_agent() dell’oggetto CORBA MAFAgentSystem (che è situato in A) passando il suo stato come parametro. In questo modo avremo in pratica trasferito da B ad A l’agente attraverso la chiamata remota. A questo punto il place gestore dovrà provvedere a rispedire l’agente sul nodo C, in modo da completare l’operazione di migrazione.

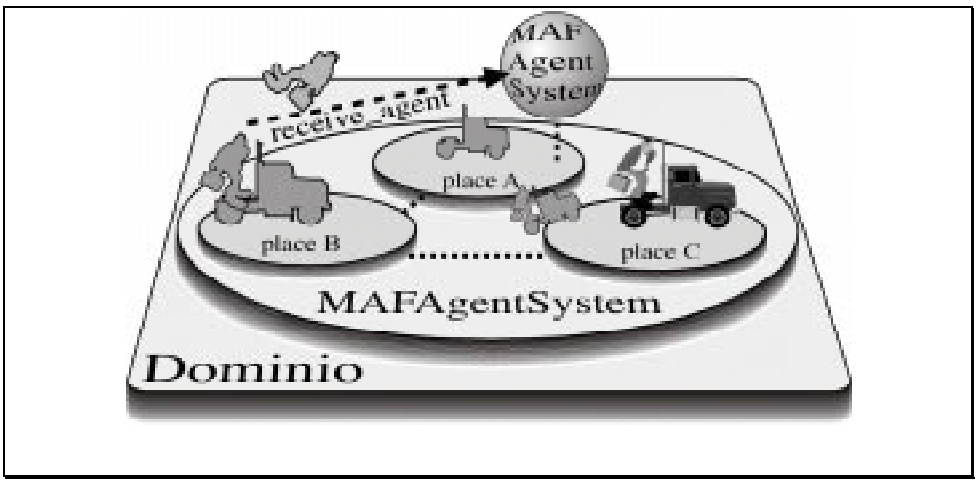

Figura 29. L'agente su B invocando il metodo del MAFAgentSystem si trova trasferito su A. Avendo specificato di andare su C, il MAFAgentSystem, utilizzando il livello di trasporto del place A, lo spedisce su C. Da notare che durante il trasferimento da B ad A l'agente di trasporto viene perso.

Da questo piccolo scenario si capisce come ogni trasferimento richiede due passaggi sulla rete (se consideriamo ogni place su un diverso host). Inoltre quello che viene trasferito nel primo passo attraverso la chiamata CORBA, è solo l'agente straniero, in quanto l'agente di trasporto non è percepito in alcun modo dall'oggetto trasportato che serializzerà solo il proprio stato. Il MAFAgentSystem dovrà quindi ricostruire l'agente di trasporto prima di poter spedire il tutto al nodo C.

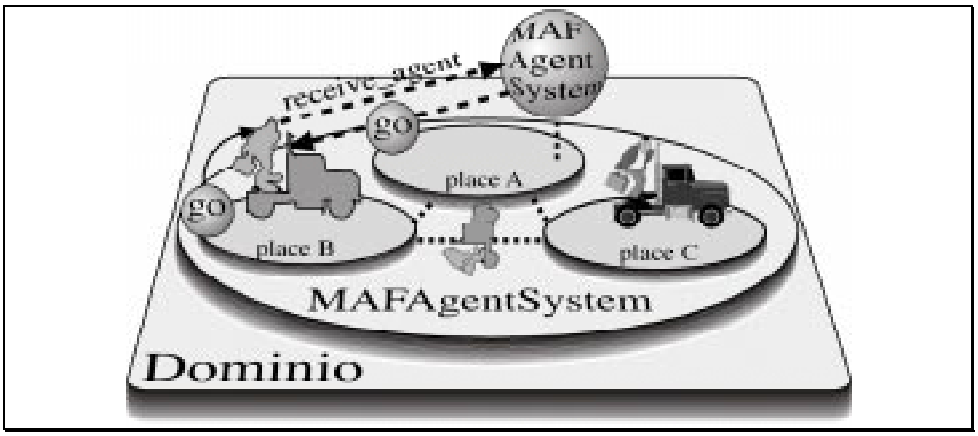

Figura 30. L'agente su B richiede al MAFAgentSystem su A di migrare. Il MAFAgentSystem risponde mandando un comando in grado di indurre la migrazione su C.

Se la risposta da parte del MAFAgentSystem all'invocazione dell'agente straniero su B, del metodo receive_agent(), fosse stata l'invio di un comando al place B che inducesse la migrazione da B a C dell'agente, avremmo ottenuto un altro tipo di comportamento. In questo caso le latenze di rete sarebbero state tre, ma si sarebbe potuto conservare l'agente di trasporto. Questa soluzione risulta poco praticabile in quanto essendo la chiamata receive_agent() l'ultima istruzione che un agente fa su un certo place, dopo di essa l'agente sarebbe eliminato dal gestore locale, e il comando di migrazione giungerebbe quando l'agente non esiste più. Questo a meno che non si utilizzino particolari stratagemmi, come ad esempio ritardare la terminazione dell'agente, aggiungendo un ritardo fisso, come ultima istruzione nel metodo della classe TransportAgent che viene sempre invocato (e che poi discrimina quale è il metodo da invocare sul vero agente). In questo modo il ritardo terrebbe fermo l'agente che non concludendo le istruzioni del metodo in esecuzione non sarebbe eliminato.

La scelta è comunque ricaduta sulla prima soluzione che è più efficiente, e pur perdendo, ad ogni migrazione, lo stato dell'agente di trasporto, le informazioni gettate via sono poco significative per gli agenti che migrano utilizzando il MAFAgentSystem in maniera diretta. Tali agenti sono infatti quelli che non hanno nessuna conoscenza della struttura di SOMA e non sono quindi in grado di sfruttare le potenzialità caratteristiche di questo sistema ad agenti (come ad esempio la mailbox). Gettare via i messaggi accumulati nella mailbox quando si migra può quindi essere tollerato.

Nel caso in cui l'agente utilizzi la migrazione nativa, la mailbox viene conservata. Questo è in perfetto accordo con il grado di conoscenza che

un agente ha di SOMA, se infatti sta usando dei metodi proprietari per migrare, sarà probabilmente anche in grado di usare metodi proprietari per scambiare mail con altri agenti (e non sarebbe tollerabile gettare la mailbox).

4.2.7. Modello per il management di agenti remoti

Nella specifica MASIF sono previste le funzioni per consentire il management degli agenti. Le più significative sono le operazioni di sospensione, terminazione, riattivazione e rilevazione dello stato di un agente. Vi è poi la creazione che però è forse da considerare più vicina alla funzione di ricezione, che a quelle di management. L'implementazione della creazione di un agente comporta problematiche legate alla mobilità del codice, in quanto in generale quando si crea un agente è necessario l'invio delle classi che serviranno per generare una nuova istanza dell'agente. Per questo, anche se spesso si considera la funzione di creazione come una funzione di management, non verrà trattata in questo contesto.

Le richieste di management provenienti dal mondo esterno passano sempre attraverso il MAFAgentSystem che funge da interfaccia standard. Come abbiamo specificato in precedenza, il MAFAgentSystem è situato in un place gestore. Ecco allora che per il management degli agenti occorre una gestione distribuita effettuata dal place gestore.

Per capire quale modello si possa utilizzare per sospendere, terminare o riattivare un agente che si sposta da un place all'altro, è necessario comprendere quali siano i problemi che nascono quando si cerca di fermare o raggiungere un'entità in movimento.

Prima di tutto bisogna specificare cosa succede quando un agente si muove da un place all'altro dello stesso agent system, se ad esempio l'agente notifica i suoi spostamenti, oppure se i place che attraversa tengono traccia del suo passaggio, oppure ancora se non si sa nulla dell'agente che si muove. Questo è in realtà un grado di libertà che

abbiamo per progettare il modello, e che verrà vincolato dopo avere presentato uno scenario che sarà utilizzato per descrivere i problemi da superare.

Consideriamo quattro place denominati Parigi, Roma, NewYork, Firenze e scegliamo NewYork come gestore. Immaginiamo che un agente sia arrivato dal mondo esterno a NewYork e che poi abbia cominciato a girare tra Parigi, Roma e Firenze circolarmente senza mai fermarsi più del necessario per compiere un'azione (come visitare un monumento). Immaginiamo di volere fermare il nostro agente avendo a disposizione un mezzo di comunicazione della stessa efficienza del mezzo di locomozione che l'agente usa per spostarsi da un place all'altro, e ipotizzando anche che l'agente non comunichi i suoi spostamenti né a NewYork e neanche a nessuno degli altri place. Supponiamo di sapere quale sia il primo place che l'agente ha raggiunto quando ha lasciato NewYork (per esempio Parigi). Potremmo sperare che l'agente sia ancora a Parigi e mandare un messaggio per dire all'agente che deve fermarsi, ipotizzando che l'operazione abbia avuto successo. In questo contesto sorge spontaneo chiedersi se l'entità che si spedisce per fermare l'agente sia o meno in grado di un esistenza permanente una volta giunta a destinazione, nel senso che ci dobbiamo aspettare che le cose vadano a buon fine solamente se l'agente è a Parigi quando arriva il messaggio, oppure è possibile che l'agente arrivi in un secondo momento rispetto al messaggio, ed ottenere comunque il risultato desiderato. Nel secondo caso avremmo la certezza di ottenere il successo, ma solo in questo esempio, perché se l'agente ad un certo punto cominciasse a spostarsi tra Firenze e Roma, il messaggio a Parigi non potrebbe più essere ricevuto.

Immaginiamo allora di mandare un messaggio ad ogni place (la cosa comincia a diventare costosa, ma è accettabile in quanto siamo all'interno di un agent system e possiamo ipotizzare la località), in questo modo otteniamo il risultato voluto se però ipotizziamo ancora che un messaggio venga ricevuto dall'agente anche in un secondo momento, in caso contrario i messaggi potrebbero arrivare quando l'agente è in viaggio e non essere quindi ricevuti. Questa potrebbe sembrare già una buona

soluzione, se però pensiamo di mandare dei messaggi di sospensione seguiti in un secondo tempo da un messaggio di riattivazione (se sappiamo dove è stato sospeso l'agente per la riattivazione basta un messaggio), quello che potrebbe succedere è che la migrazione in un place in cui sia presente un vecchio messaggio di sospensione, provocherebbe un arresto indesiderato dell'agente. In pratica è necessario un meccanismo che invalidi i messaggi vecchi.

A questo punto possiamo dire di avere trovato un modello che possa funzionare, ma che non sarà quello scelto per motivi di efficienza. Da un lato abbiamo l'utilizzo di broadcast che anche se accettabile nel contesto di località in cui ci troviamo, tende a consumare un certo numero di risorse di rete, e dall'altro nasce l'esigenza di determinare un intervallo di time-out se si vuole capire se l'operazione è andata a buon fine o meno. Nel caso in cui l'agente abbia lasciato questo agent system, nessuno dei messaggi potrebbe tornare indietro (a NewYork) per avvertire del successo avvenuto. L'insuccesso e la scomparsa dell'agente deve quindi essere dedotta implicitamente se, dopo un certo tempo, nessuna risposta di successo viene ricevuta. L'introduzione di time-out porta ad inefficienza ogni qual volta si deve aspettare un tempo fisso per un'informazione che potenzialmente potrebbe essere reperita più rapidamente utilizzando altre metodologie. Inoltre determinare quale sia il giusto intervallo non è cosa immediata, in questo caso si dovrebbe capire quanto è il tempo massimo che un agente impiega per fare una migrazione, cosa che dipende anche dal carico del sistema nel suo complesso.

La soluzione adottata cerca di risolvere questi problemi, ma apprende dalla precedente la necessità dell'utilizzo di messaggi che possano essere ricevuti anche nel caso il loro arrivo sia precedente a quello dell'agente. Facendo una similitudine un po' strana, si potrebbe dire che volere utilizzare messaggi, che per essere ricevuti, devono arrivare a destinazione quando l'agente è già presente, è un po' come cercare di sparare ad un bersaglio mobile che si muove alla velocità dei nostri proiettili. In pratica è un caso prenderci anche se al momento in cui premiamo il grilletto conosciamo la posizione precisa del bersaglio. Per

questo motivo non si prende nemmeno in considerazione la possibilità che un agente comunichi la sua posizione al place gestore ogni volta che migra (un po' come fanno gli agenti traceable con il place di origine), in quanto oltre a sovraccaricare il gestore, non si potrebbe sfruttare proficuamente l'informazione.

A questo punto è possibile presentare il modello scelto per l'implementazione delle operazioni di management.

I punti chiave che caratterizzano la gestione remota sono:

- accumulo dei messaggi spediti dal gestore nei place di destinazione per l'eventuale consegna ritardata;

- meccanismo di riferimenti che seguiti conducono al place di residenza attuale dell'agente.

Del primo punto si è già discusso a sufficienza, il secondo punto è introdotto per fornire un supporto informativo sul quale si basa questo modello e che verrà spiegato di seguito. Facendo riferimento allo scenario presentato immaginiamo ancora una volta che l'agente arrivi a NewYork dal mondo esterno e cominci a muoversi nei tre restanti place. Ora abbiamo ancora la conoscenza di quale sia il primo place che ha raggiunto l'agente dopo avere lasciato NewYork, ma in più abbiamo l'informazione del percorso che l'agente ha effettuato. In questo modo è possibile spedire un messaggio che segua la pista lasciata dall'agente e capire se l'agente è sparito non appena non si trovino più riferimenti. Quando un agente cambia agent system lo fa utilizzando come mezzo di migrazione il MAFAgentSystem del sistema che vuole raggiungere, questo modo di andarsene non lascia riferimenti. Normalmente invece l'agente si sposta in modo nativo, utilizzando le funzioni fornite dal supporto sottostante. È così possibile tenere traccia del place di destinazione lasciando un riferimento. Nel caso l'agente si muova da un place all'altro dello stesso agent system utilizzando direttamente il MAFAgentSystem, non vengono

lasciati riferimenti, ma il passaggio per il place gestore rende possibile l'aggiornamento della posizione dell'agente.

Con questo tipo di soluzione si ottiene il vantaggio che se un agente lascia l'agent system, un messaggio mandato ad un place e il suo ritorno è il tempo che, nel caso migliore, bisogna attendere per scoprirlo. Anche se nel caso peggiore il messaggio deve fare il giro di tutti i place per accorgersi che l'agente è scomparso, non è necessaria l'introduzione di time-out e si evita l'utilizzo di broadcast.

Evidentemente il messaggio potrebbe girare all'infinito dietro all'agente, ma questo non succede in quanto su ogni place che viene visitato, si lascia un messaggio pronto a fermare l'agente (nel caso l'agente dovesse ritornare su uno di questi place). Volendo schematizzare il comportamento del messaggio che insegue l'agente si potrebbe dire che:

- se l'agente è sul place corrente fermalo;

- se c'è un riferimento seguilo (dopo aver lasciato una copia del messaggio), a meno che il riferimento non indichi un place già visitato, in questo caso torna a casa e indica ciclo chiuso;

- se non ci sono riferimenti torna al gestore ed indica che l'agente è scomparso.

In questo modo il place gestore deve attendere fino alla ricezione di uno o due messaggi. Nel caso in cui l'inseguitore colga al volo l'agente si otterrà un solo messaggio di successo, nel caso in cui l'agente non ci sia, si riceverà un insuccesso, nel caso l'agente capiti in un place dove l'inseguitore ha lasciato un messaggio, al place gestore arriveranno due indicazioni, quella spedita al momento del congiungimento tra l'agente ed il messaggio e quella di ciclo chiuso, condizione che l'inseguitore sicuramente troverà (dato che l'agente è andato su un place che l'inseguitore aveva già esaminato).

Non ci si deve preoccupare del fatto che un agente crei un ciclo chiuso di riferimenti e poi se ne vada. Ogni volta che un agente arriva su un place cancella l'eventuale riferimento lasciato in precedenza e se a questo punto cambia agent system, il ciclo non rimane chiuso (un ciclo chiuso implica sempre la presenza dell'agente).

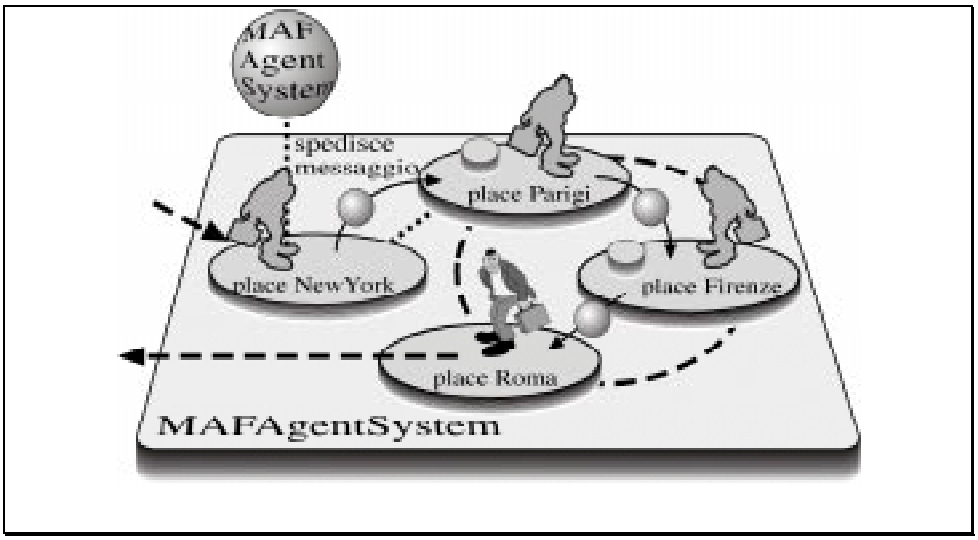

Figura 31. Ovunque l'agente vada viene fermato, l'unico modo che ha per sfuggire alla terminazione è uscire dall'agent system. In questo caso il messaggio che arriva a Roma riporterà al place gestore che l'agente è uscito dal sistema in quanto non ci sono più place da visitare.

Con un modello di gestione distribuita di questo tipo è possibile garantire l'affidabilità delle operazioni di management, e si fornisce SOMA di un supporto affidabile per il recapito di qualunque tipo di messaggi anche ad agenti che si muovono senza registrare gli spostamenti ad un name server.

4.2.8. Il problema del nome delle classi

Durante l'esposizione di MASIF è stato accennato a come sia necessario tenere le classi provenienti da diversi sistemi ben distinte, in modo da evitare collisioni nei nomi. Per ottenere questo obbiettivo si doterà il

sistema ad agenti di un meccanismo per consentire di specificare il percorso di ricerca sul file system per ogni agente creato. Grazie ad un tale meccanismo è possibile scegliere la politica di suddivisone delle classi in diversi percorsi, partendo da un percorso diverso per ogni agente, fino ad arrivare ad un unico percorso per tutti. Analizzando la situazione si può capire come suddividere le classi in percorsi diversi per ogni agente, possa limitare il riutilizzo di quelle classi comuni a diversi agenti (basti pensare ad una applicazione costituita da più agenti i quali condividono tra loro un package di funzioni comuni). All'altro estremo mettere insieme tutte le classi rende probabile la collisione dei nomi. Una soluzione intermedia che tenga conto di ambo le esigenze consiste nel scegliere percorsi di ricerca diversi, per agenti che hanno authority diversa. In questo modo ogni authority dovrà preoccuparsi esclusivamente dell'unicità dei nomi delle proprie classi. La soluzione scelta costruisce il percorso anche in base al tipo di sistema in cui l'agente è nato. In questo modo si evitano le collisioni di nomi dovute ad authority uguali generate da sistemi ad agenti differenti.

CAP 5.

IMPLEMENTAZIONE DELL'INTEROPERABILITÀ IN SOMA

L'implementazione delle specifiche progettuali fornite nel capitolo precedente ha comportato modifiche ed estensioni del sistema ad agenti SOMA. In questo capitolo verranno descritti i particolari più salienti dell'implementazione dell'interoperabilità, e saranno descritti alcuni dei risultati ottenuti dai test di prestazione e interazione, a cui il sistema finale é stato sottoposto.

Visibroker è l'implementazione CORBA che è stata utilizzata e Java è il linguaggio in cui SOMA è interamente scritto [Java98].

5.1. CORBA Facility

Per semplificare il più possibile la scrittura di agenti SOMA in grado di assumere il ruolo di servitori o clienti CORBA, è stato sviluppato un package Java denominato CORBAFacility.

Il processo di sviluppo di un server CORBA si differenzia in base alla modalità di skeleton che si ha intenzione di sfruttare. Nel caso si voglia costruire un oggetto basato sullo skeleton statico si dovrà produrre l'interfaccia IDL, compilarla per ottenere lo skeleton, ed in fine scrivere il codice del server. Nel caso si voglia costruire un implementazione dinamica, ossia un oggetto CORBA che sfrutta le caratteristiche dello skeleton dinamico, non ci sarà bisogno di utilizzare alcun compilatore IDL, si dovrà però effettuare una gestione più di basso livello delle

richieste. Dal lato del cliente le cose vanno in maniera analoga. Se si vuole fare utilizzo della invocazione statica è necessario possedere a tempo di compilazione l'interfaccia IDL del server e da essa generare gli stub necessari al client; nel caso si utilizzi l'invocazione dinamica scompare la necessità dell'interfaccia IDL a tempo di compilazione (che dovrà però essere ritrovata a runtime per effettuare il controllo dei tipi) e si dovrà gestire un meccanismo di basso livello per preparare le invocazioni. Questo è il riassunto del normale processo di sviluppo del codice di clienti e servitori CORBA che si è cercato di semplificare. I risultati ottenuti dalla sua semplificazione possono essere riassunti in questo modo:

- automatizzazione della produzione dell'interfaccia IDL senza l'utilizzo di compilatori. L'interfaccia IDL che è sempre necessaria (in alcuni casi a tempo di compilazione per produrre stub o skeleton in altri casi a runtime per il controllo di tipo) viene automaticamente prodotta e depositata nell'Interface Repository (si veda paragrafo 1.1.6);

- eliminazione della necessità di generazione di stub e skeleton;

- semplificazione delle registrazioni all'ORB e al servizio di nomi CORBA.

5.1.1. Utilizzo di CORBAFacility

Prima di addentrarci nelle modalità con cui le facility sono state realizzate analizziamo quale sia la prospettiva dell'utente finale che voglia farne uso, considerando anche limiti e vantaggi che derivano dal loro impiego.

Per scrivere un agente in grado di essere servitore CORBA bisogna semplicemente ereditare il comportamento della classe CORBAFacility.CORBAAgent scrivendo il codice dei metodi che si vogliono pubblicare a CORBA e il codice di quelli necessari alle migrazioni dell'agente. Una volta raggiunto un sito idoneo alla

registrazione all'ORB si potranno usare i seguenti metodi per diventare un server CORBA:

- register_to_ORB(String interface_name); provvede alla registrazione all'ORB e all'assegnamento del nome dell'interfaccia IDL corrispondente al server CORBA;

- register_to_NameServer(String name); semplicemente registra il server al servizio di nomi CORBA con il nome specificato. I client potranno ottenere un riferimento a partire da questo nome;

- register_to_IR(String[] operazioni); questa funzione provvede alla generazione dell'interfaccia IDL di tutti i metodi i cui nomi sono specificati nel parametro operazioni.

Dopo l'invocazione di queste tre funzioni le richieste entranti potranno essere servite dall'agente che potrà comunque invocare un metodo di deregistrazione in ogni momento (disconnect_from_ORB) e cambiare poi sito. Il cambiamento di sito è possibile ma è necessario ogni volta aggiornare anche il servizio di nomi per mantenere validi i riferimenti che possono essere ottenuti da esso. Durante il periodo di migrazione è possibile decidere di lasciare il vecchio riferimento che sarà riaggiornato alla prossima invocazione di register_to_NameServer, oppure di eliminarlo con l'invocazione di remove_from_nameServer(String Name).

```
import CORBAFacility.*;
public class CountAgent extends CORBAAgent{
   private int contatore=0;

   //Metodi per il server CORBA
   public void inc(){contatore++;}
   public void dec(){contatore--;}
   public void reset(){contatore=0;}

   //Metodi per l'agente
   public void putArg(Object obj){
     contatore=((Integer)obj).intValue();
   }
   public void run(){
```

```
        String[] operazioni={"inc","dec","reset"};
        register_to_ORB("Counter");  //Nome dell'interfaccia
        register_to_NameServer("Contatore_1"); //Nome al name
server
        register_to_IR(operazioni);
    }
 }
```

Il codice per scrivere un server CORBA con l'uso delle Facility.

I clienti che vorranno invocare i servizi di un tale agente potranno farlo come se fosse un normale oggetto CORBA. Nel caso di utilizzo dell'invocazione dinamica ci sarà l'interfaccia IDL già caricata nell'IR, nel caso di invocazione statica si dovrà ricorrere alla produzione degli stub (l'IR consente l'estrazione delle interfacce IDL in formato testo).

Tutte le operazioni superflue necessarie nella scrittura di oggetti CORBA sono state eliminate; non è più necessario scrivere codice IDL, utilizzare compilatori per la generazione dello skeleton statico, o gestire i complessi dettagli dell'invocazione dinamica.

Per terminare l'opera si è cercato di semplificare anche il processo di sviluppo del lato client. Un agente (o un qualunque oggetto Java) che voglia essere cliente di un server CORBA (un agente SOMA o meno) può, oltre ad utilizzare i normali meccanismi di invocazione, sfruttare le facility fornite. Vediamo quali sono i passi da seguire per la scrittura di un cliente che sfrutta il package CORBAFacility.

- Ottenere un riferimento al server CORBA mediante l'utilizzo di CORBAFacility.getCORBAObject(String name).

- Preparare un vettore di oggetti con i parametri necessari nell'invocazione.

- Effettuare l'invocazione specificando il nome del metodo e i parametri.

Il parametro name specificato nel primo punto va considerato come il nome utilizzato dal server CORBA per registrarsi al servizio di nomi. Dopo il primo punto si dispone di un oggetto di classe CORBAFacility.CORBAObject sul quale è possibile effettuare l'invocazione del metodo call(String nome_metodo, Object[] parametri) che tradurrà l'invocazione locale in una richiesta remota per il server.

```
CORBAObject obj=CORBAFacilityManager.getCORBAObject(name);
//o in alternativa obj=CORBAFacilityManager.newCORBAObjcet(rif);
Object[] par=new Object[2];
par[0]=new Integer(18);
par[1]=new Integer(2);
Object val=obj.call("Somma",par);
System.out.println("Risultato: "+val);
```

Il codice per una invocazione remoto con l'uso delle facility.

Eventualmente se si dispone già del riferimento (rif) ad un oggetto CORBA (se si possiede quindi un oggetto rif di classe org.omg.CORBA.Object) è possibile utilizzare le facility su esso costruendo un CORBAObject, attraverso l'invocazione del metodo CORBAFacilityManager.newCORBAObject(rif).

Con l'introduzione di queste facility abbiamo ottenuto il vantaggio di non dover utilizzare stub (e quindi compilatori per la loro generazione), e ci muoviamo ad un livello più alto dell'invocazione dinamica. Non è però possibile ottenere la trasparenza tra l'invocazione su oggetti locali e remoti che si poteva avere con l'utilizzo di stub.

5.1.2. Implementazione delle facility

Lato Server

Analizziamo come sia stato possibile ottenere il risultato esposto nel paragrafo precedente iniziando a considerare il lato server.

L'interazione tra ORB e server avviene con l'utilizzo di un oggetto CORBA intermedio. Come spiegato nel paragrafo 4.1.1. in questo modo è possibile la registrazione diretta degli agenti. Normalmente l'oggetto

intermedio, differente per oggetti CORBA differenti, è generato dalla compilazione di un interfaccia IDL. Per evitare questa fase si è scelto di utilizzare lo skeleton dinamico. Essendo l'interfaccia di un server CORBA dinamico indipendente dal tipo di servizio che esso fornisce, è stata possibile la costruzione di un oggetto intermedio generale che potesse fare da tramite per qualunque tipo di server. Si è trattato infatti di scrivere un server CORBA dinamico (chiamato _tie_Dynamic) che provvedesse a spedire le richieste in arrivo, all'oggetto in grado di esaudirle. Quest'ultimo non è vincolato ad essere un oggetto CORBA e quindi può essere un agente SOMA.

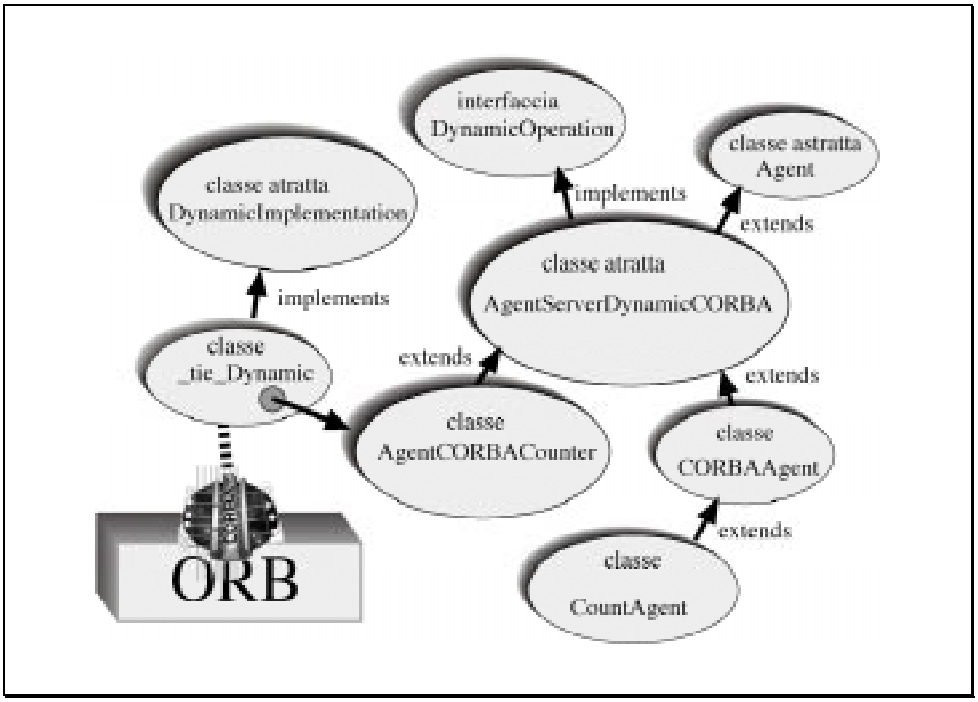

Figura 23. Il metodo register_to_ORB di AgentServerDynamicCORBA provvede a istanziare e registrare all'ORB l'oggetto di classe _tie_Dynamic, passandogli l'indirizzo dell'agente che implementa il servizio (nell'esempio l'oggetto di classe AgentCORBACounter o CountAgent). Estendendo la classe AgentServerDynamicCORBA si deve implementare il servizio tramite la realizzazione

del metodo invoke, e quindi con gli strumenti dello skeleton dinamico CORBA. Estendendo invece CORBAAgent le cose sono semplificate, basta scrivere il codice dei servizi e compiere registrarli attraverso l'invocazione del metodo register_to_IR.

Ecco allora che è stato possibile sviluppare una classe denominata AgentServerDynamicCORBA per fornire un primo grado di facility per la registrazione all'ORB e al servizio di nomi. In pratica un agente che eredita da AgentServerDynamicCORBA pur dovendo gestire le richieste a basso livello (tramite skeleton dinamico), non si deve preoccupare dei dettagli di registrazione all'ORB e al name server.

La classe CORBAAgent trattata nel paragrafo 5.1.1. estende le facility di AgentServerDynamicCORBA nascondendo la complessità della gestione dinamica delle invocazioni. Dal punto di vista dell'utilizzo, questa classe mette a disposizione il metodo denominato register_to_IR(String[] operations) che consente la generazione dell'interfaccia IDL relativa a tutti i metodi specificati nel parametro operations e la registrazione dell'interfaccia prodotta all'Interface Repository. Dopo avere invocato questo metodo è possibile la gestione automatica di basso livello delle richieste entranti. Per ogni richiesta le facility effettuano le seguenti operazioni:

- reperimento del metodo da invocare;

- conversione parametri da CORBA IDL a Java (i parametri vengono letti dalla richiesta come gli IDL corrispondenti ai tipi Java dei parametri del metodo da invocare, per motivi di efficienza si cerca di evitare di guardare nell'IR);

- invocazione del metodo;

- conversione risultato, o delle eccezioni, da Java a CORBA IDL.

Le fasi più delicate di questo processo riguardano il mapping CORBA Java. Di questo si è già parlato nel paragrafo 1.1.1.1 dove si è detto che in generale esistono compilatori (es. idl2java) che associano ai tipi IDL specificati nell'interfaccia i corrispondenti tipi Java. In pratica l'operazione di traduzione che viene fatta dai compilatori, viene riprodotta a runtime dalle facility. Se ad esempio il risultato finale di un invocazione è un oggetto Java di classe String, il parametro sarà inserito come valore di ritorno dell'invocazione come string CORBA. Questo è riprodotto per tutti i tipi primitivi Java, ma anche per strutture più complesse come ad esempio i vettori. Gli oggetti Java possono essere passati come parametro, ma devono essere dichiarati final, public, e devono avere tutti gli attributi public. Questo è necessario per poter mappare un oggetto Java in una struttura (struct) CORBA. Per gli oggetti che non possiedono queste caratteristiche Visibroker [Visibroker98] introduce la possibilità del loro inserimento come parametri di una funzione uscendo dallo standard CORBA. In pratica questi elementi sono mappati in un tipo IDL non standard denominato extensible struct, e per il loro trasferimento via rete viene utilizzato come formato esterno la serializzazione Java. Per riprodurre questa possibilità anche con l'utilizzo delle facility è stato incontrato qualche problema dovuto al fatto che Visibroker non consente di inserire dinamicamente nell'Interface Repository, IDL di tipo extensible struct (questo può avvenire solamente generando il codice IDL e caricandolo nell'IR mediante il comando di load del tool che implementa l'Interface Repository fornito con Visibroker). Questo limite non coinvolge il processo di marshalling e unmarshalling dell'oggetto che può essere fatto mediante l'utilizzo delle funzioni write_estruct e read_estruct messe a disposizione da Visibroker, bensì il controllo dei tipi. Non potendo inserire il tipo IDL extensible struct nell'Interface Repository è stato scelto di mappare tutti gli oggetti di classe non associabile a struct CORBA, in moduli IDL chiamati con il nome della classe in questione e contenenti una struct denominata EstructDef. Un oggetto di classe java.util.Hashtable viene inserito nell'Interface Repository come un IDL così strutturato:

invece di:

```
 module java {                    module java {
   module lang {                            module lang {
    module Hashtable{                         extensible struct
Hashtable;
       struct EstructDef;           };
     };                           };
   };
 };
```

In questo modo è possibile effettuare il controllo dei tipi (come ciò avviene sarà spiegato in seguito). Non si deve considerare l'introduzione di questo stratagemma come una grande perdita di compatibilità in quanto se consideriamo clienti standard CORBA, la compatibilità con essi è perduta sin dal momento che si ipotizza di utilizzare oggetti Java come parametri delle invocazioni. Esiste inoltre anche la possibilità di accettare richieste provenienti da client non standard che utilizzino l'invocazione statica (facendo uso degli stub generati con l'interfaccia IDL corretta, ossia quella che associa agli oggetti Java il tipo non standard IDL extensible struct) in quanto nessun controllo di tipo è attuato. L'unico vero problema sorge quando un cliente che voglia utilizzare l'invocazione dinamica cerchi nell'Interface Repository l'interfaccia del server, interpretando (in base alla semantica CORBA) la struct EstructDef come una struttura CORBA senza attributi.

Lato Client

Passando a considerare il lato client ci si accorge che le operazioni svolte dalle facility sono molto simili a quelle che avvengono dalla parte del server. Riassumendole in un elenco troviamo:

- recupero dell'interfaccia IDL del server;

- controllo dei tipi dei parametri passati dal client, con i tipi IDL trovati nell'Interface Repository;

- conversione da Java a CORBA IDL dei parametri;

- creazione richiesta e invocazione remota;

- conversione da CORBA IDL a Java del risultato (o delle eccezioni) e ritorno controllo al client.

La conversione dei parametri sono semplicemente scambiate rispetto al server. I parametri e il risultato devono essere convertiti da Java a CORBA IDL invece che da CORBA IDL a Java. Risulta poco differente la gestione delle eccezioni che devono essere catturate nel sito in cui avviene l'effettuazione del servizio e rigenerate per il client nel sito di invocazione. La differenza fondamentale sta invece nel controllo dei tipi che viene effettuato solamente al lato client. Questo è in accordo con lo standard CORBA che prevede il passaggio dei parametri non accompagnati dal loro tipo (si veda paragrafo 1.1.4.) e di conseguenza obbliga l'effettuazione dei controlli al lato del mittente. Il server si aspetta in pratica richieste ben compilate e non si può in generale accorgere di errori. Il processo di controllo dei tipi si basa sul confronto tra il tipo di ciascun parametro passato alle facility dal client, con i tipi IDL recuperati dall'Interface Repository. Per gli oggetti Java non mappabili in strutture CORBA ci si aspetta di trovare nell'IR una definizione di una EstructDef racchiusa in un modulo con il nome della classe dell'oggetto (come descritto poco sopra). Il controllo di tipo può quindi avvenire confrontando il nome del modulo recuperato dall'IR con il nome della classe del parametro in esame.

Le facility non consentono di ottenere trasparenza tra invocazioni remote e locali. Se insorge questa necessità si può fare uso dell'invocazione statica in maniera indipendente da come è stato implementato il server (che potrebbe fare uso delle facility). Per ottenere il codice IDL dell'interfaccia del server al fine di generare lo stub necessario, si può procedere manualmente oppure, in caso il server sia

stato implementato utilizzando parametri mappabili in tipi IDL standard CORBA, è possibile ottenere il codice IDL copiandolo dal tools Visibroker che realizza l'Interface Repository, una volta avvenuta la registrazione del server. Se si procede manualmente bisogna fare attenzione a specificare come nome dell'interfaccia IDL lo stesso nome utilizzato dal server al momento della register_to_ORB(String interface).

I tipi utilizzabili come parametri dell'invocazione

I parametri gestiti dalle facility comprendono, oltre ai tipi primitivi Java, alle stringhe, agli oggetti Java (per i quali valgono le considerazioni riportate) e al tipo Any (che come le stringhe pur essendo un oggetto Java è possibile il mapping diretto in un tipo CORBA IDL primitivo), gli oggetti CORBA. La semantica di passaggio dei parametri è sempre per valore tranne nel caso degli oggetti CORBA (per questi ultimi vale una semantica per riferimento). Gli oggetti CORBA possono essere trattati in modo generico, indicando come tipo org.omg.CORBA.Object, oppure specifico, indicando il tipo dell'interfaccia che implementano. La seconda modalità è possibile per oggetti dei quali è stato generato lo stub e fornisce la trasparenza locale-remoto delle invocazioni. Se ad esempio Count è il nome dell'interfaccia di un server CORBA che realizza l'astrazione di contatore, un agente che fa uso delle facility può dichiarare un metodo che tra i suoi parametri accetta un oggetto di tipo Count, solamente se esiste localmente la classe _st_Count che implementa lo stub necessario per l'invocazione statica. Le facility provvederanno alla creazione di una istanza dello stub (il nome della classe dello stub è ottenuto aggiungendo "_st_" al nome della classe del parametro in accordo con la sintassi usata da Visibroker), al collegamento con il riferimento ottenuto dai parametri, e solo a questo punto invocheranno il metodo. Il server all'interno del metodo potrà quindi utilizzare il normale meccanismo di invocazione statica sul contatore e potrà usufruire della trasparenza locale-remoto delle invocazioni.

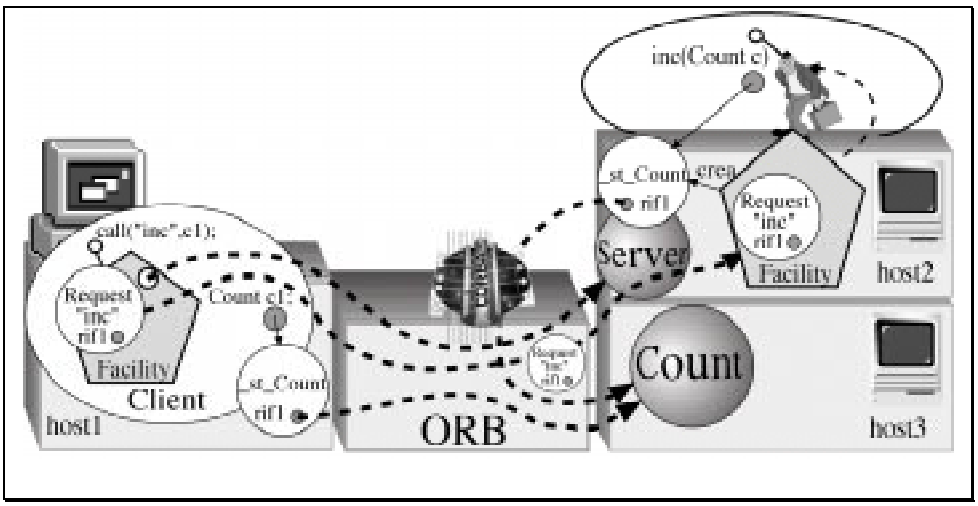

Figura 24. Nel disegno si è cercato di rappresentare quello che succede quando un oggetto CORBA viene passato come parametro. Le facility del client costruiscono una richiesta inserendo il riferimento all'oggetto (rif1) e effettuano l'invocazione remota. Le facility server ottengono il nome del metodo richiesto dal client (inc). Da esso possono risalire ai tipi dei parametri necessari all'invocazione; un oggetto di classe Count. Essendo Count un interfaccia discendente da org.omg.CORBA.Object, è possibile inferire che nella richiesta sarà inserito un riferimento ad un oggetto CORBA. Si potrà così ottenere rif1. A questo punto le facility cercheranno la classe dello stub relativo al tipo Count, ne creeranno un istanza e la assoceranno all'oggetto CORBA riferito da rif1. In fine potrà avvenire l'invocazione del metodo inc definito dall'agente passando come oggetto di tipo Count lo stub.

Quando invece lo stub non è presente, gli oggetti CORBA devono essere passati come org.omg.CORBA.Object ed eventuali invocazioni su di essi devono fare utilizzo di meccanismi dinamici o delle facility client (a partire da un oggetto di classe org.omg.CORBA.Object è sempre possibile costruire un oggetto di classe CORBAObject e utilizzare con esso le facility).

5.2. Indipendenza di alcuni meccanismi SOMA

Per dotare SOMA della capacità di accogliere e gestire agenti stranieri è stato necessario apportare delle modifiche ad alcuni meccanismi di

sistema basati su informazioni contenute all'interno degli agenti SOMA. In particolare è nata l'esigenza di aggiungere informazioni che consentissero di rendere indipendenti i meccanismi gli uni dagli altri. Purtroppo SOMA usava l'identificatore degli agenti per scopi molto diversi tra loro:

- identificare l'agente in maniera univoca;
- definire politiche di sicurezza basate sulle informazioni contenute nell'identificatore;
- come indirizzo della locazione in cui trovare le classi;
- come indirizzo del place di nascita.

Questi aspetti sono stati divisi in maniera tale da consentire un utilizzo più flessibile dei meccanismi. SOMA, ad esempio, utilizza un meccanismo by-need per reperire le classi durante gli spostamenti di un agente. Il place che ha bisogno di ottenere le classi invia una richiesta al place di origine dell'agente (che viene dedotto dall'identificatore). Se l'associazione tra identificatore, place di origine e place in cui richiedere le classi viene spezzata, si rende indipendente il meccanismo di reperimento delle classi dall'origine dell'agente. Per chiarire la necessità di indipendenza tra place di origine e place fornitore del codice si può pensare alla *mobile computing*. In questa applicazione del paradigma ad agenti mobili il place di origine di un agente è molto spesso non raggiungibile, quindi per la necessità di classi ci si deve rivolgere altrove. Anche se questo esempio è al di fuori delle necessità che si sono manifestate durante l'ampliamento di SOMA, il suo scopo è quello di fare comprendere che l'indipendenza dei meccanismi di sicurezza, di reperimento di classi, di identificazione e basati sull'origine dell'agente, è bene siano distinti l'uno dall'altro. Per ottenere questo risultato sono state

inserite all'interno degli agenti SOMA le informazioni che saranno di seguito esposte.

5.2.1. La funzione di identificazione

Gli identificatori degli agenti SOMA sono costruiti a partire dal nome del dominio e del place di origine, dal nome della classe dell'agente e da un identificatore che è unico all'interno del place di quel dominio (in pratica un numero intero che viene incrementato ad ogni creazione).

A questo tipo di identificatore è stato sostituito un nome MASIF (vedi paragrafo 3.5.4.5) per consentire una migliore gestione degli agenti in relazione alla presenza di agenti stranieri nel sistema. L'identificatore MASIF è costituito da identity, authority e type. Come identity è stato utilizzato il vecchio identificatore SOMA convertito in una sequenza di byte, per l'authority si utilizza il responsabile di sistema (che va impostato nella fase di configurazione di SOMA), e per il type si utilizza una costante intera associata al sistema ad agenti SOMA (da concordare con i responsabili dello standard MASIF).

Utilizzando il nome MASIF al posto del vecchio identificatore è possibile creare degli agenti di trasporto identificati con il nome dell'oggetto trasportato. In questo modo si evita di dover mantenere una continua corrispondenza tra l'identificatore dell'agente straniero e quello dell'agente che lo trasporta, inoltre l'agente straniero viene identificato all'interno del sistema SOMA con il suo vero nome. Le interazioni che richiedono l'uso di identificatori possono così avvenire in maniera trasparente rispetto al tipo di agente con cui si vuole interagire (SOMA o straniero). Ad esempio l'invio di mail (si veda il paragrafo 5.7.2.) avviene in maniera trasparente al tipo di agente destinatario del messaggio.

5.2.2. Informazioni relative agli accessi alle risorse

Le politiche di accesso alle risorse vengono definite a partire dalle informazioni di origine dell'agente, dalle credenziali, nonché dal nome della classe dell'agente. Quando un agente cerca di accedere ad una risorsa, il sistema controlla se ne ha il diritto verificando i permessi associati alla provenienza, e controllando eventuali credenziali possedute dall'agente. Se i permessi associati all'agente lo permettono, la risorsa può essere utilizzata.

La provenienza dell'agente e il nome della classe viene ricavata ancora una volta dall'identificatore. Per potere gestire le cose in modo separato sono state aggiunte le informazioni che vengono utilizzate per la verifica dei diritti di accesso. In questo modo si è reso possibile definire politiche anche per gli agenti che, provenendo da altri sistemi, non consentono di ricavare le informazioni di origine a partire dall'identificatore. Nel paragrafo 5.3.2. si parlerà di quali siano gli attuali controlli di accesso alle risorse per gli agenti stranieri.

5.2.3. La sorgente di codice

SOMA utilizza un meccanismo dinamico di reperimento delle classi dei suoi agenti. Quando un agente migra e giunge a destinazione viene ripristinato il suo stato. In questa fase possono essere necessarie una serie di classi che vengono ricercate nel place di origine ogni qual volta non siano presenti localmente. Questo meccanismo (di tipo by-need) evita l'inutile trasferimento delle classi a priori, in quanto se le classi sono già state trasferite in una prima migrazione in un certo place, ad una seconda migrazione in quel place le classi non verranno più richieste essendo già presenti. Il place di origine viene dedotto facendo uso ancora una volta dell'identificatore, ma se l'agente è straniero il place di origine che viene dedotto non può essere utilizzato in maniera diretta per trovare le classi. Bisogna passare attraverso il MAFAgentSystem essendo questo l'unico collegamento con l'agent system di origine dell'agente straniero. Per questo viene introdotto nell'agente un nuovo attributo che indica la sua sorgente di codice (SourceOfCode) e che comprende oltre al place il nome

della directory in cui ricercare le classi, (il motivo dell'introduzione di una directory di ricerca verrà spiegato in seguito a proposito del problema del nome delle classi).

Per gli agenti non stranieri la sorgente di codice viene sempre fatta coincidere con il place di origine, mentre per gli altri agenti si utilizza il place gestore, ossia quello che ha gestito l'arrivo dall'esterno dell'agente e nel quale è presente l'oggetto CORBA MAFAgentSystem.

5.2.4. Rendere gli agenti SOMA gestibili tramite il MAFAgentSystem

Per fornire la possibilità di scegliere se un agente SOMA debba oppure no essere visibile dall'esterno della sua località, è stata inserita la possibilità di creare agenti *visibili* che vengono registrati al place gestore che dirige l'agent system in cui nascono. Una volta registrato, l'agente è gestibile dall'esterno, e su di esso è applicabile tutto ciò che MAFAgentSystem specifica. Ad esempio su un agente SOMA visibile è possibile, dall'esterno, invocando la suspend_agent del MAFAgentSystem, ottenere la sua sospensione, oppure mediante le opportune chiamate la riattivazione, la terminazione, o semplicemente richiedere quale sia correntemente il suo stato.

Per fare questo è stato inserito un attributo che consente di discriminare tra gli agenti *visibili* e non (un po' come si fa per distinguere gli agenti traceable da quelli non traceable). Sarà poi analizzato in seguito a cosa serva questa informazione all'interno di un agente.

5.3. Integrazione con il modello di sicurezza esistente

Si è cercato di integrare il più possibile le nuove funzionalità con il modello di sicurezza già esistente. Esiste certamente la necessità di un estensione del modello di sicurezza per gestire le nuove esigenze di apertura verso il mondo esterno, ma per ora ci si è limitati al minimo

indispensabile, in maniera tale da indirizzare ogni sforzo per cercare di ottenere un buon livello di interoperabilità.

5.3.1. Riservatezza nelle migrazioni locali

Nelle migrazioni locali (intese come migrazioni da un place all'altro dello stesso agent system) la sicurezza è già ottenuta se si utilizza il metodo di migrazione nativa. Quando si utilizza direttamente il MAFAgentSystem sono due le entità che entrano in gioco: da un lato CORBA attraverso il quale avviene l'invocazione della receive_agent; dall'altro l'eventuale invio dell'agente al place remoto tramite l'utilizzo di SOMA. Nel primo caso dovrà essere CORBA che implementa i servizi di segretezza nella comunicazione, e nel secondo caso la sicurezza è già ottenuta mediante gli algoritmi di cifratura che il sistema ad agenti usa.

5.3.2. Controllo degli accessi

Quando un agente straniero entra nel sistema viene costruito attorno ad esso un agente di trasporto nel quale vengono inserite le informazioni opportune relative alla sicurezza negli accessi. Le informazioni su cui vengono costruite le politiche di sicurezza sono dominio, place di origine e nome della classe dell'agente. Gli agenti stranieri non è detto che provengano da sistemi in cui esiste l'astrazione di dominio e quella di place. Si dovrà quindi dare una semantica di tipo diverso alle informazioni che si useranno durante la definizione di politiche per gli agenti stranieri. Per ora quando un nuovo agente di trasporto viene costruito tutte le informazioni sono impostate al valore "Foreign". In questo modo si sfrutta il meccanismo di controllo degli accessi già esistente e reso indipendente dall'identificazione.

Per assegnare agli agenti stranieri i minimi diritti che consentano loro di invocare i metodi del MAFAgentSystem attraverso CORBA (sarà necessario ad esempio il diritto di creare connessioni attraverso socket)

basterà indicare che gli agenti di dominio, place e classe “Foreign” abbiano i diritti desiderati (si veda il tool AgentPolicy[Tenti98]).

In quanto al meccanismo delle credenziali per ora è non è possibile farne uso, in futuro basterà associare le credenziali, che si riconoscono all’agente straniero, all’agente di trasporto. Per riuscire a riconoscere le credenziali di un agente straniero si deve fare utilizzo dei servizi di sicurezza forniti da CORBA, l’interazione con sistemi eterogenei deve infatti avvenire sempre tramite collegamenti standard.

5.4. Esecuzione remota affidabile di comandi su agenti mobili

Per potere realizzare il modello per il management di agenti remoti che è stato specificato nel paragrafo 4.2.7. è stato necessario effettuare alcune modifiche ed estensioni del sistema SOMA. In particolare sono state rese atomiche le operazioni di partenza ed arrivo in un place, e sono stati aggiunti meccanismi per il rilascio di riferimenti e per la loro rimozione una volta diventati inutili.

In pratica è stato esteso il meccanismo basato sullo scambio di comandi che SOMA utilizza per la comunicazione tra place, rendendolo capace di inviare comandi alla ricerca di un agente. Il supporto ottenuto si può dire affidabile in quanto anche in caso di spostamenti dell'agente, il comando verrà eseguito, in un tempo finito, sul place in cui incontrerà l'agente.

5.4.1. Atomicità delle operazioni di migrazione

Nel paragrafo 4.2.7. si è sempre parlato di messaggi che venivano spediti nei vari place e che eventualmente si duplicavano. SOMA utilizza i comandi per le operazioni che richiedono la comunicazione tra due place. Un comando che arriva in un place viene eseguito da un thread gestore dei

comandi. Volendo realizzare le operazioni di management, l'utilizzo di questo meccanismo risulta adatto. Se ad esempio si vuole fermare un agente, si può pensare di scrivere un comando che, giunto nel place dove l'agente risiede, sospenda il thread di esecuzione utilizzando le normali funzioni di gestione dei thread che il linguaggio mette a disposizione. Per ottenere un comportamento corretto è stato necessario rendere atomica l'operazione di trasmissione di un agente. Le operazioni che un place fa quando un agente richiede di migrare, per quanto interessa ai problemi di sincronizzazione, possono essere così schematizzate:

- invocazione del metodo go;
- operazioni varie (quali il controllo dei permessi per la migrazione);
- rilascio di un riferimento verso il place di destinazione;
- spedizione dell'agente serializzato;
- cancellazione delle tracce dell'agente dal place mittente.

Nel caso un comando di sospensione arrivi dopo la spedizione del messaggio, ma prima della cancellazione delle tracce dell'agente, si avrà l'illusione di avere completato con successo l'operazione ma in realtà l'agente non sarà fermato in quanto continuerà la sua esecuzione una volta arrivato sul place di destinazione.

Potrebbe sembrare che per evitare il problema sia sufficiente cancellare le tracce dell'agente prima della sua spedizione. In questo modo l'agente non sarebbe trovato nel place che lo sta spedendo, e in funzione del fatto che il riferimento al nuovo place sia già stato lasciato o meno, avremmo due diversi comportamenti (entrambi errati). Un riferimento non presente farebbe capire che l'agente ha lasciato l'agent system, un riferimento già presente farebbe proseguire il messaggio nel nuovo place. Nel caso arrivi prima il messaggio dell'agente, si ricadrebbe in errore in quanto si

potrebbe trovare un vecchio riferimento non aggiornato, oppure non trovare nessun riferimento.

Oltre a quanto detto bisogna aggiungere che il thread che esegue le operazioni di migrazione sopra elencate è il thread dell'agente invocante il metodo go. Le operazioni di migrazione sono racchiuse dentro metodi sincronizzati. Questo non basta per rendere atomica la migrazione (si ottiene atomicità solo rispetto alla visione di altri agenti che vogliono migrare), nel senso che un altro thread potrebbe sospendere l'agente proprio mentre questo sta migrando. Evidentemente il fatto che le operazioni siano sincronizzate, implica che se un agente viene sospeso proprio mentre sta migrando, il sistema venga a trovarsi in una condizione di blocco.

Da tutto ciò l'esigenza di fare diventare atomica l'operazione di migrazione agli occhi dei comandi che vogliono eseguire operazioni di sospensione sugli agenti. Per ottenere il risultato aspettato sono stati utilizzati gli stessi riferimenti dotandoli di un meccanismo di sincronizzazione. Per la gestione dei riferimenti è stata utilizzata una tabella che come entry ha il nome degli agenti. Quando un comando arriva cerca di bloccare la entry corrispondente all'agente che vuole sospendere (questo può provocare anche la temporanea sospensione del comando se la entry era già stata fermata in precedenza dall'agente). Quando l'agente comincia la migrazione per prima cosa (ancora quando il thread non è entrato in un blocco di codice sincronizzato) lascia un riferimento dopo avere bloccato la entry. In questo modo si evita che il comando sospenda l'agente mentre sta migrando, ma si evita anche che l'agente inizi la migrazione mentre il comando si sta preparando per la sospensione.

La sincronizzazione viene utilizzata anche quando un agente arriva in un place, infatti è questo il momento in cui vengono eseguiti i comandi arrivati per l'agente. Se un comando si sta duplicando per lasciare una sua copia sul place, e l'agente arriva un attimo prima che il comando sia posto nella sua coda, si creerebbe una situazione di errore. L'agente non riceverebbe il comando, e il comando penserebbe di avere già visitato quel place.

In questo modo sono state presi in considerazione un po' tutti gli aspetti relativi alla sincronizzazione tra comandi e agenti. Un ultima cosa che si può dire è che se si pensa di utilizzare questo meccanismo per fare altre cose bisogna tenere presente che: i comandi sono inseriti in una coda e vengono ricevuti in base all'ordine di arrivo nel place (che non è detto sia quello di spedizione); il thread che viene generato per l'esecuzione dell'agente (Worker), è utilizzato anche per eseguire i comandi (questo va ben tenuto presente nel caso di sospensione), subito prima del codice dell'agente; i comandi vengono ricevuti una volta sola dato che sono etichettati con una sigla crescente, e l'agent tiene traccia della sigla dell'ultimo comando che ha eseguito (quindi non eseguirà più comandi con sigla minore di quella corrente).

Un ulteriore problema di sincronizzazione

Se un agente decide di migrare invocando direttamente il metodo del MAFAgentSystem, proprio quando il place gestore ha spedito un comando con lo scopo di sospendere l'agente, si presenta una situazione che va gestita con cautela.

In questo caso bisogna che il place gestore non faccia completare l'invocazione della receive_agent, altrimenti il comando di sospensione non troverebbe l'agente e direbbe che è scomparso. Per evitare questo inconveniente sono state serializzate le operazioni che possono essere compiute su un agente.

In questo modo la richiesta di migrazione fatta dall'agente viene sospesa, dato che è già in corso un'operazione su di esso. Il comando di sospensione troverà l'agente e porterà il suo risultato di successo. A questo punto potrebbe essere riattivato il thread relativo alla richiesta di migrazione, se non fosse che il comando precedente aveva lo scopo di sospendere l'agente. Se infatti si riattivasse la richiesta invalideremmo il comando appena eseguito, in quanto ci sarebbe l'agente sospeso sul place da cui giunge la richiesta corrente, ma alla fine della migrazione ci sarebbe anche una copia dell'agente attiva sul place di destinazione. Proprio per questo quando si riattiva il thread di una richiesta si controlla

quale sia lo stato dell'agente, ed in base ed esso si decide di far terminare con insuccesso l'operazione (ad esempio se si cerca di sospendere un agente che nel frattempo è stato terminato), di mantenerla sospesa, oppure di farla proseguire.

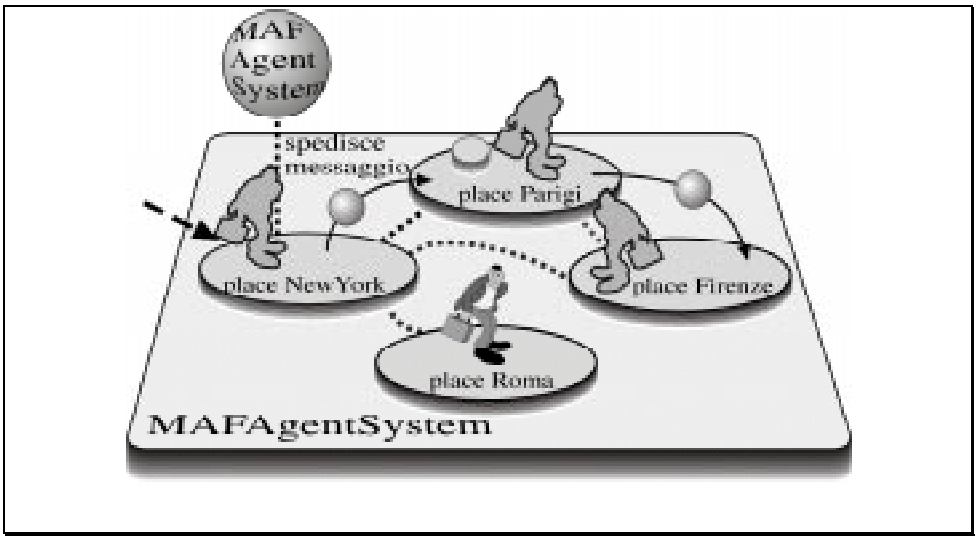

Figura 32. Il place gestore spedisce un messaggio (o comando) per sospendere l'agente quando si trova a Firenze. A questo punto l'agente chiede di andare a Roma utilizzando la migrazione MASIF (non quella nativa) e quindi non lascia alcun riferimento su Firenze per Roma. Se il place gestore lascia che la migrazione avvenga si ottiene il risultato che il comando di sospensione ritorna un risultato di insuccesso (e la notifica che l'agente ha lasciato il sistema).

Nella situazione presentata una richiesta di riattivazione deve quindi controllare se è il caso di riattivare un thread locale che aveva iniziato una migrazione, oltre che il thread remoto dell'agente che aveva effettuato la richiesta di migrazione stessa (anche esso sospeso).

Una volta spiegato quale sia il modello utilizzato per il management degli agenti stranieri, può essere chiarito come mai sia stato inserito nel corpo degli agenti SOMA l'informazione relativa al fatto che siano internazionali o meno. Per essere internazionali devono poter venire gestiti in maniera remota, e devono quindi lasciare i riferimenti ad ogni

migrazione. Il supporto di place può capire da questa informazione se deve o non deve lasciare il riferimento quando un agente SOMA migra.

5.4.2. Garbage dei riferimenti

Un problema che deriva dall'utilizzo dei riferimenti è che prima o poi bisogna eliminare quelli che non servono più. Per questo ogni volta che si scopre che un agente è terminato si invia un comando che cancella i riferimenti lasciati da quell'agente. Sicuramente se qualcuno invoca dal mondo esterno la funzione per ottenere lo stato dell'agente, viene inoltrata una ricerca per aggiornare il suo stato (naturalmente a meno che non sia già terminato). In molti casi però gli agenti possono lasciare il nostro sistema senza che il place gestore se ne accorga. Per questo è stato scritto un algoritmo di garbage collection che ad intervalli regolari, da determinare in base al carico e alle risorse del sistema, provvede all'eliminazione dei riferimenti. L'intervallo di time-out che bisogna inserire non produce alcuna inefficienza dei servizi come accadeva nel caso della soluzione presentata nel paragrafo 4.2.7. in quanto non è un tempo critico per il completamento di nessuna operazione.

L'algoritmo, partendo dai riferimenti iniziali mantenuti nel place gestore, segue tutti i cammini degli agenti, che sono ancora considerati attivi, e determina quali siano terminati (un po' come fanno i comandi che inseguono gli agenti per sospenderli). Dopo di che invia un comando che fa il giro di tutti i place dell'agent system ed elimina i riferimenti degli agenti terminati.

Per motivi di efficienza e per evitare la saturazione della rete con messaggi che cercano di capire se un agente è terminato, questa prima parte dell'algoritmo viene fatta in modo simulato. In pratica prima si reperiscono le informazione di tutti gli agenti sospetti, facendo girare un unico comando in tutti i place del sistema. Dopo di che nel place gestore si naviga tra le tabelle di riferimenti ottenute come se si stesse girando per la rete.

5.5. Servizi di nomi e MAFFinder

Nel capitolo tre sono state presentate le soluzioni di MASIF per il ritrovamento degli agenti e degli agent system. Il MAFFinder è il servizio di nomi che consente di risalire alla location (si veda paragrafo 3.5.4.5) di un agente a partire dal suo nome. L'implementazione sviluppata fornisce un supporto persistente delle informazioni basata su un file hash che consente anche la costruzione di indici secondari. Gli indici secondari sono utilizzati per la ricerca a partire da campi non chiave del record, funzione richiesta dalle caratteristiche del MAFFinder.

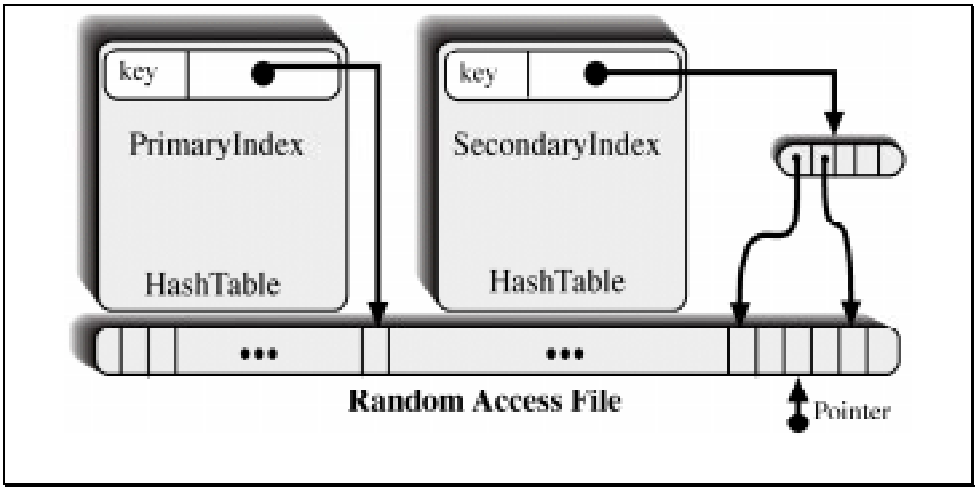

Figura 33. Hashfile basato sull'utilizzo di Hashtable e RandomAccessFile.

L'inserimento di nuove informazioni avviene in coda al file, mentre la richiesta di una registrazione di un record che era già presente provoca la modifica locale se lo spazio lo consente, oppure la invalidazione dello spazio occupato dal vecchio record e l'inserimento del nuovo in coda. Hashfile fornisce la possibilità di ricompattare gli spazi rimasti inutilizzati, ma la gestione è lasciata all'utente. Ogni volta che il MAFFinder viene messo in esecuzione provvede all'attivazione (oltre che

anche alla ricompattazione) di tre Hashfile utilizzati rispettivamente per la memorizzazione delle informazioni relative a agenti, sistemi e place. Per i file relativi ad agenti e sistemi viene anche creato un indice secondario che consente una ricerca per profilo efficiente (si evita la scansione di tutto il file).

Una volta ottenuta la location di un agent system interrogando il MAFFinder, è necessaria la sua risoluzione per ottenere il riferimento al server CORBA. Nel paragrafo 3.5.4.5 sono state messo in evidenza le due modalità di risoluzione della location che sono state entrambe implementate. È stata implementata una funzione di risoluzione delle location che esamina il formato della stringa ed inferisce quale modalità si debba adottare per costruire il riferimento. Nel caso si tratti di una location di tipo CosNaming è necessario effettuare la procedura di conversione da stringa a NameComponent e poi accedere al servizio di nomi CosNaming; nel caso invece si tratti di una location Internet si procede alla costruzione dello IOR a partire dalle informazioni direttamente ottenibili dalla location. La seconda procedura risulta essere non immediata e fa utilizzo di strutture dati di solito non utilizzate a livello di utente CORBA. Nella figura è mostrato uno schema riassuntivo del processo di costruzione.

In quanto al tipo di schema utilizzato dagli agent system SOMA per registrarsi al MAFFinder sono ancora una volta state messe a disposizione le due possibili modalità (specificabili dalla linea di comando). Per ottenere le informazioni relative all'host, alla porta e alla chiave dell'oggetto server CORBA è stato utilizzata la specifica relativa alla sintassi dei riferimenti convertiti in stringa. L'ORB è in grado di convertire un riferimento ad un oggetto in una stringa che può essere poi riconvertita ad un riferimento in un secondo tempo. La stringa prodotta rispetta le specifiche di interoperabilità tra ORB (si veda GIOP al paragrafo 1.3) e quindi il suo formato è standard. La sua costruzione (figura 34) a grandi linee avviene serializzando le strutture dati che contengono le informazioni di allocazione dell'oggetto (tra cui host, port e key) e convertendo lo stream di byte ottenuto in un formato esadecimale.

Da essa sono quindi deducibile (dopo le opportune conversioni) le informazioni necessarie alla costruzione dell'indirizzo Internet che potrà essere registrato al MAFFinder.

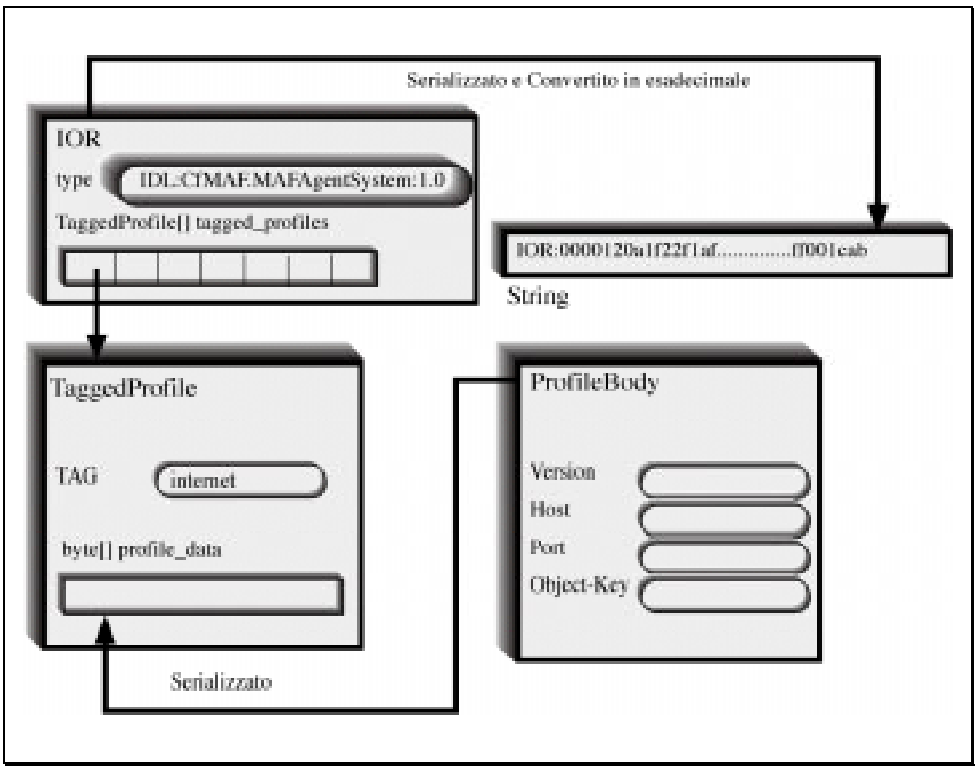

Figura 34. Come passare da host, port e object-key, alla stringa che può essere fornita all'ORB per ottenere il riferimento all'oggetto corrispondente (usando il metodo string_to_object). Si parte dalla struttura dati ProfileBody in cui devono essere inserite le informazioni iniziali, si costruisce un TaggedProfile di tipo Internet e gli si associa come profile_data il ProfileBody serializzato. A questo punto si costruisce la strutture IOR nella quale si inserisce il TaggedProfile. In fine si trasforma lo IOR in un vettore di byte e lo si converte in una stringa esadecimale dove ogni coppia di caratteri corrisponde ad un byte.

Oltre alle possibilità esposte è stato aggiunto un mezzo alternativo basato su un servizio di nomi specifico di Visibroker per trattare indirizzi Internet. URLNaming è il nome che Visibroker attribuisce al servizio che consente di associare nomi URL a riferimenti ad oggetti CORBA. In sostanza si tratta di un server web posizionato in ascolto nel sito e sulla porta specificata che associa ad ogni nome oggetto un file. In questo modo attraverso l'utilizzo di protocolli standard come http o ftp è possibile ottenere il file corrispondente all'oggetto cercato che contiene il riferimento (IOR) in formato stringa. L'astrazione fornita al client da funzioni di libreria, è quella di un normale servizio di nomi che consente di effettuare operazioni di registrazione e ricerca. Fondamentalmente ogni ricerca effettuata dal client viene tradotta in un comando HTTP di GET, mentre ogni registrazione in un comando HTTP di PUT (si deve utilizzare quindi un server web in grado di accettare la ricezione di file). Visibroker fornisce un server web chiamato Gatekeeper. L'utilizzo di questo servizio di nomi consente di ottenere ancora location rappresentate da un indirizzo Internet, ma a differenza di quello che avviene quando gli indirizzi sono costruiti a partire dallo IOR di un oggetto CORBA, utilizzando URLNaming si dispone di nomi più semplici (o user-friendly) e facili da memorizzare. Questo accade perché la parte chiave che si ottiene dallo IOR dipende dall'implementazione di CORBA che si sta utilizzando, e trattandosi di un vettore di byte è in generale un nome abbastanza complesso. Se si utilizza URLNaming il nome dell'oggetto può essere specificato a piacere. Pur essendo già attivo questo servizio non è possibile sfruttare il Gatekeeper fornito da Visibroker come server web a causa di incompatibilità tra Gatekeeper e JDK1.2beta2 nel protocollo HTTP Keep-Alive. È possibile comunque sostituire Gatekeeper con un server HTTP compatibile con il protocollo Keep-Alive del JDK1.2bata2.

5.6. Trasferimento di classi

Per il reperimento di classi si utilizza un meccanismo dinamico by-need, e si estende in questo modo il meccanismo che viene già utilizzato da SOMA. In pratica quando viene invocata la receive_agent del MAFAgentSystem non vengono passate classi, ma il solo stato serializzato dell'agente. Quando si tenta di istanziare l'agente, all'eventuale bisogno di reperire una classe risponde il classloader (LoaderSicuro) che è stato esteso per cercare le classi anche in altri sistemi.

Il nuovo comportamento del classloader può essere così schematizzato:

- si controlla se la classe è di sistema o è già caricata, altrimenti;
- si cerca il bytecode nel place locale, altrimenti;
- se per caso siamo nel place indicato dal SourceOfCode dell'agente e le classi non sono qui vuole dire che l'agente viene da fuori, chiamiamo quindi il metodo fetch_class dell'agent system di provenienza dell'agente, altrimenti;
- si richiede al place indicato nel SourceOfCode dell'agente l'invio delle informazioni necessarie.

In questo modo otteniamo il risultato voluto quando cerchiamo di istanziare un agente sul place gestore dell'agent system. Se nasce la necessità di una classe quando l'agente non si trova su tale place, quello che succede è che si arriva al quarto punto dello schema precedente e si richiedono le classi dove indicato nel SourceOfCode. Per gli agenti stranieri il sorgente delle classi è sempre il place gestore che è in grado, a fronte di una richiesta di una classe, di rinviare la richiesta anche ad un agent system remoto, nel caso in cui non trovi la classe localmente.

Una cosa da aggiungere riguarda la lista di nomi di classi che può essere tra i parametri della receive_agent e della create_agent. Quando durante la ricezione si riceve una lista di classi, il comportamento che si è

scelto di implementare è quello di fare immediatamente il fetch di tutte le classi indicate. In questo modo se il mittente si vuole scollegare una volta spedito l'agente lo può fare, stando attento solamente ad indicare tutte le classi che saranno necessarie all'esecuzione futura dell'agente.

Per la creazione le cose sono un po' differenti nel senso che se il cliente non è un agent system ci si aspetta di ricevere le classi indicate nella lista direttamente in un parametro della chiamata. Si richiede lo stesso anche nel caso il client sia un agent system, ma la differenza sta nel fatto che in quest'ultimo caso è anche possibile non indicare alcuna classe (a parte una in modo da sapere cosa si deve istanziare), in quanto le classi mancanti potranno venire richieste in futuro.

Una cosa da aggiungere a riguardo della creazione è che se il cliente non è un agent system ci si aspetta di avere tutte le classi subito, quindi come SourceOfCode dell'agente che verrà creato, si indicherà il place effettivo della creazione (nel quale verranno subito scaricate tutte le classi ricevute), e non il place gestore. Si procede in questo modo per cercare di decentralizzare il ruolo del place gestore che deve sempre fornire le classi per tutti gli agenti stranieri.

Bisogna però vedere anche l'altro lato del reperimento di classi, ossia quello in cui è il nostro agent system che viene interrogato perché qualcuno ha bisogno di classi. Questa parte si esplica nell'implementazione del metodo fetch_class, i cui parametri principali sono il nome delle classi di cui si ha bisogno e una stringa chiamata code_base che deve essere utilizzata quando si richiedono le classi. Il code_base viene fornito inizialmente dall'agent system che crea o spedisce l'agente in un altro sistema. Il sistema che riceve l'agente deve memorizzarlo ed usarlo per richiedere le classi dell'agente. SOMA come code_base fornisce il SourceOfCode trasformato in stringa. In questo modo quando al place gestore arriva una richiesta di reperimento classi, ripristinando dalla stringa il SourceOfCode, è in grado di capire in quale place e directory (nel paragrafo successivo è spiegato a cosa serva la directory nel SourceOfCode) ricercare le classi specificate nella lista. Con la struttura data al sistema fino a questo punto le cose valgono sia se

qualcuno ci sta chiedendo le classi di un nostro agente (per le quali si ricercherà sul place di origine), ma anche nel caso le classi che ci vengano richieste siano di un agente straniero che abbia attraversato il sistema.

Nel paragrafo 4.2.8. si è discusso della necessità di ottenere un meccanismo che fosse in grado di mantenere distinte le classi di ciascun agente. La parte directory del SourceOfCode serve proprio per ottenere questo obbiettivo. Se volessimo esplodere il secondo punto di ricerca del classloader analizzato nel paragrafo precedente, dovremmo dire che la ricerca della classe all'interno del place viene effettuata nella directory specificata dal SourceOfCode relativo all'agente che si sta considerando.

5.7. Aspetti tecnici e funzionalità aggiunte a SOMA

5.7.1. Struttura dell'agente di trasporto

Può essere interessante cercare di capire come avvenga l'incapsulamento di un agente straniero entro l'agente di trasporto (chiamato TransportAgent). Questo agente che è un agente SOMA a tutti gli effetti (eredita da Agent), è costituito principalmente da un riferimento ad un oggetto (che sarà l'agente trasportato), dal nome di un metodo da invocare sull'oggetto, e da un metodo run che è il metodo che viene sempre invocato sull'agente di trasporto dall'agent system (sia all'inizio che a fronte di migrazioni successive).

Mantenendo un riferimento all'agente da trasportare, il processo di serializzazione dello stato dell'agente, che si attua ad ogni migrazione, provvede ad incapsulare, insieme a tutti gli attributi dell'agente di trasporto, anche l'oggetto di cui si possiede il riferimento e che rappresenta l'agente trasportato.

L'agente di trasporto ridefinisce il metodo SetStart ed eredita il comportamento del metodo getStart. Questi metodi sono utilizzati rispettivamente nelle fasi di trasmissione e ricezione dell'agente, per impostare ed ottenere il metodo che dovrà essere invocato dopo la

riattivazione dell'agente sul place di destinazione. In questo modo l'impostazione del metodo modifica il nome racchiuso entro l'agente di trasporto, mentre il nome del metodo che viene ereditato dalla classe Agent rimane sempre "run". Non ridefinendo la getStart il polimorfismo non si attua, e il nome del metodo che viene restituito nel place di destinazione è sempre "run". Ogni volta che "run" viene invocato guarda il nuovo metodo che è stato settato (dalla setStart e grazie al polimorfismo) e lo invoca sull'oggetto trasportato.

A proposito della sicurezza bisogna dire che l'agente di trasporto essendo costituito da una classe di sistema anche se viene caricato da un classloader sicuro, viene definito con un codesource che gli permette qualunque diritto sulle risorse. L'agente trasportato viene però posto in una directory fuori dal classpath e quando viene caricato con il classloader sicuro (lo stesso dell'agente di trasporto) viene definito con il codesource che gli compete. In questo modo l'esecuzione del codice degli oggetti trasportati avviene, per il diritto dei minimi privilegi [Tenti98] che Java usa per i suoi thread, in modo sicuro.

5.7.2. Facility di migrazione per i nostri agenti

Gli agenti SOMA che vogliano migrare in altri sistemi ad agenti possono farlo utilizzando direttamente CORBA per invocare i metodi del MAFAgentSystem. Questo comporta tutta una serie di istruzioni che richiedono la conoscenza dei meccanismi CORBA che pur essendo relativamente semplici (almeno nel caso di invocazione statica), possono distogliere il programmatore che si muove ad un livello più alto e non si vuole preoccupare di questo tipo di dettagli. Per questo è stato sviluppato un agente chiamato AgentFacileMove che consente la migrazione in ambiente eterogeneo in maniera più semplice. Ereditando il comportamento di questo agente la migrazione si riduce all'invocazione del metodo go.

Accanto ad esso si possono sfruttare una serie di nuovi metodi messi a disposizione della classe AgentSystem.AgentSystem che consentono la

migrazione in altri sistemi ad agenti. Si noti però che una volta lasciato un sistema SOMA per andare in un sistema differente, ci si ritrovi senza la classe AgentSystem.AgentSystem e si perda la possibilità di usare le facility di migrazione.

Proprio per questo è nata la necessità della realizzazione di AgentFacileMove, che non si appoggia su funzioni proposte da SOMA, ma utilizza direttamente CORBA e MASIF, consentendo così l'utilizzo delle facility anche in sistemi non SOMA.

In entrambe i casi i metodi di migrazione che sono resi disponibili sono diversi in funzione della conoscenza che un agente ha della sua destinazione. Esiste la possibilità di migrare conoscendo già il riferimento all'oggetto MAFAgentSystem di destinazione, ma la migrazione è possibile anche se si conosce la locazione (Location MASIF) oppure il nome (Name MASIF). Viene data anche la possibilità di migrare conoscendo il riferimento al MAFFinder convertito in stringa (IOR), in questo modo un agente può conservare i riferimenti agli agent system che gli interessano anche durante le migrazioni, e non dover pagare il costo di accesso ai servizi di nomi ogni volta (lo IOR consente la migrazione senza accedere ad alcun name server).

Volendo inquadrare le due tipologie delle facility proposte in base a quanto detto in precedenza a proposito della migrazione, possiamo vedere l'utilizzo di AgentFacileMove come la migrazione di agenti che si muovono sfruttando l'interfaccia MASIF in maniera diretta, mentre l'utilizzo di SOMA può essere ricollegato alla migrazione che ipotizza la conoscenza (e quindi l'eventuale standardizzazione) della interfaccia che gli agent system rendono disponibile agli agenti (si ricorda che MASIF propone uno standard tra agent system, e non tra agenti ed agent system).

Quando si pensa di impiegare AgentFacileMove per spostarsi, si dovranno utilizzare esclusivamente le sue facility (non si potrà usare una volta direttamente CORBA, ed una volta AgentFacileMove.go). Questo è necessario in quanto AgentFacileMove utilizza al suo interno la nozione di Agent System corrente al fine di specificare l'AgentSystem di origine durante le invocazioni della receive MASIF. Una volta che sarà decisa

l’interfaccia standard che i sistemi ad agenti dovranno fornire agli agenti, si potrà evitare l’invocazione diretta delle funzioni del MAFAgentSystem sostituendola con una chiamata al sistema locale. In questo modo sarà il sistema che dovrà effettuare le operazioni di migrazione mediante l’interfaccia MASIF e potrà specificare se stesso come origine. A questo punto sparirà la necessità dell’utilizzo di AgentFacileMove.

5.7.3. Estensione dei servizi di mail

Per consentire l’utilizzo della mail anche agli agenti stranieri è stato esteso il comportamento della mailbox che gli agenti portano con se, in particolare si è dovuta dare la possibilità di ricevere mail ad agenti che non fossero traceable. Rendere gli agenti stranieri traceable vuole dire creare un collo di bottiglia nel place gestore (a differenza degli agenti SOMA traceable che distribuiscono il carico su tutti i place di un dominio gli agenti stranieri fanno sempre riferimento al solo place gestore). Per consentire ad agenti non traceable di ricevere mail si è deciso di utilizzare il meccanismo dei riferimenti creato per il management.

Quando si deve inviare un messaggio ad un agente straniero è necessario invocare il metodo della mailbox sendMessage il quale provvede a distinguere se il destinatario è o meno straniero. Nel caso l’agente non sia straniero si utilizza il normale metodo di invio della posta, altrimenti, se l’agente non è locale, viene creato un comando che va al place gestore, dove provvede a costruire un secondo comando che recapita il messaggio seguendo la pista dell’agente (un po’ come accade quando si effettua la sospensione) . Si noti bene che in questo modo il place gestore è richiesto solo quando si scambiano messaggi, e non ad ogni migrazione.

Questo modo di recapitare la mail è più affidabile che nel caso degli agenti traceable, per i quali esiste il pericolo che il messaggio sia perso. Anche un agente SOMA registrato al place gestore (traceable o meno) può usufruire di questa nuovo mezzo di trasporto della posta in quanto per gli agenti registrati è previsto il rilascio dei riferimenti quando migrano.

Per consentire il normale utilizzo delle funzioni di mail dopo il cambiamento dell'identificatore nel nome MASIF, è stato necessario qualche piccolo ritocco. Il mittente di un messaggio veniva specificato indicandone l'identificatore. L'informazione necessaria per potere recapitare il messaggio era il place di origine dell'agente. Essendo l'identificatore SOMA costituito in parte anche dal place di origine bastava l'invocazione di un metodo per ottenere ciò di cui si aveva bisogno. Naturalmente una volta sostituito il nome MASIF all'identificatore SOMA, la cosa è risultata un po' più complessa. Comunque la decisione di utilizzare come parte identity del nome MASIF il vecchio identificatore SOMA convertito in vettore di byte, ha reso ancora possibile ottenere il place di origine.

Una volta che si possiede il nome MASIF di un agente SOMA è sempre possibile ottenere il suo place di origine. Convertendo la parte identity è possibile riottenere un identificatore SOMA e quindi conoscere place, dominio e nome della classe dell'agente.

5.7.4. Trasferimento di package

Una caratteristica di SOMA che è stata ampliata riguarda il trasferimento di agenti costituiti da package (in precedenza non era possibile trasferire package). Gli agenti che vogliono fare utilizzo di CORBA come mezzo di comunicazione, hanno bisogno di portarsi appresso stub che gli consentano la chiamata, tali stub sono generati automaticamente da appositi strumenti che spesso li racchiudono in package. In questi casi diventa importante il trasferimento dei package.

5.7.5. Compatibilità Java1.2beta2 e Visibroker for Java 3.2

VisiBroker per Java nella versione 3.2 è stato testato con JDK1.2beta2 da JavaSoft, ma ufficialmente la compatibilità con la JVM non è ancora

garantita. Per potere utilizzare VisiBroker con la versione 1.2beta2 di Java è necessario tenere in considerazione i seguenti aspetti:

- JDK1.2beta2 include un nucleo iniziale dell'ORB che può interferire con VisiBroker. Per questo è necessario eliminare dal CLASSPATH tutte le classi fornite con il JDK1.2beta2 che si riferiscono a CORBA. Nella ricerca fatta il corretto funzionamento è stato ottenuto eliminando le directory "com", " javax", "Meta-inf", "org", e tutte le loro sottodirectory che si ottengono dall'espansione del file classes.zip;

- quando si eseguono applet progettate per VisiBroker in browser o appletviewer basate su java 1.2beta2 ci possono essere delle perdite di funzionalità. In particolare certi metodi sull'ORB e sugli oggetti CORBA non sono disponibili. Per aggirare questo problema si può installare VisiBroker nell'ambiente locale del browser così che le classi di Visibroker siano le prime nel CLASSPATH;

- Il supporto Gatekeeper non è compatibile con il protocollo HTTP Keep-Alive usato dal JDK1.2beta2. Il tunnelling HTTP e il trasferimento di file che il gatekeeper consente non possono quindi essere utilizzati, di conseguenza non è possibile utilizzare i nomi URL per gli oggetti CORBA, a meno che non si scriva un server HTTP che prenda il posto del gatekeeper.

  Per ulteriori dettagli [VisiBroker98]

5.7.6. Configurazione del sistema

Non si vuole in questo contesto dare la spiegazione di come debba essere installato SOMA, ma si vogliono mettere in evidenza i cambiamenti apportati nel processo di configurazione. Per quello che riguarda la sicurezza e le politiche di accesso nulla è cambiato. La nuova cosa che

bisogna specificare riguarda come siano suddivisi i place di un dominio tra gli agent system che si vogliono rendere visibili su CORBA.

Ogni place ha un file di configurazione, al quale è stato aggiunto un elemento che contiene le informazioni di quale sia il place gestore e del nome del MAFAgentSystem. Quando si fa partire il processo che costituisce il supporto di un place, in maniera automatica si rileva dal file di configurazione quale sia il place gestore. Tra tutti i place avviati quello che si riconoscerà come gestore provvederà a creare e registrare a CORBA, al servizio di nomi utilizzato (CosNaming o URLNaming) e al MAFFinder l'oggetto MAFAgentSystem. Il ritrovamento del MAFFinder avviene per tentativi successivi. Inizialmente si cerca nella directory corrente, o in quella specificata dalla riga di comando, il file maffinder.ior che viene generato dal programma di avvio del MAFFinder e contiene il riferimento convertito in stringa con il quale è possibile ottenere un riferimento al servizio di nomi. Se il file non viene trovato si procede la ricerca al CosNaming (il quale è rintracciabile mediante funzioni standard CORBA) e altrimenti si prova con URLNaming. In caso un qualunque sistema di nomi non venga ritrovato si ha una notifica, ma il supporto di nodo viene ugualmente attivato in quanto non tutti i servizi sono strettamente necessari.

Un dettaglio banale riguarda la configurazione dei place. Ora non è più necessaria la creazione della directory esterno. All'avvio di ogni agente è inoltre possibile specificare il percorso di ricerca sul file system delle classi che saranno necessarie per la creazione e l'esecuzione futura dell'agente. In questo modo si possono tenere divise le varie applicazioni sviluppate.

5.8. Misure di prestazione

Per capire quale sia il costo del nuovo meccanismo di migrazione sono state effettuate delle misurazioni dei tempi di spostamento di un agente.

Prima di proporre i risultati ottenuti vale la pena indicare quali siano le operazioni che bisogna compiere per fare avvenire una migrazione.

Questo porterà ad una maggiore comprensione dei valori numerici e ad un migliore confronto tra il meccanismo di migrazione nativo e la nuova funzionalità aderente lo standard MASIF.

Le funzioni che ambe due i meccanismi di migrazione devono svolgere sono le seguenti (si suppone non ci sia necessità di trasferimento di classi):

- serializzazione dell'agente;
- spedizione via rete;
- caricamento della classe;
- deserializzazione e avvio.

Mentre per il meccanismo di base queste funzioni sono le sole necessarie perché la migrazione possa avvenire, la nuova modalità di migrazione prevede una fase preliminare costituita, in generale, dalle seguenti operazioni:

- inizializzazione ORB;
- inizializzazione servizio di nomi CORBA (CosNaming);
- ritrovamento MAFFinder (ricerca nel servizio CosNaming);
- risoluzione del nome MASIF del sistema di destinazione (si ottiene la location sotto forma di stringa interrogando il MAFFinder);
- risoluzione della location per ottenere un riferimento all'oggetto CORBA MAFAgentSystem che rappresenta il sistema di destinazione.

Non è detto che le operazioni elencate siano sempre tutte necessarie. Una volta ottenuto il riferimento ad un MAFAgentSystem è possibile convertirlo in un formato trasportabile denominato IOR (una stringa; si veda capitolo 1), per conservarlo anche a fronte di altre migrazioni. Nel caso si debba tornare in un sistema del quale si possiede lo IOR, è possibile ottenere da esso il riferimento originale, senza dover accedere ai vari servizi di nomi. In alcuni casi è inoltre possibile che si conosca la location del sistema che si vuole raggiungere, in queste situazioni si può

evitare l’accesso al MAFFinder (le facility di migrazione forniscono funzioni di migrazione a qualsiasi livello, si veda il paragrafo 5.7.2.).

Per comprendere quale sia il costo effettivo delle operazioni preliminari alla migrazione, è bene indicare quale sia l’ambiente in cui queste operazioni sono svolte. SOMA fornisce come ambiente di computazione, per tutti gli agenti che siano allocati nello stesso place, un unico processo pesante. All’interno di esso possiamo identificare, oltre ad una serie di elementi che forniscono le funzioni di base per l’esistenza, la comunicazione e l’esecuzione degli agenti, un insieme di processi leggeri, o thread, che costituiscono i flussi di esecuzione degli agenti stessi. L’ambiente di computazione di agenti differenti è mantenuto distinto mediante l’utilizzo di un class loader differente per ciascuno di essi. In questo modo da un lato è possibile evitare interferenze tra gli agenti, e dall’altro usufruire di un processo comune nel quale ritrovare e utilizzare le stesse classi di sistema.

Questa premessa è assolutamente necessaria per comprendere i risultati derivati dalle misure dei tempi di migrazione. In modo particolare per quei valori che si riferiscono alla fase preliminare costituita dagli ultimi cinque punti elencati. Gli agenti, disponendo di un nucleo comune di librerie che costituiscono gran parte dell’ORB, sono soggetti ad un fenomeno generale di cacheing. Generale nel senso che le operazioni CORBA fatte da un agente possono essere utili, dal punto di vista del cacheing che l’ORB compie, anche per altri agenti. Ecco così che una volta ottenuta una connessione con il servizio di nomi MAFFinder, per tutti gli agenti che richiederanno l’utilizza di questo servizio l’ORB non dovrà creare una nuova connessione, ottenendo così un minimo spreco di risorse con il conseguente aumento delle prestazioni. Alla luce di queste considerazioni iniziali vengono proposti in seguito i risultati ottenuti.

Descrizione tempo	Dimensionedell'agente 5 K byte	50 K byte
Inizializzazione ORB	7	6
Inizializzazione CosNaming	631	587
Inizializzazione MAFFinder	7	7
Ricerca Destinazione	35	31
Serializzazione	48	947
Invio Parametri	102	243
Caricamento classe e deserializzazione	1483	3025
Altro	7	8
Totale fase preliminare	*680*	*631*
Totale seconda fase	*1640*	*4223*
TOTALE	**2320**	**4854**

Tabella 1. Migrazione MASIF. I tempi riportati sono stati ottenuti dalla media di un campione di 54 valori. La scomposizione di una migrazione nelle sue fasi mette in risalto i differenti costi e permette così una comprensione più approfondita dei risultati. I valori indicati sono da intendere come tempi espressi in milli secondi (msec).

Descrizione tempo	Dimensionedell'agente 5 K byte	50 K byte
TOTALE con sicurezza disabilitata	**1648**	**4204**
TOTALE con sicurezza abilitata	**9126**	**74624**

Tabella 2. Migrazione nativa. Nella tabella sono riportati due differenti casi di migrazione nativa. Nel primo caso il sistema di crittografia che consente la riservatezza durante gli spostamenti è disattivato. Nel secondo caso è stato attivato il sistema di sicurezza. I valori indicati sono da intendere come tempi espressi in milli secondi (msec).

Le analisi di prestazioni sono state fatte su una rete costituita da PC con processore PentiumII a 300 MHz e WindowsNT come sistema operativo. SOMA, per ogni place installato, prevede una fase di

configurazione in cui si specifica il percorso del file system che dovrà essere utilizzato per memorizzare le classi degli agenti. Per motivi pratici le directory specificate per i vari place sono state riferite al file system condiviso. Questo va specificato perché durante la fase di caricamento delle classi degli agenti l'accesso al disco condiviso ha in generale un costo differente dall'accesso al disco locale. Dai dati si può comunque rilevare quale sia questa componente di costo per valutare un possibile aumento di efficienza che potrebbe essere ottenuto con l'utilizzo di un supporto di memorizzazione più veloce. I risultati campionati accorpano tempo di caricamento delle classi e deserializzazione dello stato dell'agente. In base ad altre misure qui non riportate, è stato comunque possibile stimare il solo tempo di caricamento in un valore pari a 1400 msec (indubbiamente un tempo elevato: per come lavora WindowsNT, con l'utilizzo del file system locale ad ogni place questo valore sarebbe stato notevolmente ridotto).

Più interessanti le considerazioni sulla seconda fase del processo di migrazione. Questi valori sono vicini al tempo totale della migrazione nativa. Un risultato di questo genere era possibile prevederlo in quanto le operazioni da compiere sono del tutte analoghe in ambo le metodologie di migrazione. Una cosa che è stata rilevata suddividendo il tempo di una migrazione nelle sue componenti è che il costo complessivo è notevolmente influenzato dal processo di serializzazione-deserializzazione che si deve compiere. Se consideriamo l'agente di dimensioni pari a 50K byte relativo alla migrazione MASIF, ed eliminiamo il tempo di caricamento di 1400 msec, ci accorgeremo che su un tempo complessivo della seconda fase di 2823 msec, la parte impegnata per la serializzazione e deserializzazione è di 2572 msec. Questo discorso è valido anche per il processo di migrazione nativa (anche se non è esplicitamente evidenziato dai valori). Nel caso si voglia cercare di ottenere tempi migliori bisognerà agire principalmente sulla metodologia di appiattimento delle strutture dati degli agenti. Si potrà pensare di utilizzare un tipo di serializzazione diverso da quello di Java oppure costruire un componente dedicato che svolga questo compito.

Questo migliorerà sia i tempi di migrazione attraverso MASIF che quelli ottenuti con la migrazione nativa. Sono riportati nella tabella 3 altri tempi di serializzazione al variare delle dimensioni dell'agente da 5kbyte fino a 100k byte. Si noti come all'aumentare delle dimensioni dell'agente da serializzare si assista ad un degrado delle prestazioni.

byte agente	**5K**	**10K**	**20K**	**50K**	**100K**
tempo msec	48	111	242	947	3275
efficienza byte/msec	107	92	85	54	31

Tabella3. Tempi di seriazlizzazione e degrado dell'efficienza con l'aumentare delle dimensioni dell'agente.

La parte più interessante dei risultati raccolti è la fase preliminare dalla quale dipende la differenza tra i due meccanismi che consentono la mobilità degli agenti. Il tempo totale delle operazioni iniziali è di circa 630-680 msec, quasi interamente dovuto all'inizializzazione del CosNaming. Ad esclusione di quest'ultimo tempo tutte le altre fasi sono di costo molto ridotto, e da altri esperimenti è stato rilevata la possibilità di ridurre anche questo costo a tempi dell'ordine di grandezza di 10 msec se si mantiene, dopo la prima inizializzazione, il riferimento al CosNaming convertito in stringa (lo IOR). Da quanto esposto fino ad ora sembrerebbe che il costo della utilizzo della migrazione che si appoggia a CORBA sia molto prossima a quella nativa (facendo il cacheing descritto si arriverebbe ad una differenza di appena 50 msec circa). In realtà se le cose vanno in questo modo c'è una spiegazione, non si deve dimenticare che l'utilizzo di CORBA è sempre abbastanza costoso. Per spiegare questi risultati bisogna tenere presente la premessa fatta in precedenza in questo paragrafo, a proposito dell'ambiente in cui gli agenti eseguono, e spiegare

anche cosa succede ogni qual volta si necessiti di un servizio fornito da un server CORBA. Si dovrà anche fare riferimento alla implementazione di CORBA che è stata scelta come supporto per l'estensione di SOMA, ossia a VisiBroker for Java 3.2.

Prima che un client possa invocare un metodo su un qualsiasi oggetto CORBA è necessario che tra cliente e servitore venga stabilita una connessione TCP (questa è una scelta di VisiBroker). Il cliente non si accorge di questo in quanto è VisiBroker che provvede a tutto. La connessione viene mantenuta dalle librerie che costituiscono l'ORB per essere sfruttata in più di una invocazione richiesta dal client. Il collegamento tra client e server viene mantenuto anche nel caso in cui il cliente elimini il riferimento all'oggetto, nell'eventualità che in un secondo tempo venga ancora richiesto un servizio di quel server. Questo ultima strategia adottata da VisiBroker insieme all'utilizzo comune che gli agenti fanno delle librerie dell'ORB, fa si che una connessione aperta possa essere riutilizzata da qualunque agente in un qualunque momento (non ha importanza chi sia stato il primo che ha attivato il collegamento client server).

Ecco allora come si spiegano i bassi tempi di inizializzazione dei vari componenti. Una volta attivata la connessione con il MAFFinder, (questo viene fatto già all'attivazione di un place) le successive richieste per ottenere un riferimento ad esso non porteranno allo spreco di tempo che occorre per stabilire il collegamento. Questo vale anche per l'invocazione della receive_agent che deve essere fatta sull'oggetto MAFAgentSystem rappresentante la destinazione (dopo la prima transizione il costo di connessione sarà totalmente eliminato).

Rimane da spiegare come mai allora il costo di inizializzazione del CosNaming sia sempre così alto. Per ottenere un riferimento al servizio di nomi CORBA è possibile procedere in due modi: conoscendo lo IOR; invocando sull'ORB, un metodo denominato resolve_initial_references, specificando come parametro il servizio di nomi. Nel caso si acceda ad esso conoscendo lo IOR il comportamento dell'ORB è quello normalmente adottato per gli oggetti a livello applicativo. In pratica viene

stabilita una sola connessione iniziale e poi riciclata per le invocazioni successive. Nel caso invece si invochi resolve_initial_references, l'ORB provvede ogni volta a stabilire una nuova connessione con il server ed è per questo che si ottiene un costo superiore.

In conclusione il comportamento che si ottiene può essere considerato buono in quanto basato su un concetto di località. Le migrazioni MASIF che avvengono in una località costituita da un certo numero di sistemi sono di efficienza paragonabile alle migrazioni native (in questo caso le connessioni sono infatti già tutte stabilite). Le migrazioni per le quali non è possibile fare ipotesi di località risulteranno di costo maggiore. In particolare il costo sarà più elevato di quello riportato nei risultati del tempo necessario per stabilire una connessione, 500-800 msec in più (tempi riferiti alla velocità di connessione su LAN).

Non si è ancora detto nulla a proposito della inizializzazione dell'ORB. Questo tempo è stato campionato con l'idea che si trattasse di un'operazione costosa. In realtà la chiamata ORB.init() è risultata essere un operazione del tutto locale. Quando si lavora con VisiBroker, deve essere in esecuzione su ogni LAN, almeno un OSAgent. Si tratta di un processo demone che è responsabile della registrazione dei server e delle richieste di servizio dei client. Quando un processo ha bisogno di effettuare un operazione che coinvolge l'OSAgent, deve per prima cosa capire su quale host della LAN esso si trova. Per ottenere questa informazione viene effettuato un broadcast iniziale. Da quel momento in poi la comunicazione tra le due entità sarà di tipo punto-punto e basata sul protocollo UDP. L'operazione di inizializzazione dell'ORB non cerca l'OSAgent, che viene ritrovato solo al momento di bisogno effettivo. In ogni caso essendo presente la condivisione tra gli agenti, delle librerie di sistema, e quindi anche quelle che realizzano l'ORB, il broadcast viene effettuato una volta per tutte e l'eventuale comunicazione con l'OSAgent sarà poi sempre di tipo punto-punto.

Resta da considerare solamente l'aspetto che riguarda la sicurezza. I tempi di migrazione nativa rilevati con il sistema di crittografia attivato, sono molto più elevati dei tempi presi con i sistemi di sicurezza abbassati.

Queste differenze sono dovute alla complessità degli algoritmi di cifratura e riguardano quindi totalmente l'aspetto sicurezza. La sicurezza è sempre un punto molto delicato, ed è probabilmente l'unica cosa per la quale siamo disposti a pagare overhead così elevati. I valori della migrazione MASIF non sono influenzati dall'attivazione del sistema di sicurezza e per questo non sono riportati. Quando un agente viene passato come parametro della receive_agent è trasmesso via rete in chiaro. Si sarebbe potuto decidere di passare lo stato dell'agente dopo averlo crittografato, cosa molto facile con l'utilizzo dei sistemi di sicurezza di cui SOMA già dispone (bastava creare un TransCommand e serializzare questo al posto dell'agente [Tenti98]), ma in questo modo i sistemi non SOMA non sarebbero più stati in grado di decifrare il messaggio. Dovendo sempre tenere presente che si sta realizzando un meccanismo standard di migrazione, l'unico modo per fornire sicurezza è quello di rendere anche essa standard. Ecco allora che il problema della sicurezza viene rimandato in attesa che si possa usufruire di una implementazione dell'ORB che offra i meccanismi di sicurezza specificati da CORBA.

In realtà esiste un caso in cui il sistema di sicurezza è utilizzato anche dalla migrazione MASIF, ossia quando l'agente deve cambiare place rimanendo nello stesso Agent System o quando un agente che arriva in per la prima volta in un Agent System chiede di essere messo in esecuzione in un place che non sia quello di default. Il trasferimento tra il place di default di un Agent System (chiamato anche place gestore) e un qualunque place dello stesso Agent System avviene attraverso una migrazione nativa, quindi (se è attivato il sistema di crittografia) è sicuro e di conseguenza ha un costo molto elevato (si veda il paragrafo 5.3.1.).

Per concludere questo paragrafo vengono riportati per esteso i valori dei risultati dai quali sono stati ricavati i valori medi riportati nella tabella 1.

FASE DI INIZIALIZZAZIONE				SECONDA FASE				TOTALI		
					Durata Invocazione Remota					
	COS	MAF	Ricerca		Caricamento			Fase	Seconda	
ORB	Naming	Finder	destin.	Serializ.	Spediz.	e deserializ.	Altro	Iniz	Fase	TOTALE
0	611	0	20	1041	61	3565	10	631	4677	5308
20	531	10	10	1122	50	4056	10	571	5238	5809
10	531	10	10	891	70	3976	10	561	4947	5508
0	541	0	20	1071	61	2974	10	561	4116	4677
0	691	0	30	681	59	3085	10	721	3835	4556
0	811	10	10	691	60	3575	10	831	4336	5167
10	531	10	20	681	70	2864	0	571	3615	4186
10	531	10	20	1201	50	3395	20	571	4666	5237
10	531	10	20	841	431	2894	0	571	4166	4737
10	531	10	10	691	200	3595	10	561	4496	5057
10	531	10	10	831	420	3135	0	561	4386	4947
0	541	10	10	691	490	3816	0	561	4997	5558
0	541	10	20	1051	61	5838	10	571	6960	7531
0	540	0	20	1132	350	4146	10	560	5638	6198
10	531	10	20	691	190	4036	0	571	4917	5488
10	531	10	20	821	380	3235	0	571	4436	5007
10	812	0	20	691	730	3235	0	842	4656	5498
10	881	0	20	1022	69	3856	10	911	4957	5868
10	741	10	20	691	219	2234	0	781	3144	3925
10	541	0	150	1092	59	3876	10	701	5037	5738
0	541	10	20	1071	61	3875	0	571	5007	5578
0	540	10	441	691	61	1882	10	991	2644	3635
0	540	0	40	1142	1011	3145	20	580	5318	5898
10	540	0	31	1932	60	2644	0	581	4636	5217
0	541	10	10	1392	80	3445	0	561	4917	5478
10	541	0	20	691	1462	2073	10	571	4236	4807
10	530	10	20	1473	120	3064	0	570	4657	5227
10	601	0	20	871	71	2323	10	631	3275	3906
0	541	0	20	691	141	1752	0	561	2584	3145
0	540	10	10	1342	60	2734	0	560	4136	4696
0	541	10	30	1402	60	2143	10	581	3615	4196
0	541	0	20	1141	231	2023	10	561	3405	3966
10	531	10	10	691	190	2774	10	561	3665	4226
0	541	0	20	1191	481	2394	10	561	4076	4637
0	541	0	20	681	180	2574	10	561	3445	4006
10	531	10	10	1202	60	2203	20	561	3485	4046
0	541	0	20	821	71	3445	20	561	4357	4918
10	540	11	20	1191	711	2814	10	581	4726	5307
10	531	10	20	1171	71	2213	10	571	3465	4036
0	541	0	20	1342	61	2553	10	561	3966	4527
10	540	0	20	982	120	3455	0	570	4557	5127
10	530	10	10	691	492	2423	10	560	3616	4176
0	761	10	10	691	591	3775	0	781	5057	5838
10	901	10	10	690	60	2544	10	931	3304	4235
0	911	0	20	691	80	2634	10	931	3415	4346
10	540	0	20	691	71	2083	10	570	2855	3425
10	531	10	20	1051	70	2714	10	571	3845	4416
10	541	0	20	691	70	3455	10	571	4226	4797
0	541	10	10	691	480	2845	0	561	4016	4577
0	821	60	20	1011	220	2504	10	901	3745	4646
10	530	10	20	691	180	1823	20	570	2714	3284
10	611	10	150	691	1242	2223	0	781	4156	4937
0	541	0	20	691	70	4006	10	561	4777	5338
10	541	10	20	1432	60	3420	10	581	4922	5503
6	587	7	31	947	243	3025	8	631	4223	4854

Tabella 4.Tempi di migrazione MASIF di un agente di dimensioni pari a 50Kbyte.

FASE DI INIZIALIZZAZIONE				SECONDA FASE				TOTALI		
					Durata Invocazione Remota					
	COS	MAF	Ricerca		Caricamento			Fase	Seconda	
ORB	Naming	Finder	destin.	Serializ.	Spediz.	e deserializ.	Altro	Iniz	Fase	TOTALE
10	822	0	20	30	20	1432	10	852	1492	2344
10	541	0	20	30	10	1392	10	571	1442	2013
10	541	0	20	30	20	2223	0	571	2273	2844
10	541	0	30	30	351	1752	20	581	2153	2734
0	661	10	20	30	19	481	0	691	530	1221
10	541	0	20	30	20	441	0	571	491	1062
10	541	0	20	30	19	2364	0	571	2413	2984
10	531	10	30	30	10	651	10	581	701	1282
10	1041	10	11	30	19	1272	0	1072	1321	2393
10	671	0	20	30	21	2413	0	701	2464	3165
10	541	0	20	30	19	3806	10	571	3865	4436
0	541	10	20	30	20	1472	0	571	1522	2093
0	721	10	10	30	170	1402	10	741	1612	2353
0	541	10	20	40	461	3064	0	571	3565	4136
0	731	10	10	30	11	1852	10	751	1903	2654
0	541	10	10	30	140	731	0	561	901	1462
10	1362	10	20	30	20	1472	10	1402	1532	2934
0	541	10	20	30	19	2474	10	571	2533	3104
0	541	10	10	591	30	2844	10	561	3475	4036
0	541	10	30	30	221	2954	0	581	3205	3786
10	832	10	210	30	19	2284	10	1062	2343	3405
10	591	10	10	40	10	2774	10	621	2834	3455
10	540	0	20	30	21	2143	0	570	2194	2764
0	551	0	30	30	210	701	10	581	951	1532
10	541	10	10	40	11	1872	10	571	1933	2504
10	541	10	10	40	20	1933	10	571	2003	2574
10	541	0	20	30	20	1422	0	571	1472	2043
0	541	10	10	30	20	861	0	561	911	1472
10	711	0	431	30	81	1091	0	1152	1202	2354
10	541	0	20	30	140	631	0	571	801	1372
10	531	10	20	30	10	1282	10	571	1332	1903
10	551	10	10	40	10	1903	0	581	1953	2534
10	531	10	20	30	10	1502	20	571	1562	2133
10	541	10	10	40	150	1232	10	571	1432	2003
10	541	0	20	30	500	1042	0	571	1572	2143
10	541	0	20	30	20	671	0	571	721	1292
0	541	10	190	30	30	571	0	741	631	1372
0	540	10	11	40	9	571	10	561	630	1191
10	541	10	10	40	120	611	0	571	771	1342
10	530	10	20	31	9	601	20	570	661	1231
0	1102	10	30	370	161	781	20	1142	1332	2474
0	541	40	20	30	30	631	0	601	691	1292
10	1022	0	20	30	191	981	10	1052	1212	2264
10	971	10	20	30	170	571	10	1011	781	1792
10	741	0	20	30	31	1131	0	771	1192	1963
10	541	0	20	30	71	961	10	571	1072	1643
10	541	0	20	30	20	2073	10	571	2133	2704
10	571	10	20	30	10	611	20	611	671	1282
0	550	0	20	41	9	2414	10	570	2474	3044
10	541	0	80	30	651	1622	10	631	2313	2944
0	540	10	10	30	721	611	10	560	1372	1932
0	711	10	20	30	31	2133	10	741	2204	2945
10	721	50	60	30	40	2834	20	841	2924	3765
10	631	0	70	30	320	521	0	711	871	1582
7	631	7	35	48	102	1483	7	680	1640	2320

Tabella 5.Tempi di migrazione MASIF di un agente di dimensioni pari a 5Kbyte.

5.9. Ricezione di un agente Grasshopper

Per verificare le nuove capacità di SOMA riguardanti l'interoperabilità sono state fatte alcune prove di interazione con il sistema ad agenti Grasshopper [GRAS98]. La ricezione di un agente Grasshopper nel sistema SOMA è la prova più significativa che si può scegliere. In questo modo è possibile testare l'elemento più importante dell'implementazione SOMA di MASIF: la mobilità. Per questo motivo è stato scritto un agente Grasshopper (il cui codice è riportato sotto) che dopo avere ottenuto un riferimento al MAFAgentSystem di destinazione, costruisce i parametri necessari all'invocazione della receive_agent e migra. I due agent system sono (l'agency Grasshopper e il place SOMA) stati attivati con un contesto comune di nomi in maniera tale da consentire all'agente il ritrovamento di entrambe. Le classi necessarie all'agente (comprese quelle di sistema) sono state automaticamente richieste e trasferite. Il risultato ottenuto può essere considerato buono in quanto tutte le fasi della migrazione sono avvenute in maniera corretta.

Ecco il codice dell'agente Grasshopper:

```
import org.omg.CosNaming.*;
import java.io.*;
import java.util.Properties;
import CfMAF.*;
import org.omg.CORBA.*;
import de.ikv.grasshopper.agency.MobileAgent;
import de.ikv.grasshopper.type.ServiceInfo;
import de.ikv.grasshopper.type.ServiceSpecification;
import de.ikv.grasshopper.util.GrasshopperConstants;
import de.ikv.grasshopper.type.ClassQualifier;
```

```
public class GrassAgent extends
de.ikv.grasshopper.agency.MobileAgent {

public void live(){
 System.out.println("** Sono un agente Grasshopper appena **");
 System.out.println("** nato che vuole andare su Vivaldi  **");
 byte[] b={0};
 short[] s={0};

 try{
        Properties p=new Properties();
        p.put("ORBservices","CosNaming");
        p.put("SVCnameroot","SOMACosNaming");
        ORB orb = ORB.init((String[])null,p);
        org.omg.CORBA.Object nameServiceObj =
          orb.resolve_initial_references("NameService");
        NamingContext nameService =
          NamingContextHelper.narrow(nameServiceObj);

        //MAFAgentSystem di destinazione
        NameComponent[] location= {new NameComponent("CfMAF",""),
         new
nameComponent("Minni*MAFAgentSystem_Dom2_Vivaldi*4","")};
        MAFAgentSystem MAFdest =

MAFAgentSystemHelper.narrow(nameService.resolve(location));

        //MAFAgentSystem di origine
        NameComponent[] mafnameOr={ new NameComponent("CfMAF",""),
         new NameComponent("Chitarra","")};
        MAFAgentSystem MAForig =

MAFAgentSystemHelper.narrow(nameService.resolve(mafnameOr));

        ServiceInfo si=getInfo();
        ServiceSpecification
srv_spec=si.getServiceSpecification();

        //Ottieni code_base relativo all'agente
        String code_base=si.getCodebase();

        //Costruzione dell'identificatore MASIF
        byte[] principal =
         si.getServiceSecurityRelated().getOwnership();
        byte[] id=si.getIdentifier().toByteArray();

        Name name =
         new Name(principal,id,GrasshopperConstants.GRASSHOPPER_System);

        //Costruzione del profilo dell'agente
        short major_version=getMajorVersionNumber();
        short minor_version=getMinorVersionNumber();
        String agent_system_description="GRASSHOPPER agency";
        short agent_system_type=srv_spec.getAgentSystemType();
        AgentProfile agent_profile=new AgentProfile(
               srv_spec.getLanguageId(),
               agent_system_type,
               agent_system_description,
               major_version,
               minor_version,
```

```
                GrasshopperConstants.JavaObjectSerialization,
                new Any[0]);

        //Non specifcando le class si adotta una politica on
demand
        ClassName[] class_name=new ClassName[0];

        //Serializzazione dell'agente

        byte[] agent;
        ByteArrayOutputStream ob=new ByteArrayOutputStream();
        ObjectOutputStream oo=new ObjectOutputStream(ob);
        oo.writeObject("run2");
        oo.writeObject((Object)this);
        agent=ob.toByteArray();

        //Migrazione
        MAFdest.receive_agent(name,agent_profile,agent,"Bologna",
                     class_name,code_base,MAForig);

        //Libera l'agency grasshopper dall'agente
        remove();
 }
 catch(Exception e){
        System.err.println("Errore. Trasferimento non
completato.");
        e.printStackTrace();
 }
}

public void run2(){
 System.out.println("Sono giunto nel sistema SOMA, nel place
Bologna");
 }
}
```

La scrittura di un agente mobile Grasshopper prevede l'estensione della classe `de.ikv.grasshopper.agency.MobileAgent` *e l'implementazione del metodo* `live` *che viene invocato dopo ogni migrazione. Spetta al programmatore fare in modo che ad ogni successiva invocazione di* `live` *si ottenga l'esecuzione di una differente porzione di codice. In questo caso con la migrazione attraverso MASIF è possibile specificare il metodo che si vuole venga eseguito all'arrivo nel nuovo AgentSystem (nell'esempio il metodo "run2").*

Conclusioni

Il grande numero di sistemi ad agenti mobili che sono stati sviluppati dimostra che questa tecnologia si sta sempre più affermando. Gli esperimenti compiuti svelano capacità che consentono di superare brillantemente i problemi che si incontrano durante lo sviluppo di applicazioni di rete flessibili, aperte, e scalabili perfino su reti globali come Internet.

Per consentire che questa tecnologia si diffonda in un ambiente eterogeneo bisogna superare i problemi legati alle differenti architetture hardware e software che sono presenti sulla rete delle reti, e fornire la possibilità a diversi agenti e sistemi di interagire tra loro.

L'utilizzo di una macchina virtuale quale la Java Virtual Machine (JVM) consente di ottenere l'indipendenza dalla piattaforma hardware, e permette in questo modo lo spostamento di codice in formato binario, eliminando tutti i problemi legati al trasferimento di codice sorgente (unica alternativa per la migrazione di applicazioni tra host con differenti architetture hardware) quali la ricompilazione all'arrivo, la traduzione tra differenti linguaggi nonché i problemi legati alla sicurezza. Per questo motivo la scelta del linguaggio di implementazione di SOMA è ricaduta su Java, per ora l'unico linguaggio per la JVM, che inoltre sta diventando il linguaggio standard della programmazione su Internet.

L'uniformità a livello di macchina virtuale non basta a garantire che le differenti architetture software siano in grado di interagire. Per questo scopo è necessario fornire degli standard che consentano la libertà di sviluppo di nuovi sistemi e tecnologie, e allo stesso tempo, possano divenire un ponte di collegamento utilizzabile dalle differenti applicazioni.

Lo standard CORBA si è rilevato uno strumento in grado di fornire un meccanismo di comunicazione potente e allo stesso tempo semplice da utilizzare per superare le barriere dovute all'eterogeneità. Il rafforzamento della relazione tra CORBA e SOMA ha inoltre consentito un ancora più facile utilizzo delle possibilità di comunicazione, consentendo ai programmatori di agenti mobili di impegnarsi sempre ad alto livello, senza dover mai entrare in dettagli implementativi di scarsa rilevanza progettuale.

Accanto a CORBA, l'implementazione dello standard MASIF ha consentito di ottenere un ulteriore grado di interoperabilità, rendendo SOMA capace di accettare agenti mobili di altri sistemi e realizzando così un meccanismo di migrazione in ambiente eterogeneo.

Per verificare le nuove capacità di SOMA riguardanti l'interoperabilità, sono state fatte alcune prove di prestazione ed alcune di interazione con il sistema ad agenti Grasshopper (il primo sistema commerciale aderente allo standard MASIF). Le rilevazioni dei tempi di migrazione hanno dimostrato che anche con l'utilizzo di un middleware estremamente vario e generale come CORBA si possono ottenere buone prestazioni, che diventano addirittura paragonabili a misure fatte in ambiente non eterogeneo se si ipotizza il principio di località. I test con Grasshopper hanno messo alla prova le capacità di interoperabilità di SOMA dimostrando una reale possibilità di interazione.

Seppure il lavoro svolto abbia messo in evidenza la possibilità di ottenere interazione tra differenti implementazioni di sistemi ad agenti, per consentire un più ampio spettro di interoperabilità saranno necessarie ulteriori evoluzioni. MASIF, insieme a CORBA, ha consentito la maturazione di uno strumento di comunicazione a livello di sistema, ma non ha messo a disposizione un supporto di esecuzione in cui ritrovare le risorse in modo standard. Gli sviluppi futuri dovranno occuparsi di definire le modalità di interazione tra le applicazioni ad agenti e i sistemi. Accanto a questa necessità va posta quella di interazione con sistemi non basati sulla macchina virtuale Java. Per consentire lo scambio di codice in

un tale contesto si dovrà provvedere alla costruzione di ponti di traduzione tra i vari linguaggi utilizzati dai differenti sistemi.

Bibliografia

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

[AgSoc97] AgentSociety Home Page, 1997, http://www.cs.umbc.edu/agentslist.

[BaGP97] M.Baldi, S.Gai, G.P.Picco: “Exploiting Code Mobility in Decentralized and — Flexible Network Management”, Mobile Agents: 1 st International Workshop MA’97, vol.1219, pp.13-26, Lecture Notes on Computer Science, Springer, April 1997.

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

[CORBA95] Object Management Group: “The Common Object Request Broker: Architetture and Specification. Rev 2.0”, OMG Document 96-03-04, 1995.

[CDR95] Object Management Group: “The Common Object Request Broker: Architetture and Specification. Rev 2.0”, OMG Document 96-03-04, cap. 12 par 2.1 “Common Data Rappresentation CDR” , 1995.

[CORBASec95] Object Management Group: “CORBA Security Services Specification”, 1995.

[CSI95] Object Management Group:”Common Secure Interoperability Specification (CSI)”, 1995.

[CugGPV97] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, Giavanni Vigna, “Analyzing Mobile Code Languages”, 1997.

[FIPA97] L.Chiariglione: “FIPA 97 Specification”, Foundation for Intelligent Physical Agents, October 1997.

[FIPACFP97] FIPA Third Call for Proposal, 1997, http://www.cselt.it/fipa.

[FKK96] A.O. Freier, P.L.Karlton, P.C.Kochner. “The SSL Protocol, V.3.0”. Netscape Corporation, March 1996

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

[GRAS98] IKV++: “Grasshopper: An Intelligent Mobile Agent Platform”, 1998, http://www.ikv.de/products/grasshopper.

[JanMD97] Jan Vitek, Manuel Serrano, Dimitri Thanos, “Security and Comunication in Mobile Object Systems”, 1997.

[Java98] “Java Development Kit” JDK Software Version 1.2 (beta2), 1998, http://www.java.sun.com/products/jdk/1.2/docs/, 1998.

[KSE94] Neches R., “The Knowledge Sharing Effort”, 1994, http://www.stanford.edu/knowledgesharing/papers/kse-overview.html.

[KauPS95] Charlie Kaufman, Radia Perlman, Mike Speciner,“ Network Security, Private Comunication in a Public World”, 1995.

[KQML93] External Interfaces Working Group, Specification of the KQML agent-comunication language,1993.

[Labrou94] Labrou Y. & Finin T., A semantic approach for KQML - A general purpose comunication language for software agents, Proceedings of the 3rd International Conference on Information Knowledge Management, November 1994.

[MASIF97] Crystaliz Inc., General Magic Inc., GMD Fokus, IBM Corp.: “Mobile Agent Facility Specification”, Joint Submission supported by Open Group, OMG TC Document, November 1997.

[OBJ97a] Voyager Core Package Thecnical Overview, ObjectSpace press, March 1997.

[OBJ97b] ObjectSpace Voyager, General Magic Odyssey, IBM Aglets: a Comparison, ObjectSpace press, June 1997.

[Orf98] Robert Orfali, Dan Harkey :”Client/Server Programming with Jave and Corba”, Wiley Computer Publishing, 1998.

[PilaOSI91] OSI ISO 9595 Information Tecnology, Open System Interconnection, Common Management Information Protocol Specification, 1991.

[POA98] Object Management Group: “The Portable Object Adapter”, OMG Document 98-02-14, February 1998.

[RMI98] “Remote Method Invocation for Java”, Javasoft Corporation, 1998, http://chatsubo.javasoft.com/current/rmi/index.html.

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

[Tenti98] L. Tenti, Tesi di Laurea presso la Facoltà di Ingegneria, Corso di Laurea in Ingegneria Informatica, Aprile 1998.

[TwoPhase93] J.Gray, A.Reuter: “Transaction processing: cocepts and techniques”, Morgan Kaufmann, 1993.

[Visibroker98] Visigenic “VisiBroker for Java 3.2 Release Notes”, 1998, http://Visigenic/vbroker/docs/vbjrel.html fornito con VisiBroker for Java 3.2.