Introduzione

Internet rappresenta ormai uno strumento di comunicazione unico per diffusione, versatilità e potenza. Al suo interno confluiscono esperienze e competenze fra le più disparate e si stabiliscono rapporti che non avrebbero l'opportunità di esistere altrimenti. La possibilità di accedere a grandi quantità di risorse (informazioni e potenza di calcolo) ha portato alla nascita di nuove esigenze e allo sviluppo di nuovi strumenti studiati appositamente per la rete delle reti. L'ampia disponibilità di informazioni, da un lato, e il forte incremento nel numero di utenti dall'altro, hanno ripetutamente portato a situazioni di congestione inaccettabili. Il miglioramento delle infrastrutture (con la realizzazione, ad esempio, di dorsali veloci) si è dimostrata una soluzione che deve essere applicata in modo continuo vista la crescita, pressoché esponenziale, del traffico sulla rete. La sempre crescente richiesta di applicazioni distribuite ad alto grado di scalabilità non può ottenere una risposta soddisfacente nell'utilizzo delle tecniche classiche di soluzione basate sul modello di programmazione client/server poiché l'utilizzo di tale modello presuppone un forte impegno della banda del canale di comunicazione.

Una delle più recenti proposte per superare il problema della limitatezza della banda è il modello di programmazione ad **Agenti Mobili:** un agente mobile è un oggetto attivo in grado di spostarsi sulla rete alla ricerca delle condizioni che gli permettano di portare a termine il proprio compito; queste condizioni possono concretizzarsi nella necessità di accedere ad un archivio locale ad un certo nodo, o nell'opportunità di coordinarsi con un altro agente, o ancora nell'esigenza di compiere la stessa operazione in una determinata serie di nodi della rete.

Il superamento dei limiti del modello client/server risiede nella capacità degli agenti di assumere dinamicamente, a seconda del bisogno, entrambi i ruoli di cliente o di servitore. Questo porta ad un incremento della flessibilità del

modello di programmazione che permette un approccio più intuitivo nella progettazione e realizzazione delle applicazioni distribuite.

L'adozione di questo modello presuppone la realizzazione di una infrastruttura standard che supporti l'esecuzione degli agenti stessi e l'accettazione di tale infrastruttura a livello pressoché globale.

Il problema che bisogna risolvere affinché questo possa avvenire risiede nel fatto che i proprietari degli elaboratori che formano la rete non permetteranno mai ad un elemento estraneo qual è un agente mobile di eseguire sulle loro macchine fino a quando non riceveranno sufficienti garanzie sul fatto che l'agente non può arrecare danno all'elaboratore. Si tratta, in pratica, di affiancare al modello di programmazione ad agenti mobili un adeguato Modello di Sicurezza. Tale modello di sicurezza dovrà garantire alle parti in gioco, ossia agenti ed elaboratori, un adeguato grado di protezione reciproca in modo tale che gli elaboratori possano proteggersi dagli attacchi di agenti non fidati e che, viceversa, gli agenti possano eseguire sui vari nodi della rete senza dover temere manomissioni o altri comportamenti illeciti.

Nel primo capitolo introduciamo il problema della sicurezza nei sistemi informatici analizzando quali sono le principali minacce e quali le tecniche adottate per affrontarle. Un particolare accento viene posto sul contrasto fra le esigenze di sicurezza e quelle di apertura dei sistemi distribuiti.

Nel capitolo 2 delineiamo le principali caratteristiche dei sistemi ad agenti mobili analizzando, in particolare, il concetto di mobilità e gli strumenti di comunicazione che più si prestano al nuovo modello di programmazione.

Il capitolo 3 entra più specificatamente nel merito di questo lavoro affrontando le problematiche connesse alla sicurezza nei sistemi ad agenti mobili; il tema viene esaminato in termine di controllo delle interazioni fra gli elementi dell'infrastruttura che supporta l'esecuzione degli agenti, gli agenti stessi e gli elaboratori. Attraverso una analisi dei modelli di sicurezza implementati in alcuni sistemi ad agenti attualmente disponibili si evidenziano le caratteristiche che un modello di sicurezza deve avere e i meccanismi che ne sono alla base. Una speciale attenzione è stata dedicata

allo studio del modello di sicurezza implementato nella versione 1.2 beta2 del Java Developer Kit. Poiché Java si pone ormai come standard *de facto* nella programmazione di applicazioni per la rete, diviene prioritario far riferimento alla evoluzione di questo linguaggio.

Il capitolo 4 propone, infine, un modello di sicurezza che, partendo dagli strumenti messi a disposizione dal JDK1.2 beta2, fornisce una risposta ai principali problemi connessi ad un uso effettivo del modello di programmazione ad agenti mobili. Per poter svolgere una valutazione complessiva dei vantaggi derivanti dall'adozione del modello ad agenti mobili è necessario far riferimento ad una situazione di utilizzo reale ossia in presenza di un adeguato modello di sicurezza. A tale scopo il modello implementato è stato sottoposto ad una serie di misurazioni atte alla valutazione del suo impatto sulle prestazioni complessive del sistema.

1 Sicurezza nei sistemi informatici

L'utilizzo dei calcolatori nei vari ambiti delle attività umane ha portato indubbiamente grandi vantaggi. In tutti i campi in cui le risorse principali sono le informazioni e la capacità di elaborarle, il computer è un utile e preciso strumento di memorizzazione e di elaborazione.
Ma ad ogni passo avanti nella storia della tecnologia consegue l'insorgere di nuovi problemi. Così l'uso di un nuovo strumento potrà essere accettato solo se il bilancio tra vantaggi e rischi derivanti dal suo utilizzo denota risultati favorevoli. Nel caso dei calcolatori, uno degli aspetti che ne minano l'utilizzo è il fatto che, per loro natura e complessità, essi pongono una serie di nuovi problemi di sicurezza che vanno adeguatamente affrontati e risolti. Un archivio cartaceo può essere protetto con le sbarre alle finestre ed una guardia giurata alla porta che fa entrare solo il personale autorizzato, ma una banca dati collegata ad internet introduce problemi complessi anche dal punto di vista della sicurezza. Alcune delle minacce cui la società informatica è sottoposta sono ovvie e la letteratura riporta numerosi esempi di attacchi portati ai suoi danni: in sistemi multiutente come UNIX è facile costruire un programma che, sostituendosi alla schermata di login, si impadronisca della password di altri utenti; in una LAN non è difficile ottenere copie dei messaggi che transitano sulla rete; un processo che si registri come file system remoto può appropriarsi di grandi quantità di dati.

1.1 Politiche e meccanismi

Per risolvere questi problemi è necessario adottare precise *politiche di sicurezza* e realizzare i *meccanismi* che permettano di implementarle.

Anche nell'ambito della sicurezza è importante separare le politiche dai meccanismi. I meccanismi stabiliscono *come* fare le cose. Le politiche indicano *cosa* è necessario fare.

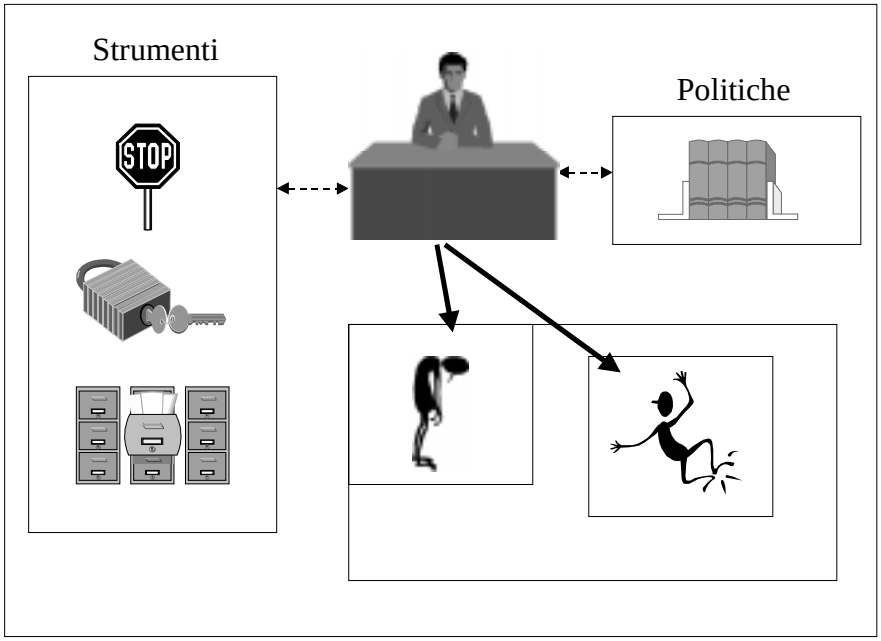

Figura 1: Separazione tra meccanismi e politiche.

Come si vedrà, i diversi ambiti in cui si pone il problema sicurezza evidenziano necessità differenti e, in generale, un diverso modo di intendere il problema stesso. E' ovvio che ogni ambito presuppone politiche di sicurezza differenti realizzate ad hoc in base ai diversi aspetti di ogni scenario. E' comunque prevedibile che molti degli strumenti atti all'imposizione delle diverse politiche siano comuni. Ciò deriva dal fatto che alcuni elementi basilari (i meccanismi appunto) del problema sicurezza non cambiano da un sistema all'altro. In questa ottica va inquadrata la comparsa di pacchetti software che forniscono tutta una serie di meccanismi generali utilizzando i quali una applicazione può imporre una propria politica, ma

anche la presentazione di sistemi che permettono lo sviluppo di politiche complesse ed il loro mappaggio su set di meccanismi differenti.

1.2 Il problema sicurezza

Per sintetizzare i termini del problema potremmo dire che:

- *oggetto del problema* sicurezza di un sistema è l'insieme delle risorse che lo costituiscono e di cui può disporre (siano esse risorse fisiche o informazioni);
- *soggetti del problema* sono gli utenti (siano essi persone, processi, gruppi o quanto altro) che possono accedere al sistema e quindi alle sue risorse;
- la *soluzione del problema* consiste nell'adozione di una adeguata politica di sicurezza che, attraverso l'utilizzo degli appropriati meccanismi, garantisca la protezione delle varie risorse da accessi illegittimi.

Possiamo dire che gli oggetti da difendere appartengono, essenzialmente, a due categorie concettualmente separate, ma implementativamente assimilabili: le *risorse fisiche* (o di sistema) e le *informazioni*.
Occorre poi osservare che nei sistemi distribuiti si potrebbe considerare una terza categoria di risorse ossia i canali di comunicazione; proteggere l'accesso agli end point di un canale di comunicazione (gli end point rientrano, nella nostra classificazione, tra le risorse di sistema) non assicura infatti la protezione delle informazioni che vi transitano poiché la controparte fisica-implementativa del canale virtuale si compone di una rete di componenti hardware e software che non sono sotto il diretto controllo di una unica autorità, mentre tutte le risorse di una macchina sono sotto il diretto controllo del sistema operativo.

Solitamente i soggetti che accedono alle risorse si indicano con il termine *principal* e si usa dire che un processo esegue un determinato compito per conto (on behalf of) di un certo principal.
Il principal sarà anche l'entità di riferimento per l'assegnazione dei *permessi* di accesso. Occorre osservare che con il termine permesso si indicherà una autorizzazione per l'utilizzo di una o più risorse del sistema. Tale definizione non limita in alcun modo né il tipo di utilizzo (azione), né il tipo (target) di risorsa considerata; sono quindi compresi i casi più specifici (come il permesso di modifica di un dato campo di certo record di un DB) e quelli più generali (come il permesso di accesso ad una certa sottorete).
La legittimità di un accesso sarà, in generale, stabilita in base all'*identità* del principal per conto di cui è stata richiesta l'operazione; l'identità potrà essere caratterizzata in molte maniere diverse e fungerà (più o meno direttamente) da chiave di ricerca per la funzione che mappa principal e permessi.

1.3 Minacce e tipi di attacco

Per comprendere quali siano i meccanismi base di un sistema di sicurezza informatico e le linee che guidano alla stesura di una politica occorre stabilire quali sono gli obbiettivi minimi e conoscere i pericoli da cui bisogna difendersi.
Per quanto riguarda la sicurezza delle informazioni possiamo identificare quattro problemi primari:

- il problema della **identità** consiste nel fatto che due controparti che vogliano comunicare, ossia scambiarsi informazioni, debbano prima di tutto identificarsi a vicenda (*mutua autenticazione*);
- il problema della **riservatezza** (o **privacy**) indica la necessità che solo coloro che sono autorizzati possano accedere ad informazioni riservate;

- il problema dell'**integrità** fa riferimento alla necessità di verificare che le informazioni cui si accede non siano state manomesse ;
- il problema della **paternità** indica il bisogno di garantire che l'autore di una certa informazione non possa ripudiarne la paternità.

Il genere di minacce cui può essere sottoposto un sistema di calcolatori, per quanto riguarda la sicurezza, si divide in quattro categorie:

- **leakage,** che indica l'acquisizione di informazioni riservate da parte di utenti non autorizzati;
- **tampering,** che è un attacco di tipo attivo e riguarda la manomissione di dati da parte di utenti non autorizzati;
- **resource stealing,** che indica l'utilizzo di risorse del sistema da parte di utenti non autorizzati;
- **vandalism,** che riguarda quegli attacchi che minano l'integrità e l'operabilità del sistema; un caso particolare, detto *denial of service*, è quello in cui l'utilizzo massiccio di una risorsa impedisce agli altri utenti di accedervi.

L'accesso ad un elaboratore avviene sempre attraverso un canale di comunicazione sia esso virtuale (l'operatore che siede davanti alla console di un MainFrame ha un accesso diretto alla macchina) o fisico (si pensi ad un login remoto). Per attuare le minacce viste bisogna accedere all'elaboratore (o agli elaboratori) e quindi appropriarsi di uno dei canali di accesso.
Nei moderni sistemi distribuiti la connessione ad una rete, permettendo la creazione di canali di comunicazione veloci e flessibili, rappresenta un ottimo mezzo di accesso, ma allo stesso tempo costituisce un comodo bersaglio per i diversi *tipi di attacco*, i quali essenzialmente mirano ad esaminare/modificare le informazioni che vi transitano.

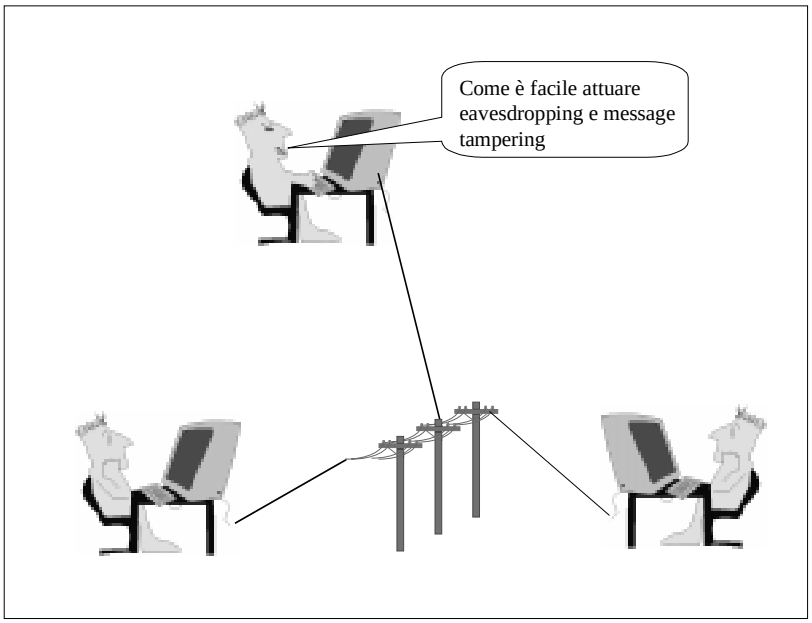

Figura 2: La rilevazione delle informazioni che transitano su di una rete è uno degli attacchi classici di più facile attuazione.

I principali tipi di attacco, ossia le modalità attraverso le quali le minacce viste possono essere attuate, sono:

- **eavesdropping:** letteralmente origliare, comprende tutti quegli attacchi che mirano ad ottenere copia dei messaggi che transitano su di un canale o che non sono adeguatamente protetti in memoria; un caso classico è quello che prevede l'utilizzo di appositi programmi per l'intercettazione e l'esame successivo dei pacchetti che transitano su di un ramo di una rete (*sniffer)*;
- **masquerading:** indica l'assumere momentaneamente l'identità di un diverso principal, impadronendosi, ad esempio, della sua password, per ricevere o spedire messaggi al suo posto o per sfruttare impropriamente i suoi diritti di accesso;

- **message tampering:** denota l'intercettazione e la modifica di messaggi prima che giungano alla legittima destinazione;
- **replaying:** consiste nell'intercettare messaggi, conservarli e spedirli successivamente, quando, ad esempio, l'accesso ad una data risorsa è stato revocato; questo genere di attacco può essere utilizzato per attuare resource stealing o vandalism, senza che sia necessario esaminare il contenuto del messaggio.

Per ottenere l'accesso a qualche macchina del sistema si può far ricorso all'appoggio di un complice interno o affidarsi ad una qualche forma di infiltrazione. Esistono diverse tecniche (virus, worms, cavalli di Troia) che sfruttano debolezze del sistema per aprire un varco nelle sue difese nascondendo le proprie intenzioni dietro sembianze apparentemente innocue o addirittura utili.

1.4 Sistemi aperti vs. sistemi sicuri

Gli argomenti finora affrontati ci permettono di trarre alcune utili conclusioni riguardo alla sicurezza nei sistemi distribuiti:

- il loro principale punto debole è l'apertura dei canali di comunicazione ossia la facilità con cui è possibile accedere alle risorse di comunicazione tra client e server; per tenerne conto è necessario considerare i canali di comunicazione come risorse a rischio ad ogni livello del sistema;
- a causa della distribuzione e della apertura di questi sistemi, occorre partire dal presupposto che ogni comunicazione avviene con una controparte non fidata finché non sia dimostrato il contrario;

Quest'ultima affermazione ci porta ad una osservazione importante: se è vero che, in linea di principio, in un sistema aperto e distribuito ogni entità

andrebbe considerata non fidata e quindi anche i singoli componenti del sistema dovrebbero diffidare gli uni dagli altri, così facendo non sarebbe possibile realizzare sistemi utilizzabili. E' allora necessario considerare un insieme minimo di componenti (la cosiddetta *trusted computing base*) come fidati per costruire su questi l'infrastruttura necessaria alla realizzazione dei meccanismi per la sicurezza.
Volendo calare queste considerazioni in un significativo esempio di applicazione, possiamo dire che, in un sistema distribuito basato su di una architettura client/server, per prevenire le minacce viste occorrerà:

- rendere sicuri i canali di comunicazione per garantire la privacy;
- realizzare client e server in modo tale che, diffidando gli uni dagli altri, essi utilizzino un protocollo che permetta la mutua autenticazione: i server devono avere la garanzia che i client agiscono per conto dei principal che dichiarano, i client devono avere la garanzia che il server cui si rivolgono è l'autentico server fornitore del servizio di cui hanno bisogno;
- assicurare la "freschezza" delle comunicazioni in modo da evitare il message replaying.

In seguito alla diffusione di Internet ed alla conseguente crescita esponenziale del rischio di intrusione per tutte quelle macchine collegate alla rete delle reti alcune softwarehouse hanno realizzato sistemi detti *firewall* con i quali è possibile proteggere una sottorete controllando opportunamente il canale che la collega ad Internet. Si tratta di applicazioni la cui esecuzione è solitamente affidata ad un elaboratore dedicato che funge di fatto da ponte di collegamento tra la sottorete da proteggere e l'esterno. Il firewall opera a livello applicativo regolando la creazione di connessioni per protocolli di alto livello (quali FTP o HTML) attraverso un sistema password. In questo modo solo gli utenti esplicitamente autorizzati possono *attraversare* il firewall.

1.5 I meccanismi

Come abbiamo già osservato la politica che si attua in un certo sistema di sicurezza ha un ruolo molto importante nel garantire la protezione delle risorse. Il punto di forza della politica sta nel riuscire a comprendere le necessità più profonde e peculiari del sistema da proteggere in modo da adattarsi al meglio ad una situazione forse simile ad altre, ma certamente unica. Una buona politica deve cioè essere fatta su misura, ma, se è difficile tracciare una classificazione delle politiche (questo non rientra tra i nostri scopi), è facile invece identificare alcuni meccanismi di utilità generale che ormai tutti i sistemi di sicurezza utilizzano. I meccanismi necessari all'imposizione di un certa politica di sicurezza in un sistema distribuito sono essenzialmente di tre tipi: strumenti per l'**autenticazione**, strumenti per l'**autorizzazione** e meccanismi per l'**access control**. I diversi meccanismi fanno capo a fasi diverse del funzionamento del sistema di sicurezza: gli strumenti di autenticazione ed i meccanismi di access control vengono utilizzati in due momenti specifici e, precisamente, all'ingresso di un soggetto nel sistema, ed ogni qual volta questo tenti di accedere a qualche risorsa. Ci sono poi alcuni strumenti accessori che vengono utilizzati in diverse fasi: la crittografia, ad esempio, entra in gioco ogni volta che occorre nascondere qualcosa (uno dei requisiti minimi per la sicurezza è ad esempio la difesa dall'eavesdropping).

1.5.1 Autenticazione

Nei sistemi non distribuiti ad ogni sessione l'utente viene autenticato dal sistema operativo che di conseguenza fissa i limiti allo spazio in cui l'utente può muoversi; lo stesso sistema operativo regola, attraverso meccanismi hardware-software, gli accessi alle risorse del sistema accentrando su di sé

tutte le funzioni rilevanti ai fini della sicurezza. L'autenticazione di un utente si basa tradizionalmente (vedi [SG94]) sulla combinazione di tre set di elementi: qualcosa che l'utente possiede (come una chiave o una carta magnetica), qualcosa che l'utente conosce (come una password) o qualcosa che l'utente è o meglio un qualche suo attributo (come le impronte digitali o il disegno della retina).

In un sistema distribuito il problema della autenticazione si complica: le entità che cooperano sono client e server che eseguono su macchine diverse, probabilmente su sistemi operativi diversi, a chilometri di distanza; non può esistere una singola autorità che si occupi di regolare l'accesso a tutte le risorse e quindi i diversi meccanismi vanno isolati e sviluppati separatamente. In questa ottica è necessario realizzare un *servizio di autenticazione* che funga da autorità per la generazione, l'immagazzinamento, la gestione, e la distribuzione delle informazioni necessarie alla mutua autenticazione di due qualsiasi controparti.

1.5.2 Autorizzazione

Una volta stabilita e verificata l'identità di un principal sarà necessario definire i confini del suo campo di manovra ossia attribuirgli, attraverso la consultazione della politica di sicurezza attiva, il set di autorizzazioni che gli compete. In poche parole, successivamente all'identificazione, ogni soggetto che vuole accedere al sistema andrà sistemato nel *dominio di protezione* più appropriato (vedi [SG94]). Un dominio di protezione è un contesto (ogni principal si troverà in ogni momento in un determinato dominio) cui viene associato un set di autorizzazioni di accesso che stabilisce le risorse ed il tipo di operazioni su queste che il principal è autorizzato a compiere.

1.5.3 Access control

Una volta stabilito qual è il target di una certa operazione di accesso, il soggetto che vuole eseguire l'operazione e le limitazioni da porre a tale accesso, occorre prevedere un meccanismo in grado di controllare e regolare l'operazione in modo che tutto si svolga come previsto dalla politica di sicurezza.

In un sistema distribuito si ampliano tutte e tre le classi di elementi coinvolti in questa procedura: la classe delle risorse, quella dei principal e quella delle operazioni di accesso. Fra le risorse, che non comparivano come target nei moduli di sicurezza di un sistema operativo non distribuito, ci saranno quelle per l'utilizzo della rete ai vari livelli. Come già detto, i soggetti dei controlli non saranno più solo utenti ma anche processi (che operano più o meno direttamente per conto dei primi o meno) o gruppi. Nel novero delle operazioni critiche, infine, entreranno anche quelle relative all'accesso all'uso della rete e dei suoi servizi ed altre ancora.

L'implementazione più generale del meccanismo di access control si basa sull'utilizzo di una matrice (*matrice di protezione*) che stabilisce, per ogni coppia dominio-risorsa, che tipo di accesso è consentito. In pratica poi questa matrice viene ridimensionata secondo due possibili strategie che riflettono due diversi modi ci caratterizzare il problema dell'accesso: il primo approccio (detto a *lista di accesso*) si pone dalla parte delle risorse nel senso che, per ogni risorsa, vengono specificati i soggetti autorizzati all'accesso, il secondo (detto a *capability*), viceversa, prevede che, per ogni dominio di protezione ossia soggetto, venga specificata la lista delle risorse cui può accedere.

1.5.4 Crittografia

Alcuni dei problemi che abbiamo affrontato possono essere efficacemente risolti utilizzando le tecniche proprie della crittografia. In tutti quei casi in cui la protezione delle informazioni è una necessità di prima importanza la crittografia fornisce tutta una serie di strumenti che hanno ormai raggiunto una notevole maturità.

La crittografia è la scienza che si occupa del problema di rendere inintelleggibile un messaggio per tutti quelli che non ne sono il legittimo destinatario. Ha sviluppato diverse tecniche per realizzare la cosiddetta cifratura (encryption) ossia la manipolazione di un messaggio in modo che solo il destinatario possa leggerlo una volta decifrato.

La cifratura avviene utilizzando un *algoritmo di cifratura* ed una *chiave,* detta appunto *di cifratura,* mentre la decifrazione può essere svolta, attraverso il corrispondente *algoritmo di decifrazione*, solo conoscendo la corrispondente *chiave di decifrazione*. E' evidente che l'algoritmo di decifrazione dovrà eseguire la trasformazione inversa di quella operata dall'algoritmo di cifratura. Tale trasformazione (e di conseguenza anche gli algoritmi che la descrivono) potrà essere segreta o pubblica, nel primo caso mittente e destinatario del messaggio (o in generale dell'informazione) dovranno essere i soli a conoscere la trasformazione, nel secondo la necessità di segretezza avrà come oggetto le chiavi. Di fatto si può guardare ad una chiave come ad un indice che seleziona l'algoritmo voluto in una famiglia di algoritmi della stessa specie, ciò che è pubblico è la struttura della famiglia mentre la segretezza delle chiavi garantisce l'efficacia della trasformazione.

In relazione alla natura delle chiavi utilizzate si parlerà di *algoritmi simmetrici* o *in chiave segreta* e di *algoritmi asimmetrici* o *in chiave pubblica*. I primi utilizzano la stessa chiave per entrambe le trasformazioni (cifratura e decifrazione), utilizzando invece gli algoritmi asimmetrici al destinatario vengono associate due chiavi: una *chiave pubblica* ed una *privata*; in pratica il mittente cifrerà, utilizzando la chiave pubblica, e spedirà

il messaggio al destinatario che, come unico depositario della chiave segreta, sarà il solo in grado di decifrarlo. Ogni possibile destinatario dovrà quindi possedere una coppia chiave pubblica-chiave segreta, le due chiavi sono ovviamente correlate ed è importante osservare che, per garantire l'inviolabiltà delle informazioni cifrate, la deduzione della chiave segreta da quella pubblica deve essere computazionalmente impossibile.
Le tecniche crittografiche permettono la soluzione dei quattro problemi relativi alla protezione delle informazioni:

- il problema della **riservatezza** viene efficacemente affrontato attraverso la cifratura delle informazioni riservate;
- il problema della **integrità** trova nell'utilizzo delle *firme digitali* (*signature*) una naturale soluzione. Una signature è un riassunto firmato del messaggio ottenuto cifrando con la chiave segreta del mittente il risultato dell'applicazione di una certa funzione (*hash*) al messaggio stesso; la signature, che ha dimensione fissa ed indipendente da quella del messaggio, viene spedita insieme al messaggio, il destinatario può così decifrarla con la chiave publica del mittente, ricalcolare l'hash del messaggio e confrontarlo con quello decifrato; se i due corrispondono il destinatario potrà essere sicuro che il messaggio non ha subito manomissioni;
- il problema della **paternità** può essere risolto ancora una volta con le signature; una volta verificata un signature, infatti, il destinatario potrà dimostrare che essa può essere stata prodotta solo dal mittente poiché solo il mittente è in possesso della chiave segreta necessaria;
- il problema della **identità** può essere affrontato con l'utilizzo di differenti protocolli in cui, ad esempio, l'identificando è tenuto a fornire un qualche segreto che l'identificatore verificherà.

2 Sistemi ad agenti mobili

2.1 Introduzione

Negli ultimi anni Internet ha subito una grande espansione trasformandosi in un contenitore d'informazioni e servizi che si estende in tutto il mondo. Con il crescere del numero di macchine collegate è cresciuta la connettività, ma allo stesso tempo sono aumentati i tempi di latenza e la probabilità di incorrere in errori della rete.

Nata come semplice interconnessione delle reti è divenuta in breve tempo uno strumento molto più flessibile e complesso, ma allo stesso tempo anche molto più caotico. Le nuove tecniche per le comunicazioni sicure ed il costo ridotto per l'accesso alla rete ne hanno aumentato l'attrattiva agli occhi delle aziende in cerca di canali per il coordinamento delle varie sedi e più in generale di strumenti di comunicazione rapidi e flessibili. Il mito del villaggio globale è ormai a portata di mano anche perché le più svariate organizzazioni hanno allestito la loro pagina web attraverso cui mantengono una rete di collegamenti difficilmente gestibile altrimenti. La recente comparsa dei cosiddetti assistenti digitali personali (PDA) e la possibilità di connettere alla rete un computer portatile hanno esteso ulteriormente le possibilità di collegamento, ma, hanno introdotto nuovi problemi legati alla connettività intermittente ed alla limitata banda passante di tali collegamenti.

Ormai è lecito pensare ad internet come ad un immenso computer globale e ci si può chiedere in che modo sia possibile programmarlo. In questa ottica, la cosiddetta programmazione di rete ha acquisito ultimamente un sempre più crescente interesse cercando di stabilire il modo migliore per sfruttare le nuove possibilità. E' chiaro ormai che l'estensione di tecnologie e modelli consolidati alla nuova realtà è inapplicabile non solo per questioni di connettività e scalabilità (i tentativi di adattare le tecnologie dei sistemi

operativi distribuiti alle WAN sono falliti fornendo comunque un utile insegnamento), ma soprattutto perché le soluzioni tradizionali trascurano aspetti propri di questo nuovo scenario quali:

- la distribuzione geografica, fonte di latenze che occorre prendere in considerazione;
- la mancanza di uno schema globale dei dati;
- la necessità di introdurre il concetto di qualità di servizio per poterlo associare alle varie risorse;
- la presenza di domini chiusi protetti da firewalls.

Bisogna, in sintesi, confrontarsi con un insieme di nuovi fenomeni che vanno adeguatamente analizzati e controllati.

2.2 La crisi del modello cliente-servitore

Il modello cliente-servitore (e le sue variazioni) divenuto in poco tempo il paradigma standard nella costruzione di applicazioni distribuite mostra ormai i suoi limiti dovuti essenzialmente all'uso inefficiente del canale di comunicazione, nonostante l'incremento di prestazioni della rete conseguente all'utilizzo di dorsali molto veloci e di macchine sempre più potenti nei nodi intermedi. Inoltre, la flessibilità richiesta alle moderne interfacce pretende dai servitori che forniscono i vari servizi una granularità di intervento che li rende componenti complessi, difficili da gestire e da aggiornare. Infine, le esigenze del cosiddetto mobile computing sono difficilmente gestibili con un modello quale quello cliente-servitore che richiede il mantenimento di un canale permanente e lo scambio di una certa quantità di messaggi anche per l'esecuzione di compiti relativamente semplici.
Per risolvere questi problemi sono stati proposti approcci specifici che tendono ad ampliare il modello originale per aumentarne le capacità. Sono

stati studiati linguaggi per la comunicazione tra cliente e servitore estesi attraverso costrutti di più alto livello espressivo per poter adattare l'interfaccia del servitore ai bisogni del cliente. Sono stati creati i *proxy*, una sorta di rappresentanti, che fanno le veci di quelle macchine che si connettono solo ad intermittenza. L'uso di *linguaggi script* e di altri linguaggi che eseguono su di una macchina virtuale invece che sull'architettura reale dell'elaboratore hanno permesso di superare lo scoglio della eterogeneità delle architetture. E' cosi stato possibile sperimentare l'utilizzo dei cosiddetti executable content (documenti che incorporano una parte di codice che viene eseguito quando si accede al documento per aggiungere effetti quali filmati o colonne sonore).

Tutti questi tentativi si sono però dimostrati in fretta quali quelli che sono, ossia semplici aggiustamenti ed estensioni di un modello che ormai ha segnato il passo.

2.3 La necessità di nuovi approcci

La ragione prima dell'inadeguatezza del modello client/server risiede nel fatto che nacque nell'ambito delle reti locali in cui il costo di un canale permanente e la sua affidabilità possono, con buona approssimazione, essere assunti come dati di fatto e l'adozione di semplici protocolli ottimistici è, di conseguenza, accettabile (per ragioni analoghe le tecniche proprie dei sistemi operativi distribuiti hanno avuto notevole applicazione nel contesto delle reti locali). Ma il contesto è cambiato, se non altro per la dimensione della rete considerata, e solo un approccio differente al problema può portare al superamento di questo ostacolo ed aprire nuove strade.

L'instaurarsi di un ambiente fisicamente ed artificialmente partizionato in diversi domini ha poi fatto sì che la trasparenza alla allocazione di risorse e processi (obiettivo principe dei sistemi operativi distribuiti) passasse in

secondo piano e acquistasse, viceversa, importanza la capacità di adattare l'esecuzione dei processi all'ambiente in cui si trovano ad agire.
Risulta così evidente la necessità di affrontare la sfida della programmazione della rete con approcci innovativi che, forti degli insegnamenti forniti dalle esperienze precedenti e consci dell'ampiezza del problema, riescano a fornire una risposta soddisfacente, flessibile, coerente e convincente. Tale risposta è destinata a diventare la base per la creazione di una nuova generazione di servizi informatici.
In questo scenario si inserisce l'approccio ad agenti mobili la cui novità non sta nella capacità di risolvere problemi altrimenti non attaccabili, e neppure in una singola capacità specifica che lo rende vantaggioso rispetto agli approcci precedenti, ma nel fatto che racchiude in sé una gamma di vantaggi e possibilità più ampia e interessante nell'ottica dello sfruttamento delle nuove possibilità offerte dalla rete.

2.4 Agenti ed agenti mobili

Per stabilire cosa sono questi agenti mobili occorre innanzitutto puntualizzare cosa intendiamo con la parola **agente** ed in che termini ne interpretiamo la mobilità. Il termine agente è usato in vari contesti con significati diversi e non sempre ben chiari. Nell'ambito dell'intelligenza artificiale si parla di "agenti intelligenti" per identificare entità autonome cui si richiede la capacità di esibire un comportamento sociale ed "intelligente". Nell'ambito della robotica gli agenti sono piccole macchine in grado di muoversi autonomamente in un certo ambiente più o meno noto nell'inseguimento di un certo obiettivo. Alcuni programmi per la scelta della posta elettronica affidano a propri "agenti" la scansione della posta in arrivo per scegliere quella di interesse. Infine il termine "**agenti mobili**" compare ripetutamente in una serie di articoli più o meno recenti vicino a parole come sistemi

distribuiti e mobilità del codice; questi articoli descrivono nuovi ambienti di programmazione che estendono per lo più vecchi linguaggi con l'inserimento di una nuova astrazione diversamente caratterizzata dai vari autori, quella degli agenti.
Volendo allora fare una caratterizzazione degli agenti possiamo dire che gli agenti:

- possono assumere dinamicamente sia il ruolo di cliente che quello di servitore;
- sono entità autonome ossia hanno propri obiettivi ed un proprio flusso di esecuzione;
- sono inseriti in un certo ambiente di esecuzione di cui percepiscono alcune caratteristiche in base alle quali modificano il loro comportamento;
- hanno visibilità della loro allocazione ossia sanno sempre dove si trovano;
- dovranno rispettare certe "regole" imposte dalle macchine che ne ospitano l'esecuzione;
- agiscono, più o meno direttamente, per conto di un determinato utente.

Gli agenti mobili, hanno la capacità di migrare ossia di interrompere la propria esecuzione, "spostarsi" su di un'altra macchina, non necessariamente con la stessa architettura, e riprendere qui ad eseguire come se nulla fosse successo (a parte il cambiamento di ambiente che, come detto sopra, deve necessariamente essere percepito).

2.5 Il "modello" ad agenti mobili: aspetti salienti

Le virgolette attorno alla parola modello sono d'obbligo se si considera che ancora non si è raggiunto un accordo unanime su cosa si debba intendere per

agente e che quindi non esiste ancora un vero modello complessivo ed organico. I diversi aspetti che occorre considerare nella realizzazione di un tale modello sono però ormai chiari e molti ricercatori li stanno indagando.

2.5.1 Mobilità

Quello di mobilità è un concetto che sorge spontaneamente nel contesto delle reti di calcolatori, in quanto è normale, avendo a disposizione un canale di comunicazione tra due macchine, pensare di utilizzare questo canale per spostare informazioni, dati, programmi. Il concetto di mobilità del codice (*code mobilty*) è già stato esplorato, ne sono esempi l'uso del linguaggio PostScript per il controllo delle stampanti laser e le ricerche fatte nell'ambito dei sistemi operativi distribuiti, dove viene utilizzato come meccanismo per la soluzione del problema del load balancing, ma anche gli studi (in un certo senso precursori) svolti riguardo al modello *Remote Evaluation* (REV, vedi [SG90]) o al suo complementare *Code On Demand* (COD, vedi [FPV96]).
Per capire meglio le novità del modello ad agenti mobili analizziamo la macchina virtuale che supporta questo modello ed i diversi aspetti che contraddistinguono la mobiltà.

2.5.1.1 La macchina virtuale dei sistemi ad agenti mobili

I sistemi che permettono la code mobility si differenziano dai tradizionali sistemi distribuiti innanzi tutto per la struttura della macchina virtuale (vedi [FPV96]) su cui eseguono le loro applicazioni.

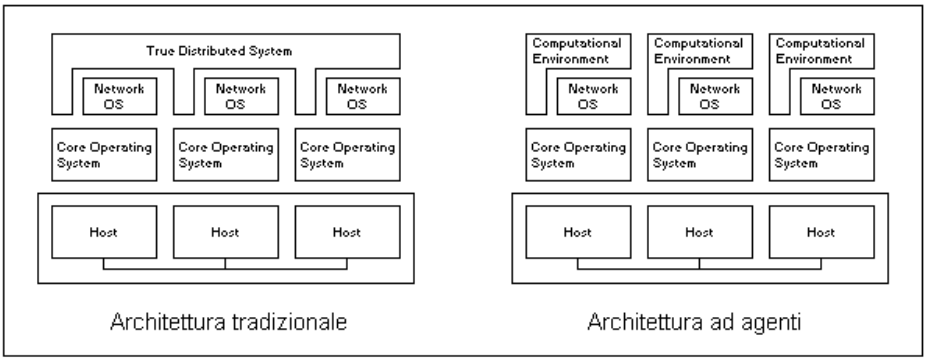

Figura 3: La caratteristica peculiare dell'architettura di un sistema ad agenti è il fatto che la locazione diventa un parametro fondamentale, in base al quale gli agenti modificano il loro comportamento.

Per riflettere i diversi obiettivi e le diverse assunzioni di base in una architettura tradizionale, sopra il sistema operativo di rete veniva costruito uno strato detto True Distributed System (vedi Figura 3) che realizzava, tra l'altro, la trasparenza alla allocazione ed il load balancing. In sistemi ad agenti tale strato esterno è stato sostituito con un Computational Environment (CE) che non nasconde più la propria località essendo divenuto requisito essenziale la possibilità da parte degli agenti di conoscere con esattezza la propria "posizione". In pratica il CE fornisce i meccanismi per la mobilità vera e propria e, come vedremo, tutta una serie di altri servizi correlati.

2.5.1.2 I soggetti della mobilità

Le entità che un CE si trova a gestire si possono suddividere in due classi:

- *Computazioni*: ossia i processi, i thread o comunque i flussi di esecuzione che stanno eseguendo sotto il controllo del CE;
- *Risorse*: ossia quelle entità passive che possono essere condivise da diverse computazioni, come ad esempio i file.

Le computazioni sono l'insieme di un codice e di uno stato, più o meno complesso, ma scomponibile in due parti fondamentali lo *stato della esecuzione* e lo *spazio dei dati*. Quest'ultimo può essere utilmente pensato come l'insieme dei riferimenti agli "oggetti" cui la computazione può accedere.
In linea di principio, codice, stato della esecuzione e spazio dei dati potrebbero spostarsi indipendentemente, ma non tutte le combinazioni possibili avrebbero una utilità pratica. Ha senso però analizzare separatamente la mobilità dello stato di esecuzione e dello spazio dei dati in relazione a quella del codice.

2.5.1.3 La mobilità del codice e dello stato della computazione

Questa caratterizzazione ci permette di tracciare una prima linea di demarcazione fra i diversi meccanismi per la mobilità del codice (vedi Figura 4):

- per *Mobilità forte* si intende la contemporanea migrazione del codice e dello stato della esecuzione;
- con *Mobilità debole* si indica la migrazione del solo codice.

La mobilità forte contempla due forme di spostamento: la migrazione vera e propria in cui la computazione sospende il proprio flusso di esecuzione, congela il proprio stato, si sposta su di un'altra macchina e qui riprende l'esecuzione da dove si era interrotta ripristinando lo stato. In alternativa è possibile che la computazione cloni se stessa inviando una propria copia su di un altro calcolatore.

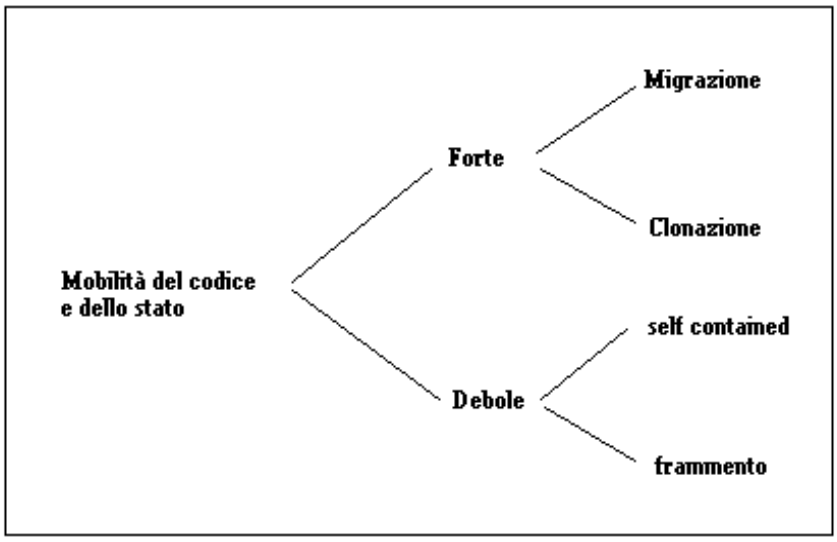

Figura 4: I meccanismi per la mobilità del codice si dividono innanzitutto in meccanismi per la mobiltà forte e meccanismi per la mobiltà debole.

Per quanto riguarda la mobilità debole si può distinguere fra i casi in cui a migrare è un codice autocontenuto che provoca l'istanziazione di una nuova computazione nel CE di destinazione, e quello in cui si sposta un frammento di codice, quale ad esempio una procedura, che viene linkato nel contesto di una computazione già esistente.

A ben guardare mobilità forte e debole stanno in un preciso ordine cronologico, infatti, in riferimento a quanto affermato nell'introduzione, la mobilità debole è stata esplorata prima di quella forte come meccanismo per estendere il modello cliente-servitore e dotarlo di funzionalità più avanzate e

maggiore flessibilità (lo stessso Java ne è un esempio). Molti dei problemi incontrati in questi studi e di soluzioni proposte sono stati ripresi ed utilizzati anche nelle più recenti ricerche sulla mobilità forte, fra questi la ricerca di un linguaggio dotato di una potente semantica per lo scambio di messaggi fra agenti e la realizzazione di meccanismi per affrontare l'eterogeneità delle architetture.

2.5.1.4 La mobilità dello spazio dei dati

Per quanto concerne la mobilità dello spazio dei dati, avendolo definito come l'insieme dei riferimenti alle risorse accessibili alla computazione (vedi paragrafo 2.5.1.2), dovremo, per prima cosa, stabilire come modellare le risorse e stabilire di quali tipi possono essere i riferimenti. Consideriamo una risorsa come una terna formata da un *identificatore*, un *tipo*, ed un *valore* (intendendo valore in senso lato, ad esempio il contenuto di un file o di una area di memoria, ma anche il driver di una stampante). Il tipo della risorsa determina sia la sua struttura che la sua interfaccia, e stabilisce se la risorsa sia *trasferibile* (un file) o meno (una stampante o un file di configurazione del sistema).

I diversi tipi di riferimento sono soggetti a trattamenti differenti in seguito alla migrazione dello spazio dei dati di una computazione. Essi possono essere catalogati, in ordine decrescente di consistenza, come *riferimenti per identificatore*, *per valore* e *per tipo*. I primi, *per identificatore*, sono quelli in cui la risorsa è unica e non può essere sostituita da una diversa dello stesso tipo (ad esempio la porta cui è collegata una ben precisa stampante). I riferimenti *per valore* sono quelli in cui la risorsa originaria può essere sostituita da una dello stesso tipo e valore (ad esempio un file di testo). In quelli *per tipo* si richiede solo l'uguaglianza del tipo (un agente che necessita

di un display per interagire con l'utente può, in seguito alla migrazione, ripristinare il collegamento con il display locale).

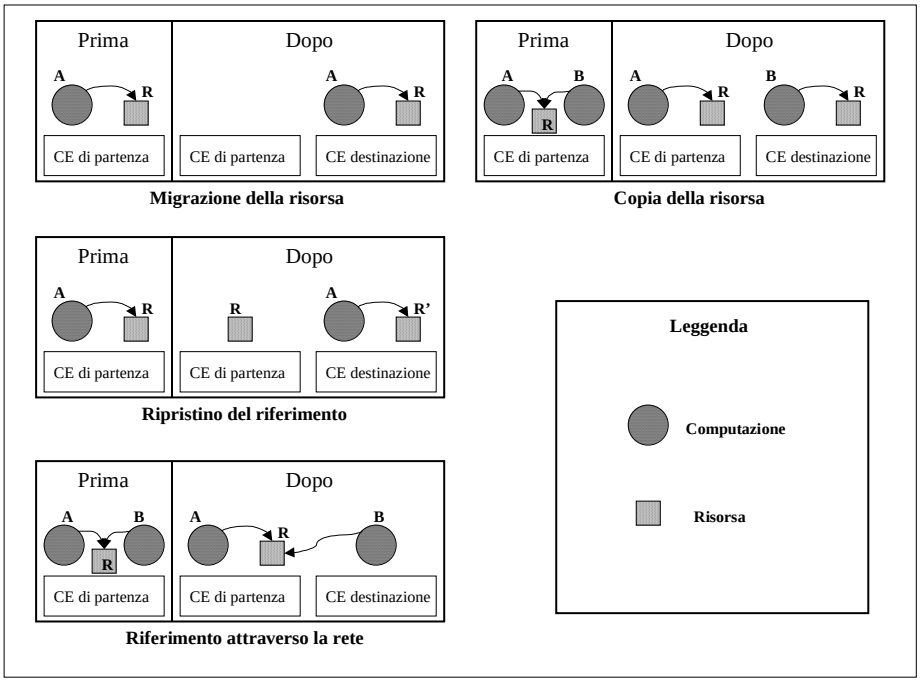

Figura 5: I meccanismi per la mobilità dello spazio dei dati.

Quando lo spazio dei dati di una computazione si sposta, i diversi riferimenti andranno manipolati per consentire il proseguimento della esecuzione in maniera corretta; il trattamento dei riferimenti cambierà a seconda del tipo di riferimento e di risorsa coinvolti (vedi [FPV96]):

- con riferimenti *per identificatore* a risorse trasferibili l'approccio più efficace è senza dubbio quello della migrazione della risorsa insieme alla computazione; questo però potrebbe comportare dei problemi se altre computazioni riferiscono quella risorsa, inoltre occorre prendere in considerazione il costo di tale migrazione e valutare eventualmente un approccio alternativo; così come avviene nei casi in cui la risorsa non possa essere rimossa, si può ricorrere ad un *riferimento attraverso la rete*

ossia ad un meccanismo che permetta di riferire (e quindi anche di modificare) la risorsa a distanza (in remoto);

- con riferimenti *per valore* nel caso di risorse trasferibili il meccanismo più adatto è quello della *copia*; si potrebbe utilizzare anche la migrazione, ma questo porta ai problemi già visti; nel caso di risorse non trasferibili (ed eventualmente anche negli altri casi) si ricorrerà di nuovo al riferimento attraverso la rete;
- con riferimenti *per tipo* il trattamento più appropriato è senza dubbio il *ripristino* del riferimento ad una risorsa locale dello stesso tipo.

	Risorse trasferibili	**Risorse non trasferibili**
Riferimenti per identificatore	Migrazione Riferimento attraverso la rete	Riferimento attraverso la rete
Riferimenti per valore	Copia Migrazione Riferimento attraverso la rete	Riferimento attraverso la rete
Riferimenti per tipo	Ripristino Riferimento attraverso la rete Copia, Migrazione	Riferimento attraverso la rete

Figura 6: A seconda del tipo di risorsa e di riferimento si stabilisce una graduatoria delle possibili strategie di trattamento.

Da questa classificazione rimane escluso il meccanismo più semplice, ma anche il più usato nei sistemi attuali e cioè la *rimozione* del riferimento; utilizzando questo meccanismo quando la computazione tenta di riferire la risorsa viene scatenata una eccezione (o qualcosa di equivalente) cui può

conseguire, ad esempio, il tentativo di ripristinare il riferimento in una forma diversa o attraverso un meccanismo differente.

2.5.2 Comunicazione

Quella ad agenti si delinea come un'architettura molto più dinamica rispetto a quella basata sul paradigma cliente-servitore, giovandosi di un ulteriore grado di libertà: la mobilità. Per ottenere da questa nuova architettura il massimo delle prestazioni in termini di potere espressivo occorre dotarla di strumenti di comunicazione inter-agente molto avanzati, è evidente infatti che, al di là della mobilità, uno dei maggiori punti di forza degli agenti risiederà nella loro capacità di coordinamento e di collaborazione.
Sarà così opportuno prevedere una gamma di meccanismi di comunicazione comprendente message passing asincrono, canali di comunicazione con connessione, RPC o equivalenti, ma anche strumenti più evoluti per il coordinamento come la comunicazione di gruppo, la gestione di eventi e nuovi meccanismi più adatti al mondo degli agenti. La maggior parte dei sistemi ad agenti proposti finora mettono a disposizione degli agenti solo il message-passing o poco più. Dopo una modellizzazione dei possibili tipi di comunicazione che possono interessare gli agenti, vedremo allora alcune proposte più articolate (vedi [BHR+97]).

2.5.2.1 Comunicazione tra agenti

Parlando di comunicazione fra agenti le entità di cui più spesso si tratterà sono di due tipi: gli agenti stessi ed i *place*; i place rappresentano ambiti circoscritti in cui gli agenti eseguono e si incontrano (vedi [BHR+97],

[PEI97]); i place sono i punti di partenza e di arrivo degli agenti che si spostano per eseguire il loro compito; nei place gli agenti hanno accesso a tutta una serie di servizi e di risorse, possono utilizzare diversi strumenti di comunicazione e di sincronizzazione; i place possono rappresentare ambiti omogenei dal punto di vista logico (se, ad esempio, forniscono solo servizi di un certo tipo) o di sicurezza (un certo place potrebbe essere l'unico punto per entrare in una certa rete). Faremo poi riferimento ad un architettura in cui gli agenti vengono divisi in *mobili* e *di servizio.* Questi ultimi sono agenti stazionari e forniscono i servizi disponibili in quel place, siano essi servizi di accesso alle risorse locali o servizi di più alto livello (alcuni degli agenti di servizio propri di place diversi potrebbero, ad esempio coordinarsi per creare un servizio di directory dei servizi forniti dai vari place). L'ipotesi che l'accesso alle risorse locali sia fornito da agenti di servizio trova riscontro in molti dei sistemi realizzati, ma allo stesso tempo è abbastanza generale per i nostri scopi, visto che anche prevedendo un'interfaccia di accesso alle risorse locali di un tipo più tradizionale (quale potrebbe essere un API studiata per risolvere i problemi di sicurezza) le osservazioni che seguono rimangono valide.

In seguito a questa modellizzazione gli atti di comunicazione possono essere suddivisi in cinque categorie (vedi Figura 7):

- *l'interazione tra un agente mobile ed uno di servizio* (1) è del tutto analoga a quella che si ha tra cliente e servitore, è quindi appropriato utilizzare un meccanismo quale la chiamata a procedura remota, che può anche essere reso molto efficiente nel caso in cui cliente e servitore eseguano sulla stessa macchina;
- *nell'interazione fra due agenti mobili* (2) il ruolo dei due agenti non è stabilito in partenza e la comunicazione si svolge tra pari, è quindi necessario un meccanismo con un grado di flessibilità quale solo il message passing consente di ottenere. Infatti anche protocolli evoluti

quali KIF e KQML (vedi [DKS+93]) si basano su di un meccanismo di questo tipo;

- per *interazione anonima* (3) si intende l'interazione che si ha quando il mittente non conosce l'identità del destinatario; questo avviene molto spesso in un gruppo di lavoro, ad esempio, quando uno dei partecipanti al gruppo dinamico consegue un risultato parziale che potrebbe servire ad un altro membro del gruppo, ma non sa quale; per questo genere di situazione sono stati proposti nel passato diversi strumenti quali lo spazio delle tuple [CG89], l'utilizzo di sofisticati gestori di eventi o di protocolli di comunicazione di gruppo (vedi ad esempio [BR94]); in ogni caso questi andranno adeguatamente modificati per integrare ad esempio il concetto di sicurezza;
- *l'interazione tra agenti ed utente* (4) avverrà attraverso interfacce visuali;
- la *condivisione di oggetti* (5) costituisce uno dei meccanismi di comunicazione più adatti in un sistema ad agenti, permettendo, ad esempio, la realizzazione di uno schema di comunicazione anonima.

Al di là del valore di questa schematizzazione è evidente la necessità di dare soluzione a due nuovi ordini di problemi: in primo luogo, perché possano avere una adeguata diffusione, i sistemi ad agenti mobili devono garantire un alto grado di sicurezza, e questo, come vedremo, comporta un uso approfondito di strumenti quali la crittografia ed i protocolli associati; secondariamente, dato che il target di questi sistemi è l'intera rete, essi dovranno tener conto di concetti necessariamente trascurati nell'ambito delle LAN quali quello di costo del canale di comunicazione in termini di latenza e di affidabilità e quello di utilizzo delle risorse.

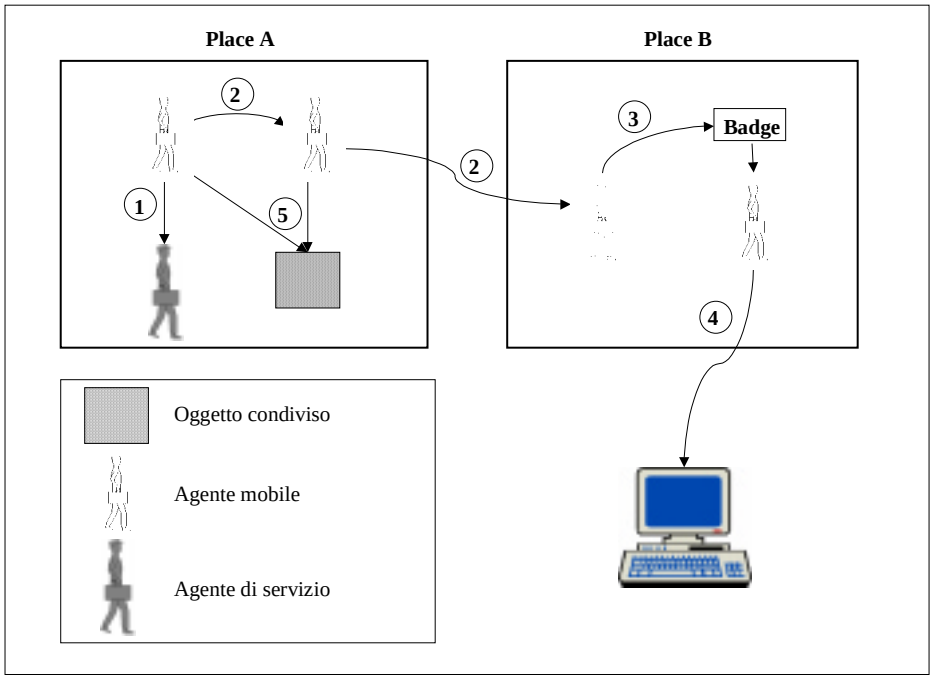

Figura 7: In un ambiente ad agenti le modalità di comunicazione sono molteplici.

Entrambe queste problematiche hanno riflessi sulle caratteristiche degli strumenti di comunicazione che ne modificano la realizzazione e l'utilizzo rispetto a quanto è avvenuto finora.

2.5.2.2 Alcune proposte di estensione

2.5.2.2.1 I badge

Un primo interessante strumento che potrebbe incrementare la versatilità del sistema di comunicazione è una estensione al sistema dei nomi che chiameremo *badge* (vedi [BHR+97]). Essa rappresenta una sorta di segno distintivo che gli agenti possono "indossare" e con cui si possono far riconoscere, per affermare la propria appartenenza ad un certo gruppo di lavoro, oppure per annunciare l'interesse per un certo tipo di informazioni o, ancora, per pubblicizzare una propria caratteristica. In pratica forniscono la possibilità di individuare un agente, di cui non si conosce l'identità in precedenza, in base a una qualche sua caratteristica; lo schema da utilizzare per questa identificazione si compone di una coppia: identificatore del place, predicato di badge, dove con predicato si intende un'espressione logica del tipo "Applicazione PIPPO"AND ["Coordinatore" OR "Risultato"]. Il fatto che entrambi gli elementi di questa coppia possono, in linea di principio, essere opzionali rende questo schema molto flessibile, infatti può essere utilizzato per specificare un incontro con un agente di un determinato tipo in un determinato place, ma anche, ad esempio, con il primo agente disponibile in un certo place (predisponendo un predicato di badge sempre vero).
I badge potrebbero essere utilizzati per la realizzazione di un modello di access control basato sui *ruoli* (vedi [SCF+96]), infatti l'utilizzo naturale di un badge è proprio quello che attribuisce a chi lo indossa una determinata funzione ossia un determinato ruolo.

2.5.2.2.2 Le sessioni

Un secondo contributo alla potenza del sistema di comunicazione ci viene dall'estensione di uno strumento che in alcuni sistemi è detto *meeting* (vedi

[PEI97], [GCK+96], [WHI94]), ma che noi chiameremo *sessione*. La sessione, intesa come relazione di comunicazione tra due agenti, è un concetto necessario per almeno due ragioni: l'instaurarsi di una sessione è una forma di sincronizzazione tra agenti che vogliono cooperare; inoltre lo strumento sessione è indispensabile quando sia richiesta una interazione con stato. La novità di questo strumento risiede nel fatto che la richiesta di creazione di una sessione può essere effettuata con due differenti modalità: *attiva* o *passiva*. Con una richiesta passiva l'agente esprimerà il desiderio di partecipare ad una sessione, ma l'operazione sarà non bloccante per tale agente; con una richiesta attiva, viceversa, l'agente sarà bloccato fino allo stabilirsi della sessione o fino allo scadere di un time out.
Dalla combinazione di queste due modalità derivano semantiche di sincronizzazione diverse:

- se uno dei due agenti coinvolti nella sessione invoca una richiesta attiva mentre l'altro fa uso di quella passiva, si realizzerà il classico schema cliente-servitore in cui il primo agente gioca il ruolo del cliente richiedendo esplicitamente l'instaurarsi della sessione per poter continuare, il servitore è molto più libero nel caso della sessione;
- se entrambi utilizzano la modalità attiva si realizza invece uno schema di comunicazione tra pari con un classico rendezvous che bloccherà i due agenti fino all'instaurarsi della sessione.

Le sessioni potranno essere sia intra-place che inter-place, sembrerebbe infatti limitativo limitare le sessioni solo ad agenti residenti sullo stesso place, anche perché il costo di una sessione potrebbe essere anche molto inferiore a quello richiesto per spostare un agente in un nuovo place perchè possa stabilire una sessione. Al fine di stabilire una sessione tra due agenti occorre, ovviamente, che entrambi acconsentano, inoltre è verosimile prevedere che entrambe i partecipanti evitino la migrazione fino a che la

sessione non venga chiusa sarebbe altrimenti necessario prevedere un meccanismo di forwarding molto costoso.
L'unione dei concetti di sessione e di badge fornisce scenari completamente nuovi in cui, ad esempio, ad un agente è data la possibilità di richiedere in maniera bloccante un canale di comunicazione con un pari di cui non si conosce l'identità, ma cui si richiede di trovarsi in un determinato place (in cui ad esempio avrà l'accesso a determinate risorse) e di indossare un determinato badge.

2.5.2.2.3 Gli eventi

Per realizzare la cosiddetta comunicazione anonima, introduciamo uno strumento molto studiato in ambiente centralizzato: gli *eventi*. Osserviamo innanzitutto che quello ad eventi è un modello molto comodo nel distribuito perché astrae dalla identità del ricevente; questo permette la realizzazione di protocolli di comunicazione complessi senza che i partners della comunicazione debbano conoscersi in anticipo. Intenderemo gli eventi come entità di un determinato *tipo* cui sono associate alcune *informazioni*. Essi sono generati da *produttori* (che possono essere agenti, utenti o il sistema stesso) e utilizzati da *consumatori*. Sia i consumatori che i produttori dovranno in qualche modo registrarsi, i primi per rendere noti i tipi di eventi che produrranno, i secondi per evidenziare i tipi di eventi cui sono interessati.
Fin qui i concetti introdotti sono analoghi a quelli che si riscontrano in ambienti event-driven centralizzati; la novità sta nell'introduzione degli *oggetti di sincronizzazione* il cui comportamento, simile a quello di complessi filtri di eventi, è regolato da *regole*, *stato interno* e *timeout* (vedi Figura 8). Le regole sono formate da una parte condizioni e da una parte azioni. La parte condizioni è costituita da una espressione arbitrariamente complessa i cui termini esprimono condizioni sugli eventi in ingresso e sullo

stato interno; se tale espressione risulta vera viene intrapresa l'operazione specificata nella parte azioni della regola.

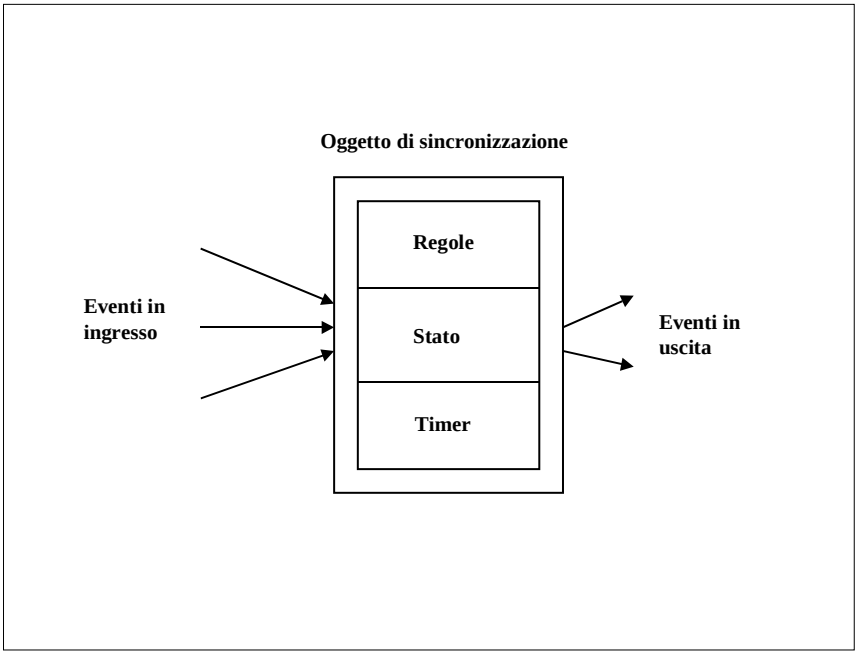

Figura 8: La comunicazione anonima può essere realizzata attraverso degli oggetti evento che notificano al destinatario il messaggio voluto.

La parte azioni può comprendere l'emissione di un evento in uscita o un cambiamento di stato interno. I timeout rappresentano particolari regole, senza eventi in ingresso, che eseguono la parte azioni dopo un certo periodo di tempo. Lo stato interno è costituito da un set di variabili.
Vediamo, ad esempio, come viene definito un oggetto di sincronizzazione che comprende una semplice regola ed un timer:

```
Rule 12: if((event98 and event51) or (event3 and
            varaible1==true)) then {
           send(event90);
           variable1=false;
         }
variable1: boolean
timer1= after 10 seconds { send(event51)}
```

2.5.2.2.4 La blackboard

La comunicazione anonima può essere efficacemente implementata attraverso una *blackboard*. Si tratta, in pratica, di uno spazio, gestito da un apposito gestore, in cui è possibile depositare oggetti costituiti da una stringa ed un messaggio; per ottenere un dato messaggio sarà sufficiente conoscere la stringa. L'accesso alla blackboard può essere regolato in diversi modi. Può essere possibile leggere tutti i messaggi specificando al posto della stringa una wildcard. Si può utilizzare la stessa blackboard per distribuire la stringa che da accesso ad un certo messaggio. Un messaggio può essere rimosso dopo che è stato letto la prima volta o meno.

La comunicazione di tipo anonimo è resa possibile dal fatto che non è necessario conoscere il mittente (o meglio il produttore) del messaggio per poterlo leggere, e che, viceversa, chi mette un messaggio nella blackboard non può sapere chi lo leggerà.

3 Sicurezza in ambienti ad agenti mobili

La comparsa dei sistemi ad agenti mobili apre un nuovo capitolo nello studio del problema della sicurezza dei sistemi informatici. Ciò è dovuto essenzialmente al fatto che, in quanto entità attive, gli agenti costituiscono una nuova classe di soggetti abilitati all'accesso alla rete e quindi alle macchine che la costituiscono. Si tratta di una classe nuova perché i suoi membri hanno caratteristiche differenti dagli altri, prima fra tutte il fatto che autonomamente "entrano", negli host che ne ospitano l'esecuzione, muovendosi sulla rete.

La questione sollevata dai cosiddetti Executable Content (vedi le applet di Java) riguardo al codice che viene eseguito all'insaputa dell'utente del browser diventa, infatti, di primaria importanza in un sistema in cui gli agenti si trovano ad interagire solo con l'infrastruttura che li supporta e che deve occuparsi dei problemi della sicurezza.

Come abbiamo visto, garantire la sicurezza significa garantire che le varie entità che si trovano ad interagire (siano essi utenti, agenti o processi di sistema) lo facciano in maniera lecita. Per far questo è necessario controllare le interazioni che si hanno alle interfacce fra i vari componenti del sistema (vedi [VST97]) al fine di creare delle barriere che proteggano le varie unità reciprocamente.

In un sistema ad agenti l'insieme delle classi di oggetti che vanno protette si arricchisce di un nuovo membro: gli agenti. Nei sistemi client/server le entità attive eseguono sotto il controllo del sistema operativo della macchina su cui sono state create; l'utente che lancia una applicazione ha un account che lo identifica e ne stabilisce i diritti; i meccanismi di confinamento del sistema operativo assicurano la protezione delle risorse critiche di sistema.

Il *principal*, per conto del quale l'agente esegue può non essere un utente noto all'amministratore di un sistema ospite, questo dovrà allora stabilire

come trattare l’agente solo in base alla sua provenienza; questa, però, potrebbe essere una informazione o non sufficiente in alcuni casi o non necessaria se l’agente è in grado di dimostrare (in qualche modo) che può essergli concessa una certa fiducia. Il paradigma ad agenti modifica, insomma, i termini del problema sicurezza; di conseguenza, è necessario modificare l’approccio alla sua analisi.

3.1 Il problema della sicurezza nei sistemi ad agenti mobili

Per stabilire quali sono i nuovi termini del problema sicurezza è necessario analizzare l’architettura del sistema ed individuarne, così come era stato fatto per l’architettura client/server, i punti deboli.
Nella nuova architettura il Computational Environment (CE) si sostituisce al sistema operativo come autorità locale incaricata del rapporto diretto e quindi anche della sicurezza (vedi 2.5.1.1). Il CE deve allora essere in grado di fornire una gamma completa di meccanismi per poter regolare tutte le interazioni che avvengono alle interfacce (vedi Figura 9) tra i vari componenti dell'architettura e precisamente (vedi [VST97]):

- gli agenti e le altre informazioni (1) devono muoversi su canali sicuri per evitare, ad esempio, la possibilità di attacchi passivi ed attivi alla integrità sia del codice che dei dati di una computazione che migra;
- gli agenti in ingresso in un sistema (2) devono essere autenticati per stabilirne la provenienza e di conseguenza 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 (*autorizzazione*);
- l'accesso da parte degli agenti alle risorse locali (3) deve essere controllato per proteggere l'host (*access control*);

- CE ed agenti (4) devono essere protetti gli uni dagli altri per eliminare la possibilità di comportamenti illeciti;
- gli agenti che eseguono sullo stesso CE devono essere isolati fra loro (5);

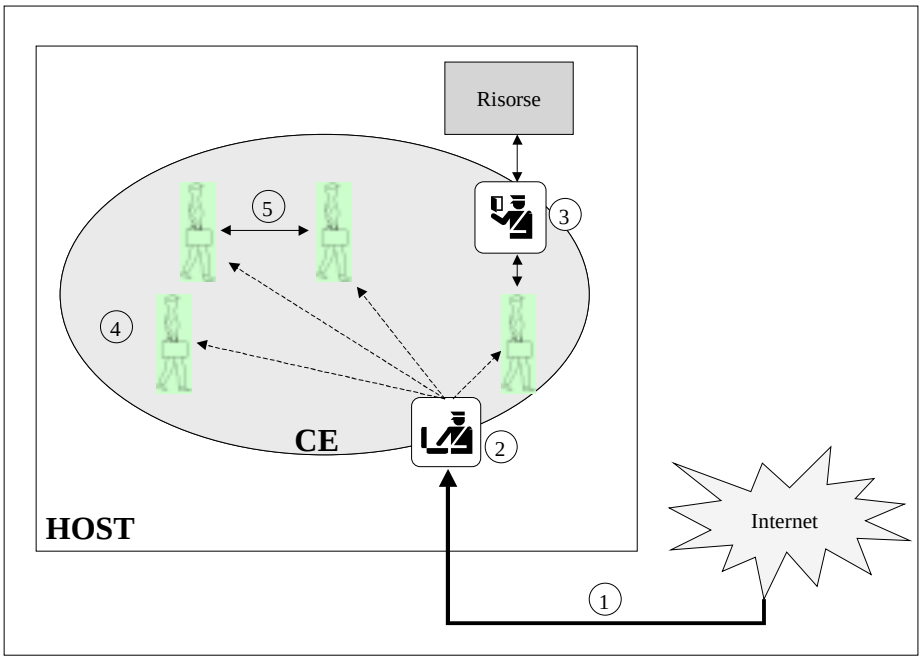

Figura 9: Garantire la sicurezza nei sistemi ad agenti mobili significa controllare l'interazione alle varie interfacce.

3.2 Il disegno dei meccanismi

3.2.1 La protezione dei canali di comunicazione

Per proteggere i canali di comunicazione tra i vari place saranno utilizzate le ben note tecniche crittografiche; in tal modo sia gli agenti che le informazioni che essi si scambiano attraverso al rete avranno al garanzia di giungere a destinazione senza aver subito modifiche.

3.2.2 Il controllo degli accessi

Per quanto riguarda il controllo degli accessi gli strumenti messi a disposizione dagli attuali sistemi operativi permettono la creazione di domini di protezione piatti mentre non consentono la realizzazione di sotto domini ricorsivi; tale astrazione è necessaria per esprimere, ad esempio, il fatto che, essendo sotto il controllo del CE, gli agenti eseguiranno con un sotto insieme dei diritti del CE stesso, in altre parole il dominio di protezione del CE incapsula quello dell'agente. Inoltre, gli schemi classici (quali ad esempio quelli che prevedono l'utilizzo delle access list) non forniscono un adeguato grado di flessibilità (nei sistemi ad agenti il concetto di principal si ampia e questo porta alla necessità di un sistema di naming più articolato).
Ma il difetto principale dei meccanismi di access control classici è un altro: essi prevedono una classificazione delle risorse statica, mentre ora occorre poter ampliare la gamma dei target in maniera dinamica durante l'esecuzione del sistema (nuove applicazioni creeranno nuove risorse, il sistema deve permettere una facile gestione integrata anche di queste) ed estendere il concetto di risorsa così da ricomprendervi sia il classico singolo file che risorse proprie del livello applicativo quali possono essere i record di un Data Base.

3.2.3 La protezione del CE dagli attacchi di agenti non fidati

Quello dell'interfaccia tra agenti e CE è un altro importantissimo problema simile a quello che i sistemi operativi devono risolvere per proteggere il sistema dai processi che eseguono su di esso e tra loro. I sistemi operativi lo risolvono prevedendo spazi di indirizzamento separati per ogni processo e per il kernel, ma questo approccio non può essere adottato nei sistemi ad agenti

in cui gli agenti ed il CE condividono lo stesso spazio di indirizzamento per questioni di efficienza nella comunicazione (in generale gli agenti saranno computazioni snelle e veloci). La soluzione è imporre agli agenti una specifica interfaccia con cui interagire con il CE, una interfaccia che elimina o almeno limita gli accessi insicuri al sistema. In questo modo il problema diventa quello di garantire il rispetto dell'interfaccia da parte del codice degli agenti, sia esso codice sorgente, in formato intermedio (bytecode) o codice oggetto nativo. Questo può essere fatto in diversi modi:

- il codice dell'agente può essere certificato come sicuro da una terza parte fidata;
- il codice può portare con se (sotto forma, ad esempio, di firma) una prova formale della propria correttezza;
- il linguaggio di alto livello in cui il codice è scritto è sicuro in se e si può verificare che il codice nel formato intermedio utilizzato per la migrazione è conforme alla semantica di alto livello (vedi JAVA); in questo modo il controllo dell'interazione viene regolato da opportune politiche durante l'esecuzione del codice invece che a priori; viceversa, senza eseguire alcun controllo statico, il codice può essere confinato in un ambiente strettamente controllato a tempo di esecuzione in cui semplicemente non è autorizzato ad eseguire nessuna istruzione illegale (come in agent TCL [GCK+96]);
- il codice viene riscritto in una forma sicura (vedi la tecnologia SFI [WLA+93]);

3.2.4 La protezione degli agenti in place non fidati

Purtroppo per quanto riguarda le garanzie che possono essere date agli agenti, riguardo a comportamenti scorretti del CE, sono stati proposti alcuni approcci (vedi [VIG97]), ma non sono ancora emersi strumenti convincenti. Ciò è

dovuto al fatto che una volta entrate nell'ambiente di un CE le computazioni sono sotto il suo pieno controllo ed è impossibile, ad esempio, evitare che il CE faccia una copia del codice dell'agente per poi modificarlo.
Una degli approcci emersi in questo campo prevede, per un agente che deve spostarsi in un place non fidato, un protocollo che si articola nei seguenti passi:

- l'agente cifra il suo stato prima di muoversi;
- si sposta nel place sospetto ed esegue memorizzando i cambiamenti da apportare al suo stato in un area temporanea che sarà considerata a rischio fino a che l'agente non raggiunga un place fidato;
- una volta terminato il suo compito nel place non fidato si sposta in un place fidato dove può tranquillamente aggiornare il suo stato;

questa strategia permette all'agente di proteggere parte di se, ma molta altra strada deve essere fatta per difendere un agente da tutti i possibili attacchi di un CE non fidato.

3.2.5 Il confinamento

I problemi che si instaurano nei rapporti tra CE ed agenti e quelli tra agenti ed agenti possono essere affrontati efficacemente realizzando un *confinamento* delle entità (attive o passive) che vanno protette le une dalle altre. In particolare occorrerà definire le regole attraverso cui decidere quali entità attive possano accedere a quali entità passive (risorse) o attive (agenti) ed in che modo.
Questo processo si articola in due distinte fasi:

- *autenticazione*: l'agente che chiede l'accesso ad una risorsa andrà innanzitutto identificato per stabilire con certezza per conto di quale principal opera;
- *autorizzazione*: andrà poi consultata la politica di sicurezza prevista al fine di stabilire se a tale agente è permessa l'operazione richiesta;

L'obbiettivo della fase di identificazione non è, in generale, solo quello di stabilire il principal per conto di cui l'agente esegue, tale informazione potrebbe infatti essere non sufficiente o non necessaria, si pensi al caso di un agente che torni ad eseguire su di un host che gli aveva già concesso un certo grado di trustness, l'autenticazione potrebbe non essere più necessaria (sempre che l'agente non sia cambiato).

Le politiche di sicurezza rappresentano il terreno su cui combattere la battaglia che vede opposte le esigenze della sicurezza e quelle di apertura degli ambienti ad agenti. Andranno sviluppate modalità attraverso cui gli utenti e gli amministratori di questi sistemi possano specificare le diverse politiche in una maniera che permetta loro di comprendere appieno le conseguenze delle diverse scelte.

Occorre, a questo punto, osservare che i meccanismi di imposizione dei limiti determinati da una data politica possono essere realizzati a diversi livelli dell'architettura, e possono agire in momenti diversi del ciclo di vita di un agente. La scelta del livello che li fornisce ha grande ripercussione sull'efficienza dei meccanismi (si pensi all'overhead introdotto), mentre non cambia molto il genere di funzionalità che è possibile fornire con le diverse soluzioni.

3.3 Alcuni modelli implementati

Negli ultimi anni sono stati realizzati molti sistemi (commerciali e non) per la creazione di applicazioni che si basano sulla tecnologia degli agenti mobili. Nell'intento di esplorare le possibilità del nuovo paradigma gli sforzi maggiori sono stati fatti nella direzione dello sviluppo dei meccanismi più adatti per la migrazione e la comunicazione, mentre il problema sicurezza è, inizialmente, passato in secondo piano; ora, però, comincia a sentirsi anche l'esigenza di un modello di sicurezza abbastanza robusto da convincere il mercato ad utilizzare la tecnologia dei sistemi ad agenti mobili. Per capire quali sono le proposte analizziamo alcuni dei modelli di sicurezza implementati.

3.3.1 Il modello Padded Cell

Questo modello, utilizzato in safe-tcl [OLW96] isola gli agenti non fidati facendoli eseguire da un interprete detto *safe-interpreter* (vedi Figura 10) in cui vengono nascosti i comandi critici per la sicurezza (*hidden commands*). Tale interprete è totalmente controllato da un *master-interpreter* il quale può estendere la *safe-base* (ossia l'insieme dei comandi che non sono hidden) del safe-interpreter definendo all'interno di quest'ultimo degli *aliases* dei comandi nascosti. Quando un alias viene invocato all'interno del safe-interpreter esso attiva una procedura nel master-interpreter la quale può così eseguire dei controlli prima di eventualmente invocare il relativo comando nascosto nel safe-interpreter.

Scrivere una politica di sicurezza significa allora scrivere tutte le procedure che realizzano gli aliases che la politica intende creare.

E' importante notare che le politiche possono essere create indipendentemente dagli agenti; una politica che "viaggi" con un agente non

fidato può ugualmente essere utilizzata se prodotta da una terza parte fidata, questo offre molta flessibilità nella gestione delle politiche.

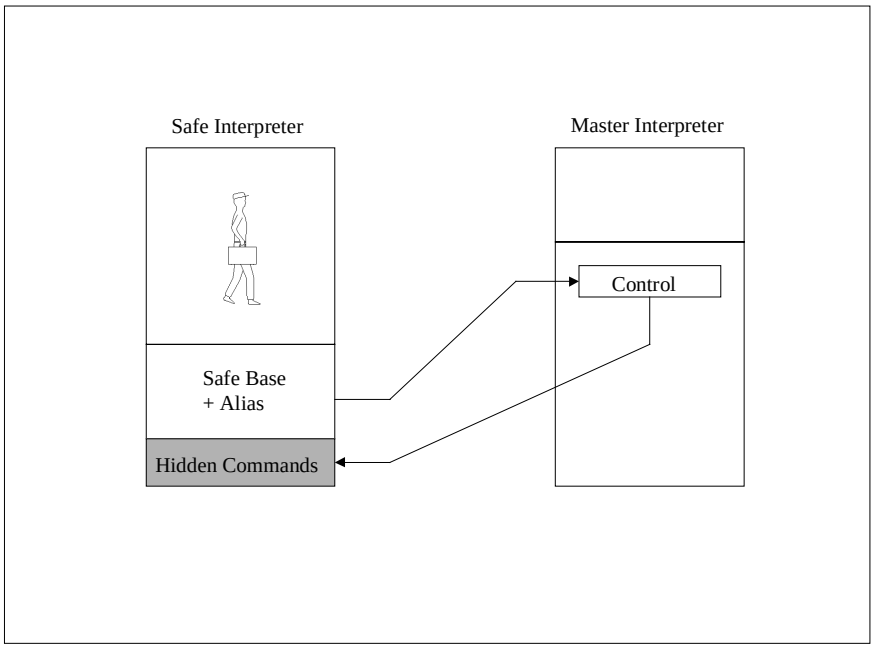

Figura 10: Nel modello safe-tcl i comandi pericolosi vengono nascosti e poi riattivati in forma controllata attraverso il meccanismo degli alias.

Un altro aspetto interessante è il fatto che, a differenza di ciò che avviene in altri modelli, ogni agente esegue su una diversa macchina virtuale cui può essere associata una diversa politica di sicurezza. I confini dei vari domini di sicurezza sono ben chiari; nei sistemi ad oggetti, invece, ogni invocazione ad un metodo può provocare un cambio di dominio.

La possibilità della contemporanea presenza di agenti, sottoposti a diverse politiche, che possono cooperare, apre, però, la strada ad un nuovo tipo di problema quello della composizione delle politiche. Anche se due politiche sono separatamente sicure, la loro composizione potrebbe infatti non esserlo: si pensi al caso in cui a un agente è applicata una politica che gli permette di accedere ad informazioni al di fuori della rete locale e a un secondo agente si applichi invece una politica che gli permette di accedere ad informazioni

all’interno della LAN, se i due agenti possono comunicare è possibile scavalcare il firewall locale.
In Agent-tcl [GCK+96], un agente dedicato, detto *resource-manager*, è incaricato dell’autenticazione degli agenti entranti e della scelta della politica le cui regole sono poi imposte mediante l’utilizzo dei meccanismi forniti dal safe-tcl.

3.3.2 Il modello di Ara

Nel sistema Ara [PEI97] ad ogni agente viene attribuito un set di *allowance* all’atto della creazione; si tratta di capabilities, esplicitamente concesse dal sistema, che l’agente porterà con sé ogni volta che migra. Le allowance permettono di condividere risorse fra gli agenti, di limitare i permessi degli agenti da parte dei nodi ospiti, ma anche di porre, da parte del principal dell’agente, un limite alle sue capacità. In realtà è previsto che ogni agente venga creato all’interno di un gruppo, in cui condividerà alcune allowance con altri agenti, ottenendone una parte quando voglia abbandonare il gruppo per migrare. È previsto poi un meccanismo per lo scambio di allowance tra agenti che ovviamente impedisce ad agenti maliziosi di acquisire poteri a loro piacimento.
Gli agenti si muovono tra places, astrazioni della locazione fisica dei nodi, i quali implementano una propria politica di ingresso e di accesso stabilendo cioè quali agenti ammettere (quelli non ammessi sono rispediti al place di partenza) ed il set di *local allowance* che sarà imposto loro (vedi Figura 11). A sua volta un agente può specificare un *local allowance set* (più o meno ristretto rispetto al proprio *global allowance* set) come minimo richiesto al place di destinazione perché la migrazione possa aver luogo.

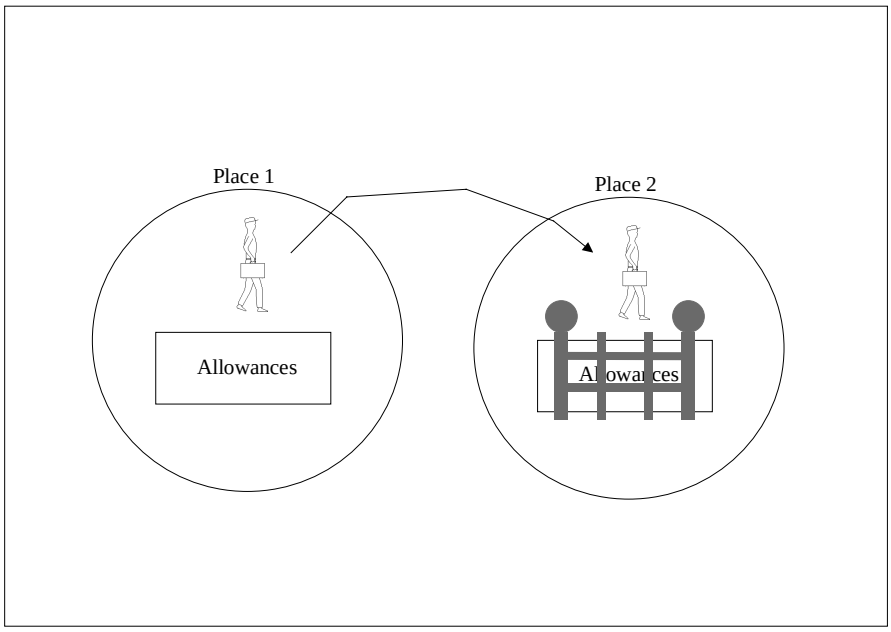

Figura 11: In Ara i permessi di un agente vengono limitati attraverso il concetto di local allowance.

Il meccanismo di imposizione dei limiti rappresentati dalle allowance è interamente a carico dell'ambiente di esecuzione degli agenti, nel senso che i controlli sono tutti eseguiti negli stub delle chiamate critiche al sistema (l'accesso alle risorse locali, ad esempio, è fornito da una API specifica detta Ara Host Interfacce). Questa non sembra essere una soluzione molto appropriata poiché la creazione di un nuovo tipo di allowance comporterà la necessità di modificare tali API.

Inoltre nella prima release del sistema non è stata realizzata un interfaccia di programmazione dei place quindi non è molto chiaro come si possano scrivere le politiche al loro interno.

3.3.3 Il modello di sicurezza di java

Java si è imposto ormai come standard di fatto per la realizzazione di applicazioni per la rete grazie al paradigma ad oggetti che permette una veloce prototipazione ed un rapido sviluppo ed alla vasta gamma di strumenti che permettono una comoda gestione delle risorse di rete. Anche Java deve però fare i conti con i problemi della sicurezza, analizziamone dunque il modello di sicurezza.

3.3.3.1 Il modello originale

Il modello di sicurezza originale di java prevede la suddivisione del codice in *trusted* e *untrusted*: è considerato trusted solo il codice cui è possibile accedere attraverso la variabile di sistema CLASSPATH, mentre tutte le altre classi, anche se residenti sul file system locale, sono considerate untrusted; le Applet, in quanto provenienti dalla rete, sono quindi classificate come untrusted.

Il codice untrusted viene confinato in una *sandbox* ossia un area limitata in cui gli viene praticamente impedito l'accesso a tutte le risorse critiche del sistema. Il modello sandbox si serve di diversi meccanismi per imporre la restrittiva politica di sicurezza. Innanzitutto il linguaggio è stato studiato per essere *type-safe* ossia per eliminare tutti quei costrutti che permettono accessi pericolosi alla memoria come l'aritmetica dei puntatori del C. Inoltre, alcune caratteristiche del linguaggio, come la gestione automatica della memoria, il garbage collection ed il controllo automatico degli indici nelle stringhe e negli array, alleggeriscono il compito del programmatore aiutandolo a scrivere codice sicuro.

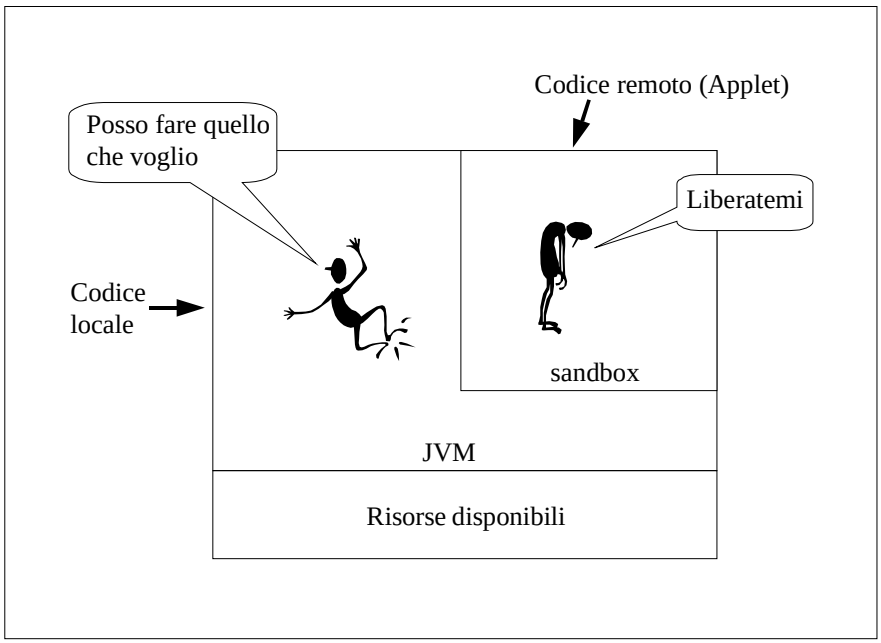

Figura 12: Il modello di sicurezza originale prevede che il codice locale abbia accesso completo mentre alle Applet concede un raggio d'azione ridottissimo.

L'unico modo in cui è possibile importare codice al di fuori del classpath è quello di utilizzare un componente del package java.lang: il *ClassLoader*; questo componente definisce una classe astratta che supporta l'importazione di classi, i suoi metodi principali sono:

- protected abstract Class loadClass (String name, boolean resolve) throws ClassNotFoundException

  Implementando questo metodo astratto si insegna al ClassLoader come caricare dati da una fonte diversa dal CLASSPATH per poi poterli convertire in una istanza di java.langClass;
- protected final Class defineClass(byte data[], int offset, int length)

  Trasforma dati grezzi (un array di byte) in una istanza di java.lang.Class;
- protected final void resolveClass(Class c)

Risolve (chiamando loadClass()) i riferimenti ad altre classi dalla classe specificata;

per utilizzare un ClassLoader occorre definirne il metodo loadClass (in modo tale, ad esempio, che importi i dati necessari alla definizione di una data classe attraverso una connessione con un server).
Ogni classe importata con un ClassLoader contiene un riferimento ad esso ottenibile con il metodo Class.getClassLoader(); i riferimenti a nuove classi, effettuati all'interno di una classe importata, vengono risolti attraverso lo stesso ClassLoader; in questo modo viene creato uno spazio dei nomi separato per il codice importato il quale non può sovrapporsi al codice di sistema o a quello importato da un diverso ClassLoader.
La possibilità di inserire delle Applet all'interno di pagine HTML ha reso necessario l'inserimento, nei browser utilizzati per navigare il web di estensioni in grado di eseguire tali Applet: di fatto ogni browser avvia una propria Java Virtual Machine configurata in modo da creare un ClassLoader diverso per ogni URL da cui viene importata un Applet, in questo modo Applet provenienti da URL differenti eseguono in sandbox differenti ed in spazi dei nomi separati.
Il ClassLoader inoltre verifica la correttezza del codice che importa invocando il *bytecode verifier*: ciò che un ClassLoader importa è il codice della classe in formato bytecode ossia tradotto nel linguaggio macchina della macchina virtuale Java, questo codice però potrebbe essere stato ottenuto compilando, ad esempio, un programma scritto in C++ o manipolando direttamente il bytecode di una classe normale e quindi potrebbe non rispettare più le specifiche del linguaggio, ciò potrebbe permettere al codice di imbrogliare il sistema di sicurezza; per questo motivo tutto il codice importato da un ClassLoader viene sottoposto all'analisi di un componente (detto Verifier) il quale lo sottopone ad un processo che si articola in quattro fasi: le prime due verificano la correttezza del formato dei dati, le altre la rispondenza alle specifiche del linguaggio da parte del bytecode.

Infine ogni accesso alle risorse critiche del sistema è mediato dalla Java virtual machine e controllato dal *security manager* in modo tale da limitare i poteri del codice untrusted al minimo indispensabile. La classe java.lang.SecurityManager definisce un componente che implementa di fatto una politica di sicurezza. Una applicazione Java può impostare una e una sola volta un SecurityManager invocando il metodo System.setSecurityManager(SecurityManager), una seconda invocazione provoca il sollevarsi di una SecurityException.

Scopo del SecurityManager è quello di centralizzare il controllo sull'accesso alle risorse a run-time; le operazioni critiche per la sicurezza sono state classificate e sottoposte al controllo del SecurityManager; tutte le classi di sistema interpellano il SecurityManager (se ne è stato specificato uno) prima di svolgere queste operazioni; in pratica ad ogni operazione critica XXX corrisponde nel SecurityManager un metodo checkXXX() così definito:

```
Public void checkXXX(arguments) {
 // Decidi se concedere l'accesso
 ….
 If  (accesso_consentito)
       Return;
 throw new SecurityException("XXX");
}
```

ogni metodo che desidera eseguire l'operazione XXX deve contenere il seguente codice:

```
….
SecurityManager security=System.getSecurityManager();
if(security!=null)
       security.checkXXX(arguments);
….
```

Attraverso tutta una serie di metodi interni il SecurityManager ispeziona la stack per stabilire se la classe che richiede l'operazione è trusted o meno e, di conseguenza, ritorna normalmente o solleva un eccezione.

3.3.3.2 Le modifiche inserite nel JDK1.1

Con la versione 1.1 del java development kit è stato fatto un tentativo per estendere la flessibilità della politica di sicurezza introducendo il concetto di applet firmate. Si tratta di applet che viaggiano sulla rete impacchettate e criptate in archivi detti jar file (jar sta per java archive) in cui vengono incluse anche le firme digitali che ne attestano l'integrità e l'autenticità. Con il termine *firma digitale* di un documento si indica un oggetto ottenuto sottoponendo il documento ad una funzione (detta *hash*) che ne fornisce una sorta di riassunto di dimensioni fissate e indipendenti da quelle del documento e successivamente cifrando tale riassunto con una opportuna chiave. Se l'applet è riconosciuta come correttamente firmata (ossia se si verifica la correttezza delle sue firme attraverso un processo che garantisce l'integrità e la paternità dell'applet), essa viene considerata trusted ed eseguita come se fosse stata caricata dal CLASSPATH.

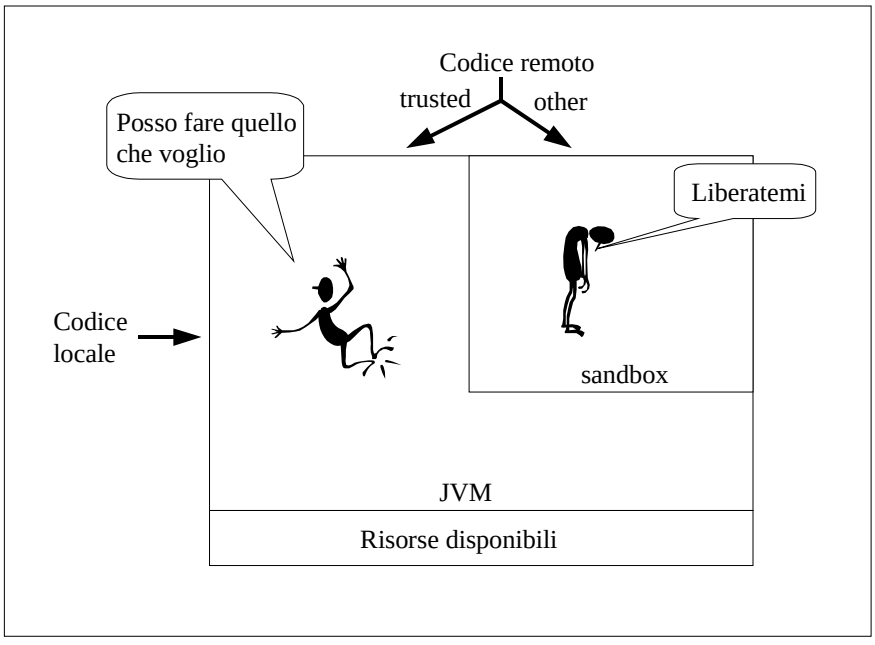

Figura 13: In jdk1.1 anche alle Applet correttamente firmate è permesso di eseguire come codice fidato.

Il nuovo package java.security fornisce inoltre al programmatore una serie di interfacce variamente configurabili per l'utilizzo di strumenti di crittografia tra cui la gestione di chiavi pubbliche e private, la generazione e verifica di firme e la generazione di certificati; l'architettura del sistema di crittografia si basa sul concetto di *provider*: le varie interfacce forniscono le caratteristiche esteriori di tutti gli elementi che rientrano in un sistema crittografico demandando poi ad una serie di provider dinamicamente selezionabili la loro implementazione; il provider standard, denominato SUN, fornisce alcune implementazioni dei diversi algoritmi (DSA per le firme digitali e la generazione delle chiavi relative, gli algoritmi di hash SHA-1 e MD5, il protocollo X.509 per i certificati).
Infine è messa a disposizione del programmatore la possibilità di creare esplicitamente delle *Access Control List* (ACL): una ACL è una struttura dati in cui ogni entry contiene una associazione tra un set di permessi e un principal (utente) al quale questi permessi vanno concessi (entry positiva) o negati (entry negativa); tale struttura è indipendente dai seguenti aspetti:

- lo schema utilizzato per l'autenticazione dei principal: l'ACL viene utilizzata dopo l'autenticazione;
- il fatto che l'applet viaggi in forma cifrata;
- il tipo di risorsa che si vuol proteggere;

definendo allora un SecurityManager che faccia uso di questo nuovo strumento è così possibile creare gradi di trustness intermedi permettendo ad un applet correttamente firmata di uscire dalla sandbox attribuendole invece che tutti i permessi del codice trusted un qualsiasi subset di questi.

3.3.3.3 L’evoluzione del modello sandbox in jdk 1.2

Con la nuova release del JDK, la Sun affronta finalmente in maniera concreta il problema della sicurezza, esplicitandone i termini ed alcuni dei meccanismi per risolverlo a livello di linguaggio; la dimostrazione più evidente di ciò è la comparsa di tutta una serie di nuove classi nel package java.security.

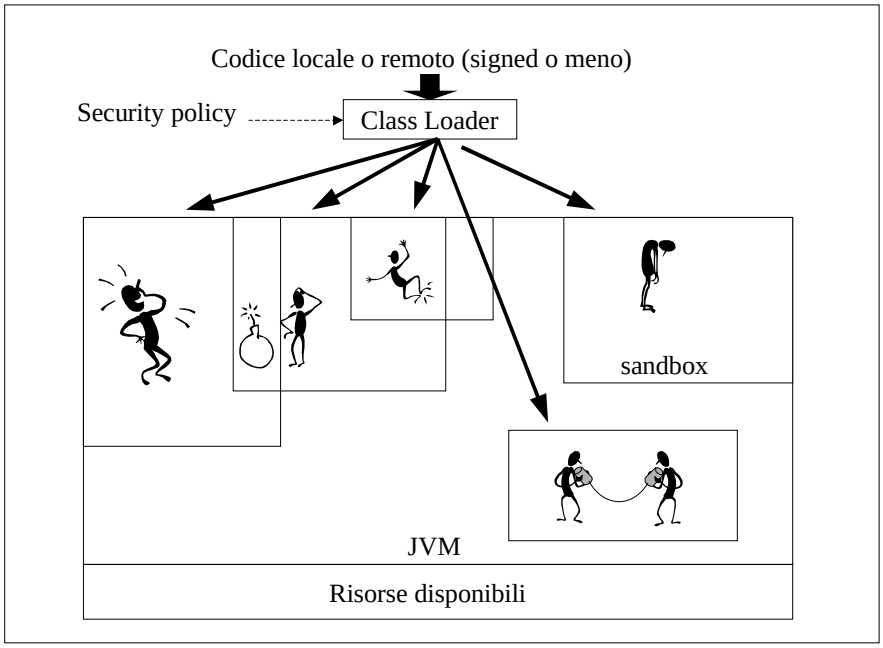

Figura 14: Nel modello di sicurezza del jdk1.2 il class loader ha un ruolo fondamentale, infatti è l'unico in grado di creare nuovi domini di protezione.

La direzione scelta è, ovviamente, quella che porta alla gestione sicura delle Applet. Comunque, molti concetti e strumenti sono abbastanza generali da poter supportare una architettura più complessa quale quella ad agenti mobili. Il nuovo modello incorpora tutta una serie di caratteristiche che hanno lo scopo di aumentarne la flessibilità è la facilità d’uso, i suoi obiettivi principali sono:

- **access control a grana fine**: fornire un meccanismo di access control facile da usare e in grado di proteggere risorse di diverso livello;
- **politiche di sicurezza facilmente configurabili**: permettere la stesura di nuove politiche e la modifica di politiche preesistenti in maniera veloce ed intuitiva;
- **access control estensibile**: rendere agevole l'estensione del meccanismo di access control per la protezione di nuove risorse aggiunte dinamicamente al sistema;
- **sicurezza anche per il codice locale**: permettere l'imposizione di una politica di sicurezza anche al codice che risiede sul CLASSPATH.

3.3.3.4 Gli elementi principali della nuova architettura

Per ottenere questi risultati vengono innanzitutto esplicitati alcuni dei fondamentali elementi che caratterizzano un modello di sicurezza.

Principal: con questo termine si intende una entità del sistema cui possono essere concessi dei permessi ed a cui, eventualmente, potrebbe essere imputato il costo delle varie operazioni di accesso.

Dominio di protezione: è il tramite che stabilisce il legame tra un principal e l'insieme di risorse cui questo può accedere, o meglio tra un principal ed il set di permessi di accesso che gli vanno concessi. Ogni dominio potrebbe proteggere le sue risorse (anche se, per ora, questa opzione non è implementata) isolandole e facendo in modo che le interazioni con l'esterno avvengano solo attraverso codice fidato e con il consenso del dominio stesso. Ogni classe appartiene ad uno e un solo dominio di protezione. Questo rappresenta un vincolo forte poiché tutte le istanze di una stessa classe si troveranno ad eseguire con lo stesso set di permessi. Un singolo thread potrà eseguire completamente all'interno di un dato dominio di protezione, ma potrebbe anche richiedere l'accesso ad un dominio diverso (si pensi ad un

metodo di una certa classe A in cui si invoca un metodo di una classe B che si trova in un dominio differente).

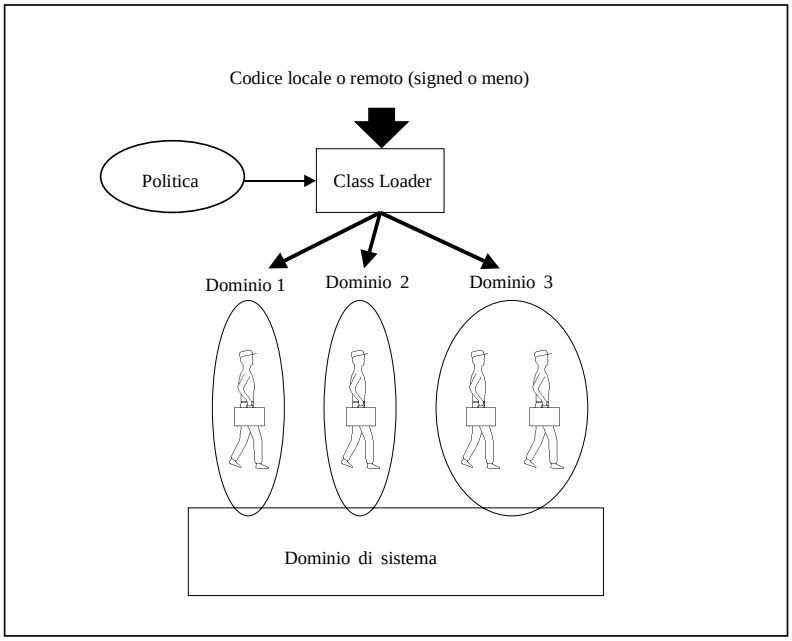

Figura 15: Il modello di architettura prevede la creazione di un differente dominio di protezione per ogni differente principal.

Questo porta alla formulazione di un problema: è giusto che, in seguito all'ingresso in un dominio diverso, al thread siano concessi nuovi permessi ? La risposta, in generale, è no; estendendo il discorso al caso di una catena di chiamate che portino all'attraversamento di diversi domini, la regola da seguire sarà:

- i permessi da attribuire ad un thread di esecuzione si ottengono come intersezione dei set di permessi concessi al thread da tutti i domini di protezione che ha attraversato.

Il principio dei minimi privilegi: la regola sopra enunciata deriva dal principio dei minimi privilegi il quale richiede che ad un thread vanno in ogni momento associati tutti e soli quei permessi che realmente gli spettano; praticamente possiamo dire che un dominio meno potente (ossia con meno permessi) non può estendere i propri permessi invocandone uno più potente e, viceversa, un dominio più potente vedrà la restrizione del proprio set di permessi invocando un dominio meno potente.

Codice Privilegiato: un caso particolare si ha quando una classe di sistema (o meno) voglia esplicitamente estendere il set di permessi di chi invoca un suo metodo per permettergli di accedere indirettamente ad una risorsa cui non può accedere direttamente (si pensi ad un applet che, volendo stampare un messaggio, abbia bisogno di uscire dalla sandbox per accedere al file che codifica un certo font che risiede sul file system locale, per far questo, il codice di sistema che realizza la stampa, dovrà temporaneamente estendere il set di permessi del thread dal quale è stato invocato) a tale scopo la parte del codice del metodo che effettua l'accesso alla risorsa critica sarà dichiarata *privilegiata*.

Per completare allora la regola da seguire nella attribuzione dei permessi possiamo dire che il set di permessi di un thread può essere esteso, se il codice eseguito in uno dei domini intermedi è privilegiato; in tal caso, infatti, i permessi associati ai domini che precedono quello in cui viene eseguito il codice in questione vengono trascurati, e la regola dell'intersezione viene applicata a partire da tale dominio che, in generale avrà un set di permessi più esteso dei precedenti.

Questa regola sarà applicata ogni volta che sia richiesto l'accesso ad una risorsa critica da parte del codice incaricato di gestire tale risorsa; quest'ultimo chiederà al sistema di sicurezza di valutare la richiesta per stabilire se debba essere accettata o meno.

3.3.3.5 Architettura del sistema e nuove classi

Nel nuovo modello il class loader ha un ruolo molto importante: deve creare un nuovo dominio di protezione per ogni diverso principal associato alle classi caricate. Per il momento solo il class loader può creare nuovi domini e concedere loro i permessi specificati dalla politica di sicurezza in uso. Non è infatti possibile richiedere la creazione di nuovi domini attraverso un'apposita API. Questo, se da un lato è una garanzia in termini di sicurezza (se i domini potessero essere creati da diverse entità sarebbe necessario considerare la possibilità di interferenze fra i domini creati da due entità diverse), dall'altro limita la possibilità di una gestione dinamica dei permessi associati ad un certo principal (sarebbe comodo poter revocare selettivamente alcuni dei permessi associati ad un certo thread creando un nuovo dominio e spostandoci il thread). Oltre ai domini creati ne esiste sempre uno che gestisce le risorse di sistema detto *System Domain* . Le applicazioni che risiedono sul CLASSPATH locale eseguono normalmente in questo ultimo dominio, avendo quindi accesso completo all'host. E' però stato inserito un meccanismo il quale fa in modo che tali applicazioni siano "caricate" dal nuovo class loader e che siano quindi anche esse sottoposte alla politica adottata.

3.3.3.5.1 Identificazione del principal

Nel modello originale di java (quello del jdk1.0.2), il principal, inteso come soggetto cui possono essere concessi dei permessi, è rappresentato da un URL, nel senso che al codice proveniente dallo stesso URL sono associati i medesimi permessi. Il nuovo modello (jdk1.2) estende tale concetto associando ad un URL anche il set di public key da utilizzare per la verifica del codice firmato proveniente da tale locazione. La coppia URL set di key rappresenta un *CodeSource*.

```
……
PublicKey[] signs = getAllSigners();
URL u = new URL(“http:\\www.www.com\App1\pippo.class”);
CodeSource cs = new CodeSource(u,signs);
……
```

L’estensione sta nel fatto che, come vedremo, è possibile concedere certi permessi indipendentemente dall’URL di provenienza del codice se questo è firmato da un certo principal.

3.3.3.5.2 I permessi di accesso alle risorse

Nella nuova architettura i permessi sono oggetti che possono quindi essere creati ed utilizzati da una classe per verifiche di possesso. Ovviamente, solo il sistema è in grado di creare ed associare i giusti permessi ai diversi domini di protezione.

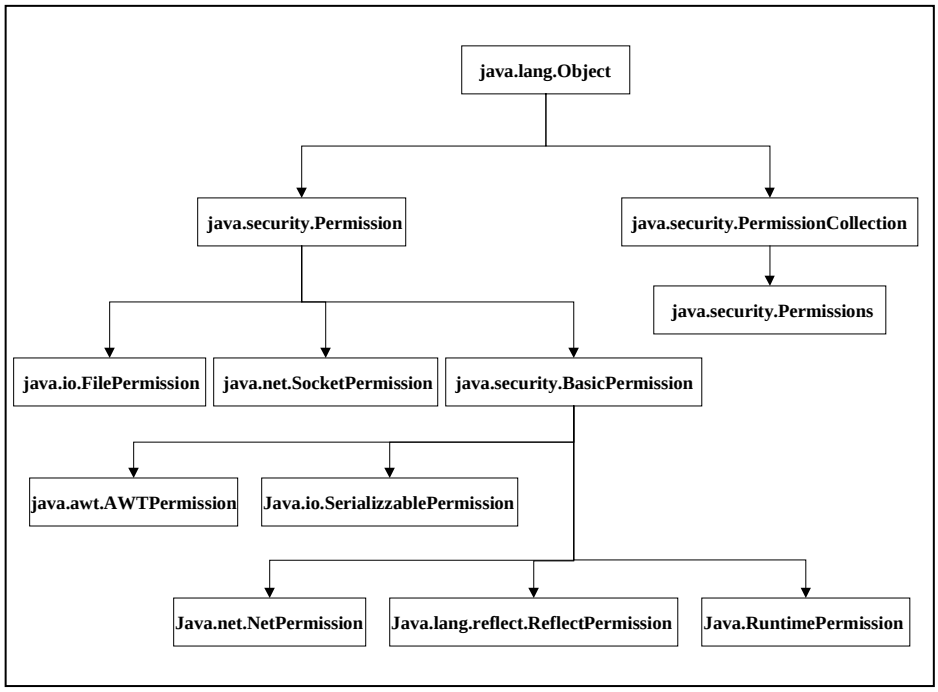

Figura 16: In ogni package sono inseriti i le classi che implementano i permessi per l'accesso alle risorse che il package permette di utilizzare.

java.securityPermission è la classe astratta base della gerarchia di classi che implementano i permessi di sistema ossia quelli che danno l'accesso al file system (java.io.FilePermission) alle risorse per l'utilizzo della rete (java.net.SocketPermission e java.security.NetPermission), alle proprietà di sistema (java.util.PropertyPermission), al sistema di supporto a run time (java.lang.RuntimePermission) e al sistema a finestre

(java.awt.AWTPermission). Per definire un nuovo tipo di permesso è necessario derivare da quest'ultima una nuova classe e ridefinire (tra gli altri) il metodo *implies*; un permesso A implica un permesso B se il fatto che è stato concesso A implica naturalmente che è concesso anche B. Per essere più precisi la creazione di un nuovo tipo di permesso prevede i seguenti passi:

- creare la classe che rappresenta il permesso;
- includerla nel package dell'applicazione che ne fa uso;
- inserire la rappresentazione in forma di stringa di tale permesso nel file della politica in modo che il sistema possa attribuire il nuovo permesso;
- inserire il codice necessario alla verifica del permesso stesso nel codice che gestisce l'accesso alla risorsa cui il permesso fa riferimento;
- sarebbe poi buona norma realizzare anche una nuova classe eccezione da scatenare nel caso di violazione del permesso;

I permessi vengono poi raggruppati attraverso le classi java.security.collection e java.security.Permissions che rappresentano rispettivamente una collezione omogenea di permessi ed un insieme di collezioni eterogenee.

3.3.3.5.3 Il meccanismo di autorizzazione

Come abbiamo già osservato, una volta creati i domini di protezione, il sistema attribuisce ad ogni classe i giusti permessi. Vediamo ora come viene eseguita la verifica dei permessi necessari all'esecuzione di una operazione critica.
java.security.AccessController è la classe statica che implementa il meccanismo per la verifica dei permessi di accesso alle varie risorse. Ogni qualvolta è necessario stabilire se un dato permesso è concesso o meno, il

codice adibito alla gestione della risorsa coinvolta invocherà il metodo *checkPermission* di questa classe.

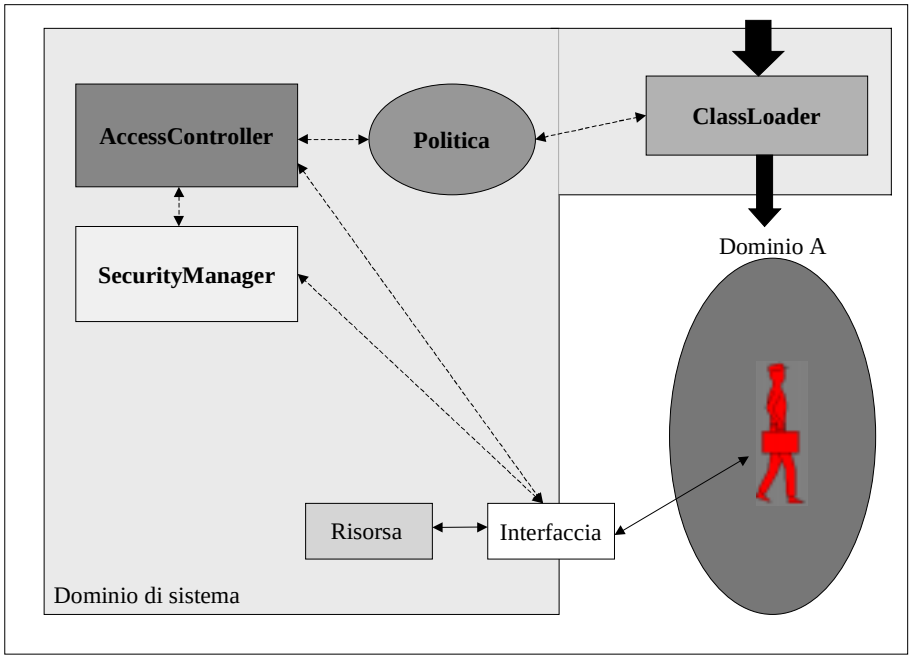

Figura 17: l'architettura del jdk1.2 semplifica l'utilizzo del meccanismo di access control; il security manager, che ora funge solo da wrapper dell'AccessController, è stato mantenuto per compatibilità

La nuova architettura snellisce il codice necessario al checking dei permessi, per verificare il permesso di lettura di un certo file con jdk1.1, ad esempio, era necessario scrivere:

```
ClassLoader loader = this.getClass().getClassLoader();
if (loader != null) {
 SecurityManager security=System.getSecurityManager();
 If (security != null) {
       Security.checkRead("path/file"); }
}
```

ora questo segmento di codice si trasforma semplicemente in:

```
FilePermission perm = new FilePermission(“path/file”
                                          ,”read”);
AccessController.checkPermission(perm);
```

La chiamata al metodo checkPermission dell’Access Controller va fatta indipendentemente dal fatto che alla classe sia associato un class loader, questo permette di estendere i controlli anche al codice locale.
Come spiegato sopra, nel caso in cui la chiamata sia fatta da un thread che rappresenta l’ultimo anello di una catena di invocazioni, la regola che si segue per assegnare un dato permesso è la seguente:

- se tutti i chiamanti della catena possiedono il permesso allora concedilo al thread;
- il permesso può essere concesso anche nel caso in cui un certo chiamante della catena lo possieda, abbia dichiarato privilegiato il proprio codice, ed anche tutti i chiamanti successivi lo possiedano;
- altrimenti scatena una eccezione del tipo AccessControlException.

Per l’implementazione di questa regola è possibile seguire due diverse strategie: una, *detta eager evaluation*, in cui all’attraversamento di ogni dominio di protezione i permessi del thread vengono dinamicamente aggiornati; la seconda, detta *lazy evaluation*, in cui il calcolo dei permessi è effettuato solo quando richiesto. La prima rende più rapida la fase di checking, ma, essendo i cambi di dominio molto più frequenti dei checking, appesantisce il thread con inutili aggiornamenti di permessi. Per questo motivo attualmente Java ha scelto la strada della lazy evaluation.
Per quanto riguarda la dichiarazione di codice privilegiato sono previste due primitive che vanno usate come segue:

```
try  {
 AccessController.beginPrivileged();
 <some sensitive code>
} finally {
 AccessController.endPriveleged();
}
```

3.3.3.5.4 La definizione di una politica di sicurezza

Stabiliti quali sono le risorse critiche, e quindi i vari tipi di permesso, e realizzato un meccanismo per il controllo degli accessi, occorre ora decidere in che modo rappresentare il legame tra i vari principal (noti o meno) ed i relativi permessi, ci serve insomma una politica ossia un oggetto al quale poter chiedere "quali permessi devo associare a questo principal ?".
java.security.Policy è la classe astratta che rappresenta la politica di sicurezza attuata dal sistema; le classi da questa derivata specificheranno il modo in cui la politica è memorizzata e può essere modificata. Possono esisterne più istanze, ma solo una di queste per volta può essere attiva. Invocando il metodo *evaluate* è possibile associare ad ogni codesource i relativi permessi. Il provider della politica ossia la classe che effettivamente viene utilizzata dal sistema è specificato nel file di configurazione ~/lib/security/java.security. A default viene utilizzata un'implementazione detta java.security.FilePolicy configurata staticamente all'inizializzazione della macchina virtuale attraverso un file (~/lib/security/java.policy) di testo formato da una serie di entry conformi al seguente schema:

```
grant [signedBy "signernames"] [, Codebase "URL"] {
 permission permissionclassname
["targetname"][,"action"]
[, signedBy "signernames"];
 permission …..
}
```

I primi due campi identificano il codesource attraverso l'URL e gli alias (*signernames*) che individuano il set di chiavi pubbliche da utilizzare per verificare l'autenticazione del codice; il campo signernames è, in generale, una lista separata da virgole; al codice si richiede di possedere una firma corretta per ognuno degli alias della lista.

Le parole *grant* e *permission* sono parole riservate che indicano rispettivamente l'inizio di una nuova entry nel file e l'inizio di una sottoentry.

Il termine *permissionclassname* denota il nome della classe che implementa un certo tipo di permesso; *targetname* è il tipo di risorsa cui il permesso si riferisce, *action* l'azione di cui viene concesso il permesso. Il secondo *signernames* specifica l'alias del principal cui è richiesto di autenticare il bytecode che implementa la classe del permesso specificato; tale autenticazione si rende necessaria in quanto un entry nel file policy esprime la fiducia che l'utente (o l'amministratore) ripone in un determinato principal e nella fonte della classe che implementa un dato permesso; il permesso specificato in tali sottoentry è concesso solo se viene verificata correttamente questa seconda autenticazione.

L'utente può specificare una ulteriore politica di sicurezza (in un file ~/.java.policy) le cui entry verranno semplicemente aggiunte a quella della politica standard.

Un dato codesource può fare match con molti altri nel senso che può uniformarsi alle informazioni riportate in diverse entry del policy file; un codesource che abbia un URL del tipo deis38.deis.unibo.it/export/home/tenti e nessun signer, ad esempio, fa match con ogni codesource il cui URL ne costituisca un prefisso come deis38.deis.unibo.it/export; per questo è necessario stabilire una regola in base alla quale determinare l'esatto set di permessi da concedere.

Tale regola si articola in quattro passi:

- se il codice è firmato cerca tutte le entry il cui campo signernames rappresenta un subset del set di signers del codice;

- se in nessuna entry compare un certo alias trascuralo; se, al primo passo, non si è trovata nessuna entry considera il codice non firmato;
- se al primo passo si è trovato un set di entry seleziona tutte quelle con il cui URL fa match l'URL del codice;
- se al terzo passo non è stato possibile selezionare nessuna entry applica al codice la politica della sandbox classica, altrimenti concedi al codice tutti i permessi delle entry trovate.

3.3.3.5.5 Il nuovo ClassLoader

Il nuovo class loader rappresenta un po' il motore della nuova architettura per la sicurezza. E' questo, infatti, il componente incaricato del confinamento delle varie classi nei diversi domini di sicurezza. Si tratta di una classe astratta che definisce un certo numero di metodi come finali per garantire la voluta sicurezza, mentre permette la ridefinizione di quei metodi che caratterizzano il loader ed in particolare il suo comportamento nella ricerca dei dati per il caricamento delle nuove classi. Se tali metodi non vengono ridefiniti il comportamento standard che si adotta nel caricamento delle varie classi è espresso dal seguente algoritmo:

- verifica se la classe è già stata caricata e risolta;
- altrimenti, se si tratta di una classe di sistema, caricala con il null loader;
- altrimenti, la classe viene caricata attraverso una chiamata al metodo *findAppClass*, un metodo non final la cui implementazione standard cerca la classe nel path specificato dalla proprietà java.app.classpath.

Come specificato nell'algoritmo, il jdk 1.2 introduce un secondo path, le classi provenienti dal questo sono caricate con un SecureClassLoader e, di conseguenza, sono soggette alla politica di sicurezza globale.

In particolare è il metodo findAppClass a poter essere ridefinito per estendere le capacità del loader, si pensi, ad esempio, al caso di un AppletLoader che immagazzina tutto il materiale contenuto in un jar file, e che, di conseguenza, ha convenienza nel cercare prima nella sua cache che altrove.

3.3.3.5.6 Sicurezza anche per il codice locale

Per permettere anche alle applicazioni locali di sottoporsi alla politica di sicurezza globale è sufficiente utilizzare la classe java.security.Main come segue:

```
java java.security.Main applicazione
```

in questo modo l'applicazione (che deve risiedere sul java.app.class.path) verrà caricata con un SecureClassLoader.

3.3.3.5.7 Il problema del contesto

Quando un nuovo thread è creato questo ha uno stack completamente vuoto e quindi non possiede molte delle informazioni che formano il contesto di sicurezza che caratterizzava il thread padre. Questo non costituisce un problema da un punto di vista della sicurezza in sé, ma può portare a subdoli errori nella programmazione del codice di sistema. La JVM è stata modificata in modo tale che alla creazione un thread erediti il contesto di sicurezza del padre ritrovandosi di fatto con un contesto di sicurezza espanso.
Un problema simile si ha quando il check di un permesso che andrebbe fatto nel contesto di un dato thread deve invece essere eseguito in un contesto differente. Si pensi al caso in cui un evento scatenato da un certo thread provoca, in risposta, la chiamata di un metodo di un altro thread il quale

avendo un contesto differente non potrà eseguire il check correttamente. Per risolvere questo problema l'AccessController mette a disposizione un metodo statico che ritorna il contesto del thread corrente:

```
AccessControlContext acc=new
AccessController.getContext();
```

in tal modo il contesto può essere passato al secondo thread dove il check può essere fatto correttamente con la seguente chiamata:

```
acc.checkPermission(permission);
```

3.3.3.5.8 Oggetti con guardia

Un problema analogo a quello che ha portato alla realizzazione della classe AccessControlContext si ha quando il fornitore di una risorsa è in un thread differente dal consumatore della stessa ed il consumatore non può fornire il proprio contesto di sicurezza per una qualche ragione (ad esempio è troppo grande).In casi simili il fornitore può proteggere la risorsa inglobandola in un GuardedObject e passando poi quest'ultimo al consumatore; nella creazione del GuardedObject il fornitore specificherà anche un oggetto di tipo Guard il quale sarà incaricato di effettuare i controlli appropriati nel contesto del consumatore; Guard è in effetti una interfaccia così che ogni oggetto può scegliere di diventare un Guard, la classe Permission ad esempio implementa tale interfaccia che ha un solo metodo chiamato checkGuard(object). Facciamo ad esempio il caso che la risorsa in esame sia il file /a/b/c.txt , allora il fornitore creerà un GuardedObject in questo modo:

```
FileInputStream f = new FileInputStream("/a/b/c.txt");
FilePermission p = new
FilePermission("/a/b/c.txt","read");
GuardedObject g = new GuardedObject(f, p);
```

poi passerà quest'ultimo al consumatore il quale, per ottenere f dovrà eseguire il seguente segmento di codice:

```
FileInputStream fis = (FileInputStream) g.getObject();
```

questo metodo, a sua volta, invocherà il metodo checkGuard() sul oggetto p e, trattandosi di un oggetto di tipo Permission il suo metodo checkGuard sarà di fatto:

```
AccessController.checkPermission(this);
```

Ovviamente tra consumatore e fornitore ci dovrà essere un accordo sull'interfaccia da adottare nell'utilizzo del GuardedObject altrimenti non potrebbe avere luogo il casting opportuno sull'oggetto restituito dal metodo getObject().

3.3.4 Un esempio completo di applicazione del modello jdk1.2

Supponiamo di aver realizzato una classe pubblica DB.BaseDiDati che memorizza informazioni riservate, questa classe avrà un metodo getRecord(int num) che restituisce il record di indice num; la nostra base di potrà essere interrogata dalle applet caricate da un apposito browser.
Supponiamo poi che un applet proveniente dall'URL "www.www.com" e firmata da un principal identificato dall'alias "pippo" voglia aprire un file "file1" in scrittura nella directory "/tmp", e che, successivamente l'applet voglia invocare il metodo getRecord(12); supponiamo infine di voler concedere all'applet sia il permesso di scrittura nella directory "/tmp" sia il permesso di tipo getRecordPermission (necessario per un buon esito della seconda operazione) per i record compresi tra 0 e 20.

Innanzitutto creiamo la classe getRecordPermission derivandola dalla classe astratta java.security.Permission:

```
package DB;
import java.security.Permission;

class getRecordPermission extends Permission{
 /* Rappresentazione interna del permesso:
  * trattandosi di un permesso di accesso ad un set di
  * record potrebbe aver senso memorizzare i limiti
  * del set
  */

 public getRecordPermission(String RecSet) {
   /* Elaborazione della forma interna:
    * dovremo estrarre i limiti del set dalla
    * stringa "0-20"
 }

 public boolean implies(Permission) {
   /* un set "0-20" ad esempio implicherà un
    * set "5-15"
    */
 }

 ……..
}
```

Poi apriamo il file .java.policy presente nella nostra directory di lavoro (come abbiamo già detto l'implementazione standard della politica carica, all'inizializzazione, un file di configurazione java.policy che si trova nella directory ~/lib/security, dove con ~ si indica la directory dove è stato installato il jdk1.2, poi, se presente, aggiunge le voci del file .java.policy che si trova nella directory di lavoro) con un text editor e aggiungiamo la seguente voce:

```
grant signedBy "Pippo" {
 permission java.io.FilePermission "/tmp" "write";
 permission DB.getRecordPermission "0-20";
}
```

Infine inseriremo nel codice del metodo getRecord() la chiamata necessaria al controllo del permesso:

```
public final getRecord(int RecNum) throws
SecurityException {
 Integer rni=new Integer(RecNum);
 String rns=rni.toString();
 getRecordPermission perm=new getRecordPermission(rns);
 try {
  AccessController.checkPermission(perm);
 } catch(SecurityException e) {
       throw e; }

 ……..
}
```

Per gestire la situazione in cui non ha il permesso di eseguire una certa operazione, l'applet dovrà raccogliere le eccezioni di sicurezza eventualmente sollevate e agire di conseguenza; così, ad esempio, per prendere in considerazione il caso in cui non avesse il permesso di scrittura nella directory "/tmp" nel codice dell'applet potrebbe comparire il seguente frammento:

```
……..
FileOutputStream fos=null;
FilePermission perm=new FilePermission("/tmp","write");
try {
 fos=new FileOutputStream("/tmp/file1");
} catch(SecurityException e) {
       System.err.println(" Non posso scrivere in
tmp!");
}
……
```

4 Implementazione di un modello di sicurezza per un ambiente ad agenti

4.1 Caratteristiche del sistema ad agenti

4.1.1 Place e Domini

Nel sistema ad agenti che consideriamo (vedi Figura 18), il mondo appare agli agenti come un insieme di *Place* raggruppati in *Domini*.

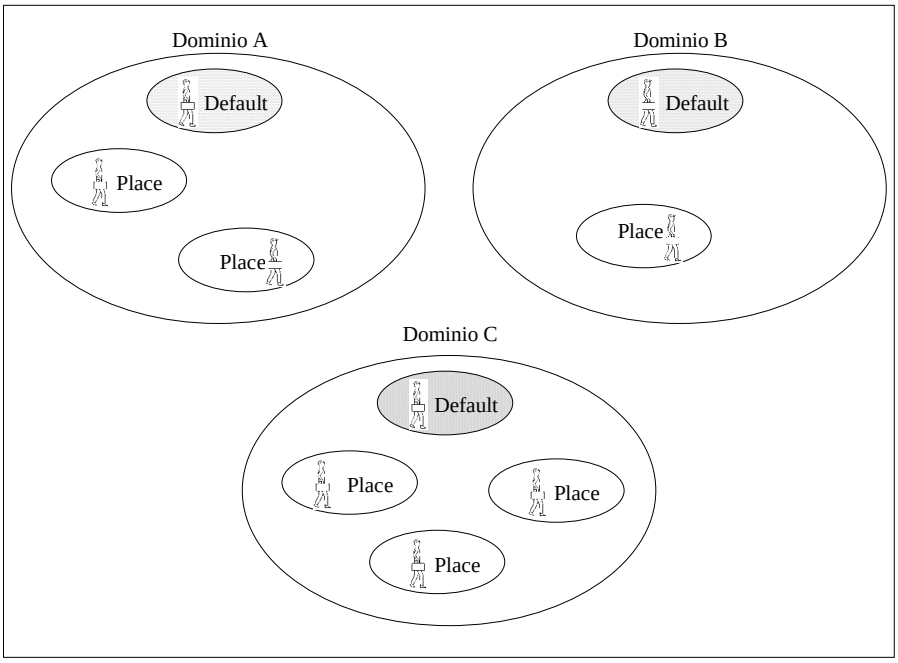

Figura 18: Gli agenti vedono la rete come un insieme di place raggruppati in domini.

Un place rappresenta il contesto in cui gli agenti eseguono ed interagiscono tra loro e con i servizi messi a disposizione, mentre un dominio rappresenta

una località all'interno della rete (località che può essere fisica o soltanto di carattere logico).
In ogni dominio è presente un *place di default* che funge da porto di approdo degli agenti provenienti dagli altri domini; una volta arrivato nel place di default, un agente ha visibilità di tutti gli altri place del dominio e può così scegliere la prossima tappa.

4.1.2 Il contesto di esecuzione degli agenti

Nel place l'agente interagisce con tre moduli che forniscono funzionalità diverse:

- **L'Agent System** gestisce tutte la funzioni di base: la *mobilità*, lo *scambio di messaggi*, la *gestione delle risorse fisiche* ed il *monitoraggio*;
- **Il Place Manager** mette a disposizione degli agenti una *Blackboard*, un *servizio di naming* locale per agenti servitori e dà la possibilità ad ogni agente di *registrarsi con un alias* presso il place, in questo modo è facile realizzare un'interazione di tipo anonimo;
- **L'Information Service** permette la *ricerca remota di informazioni* quali la posizione di un agente o la risoluzione di un alias in un determinato place; la ricerca avviene sempre all'interno di un dominio che per default è quello di residenza.

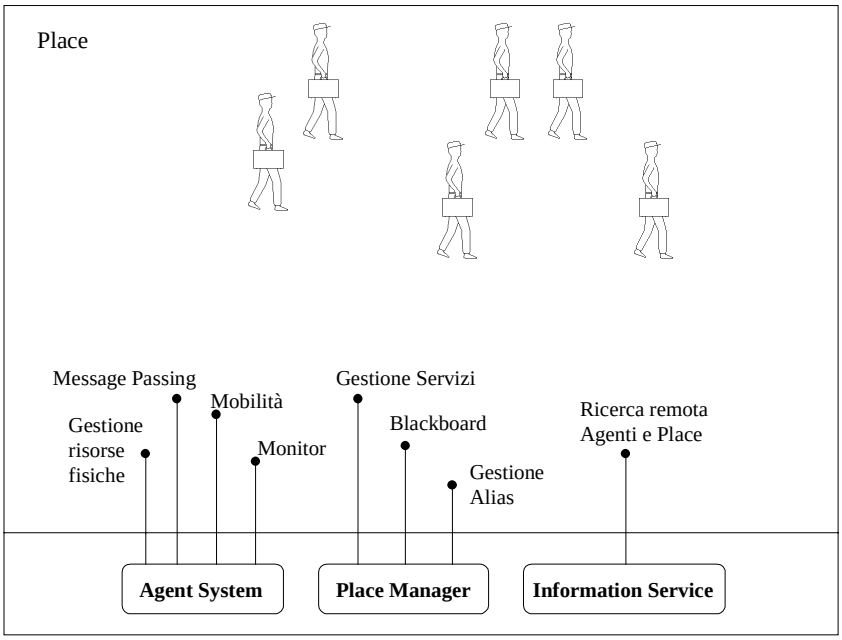

Figura 19: Nei place gli agenti interagiscono con le diverse parti del sistema.

Ogni agente e' identificato attraverso il place e il dominio di origine, il nome della classe che lo implementa ed un identificatore numerico unico, all'interno del place di origine; e' facile, quindi, per l'information service procurarsi la posizione di un agente di cui si conosce l'identificativo, interrogando il place di origine dell'agente, ossia quello incaricato di tenerne traccia.

4.1.3 La Comunicazione

La comunicazione fra agenti può avvenire mediante due distinti meccanismi:

- lo **scambio di messaggi** come *interazione lasca*; ogni agente possiede una Mailbox ed ha la possibilità di spedire un messaggio ad ogni altro agente di cui conosce l'identificatore; il meccanismo è trasparente alla

locazione nel senso che l'agente destinatario viene raggiunto anche se si trova in un dominio differente; attraverso l'identificatore, infatti, il sistema e' in grado di conoscere l'attuale posizione dell'agente destinatario e di effettuare la consegna del messaggio;

- la **condivisione di oggetti** costituisce invece una *interazione stretta* ed è il meccanismo di comunicazione più naturale in un linguaggio ad oggetti come Java. Ogni place gestisce un contenitore nel quale gli agenti possono depositare gli oggetti che vogliono condividere ed ottenerne in cambio un identificatore unico per ogni oggetto; l'identificatore può essere scambiato per passare da un agente ad un altro il riferimento all'oggetto.

Sopra questi si possono poi costruire altri tipi di meccanismo più complessi come una **blackboard**; una blackboard è un contenitore in cui ogni agente può depositare un messaggio composto da una stringa che lo identifica e da un oggetto; chiunque conosca la stringa di identificazione potrà successivamente leggere il messaggio; con un meccanismo di questo tipo si realizza facilmente uno schema di comunicazione anonima.

Rispetto a quanto abbiamo osservato nel capitolo 2, possiamo dire che lo scambio di messaggi rappresenta la più naturale forma di interazione fra agenti mobili che non si trovano nello stesso place; viceversa la condivisione di oggetti si presta ad una interazione più stretta quale quella che si può avere tra un agente mobile ed uno di servizio o tra due agenti mobili che si trovino nello stesso place; infine la blackboard si presta comodamente alla realizzazione di una interazione di tipo anonimo.

4.1.4 La mobilità

La mobilità degli agenti si esplica nella invocazione del metodo **go**() il cui primo argomento può essere di due tipi:

- **PlaceName**: all'interno di un dominio è sufficiente specificare il "nome" del place di destinazione;
- **DomainName**: se invece l'agente deve spostarsi verso un place in un dominio differente, prima entrerà in tale dominio (ossia raggiungerà il suo place di default specificando, come parametro della go(), il "nome" del dominio) poi raggiungerà il place desiderato.

Il secondo parametro del metodo go() specifica lo *StartMethod* ossia il metodo dell'agente che verrà invocato una volta giunto a destinazione. Il sistema che consideriamo, infatti, non prevede una mobilità di tipo strong, ma cerca di avvicinarsi a questa permettendo di modularizzare il comportamento dell'agente durante il suo viaggio attraverso la possibilità di specificare, al limite, ogni volta uno StartMethod differente.
Quando la migrazione, non è possibile, per un qualche motivo, viene lanciata una eccezione del tipo **CantGoException**, l'agente ha così modo di comportarsi di conseguenza.

4.2 Caratteristiche del modello di sicurezza

Dotare di sicurezza un sistema come quello che abbiamo descritto nei paragrafi precedenti significa dotarlo dei meccanismi necessari a garantire la privacy e l'integrità di agenti e place, di quelli che occorrono per eseguire una corretta autenticazione ed autorizzazione e degli strumenti con cui poter scrivere appropriate politiche di sicurezza. Lo schema di identificazione degli

agenti dovrà essere sufficientemente flessibile da rendere la stesura delle politiche rapida ed intuitiva; il meccanismo di AccessControl dovrà permettere la protezione di risorse di diverso tipo ed appartenenti a diversi livelli dell'architettura; dovrà essere possibile definire domini di protezione differenti ogni qualvolta nasca la necessità di isolare agenti che operano con gradi di fiducia differenti all'interno del place, ogni dominio di protezione conterrà gli agenti cui è riconosciuto lo stesso grado di fiducia; lo schema delle politiche dovrà rendere significativo il modello stesso. I canali di comunicazione dovranno essere protetti per garantire la privacy delle informazioni che vi transitano. I meccanismi di sicurezza devono avere un costo ridotto sia in termini di utilizzo delle risorse sia per quanto riguarda l'impatto sulle prestazioni complessive del sistema.

4.2.1 L'identificazione degli Agenti

L'identificatore di un agente deve svolgere due ruoli:

- deve *identificare l'agente* all'interno del sistema in modo da poter sempre distinguere due agenti;
- deve fungere da *identificatore del principal* per cui l'agente esegue o, più in generale, deve permettere di attribuire all'agente il giusto grado di trustness che gli compete all'interno del sistema.

Teoricamente questa distinzione porterebbe alla previsione di due identificatori separati, ma in pratica uno schema di identificazione unico degli agenti abbastanza flessibile è adatto anche alle esigenze della sicurezza; tale identificatore si comporrà di due parti:

- la prima parte è l'identificatore dell'agente ed è composto dal nome del dominio e del place in cui l'agente è stato creato, dal nome della classe

che implementa l'agente, e da un identificatore numerico unico all'interno di tale place;

- la seconda parte è costituita da un set di *credenziali* ossia di informazioni, cui l'agente non ha accesso, che certificano il fatto che l'agente riscuote un certo grado di fiducia da parte dei soggetti che le hanno rilasciate. La prima di queste credenziali è quella che l'utente creatore dell'agente (col termine creatore si intende l'utente che lo ha messo in esecuzione) rilascia all'agente per permettere ai vari place che l'agente attraverserà di risolvere il problema della paternità dell'agente stesso. Le altre credenziali saranno acquisite dall'agente durante il suo viaggio e potranno accreditarlo di permessi supplementari.

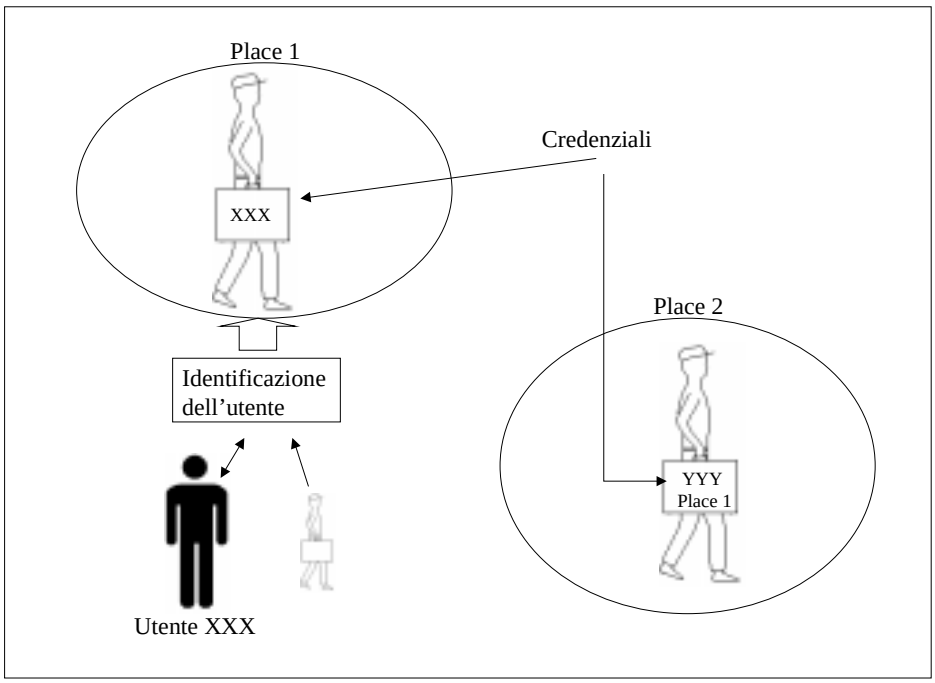

Figura 20: Alla creazione di un agente l'utente viene identificato per poter dotare l'agente della prima credenziale.

La presenza di questa seconda parte conferisce la voluta flessibilità permettendo ad un agente di modificare dinamicamente il proprio grado di trustness nel sistema.
Per ampliare il set delle proprie credenziali un agente potrà richiederle ai place che visita, i quali, in base alla propria politica, decideranno se concederle o meno.

4.2.2 L'autorizzazione degli agenti

Il modello di protezione do noi adottato è quello delle *capability list* (vedi [SG94]): ad ogni dominio di protezione è associata una lista contenente tutti e soli i diritti di accesso (*permessi*) che gli agenti acquisiranno entrando in tale dominio.
Per poter accedere alle risorse messe a disposizione di un place, ogni agente dovrà quindi avere gli adeguati *permessi.* Se il permesso richiesto per una certa operazione non rientra tra quelli concessi all'agente l'operazione fallirà. Nel nostro caso i *permessi* sono da intendere come autorizzazioni positive all'accesso. Come vedremo successivamente, è però necessario prevedere anche la presenza di permessi negativi da intendersi come divieti espliciti di accesso ad una certa risorsa. Sono prevedibili tutta una serie di permessi di base relativi all'utilizzo delle risorse fisiche. Sarà inoltre presente un'interfaccia che permetta una facile estensione alla gerarchia dei permessi per la creazione di permessi relativi a risorse di più alto livello: gli agenti di servizio, infatti, dovranno poter definire la loro politica di accesso per regolare l'utilizzo delle loro risorse interne e dei loro servizi.
Gli oggetti permesso si compongono di due parti:

- il *target*: ossia la risorsa cui il permesso concede l'accesso (ad esempio "/temp");

- l'*azione*: ossia il tipo di accesso alla risorsa che viene concesso (ad esempio "write");

alcune categorie di permessi non presenteranno il secondo campo (si pensi al permesso che concede ad un agente di clonarsi).
Un tipo di permessi particolarmente importante è quello detto dei *permessi quantitativi* (per distinguerli da quelli di cui abbiamo parlato prima i quali vengono detti *qualitativi*), tra questi ad esempio potrebbero rientrare lo spazio a disposizione sul file system locale o il numero totale di passi (intesi come trasferimenti da un place ad un altro) a disposizione di un agente.

4.2.2.1 Categorie di permessi

I diversi tipi di permessi possono essere suddivisi in base a diversi criteri di caratterizzazione; innanzitutto distingueremo fra permessi *di accesso alle risorse* e permessi *per la configurazione del sistema*; i primi riguardano l'accesso diretto alle risorse fisiche o logiche del place, i secondi invece, si riferiscono agli strumenti di interazione con il sistema di sicurezza.
Fra i permessi di accesso alle risorse troviamo quelli riguardanti l'accesso al file system, al sistema a finestre (o più in generale alla console, ossia all'interazione con l'utente), ai servizi di rete (intesi, essenzialmente, come servizi di basso livello come socket e stream tcp), ma anche quelli che si riferiscono all'interazione con il sistema (come l'acquisizione e la modifica delle variabili di ambiente o l'interazione con i servizi specifici del place).
Nella categoria dei permessi di configurazione rientreranno invece i permessi che autorizzano un agente ad intervenire su parti dell'ambiente che, più o meno direttamente, interagiscono con il sistema di sicurezza come il gestore della politica di sicurezza o il modulo incaricato della crittografia, ma anche

il meccanismo di mobilità degli agenti o quello di gestione degli oggetti condivisi.

Ci saranno poi permessi la cui concessione esprime il possesso di una proprietà valida in tutto il dominio (*permessi globali*: si pensi al permesso di ingresso in un determinato place del dominio) ed altri che invece potranno concedere solo privilegi locali ad un determinato place (*permessi locali*).

4.2.3 Le politiche di sicurezza

Il disegno dell'oggetto *politica di sicurezza* è molto delicato in quanto deve rispondere contemporaneamente ad una serie di esigenze contrastanti:

- deve permettere la dichiarazione di schemi di protezione abbastanza articolati;
- deve prevedere la possibilità di distribuire la responsabilità ;
- deve essere facile redigere nuove politiche o modificare quelle vecchie.

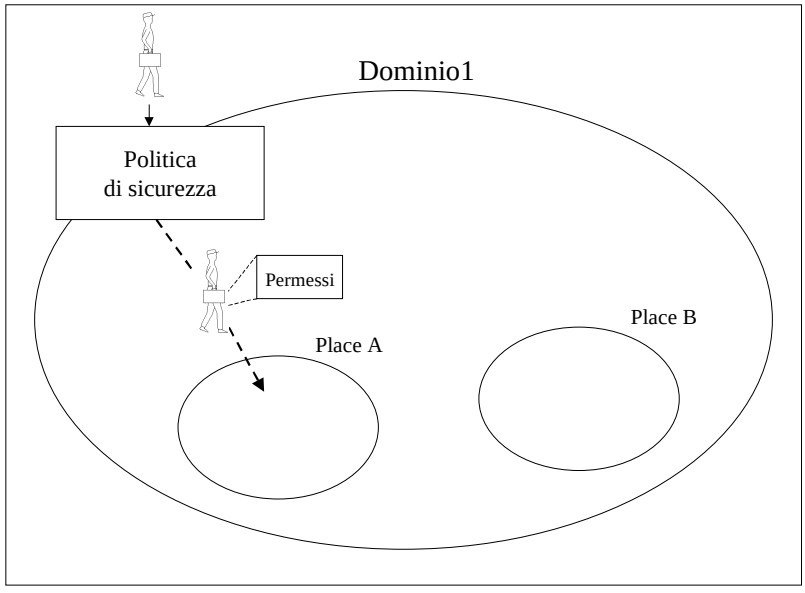

Figura 21: Una politica globale tratta tutti i place del dominio nello stesso modo.

In relazione all'ultimo punto possiamo osservare che, rispetto ad un dominio, la *politica* potrà essere più o meno *globale*:

- se è *globale,* i vari place saranno considerati come uguali nel senso che un agente avrà gli stessi permessi indipendentemente dal fatto di trovarsi in un certo place o in un place diverso dello stesso dominio (vedi Figura 21);
- se la politica non è globale ed è una *politica locale di place,* il set di permessi associati ad un agente cambierà anche all'interno del dominio, a seconda del place in cui l'agente si trova (vedi Figura 22);
- con una politica *ibrida,* alcuni permessi saranno concessi in maniera globale, poi i singoli place avranno la possibilità di modificare tale politica di default (vedi Figura 23) aggiungendo o rimuovendo permessi (definendo cioè una politica locale).

Occorre evidenziare che, nel caso di politica ibrida, la personalizzazione della politica di dominio che opera ogni place può seguire due distinte filosofie:

- considerando prioritaria la politica di place, la politica di dominio rappresenterebbe il nucleo minimo di permessi (tutti positivi) garantiti da tutti i place del dominio; ogni place avrebbe poi la possibilità di *estendere* la politica di dominio aggiungendo permessi positivi per permettere l'accesso a risorse specifiche o, semplicemente, per rendere meno restrittiva la propria politica. In questo scenario, i permessi di tipo negativo non sono necessari, è sufficiente realizzare politiche di dominio abbastanza generali. Questa soluzione è quella di costo minore, infatti l'aggiornamento della politica di place consiste in una semplice aggiunta a quella di dominio;

- se, viceversa, consideriamo prioritaria la politica di dominio, quella di place potrà solo eliminare dalla prima i permessi che ritiene inadeguati, ossia attuare una *restrizione* della politica di dominio. In uno scenario di complessità superiore (in una situazione di compromesso fra le due strategie), si possono prevedere permessi positivi e negativi sia nella politica di dominio sia in quella di place; la regola di personalizzazione prevederà, in questo caso, che la politica di place attui una restrizione della parte positiva della politica di dominio in base alla propria parte negativa ed una estensione in base alla porzione dei propri permessi positivi che non violano la parte negativa della politica di dominio; in poche parole il place potrà liberamente restringere la politica di dominio, mentre la sua estensione dovrà essere filtrata in base ai divieti che il dominio pone esplicitamente.

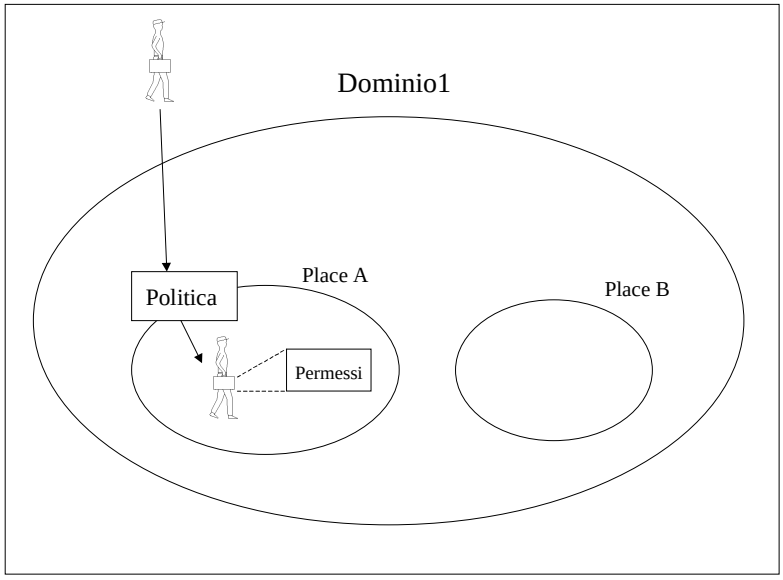

Figura 22: Nel caso di politiche locali, ogni place potrà definire una propria politica di sicurezza legata alle proprie esigenze.

L'adozione di una politica globale si sposa bene con la ragionevole prospettiva di considerare il dominio un ambito omogeneo anche dal punto di

vista della sicurezza, ma presenta anche un fondamentale difetto: non prende in considerazione il fatto che il dominio rappresenta un insieme di place logicamente correlati, ma, in generale, mappati su host diversi. Questo, infatti, comporta una oggettiva difficoltà nella definizione, a livello di dominio, di permessi specifici: consideriamo, ad esempio, il caso che a un determinato agente si voglia concedere l'accesso ad una parte del file system, in generale occorrerà stabilire di quale file system si parla visto che place mappati su hosts diversi accederanno a file system diversi.
Prevedendo invece una politica non globale, all'interno del dominio ogni place definirà una propria politica. In questo caso ogni place definirà solo i permessi di accesso alle risorse cui effettivamente può accedere: ad esempio al file system della macchina su cui è mappato.

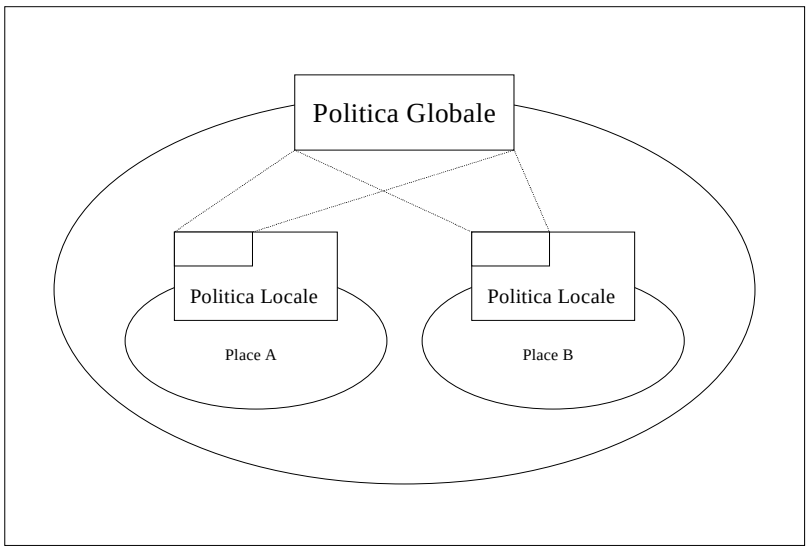

Figura 23: Una situazione ibrida racchiude i vantaggi di entrambi le situazioni precedenti.

Una politica completamente decentrata, però, renderebbe problematica la definizione di una politica di dominio se non come "riassunto" delle varie politiche dei place del dominio. Una politica del tutto decentrata non

permette, cioè, di imporre dei limiti all'agente in modo tale che il suo set di permessi sia il più ristretto possibile compatibilmente con le esigenze del dominio.

Nel nostro caso adotteremo, allora, una politica ibrida che prevede la compresenza di una politica globale (che chiameremo *politica di default* o *di dominio*) e di una serie di politiche locali ai vari place che raffinano la prima, restringendola (da qui la necessità di permessi negativi) o estendendola con i limiti visti.

Quando un utente decide di inserire nel sistema propri agenti per un certo scopo, può succedere che egli voglia restringere le capacità dei vari agenti per differenziare, ad esempio, due agenti dello stesso tipo che opereranno in domini diversi, se i diversi agenti venissero distinti solo in base alla loro identificazione, due agenti dello stesso tipo che operano per conto dello stesso utente avrebbero sempre gli stessi permessi; associando un set di *preferenze* (vedi Figura 24) all'agente è possibile personalizzarne il comportamento imponendogli dei limiti ulteriori all'interno del set di permessi che i vari place gli concederanno (così come avviene con le *local allowances* in ARA).

Per quanto riguarda l'opportunità di inserire certe categorie di permessi nella politica di dominio o in quelle locali possiamo osservare che i permessi globali troveranno il loro più appropriato punto di attribuzione nella politica di dominio mentre i permessi locali andranno specificati nelle politiche di place. Volendosi conformare al principio del minimo privilegio occorrerà che la politica di default del dominio risulti più specifica possibile, sia nel concedere permessi positivi che nel imporre permessi negativi. Sarà perciò necessario che la categoria dei permessi globali comprenda una gamma abbastanza vasta, in modo tale da poter caratterizzare abbastanza precisamente il comportamento standard dei place.

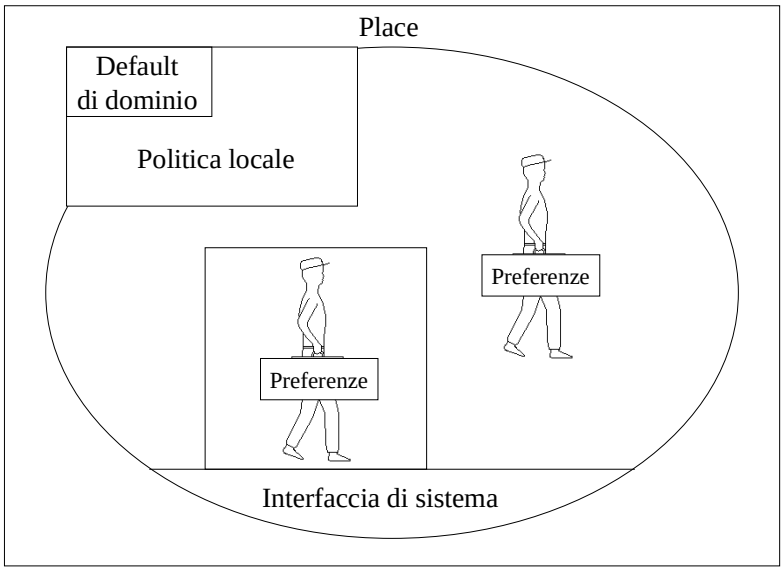

Figura 24: Il modello di riferimento mostra gli elementi principali del sistema.

Ogni place si riserverà poi, attraverso il meccanismo di personalizzazione previsto, la capacità di restringere la politica di default dettata dal gestore di dominio per adattare alla propria situazione quel subset di permessi che avrebbero pari diritto di rientrare nella categoria dei globali o in quella dei locali, ma che, per la scelta di priorità della politica di dominio, vengono inseriti in quest'ultima.

4.2.4 Agenti figli e domini di protezione

Il nostro modello si basa essenzialmente sulla capacità di confinare agenti con gradi di trustness differenti in domini di protezione differenti in modo tale che ogni agente abbia, all'interno di ogni place, lo spazio di manovra che gli compete. Questo si traduce nel fatto che, all'ingresso in un place, ad ogni agente sarà associato un set di permessi che l'agente perderà uscendo dal place (ossia verrà creato un nuovo dominio di protezione per l'agente, a meno che un tale dominio non esista già nel place, in tal caso l'agente vi sarà

inserito). Per gli agenti che vengono creati da altri agenti (agenti figli) è previsto un trattamento particolare: ogni agente figlio erediterà dal padre il medesimo set di permessi, ossia verrà inserito nel medesimo dominio di protezione, ma solo all'interno del place in cui viene creato; otterrà, infatti, un set di credenziali vuoto (se si esclude quella relativa al creatore che sarà conservata) e non una copia delle credenziali del padre; trasferendosi in un place diverso, quindi, sarà inserito, in un dominio di protezione, in generale, diverso da quello in cui sarebbe inserito l'agente padre se si muovesse in tale place.

4.2.5 Estensione dell'interfaccia di place

Come abbiamo visto nel capitolo 2, il paradigma ad agenti mobili nasce, tra l'altro, dall'esigenza di dotare le applicazioni distribuite di maggiore flessibilità e di permettere una facile estensione delle funzionalità fornite. In questa ottica i servizi messi a disposizione dai vari place dovranno costituire un set dinamico facilmente estensibile con nuovi servizi o con versioni aggiornate dei servizi esistenti.

L'interfaccia del place verrà di fatto estesa con nuovi servizi dagli agenti che si registreranno come servitori di qualche tipo (li chiameremo agenti di servizio); per tali agenti prevediamo la possibilità di definire una propria politica di accesso ai servizi che implementano.

Poiché le estensioni all'interfaccia del place avvengono attraverso l'introduzione di nuovi oggetti condivisi all'interno del place stesso, per controllare l'accesso ai nuovi servizi sarà sufficiente controllare l'accesso a questi oggetti, salvo poi prevedere che i servizi stessi possano implementare un proprio controllo d'accesso più raffinato. Entrambe le funzionalità richiedono l'invio da parte dell'agente di servizio di un nuovo segmento di politica al gestore della politica di sicurezza del place.

Il ruolo degli agenti di servizio fa nascere un problema: come controllare la consistenza delle politiche di questi agenti, ossia come controllare che un agente di servizio non funga da cavallo di Troia permettendo ad altri agenti di compiere operazioni che il place aveva loro negato.
Per prevenire l'insorgere dei problemi di sicurezza si potrebbe pensare all'utilizzo di un meccanismo che controlli l'operato degli agenti di servizio impedendogli di estendere i propri permessi ad altri, ma un tale meccanismo si troverebbe a dover affrontare situazioni ambigue (si pensi al caso in cui ad un agente si nega l'accesso al file system, ma si concede di aprire una finestra, se vuole scrivere qualcosa nella finestra avrà bisogno dei font adatti, come fornirglieli se non può leggerli dal disco? Si potrebbe pensare ad un estensione temporanea e specifica del set di permessi, ma una tale soluzione è molto costosa).
Inoltre, sarebbe inutile realizzare un tale meccanismo se poi l'agente di servizio fosse libero di spostarsi in un place dove, ad esempio, rivelare informazioni cui poteva accedere ma non riversare all'esterno. Si potrebbe impedire agli agenti servitori ogni ulteriore movimento, una volta che si siano registrati come agenti di servizio nel place. Per semplicità nel nostro modello controlleremo solo gli accessi diretti demandando alla politica del place la responsabilità di limitare la capacità degli agenti di servizio di comportarsi in maniera illegittima.

4.3 Implementazione

Per l'implementazione del modello ci appoggiamo sul modello di sicurezza di Java nella versione JDK1.2beta2 (vedi paragrafo 3.3.3.3).
Per adattarne l'uso nel sistema ad agenti da noi considerato è stato necessario creare nuove implementazioni per alcune delle classi che formano il cuore del sistema di sicurezza. Le implementazioni standard di alcuni elementi

basilari, quali la politica o il classloader, sono infatti specificatamente improntate alla realizzazione di applicazioni per la gestione di applet.

4.3.1 CodeSource e identificatore d'agente

Poiché nell'ambito del JDK1.2 i principal sono identificati attraverso la classe **CodeSource** è necessario mappare l'identificatore d'agente su tale classe; abbiamo scelto una configurazione di questo tipo:

"http://NomeDominio/NomePlace/NomeClasse/IDnumerico"

per la parte URL del codesource (dove NomeDominio e NomePlace si riferiscono all'origine dell'agente). Il campo delle chiavi pubbliche è invece utilizzato per memorizzare le chiavi con cui verificare le credenziali dell'agente. Quando un utente lancia un agente egli viene identificato attraverso una password, in questo modo il sistema può legittimamente dotare l'agente della credenziale del suo creatore.

4.3.2 Permessi per la configurazione del sistema

Fra i permessi per la configurazione del sistema, ossia quelli che, in generale, riguardano l'interazione dell'agente con il sistema ad agenti, troviamo:

- il permesso di ingresso in un place (**PlaceAccessPermission**);
- il permesso di creazione di agenti (**AgentCreatePermission**);
- il permesso di registrarsi come agente di servizio (**RegisterServicePermission**);
- il permesso di ottenere un oggetto depositato nell'object store (**GetPersistentObjectPermission**);

- il permesso di modificare la politica di place (**ModifyPolicyPermission**);
- il permesso di ottenere una credenziale (**GetCredentialPermission**).

Per ognuno di questi è stata creata una specifica eccezione che viene lanciata nel caso si cerchi di effettuare un accesso senza il permesso necessario.

4.3.3 La creazione dei domini di protezione

Abbiamo visto che in ogni place un agente opera in uno specifico dominio di protezione. Questi domini dovranno essere creati ed inizializzati opportunamente in modo tale che vengano loro assegnati i giusti permessi.
Per creare un dominio di protezione è necessario specificare una classe le cui istanze saranno inserite nel dominio ed un codesource che identificherà il dominio e permetterà l'attribuzione dei permessi. Quest'ultima operazione verrà eseguita una volta sola, alla creazione del dominio per il place (vedi paragrafo 3.3.3.5), dal ClassLoader incaricato della definizione della nuova classe. Solo i ClassLoader derivati dalla classe **SecureClassLoader** possono creare implicitamente nuovi domini di protezione attraverso l'invocazione del metodo **defineClass.** Per inserire un agente in un proprio dominio di protezione è necessario definire opportunamente la sua classe. Per far questo il bytecode della classe dovrà essere a disposizione del sistema sul file system locale; se così non fosse il sistema dovrà procurarsi il bytecode, gli approcci possibili sono due:

- il sistema richiede, in maniera trasparente per l'agente, la spedizione del codice al place di origine dell'agente stesso;
- il sistema sposta, insieme all'agente (o meglio al suo stato al momento della migrazione), anche il bytecode della sua classe.

La seconda soluzione viola l'ipotesi di avere agenti snelli che si muovono rapidamente sulla rete e penalizza di fatto le prestazioni del sistema, per questo motivo è stata implementata la prima strategia.
Il nuovo ClassLoader (**LoaderSicuro**) permette, come abbiamo già visto, anche il caricamento remoto di codice *by need* dal place di origine dell'agente che richieda una classe non presente.
L'algoritmo seguito da tale ClassLoader nella ricerca del bytecode necessario alla definizione delle nuove classi si può riassumere nei seguenti passi:

- si controlla se la classe è di sistema o è già caricata;
- altrimenti si cerca il bytecode in una directory specificata da una variabile propria del place;
- altrimenti si cerca nella sottodirectory "esterno" della directory precedente, in cui il sistema pone il bytecode degli agenti entranti prima di definirne la classe;
- infine si richiede al place di origine dell'agente l'invio delle informazioni necessarie.

Poiché la superclasse di LoaderSicuro non permette la definizione di due domini diversi per una stessa classe (non accetta cioè due invocazioni successive del metodo defineClass che si differenzino solo per il codesource) l'unico modo per ottenere un confinamento a livello di singoli agenti è quello di prevedere un ClassLoader per ogni agente (con un unico ClassLoader tutti gli agenti istanza di una stessa classe finirebbero nello stesso dominio di protezione).
La seconda soluzione ha un costo solo leggermente superiore a quello della prima poiché lo stato del classloader da noi realizzato è molto ridotto, di conseguenza è ridotta anche la differenza fra le due soluzioni in termini di sfruttamento delle risorse. La compresenza di diversi classloader permette, viceversa, un loro utilizzo parallelo; l'accesso a questi componenti è , infatti, sincronizzato (non possono esserne invocati gli stessi metodi dello stesso

classloader da più thread contemporaneamente) per cui un unico classloader introdurrebbe un punto di serializzazione nel caso in cui più agenti richiedessero contemporaneamente il caricamento di diverse classi.

4.3.4 Caratteristiche della politica

L'architettura del sistema di sicurezza ha richiesto l'implementazione di una nuova classe statica policy (**AgentPolicy**); questa nuova classe ha le seguenti caratteristiche:

- all'inizializzazione del place, e anche di questa classe statica, viene richiesta al gestore di dominio la spedizione della politica di default;
- successivamente viene caricato il file che specifica la politica locale e si provvede all'integrazione delle due; l'integrazione parte dal presupposto che la politica di dominio deve sovrascrivere quella di place il quale ha la facoltà di restringere a piacimento tale politica, mentre può estenderla solo rispettando i limiti imposti dalla parte negativa della politica di dominio. Per ottenere questo risultato il set di permessi positivi della politica di place saranno filtrati in modo che non contrastino con il set di permessi negativi della politica di dominio, mentre il set di permessi positivi della politica di dominio saranno analogamente filtrati dal set di permessi negativi della politica di place;
- in un apposito spazio sono memorizzate le preferenze degli agenti presenti nel place (il sistema le inserisce nella politica quando un agente entra nel place, prima di metterlo in esecuzione);
- il metodo **evaluate** calcola i permessi da associare al codesource che gli viene passato come parametro, considerando nella maniera opportuna il mappaggio visto tra identificatore d'agente e codesource, e le preferenze dell'agente cui tale codesource si riferisce;

- attraverso l'invocazione di due metodi specifici è possibile modificare dinamicamente la politica aggiungendo o rimuovendo permessi.

4.3.5 Le Credenziali

La funzione di una credenziale è quella di permettere ad un agente di ottenere un grado di fiducia superiore attraverso la certificazione di un suo precedente rapporto con un place fidato. Ogni credenziale deve essere in grado di dimostrare che l'agente ha eseguito nel place ed ha qui ricevuto un riconoscimento del grado di fiducia ad esso associato.
Per far questo le credenziali assumono la forma di firme digitali del codice dell'agente realizzate con un algoritmo di cifratura asimmetrico (vedi 1.5.4). Per questioni di semplicità abbiamo scelto l'algoritmo identificato dalla sigla DSA all'interno del provider SUN (vedi 3.3.3.2) che utilizza la funzione hash SHA-1 e l'algoritmo di cifratura DSA (vedi [SCH95]). Attraverso la verifica di una firma digitale, un place può verificare che l'agente ha effettivamente ottenuto un riconoscimento del grado concessogli da uno dei place che ha attraversato.

4.3.6 Agenti Cifrati

Per permettere agli agenti di spostarsi senza subire attacchi di eavesdropping o di tampering è possibile fare in modo che ogni agente venga cifrato prima di affrontare il viaggio verso un nuovo place. Una volta raggiunta la sua destinazione l'agente sarà decifrato e potrà riprendere la sua esecuzione. L'algoritmo scelto per la cifratura è l'RSA (vedi [SCH95]) fornito dal provider Cryptix. L'agente viene cifrato con la chiave privata associata al place di partenza; il place di destinazione dovrà dunque utilizzare, per la decifrazione, la chiave pubblica del primo. La scelta è caduta su di un

algoritmo asimmetrico poiché, attraverso un'opportuna distribuzione delle chiavi pubbliche è possibile attuare una prima semplice forma di difesa dagli attacchi di place non fidati; è sufficiente che tali place non siano in possesso delle chiavi pubbliche fidate, per assicurare che non potranno ospitare l'esecuzione di agenti provenienti da place fidati. Gli algoritmi asimmetrici sono, però, computazionalmente molto pesanti, per cui si potrebbe pensare all'utilizzo di algoritmi simmetrici che garantiscono velocità di cifratura/decifrazione molto superiori; in tal caso, i due place che vogliono instaurare un canale di comunicazione, dovranno preventivamente concordare una chiave di sessione simmetrica. Gli strumenti che consentono la realizzazione di canali sicuri come SSL (vedi…) adottano, per le ragioni viste, quest'ultima soluzione gestendo automaticamente lo scambio della chiave di sessione attraverso algoritmi asimmetrici.

4.3.7 L'utility AgentPolicyTool

Per la stesura e la modifica delle varie politiche (o meglio dei file che le rappresentano) è stato realizzato un apposito tool visuale.

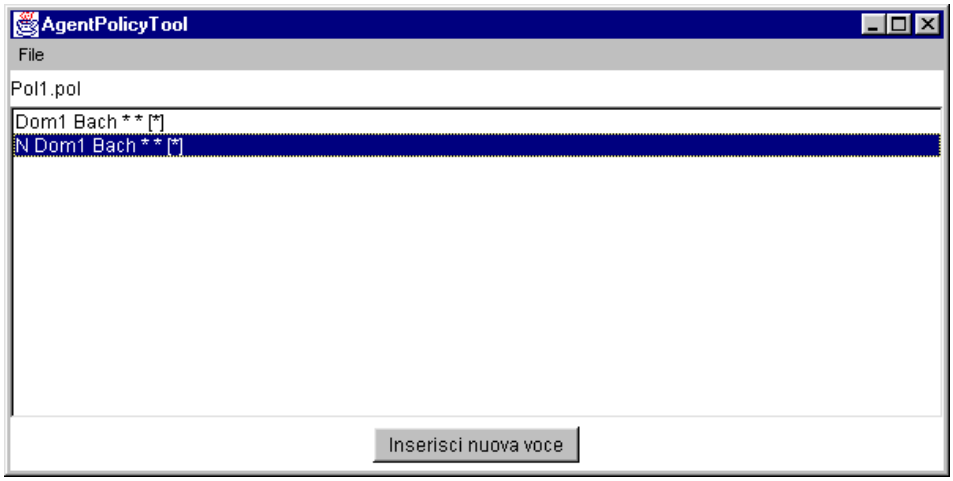

Figura 25: La finestra principale dell'utility AgentPolicyTool mostra le label associate alle voci presenti nella politica in esame.

La finestra principale (vedi Figura 25) presenta un menu a tendina che permette la scelta dell'operazione da effettuare (apertura di un file, salvataggio o uscita dal tool). Ogni politica conterrà molte voci, ognuna delle quali sarà composta da una label e da un set di permessi: la label specifica a chi tali permessi vanno concessi. Nella parte centrale della finestra principale compaiono le label delle diverse voci; si osservi che l'asterisco ("*") è considerato come wildcard in diversi contesti: se usato al posto di un nome di dominio ha il significato di "qualsiasi dominio", analogamente per un nome di place, mentre se specificato nella casella delle credenziali è da intendere come "nessuna credenziale richiesta". L'indicazione "N" prima di una label indica che la voce corrispondente è da intendersi negativa.
Una volta selezionata una voce sarà possibile modificarla (vedi Figura 26), mentre con il pulsante posto in fondo alla finestra principale sarà possibile inserire nuove voci.

Figura 26: La finestra di inserimento/modifica delle voci della politica consente una chiara visione di ogni elemento della voce in esame.

Supponiamo di voler realizzare una politica di dominio Default2 per un dominio Dom2 formato da due place di nome Escher e Godel, che contenga le seguenti voci:

- permettere a tutti gli agenti provenienti dal place Bach del dominio Dom1 di leggere e scrivere nella directory "/tmp" e di entrare nei place Escher e Godel;
- permettere a tutti gli agenti il cui utente creatore ha alias Luigi di leggere e scrivere nella directory "/privileged/luigi";
- negare a tutti gli agenti la cui classe ha nome "bad" il permesso di scrittura sulla directory "/system";

Lanciamo allora l'utility digitando:

```
java AgentSystem.AgentPolicyTool
```

inseriamo la prima voce specificando Dom1 nella casella Domain e Bach in quella Place e lasciando l'asterisco nelle altre caselle che specificano la label, infine inseriamo i permessi:

```
java.io.FilePermission  /tmp  read,write
AgentSystem.PlaceAccessPermission Escher go_in
AgentSystem.PlaceAccessPermission Godel go_in
```

nella casella al centro, e settiamo come positiva la voce (vedi Figura 27).

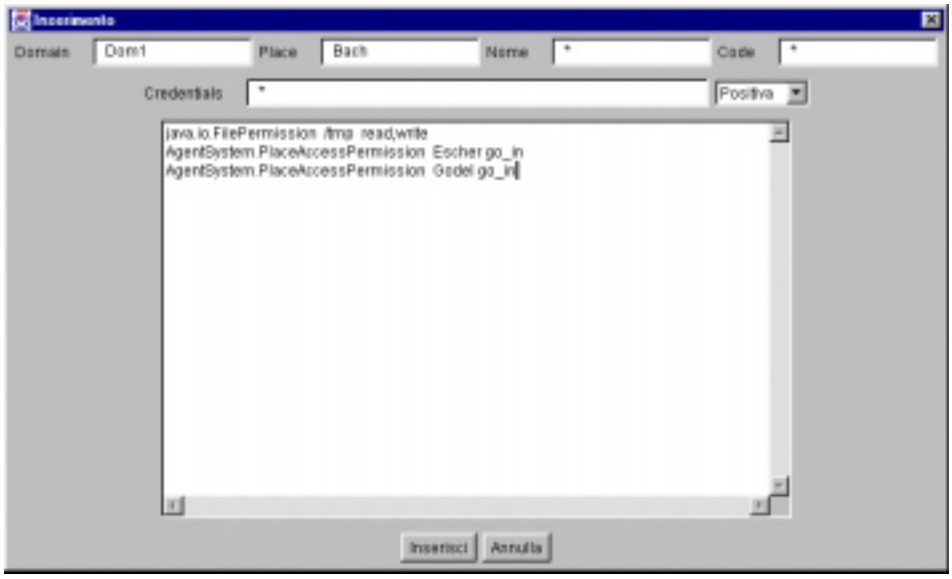

Figura 27: La prima voce specifica una serie di permessi positivi per gli agenti provenienti dal dominio Dom1 e dal place Bach.

Per una immissione corretta è sufficiente che le tre parti che compongono un permesso siano separate da uno o più spazi vuoto e che le azioni costituiscano una lista separata da virgole, senza spazi vuoti.
Per la seconda voce inseriremo l'alias Luigi nella casella Credentials ed il permesso

```
java.io.FilePermission /privileged/luigi read,write
```

in quella centrale (vedi Figura 28).

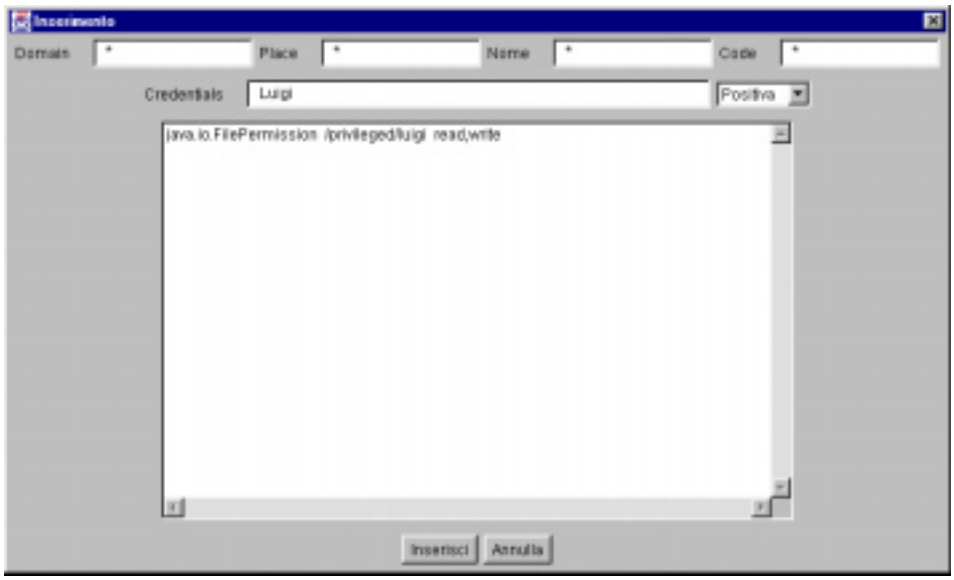

Figura 28: La seconda voce amplia il set di permessi per gli agenti di uno specifico creatore.

Infine inseriremo la terza voce indicando “bad” nella casella Nome, settando la voce come negativa e specificando il seguente permesso:

```
java.io.FilePermission /system write
```

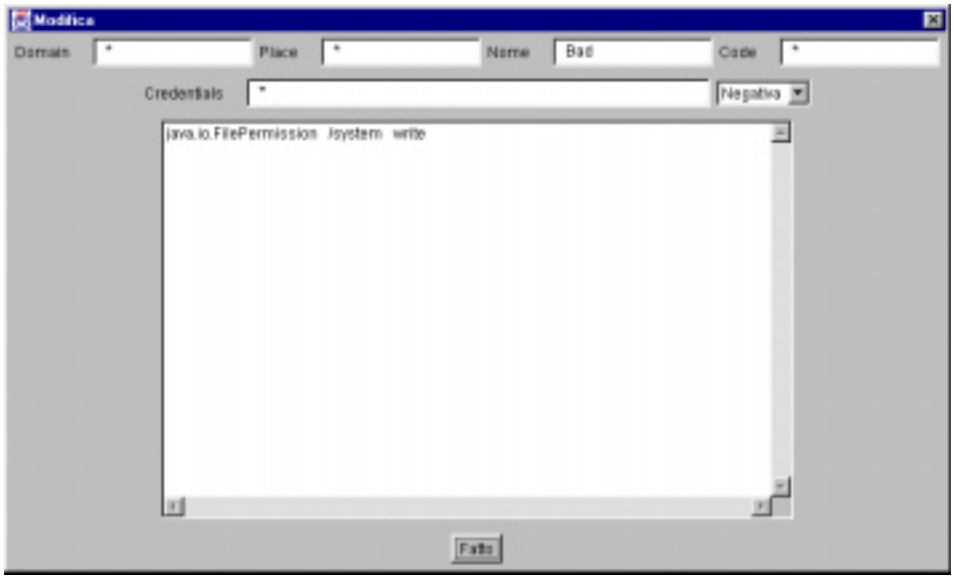

Figura 29: La terza voce limita esplicitamente i permessi per agenti di una certa classe.

Alla fine la finestra principale conterrà tre label (vedi Figura 30).

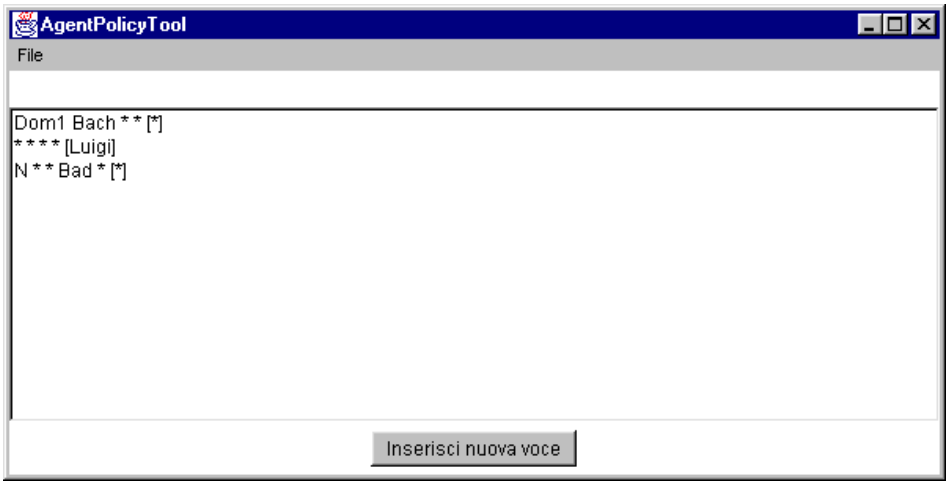

Figura 30: La finestra principale permette una visione di insieme delle voci della politica.

4.4 Valutazione dell'impatto del modello di sicurezza sulle prestazioni del sistema

L'adozione di un modello di sicurezza comporta un degradamento delle prestazioni complessive del sistema. L'intrusione dei meccanismi di sicurezza deve essere valutata per poter rapportare i benefici derivanti dal fatto di rendere il sistema più sicuro con il costo della soluzione adottata.
In uno scenario nel quale si richiede agli agenti una esecuzione snella e una grande mobilità, ci è sembrato significativo valutare essenzialmente il costo degli strumenti crittografici in relazione al tempo che un agente impiega a spostarsi tra due place.

4.4.1 Modifiche al meccanismo di mobilità introdotte dal sistema di sicurezza

L'adozione del modello di sicurezza visto ha comportato modifiche importanti nel meccanismo di mobilità degli agenti; vediamo di analizzare, un po' più in dettaglio, come gli agenti si spostano tra due place:

- il place di partenza incapsula l'agente in un oggetto TransCommand che poi spedirà al place di destinazione; TransCommand svolge un ruolo attivo nel meccanismo di trasporto, infatti, una volta giunto a destinazione, viene eseguito un suo metodo specifico (exenodo) che esegue tutte le operazioni che occorre eseguire prima che l'agente possa effettivamente riprendere l'esecuzione; si adotta, in pratica, un modello di Remote Evaluation di tipo push (vedi 2.5.1) in cui l'oggetto attivo svolge il ruolo di un articolato comando;

- l’oggetto TransCommand cifra l’agente, il suo identificatore, le sue credenziali e le sue preferenze (vedi 4.3.5 e 4.3.6) con la chiave privata del place di partenza;
- all’arrivo l’oggetto TransCommand chiede al place di destinazione la chiave pubblica del place di partenza con la quale decifra le informazioni viste;
- successivamente il TransCommand comunica alla politica locale le preferenze dell’agente;
- il TransCommand richiede l’inserimento dell’agente nel place di destinazione specificandone le credenziali;
- a questo punto il place di destinazione verifica le credenziali dell’agente, crea (se non esiste già) un dominio di protezione e ve lo inserisce.

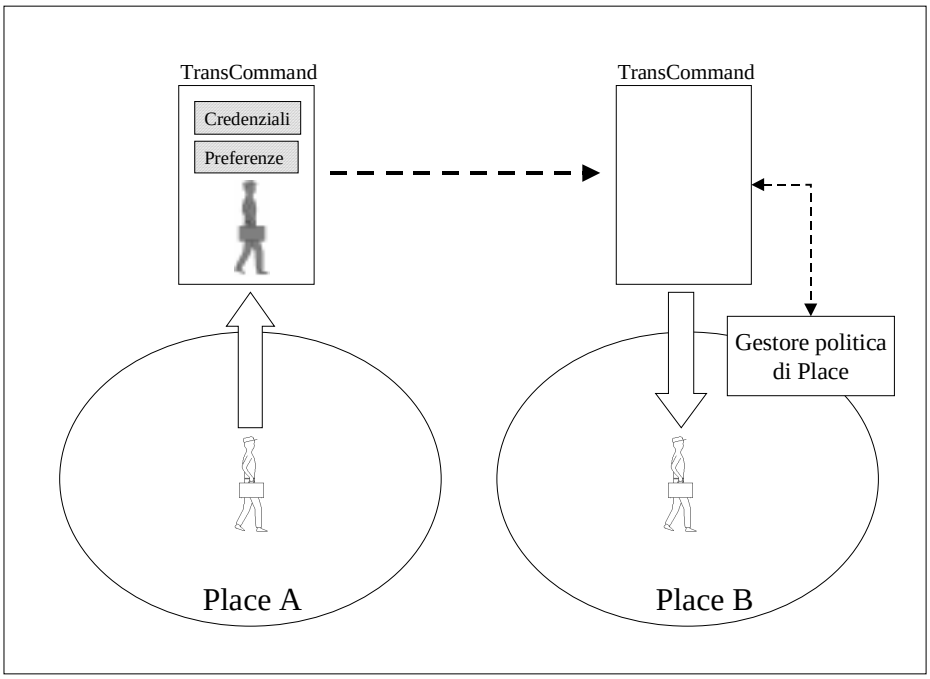

Figura 31: TransCommand cifra le informazioni che trasporta (l’agente le sue credenziali e le sue preferenze) per risolvere il problema della riservatezza.

Come vedremo, i maggiori ritardi sono dovuti alle fasi di cifratura e decifrazione e alla fase di verifica delle firme.

4.4.2 Misure effettuate

Per valutare il costo del modello di sicurezza adottato abbiamo eseguito una serie di misurazioni dei tempi necessari allo spostamento di un agente tra due place. Abbiamo analizzato sia il caso in cui i due place appartengano a due domini diversi sia quello in cui appartengano allo stesso dominio.
Le misure sono state ripetute su due architetture diverse:

- Sun Ultra1;
- PC (Pentium 166 MHz);

Abbiamo esaminato separatamente il comportamento del sistema in quattro situazioni diverse:

- a vuoto, ossia disabilitando completamente sia la verifica delle credenziali che la cifratura;
- abilitando solamente la verifica delle credenziali (*firme digitali*);
- abilitando solamente il meccanismo di cifratura/decifrazione (*riservatezza*);
- abilitando entrambi i meccanismi di cui sopra.

Le misure sono state effettuate variando la dimensione dell'agente (vedi Figura 32), e a tale proposito occorre osservare che, mentre per le operazioni di cifratura e decifrazione è lecito riferirsi alla dimensione dell'agente (o meglio alla somma fra la dimensione dell'agente e quelle del suo identificatore, delle sue credenziali e delle sue preferenze), per quanto riguarda la verifica delle credenziali bisogna far riferimento alla dimensione del codice dell'agente (vedi 4.3.5). L'algoritmo utilizzato per la cifratura degli agenti è un algoritmo di cifratura a blocchi ossia esegue la cifratura di blocchi di byte di dimensione fissata; nel nostro caso i blocchi hanno

dimensione di 128 byte per cui le dimensioni di riferimento sono, necessariamente, multipli del numero 128.

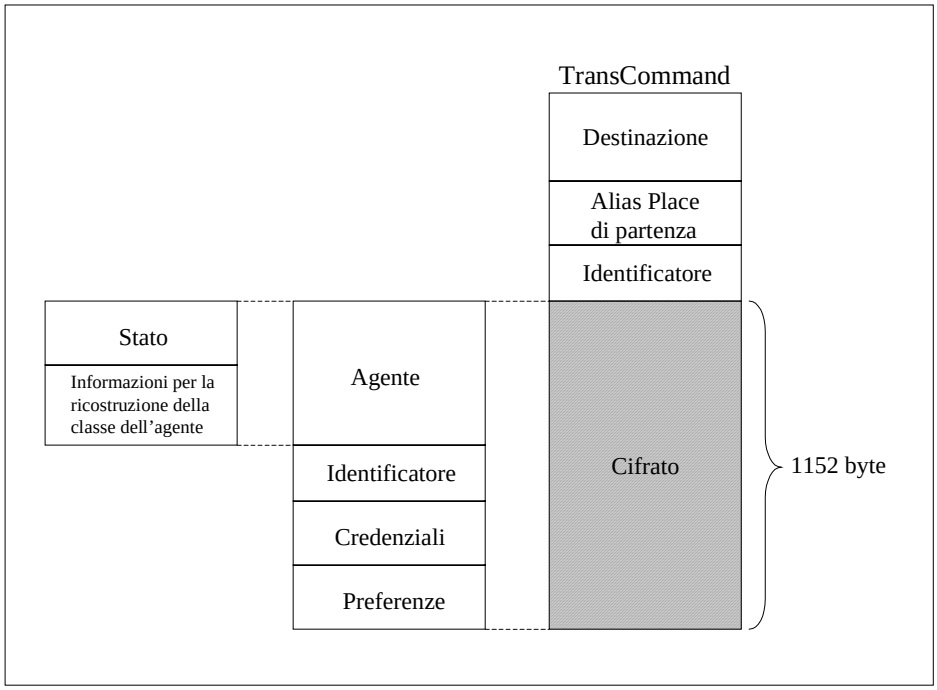

Figura 32: Le informazioni di supporto che viaggiano insieme allo stato dell'agente costituiscono una parte importante di ciò che viene cifrato, fra queste, oltre a quelle inserite dall'infrastruttura che supporto il sistema, troviamo i dati che Java inserisce nella serializzazione degli oggetti per poterne ricostruire correttamente la classe in fase di deserializzazione.

La dimensione minima per l'agente di prova utilizzato è risultata di 1152 byte senza la credenziale dell'utente creatore e di 1408 con tale credenziale, ma, per semplicità nelle tabelle si fa riferimento a una dimensione di 1 Kbyte; analogamente le dimensioni reali nel caso 10 Kbyte e 100 Kbyte sono differenti da 10 e 100, ma, in questo caso, l'errore commesso è trascurabile.
Occorre infine osservare che insieme all'agente si muovono alcune informazioni di supporto in chiaro (come la destinazione), ma la loro

presenza ha una incidenza minima sui tempi misurati, per cui possiamo tranquillamente trascurarle.

4.4.3 Valutazione dei risultati

Dai risultati riportati in Tabella 1, Tabella 2, Tabella 3 e Tabella 4 possiamo, innanzitutto osservare una netta differenza fra i tempi relativi al trasferimento tra place dello stesso dominio e quelli relativi al caso di place in domini diversi. Questo divario è dovuto al fatto che, per attraversare il confine tra due domini, un agente deve attraversare due gateway, inoltre le connessione tra place e gateway e tra i due gateway non sono permanenti, ma vengono create al bisogno: nel calcolo del tempo di transito tra place di domini differenti, bisogna quindi includere anche il tempo di creazione di tre connessioni. Nel caso di place dello stesso dominio, invece, la connessione viene mantenuta fino a che uno dei due place non interrompa la sua esecuzione. Spostandosi verticalmente lungo le colonne delle tabelle riportate si rileva come l'abilitazione del meccanismo di verifica delle credenziali, prima, e di quello di cifratura, poi, facciano aumentare complessivamente i tempi di quasi due ordini di grandezza.

	1KB	**10KB**	**100KB**
A VUOTO	280 ms	407 ms	755 ms
CREDENZIALI	489 ms	600 ms	982 ms
CIFRATURA	1621 ms	10323 ms	101218 ms
CREDENZIALI+CIFRATURA	2129 ms	10810 ms	102867 ms

Tabella 1: Tempi di spostamento tra place di domini diversi su architettura Sun Ultra1.

	1KB	10KB	100KB
A VUOTO	66 ms	147 ms	416 ms
CREDENZIALI	438 ms	520 ms	823 ms
CIFRATURA	1356 ms	9396 ms	92869 ms
CREDENZIALI+CIFRATURA	1989 ms	9976 ms	94261 ms

Tabella 2 : Tempi di spostamento tra place dello stesso dominio su architettura Sun Ultra1.

La lentezza introdotta dai meccanismi crittografici adottati è da imputare essenzialmente al fatto che gli algoritmi sono implementati in Java invece che in codice nativo, ma anche una scelta più accurata degli algoritmi porterebbe a prestazioni migliori. Le misurazioni effettuate hanno infatti un carattere di verifica e non inseguono l'obiettivo di valutare le effettive potenzialità del sistema. Un notevole miglioramento potrebbe essere raggiunto utilizzando provider più veloci per la gestione delle credenziali e un pacchetto SSL per la protezione dei canali di comunicazione (vedi [FKK96]).

	1KB	10KB	100KB
A VUOTO	375 ms	425 ms	943 ms
CREDENZIALI	716 ms	795 ms	1290 ms
CIFRATURA	2694 ms	19563 ms	211385 ms
CREDENZIALI+CIFRATURA	3603 ms	20468 ms	212930 ms

Tabella 3: Tempi di spostamento tra place di domini diversi su architettura Pentium 166MHz.

Le tabelle mostrano che la fase più pesante è senza dubbio quella di cifratura/decifrazione; all'aumentare della dimensione dell'agente, infatti, le differenze fra i tempi relativi al caso di place nello stesso dominio e quelli

relativi al caso di place in domini diversi diminuiscono notevolmente, poiché il tempo speso dai meccanismi crittografici prende il sopravvento sugli altri.

	1KB	**10KB**	**100KB**
A VUOTO	109 ms	177 ms	565 ms
CREDENZIALI	749 ms	788 ms	1175 ms
CIFRATURA	2551 ms	19288 ms	208740 ms
CREDENZIALI+CIFRATURA	3792 ms	20495 ms	209635 ms

Tabella 4: Tempi di spostamento tra place dello stesso dominio su architettura Pentium 166MHz.

Osserviamo infine che l'apparente discrepanza dei risultati riguardanti le prove eseguite su architettura Pentium (dai risultati riportati nell'ultima colonna risulta che la migrazione tra place dello stesso dominio richiede un tempo superiore al caso di migrazione fra place di domini diversi) è dovuta al fatto che, in questo caso, le prove sono state effettuate allocando i due place dello stesso dominio sulla stessa macchina; il tipo di scheduling effettuato dal sistema operativo (Windows 95) è la causa di tali stranezze.

Conclusioni

Il modello di programmazione ad Agenti Mobili sta diventando un potente strumento per la realizzazione di applicazioni distribuite ad alto grado di scalabilità e flessibilità. Il linguaggio Java, grazie all'indipendenza del codice dall'architettura, fornisce un ottima base di partenza per la realizzazione di una infrastruttura per l'esecuzione di applicazioni realizzate secondo il modello ad agenti mobili. Il modello di sicurezza del JDK1.2 beta2 include già alcune funzionalità indispensabili per l'implementazione di un modello di sicurezza per un sistema ad agenti mobili. Essendo nato nell'intento di regolare il comportamento delle applet di Java, il modello di sicurezza del JDK1.2 beta2 necessita di alcune modifiche ed estensioni, ma gli strumenti basilari forniti a livello di linguaggio permettono di affrontare tali modifiche in maniera fattibile. La stessa cosa utilizzando il modello precedente (JDK1.1) sarebbe risultata molto ardua.

Per quanto riguarda le astrazioni, la suddivisione della rete in Place e Domini ha condotto ad una naturale suddivisione delle responsabilità inerenti la politica di sicurezza da adottare in ogni determinato contesto: questo ha permesso una stesura delle politiche di sicurezza intuitiva e rapida. L'adozione di modelli di autenticazione ed autorizzazione flessibili ha consentito una precisa caratterizzazione dei gradi di fiducia associabili ad ogni agente ed un controllo approfondito e flessibile nell'accesso ai vari tipi di risorsa.

Il lavoro svolto dimostra che gli strumenti per la realizzazione di un modello di sicurezza affidabile ed allo stesso tempo abbastanza facile da gestire sono ormai a disposizione dei sistemi ad agenti mobili. Le misure effettuate evidenziano come, l'inevitabile introduzione del concetto di sicurezza in uno di tali sistemi comporta un degrado delle prestazioni accettabile considerando i vantaggi derivati dal nuovo modello di programmazione.

Il modello di sicurezza sviluppato è abbastanza flessibile da permettere una facile integrazione con le inevitabili estensioni che il modello ad agenti mobili potrà subire con l'introduzione, ad esempio, di nuovi strumenti di comunicazione. Ulteriori sforzi vanno fatti per lo studio e la caratterizzazione delle politiche di sicurezza; la realizzazione di librerie di politiche standard e di strumenti di più alto livello per la realizzazione e la gestione delle politiche di sicurezza sono elementi indispensabili nell'ottica di una larga diffusione del modello ad agenti mobili.

5 Appendici

5.1 Bibliografia

[BHR+97] Joachim Baumann, Fritz Hohl, Nikolaos Radouniklis, Kurt Rothermel, and Markus Straßer. “Communication concepts for mobile agent systems” In Proceedings of the First International Workshop on Mobile Agents, Berlin, Germany, April 1997.

[BR94] K. P. Birman, R. van Renesse. “Reliable Distributed Computing with the ISIS toolkit”. IEEE Computer Society Press, 1994.

[CAR97] Luca Cardelli.“Mobile computations” In Mobile Object Systems: Towards the Programmable Internet, pages 3-6. Springer-Verlag, April 1997. Lecture Notes in Computer Science No.1222.

[CG89] N. Carriero, D. Gelernter. “Linda in context”. CACM 32(4), Aprile 1989.

[CHK97] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. “Mobile agents: Are they a good idea?”…update. In Mobile Object Systems: Towards the Programmable Internet, pages56-48, Springer-Verlag, April 1997. Lecture Notes in Computer Science No.1222.

[DKS+93] The DARPA Knowledge Sharing Initiative External Interfaces Working Group: Tim Finin (University of Maryland), Jay Weber (Enterprise Integration Technology), Gio Wiederhold (Stanford University), Michael (Standford University), Richard Fritzon & Donald McKay (Paramax System), James McGuire & Richard Pelavin (Lockheed AI Center), Stuart Shapiro (SUNY Buffalo), Chris Beck (University of Toronto. DRAFT “ Specification of the KQML Agent-Communication Language” . June 15, 1993.

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

[FPV96] Alfonso Fugetta, Gian Pietro Picco, Giovanni Vigna. “Understanding code mobility”. IEEE Transaction on Software Engineering, to appear.

[GB97] Robert Grimm and Brian N. Bershad. “Access control in extensible systems”. Technical report, Department of Computer Science, University of Washington, March 1997.

[GCK+96] Robert Gray, Geoge Cybenko, David Kotz, Daniela Rus. “Agent Tcl”. Departmet of Cumputer Science, Dartmounth College, Hanover NH, 29 May 1996.

[OLW96] J.K.Ousterhout, J.Y.Levy, B.B.Welch. “The Safe-Tcl security model”. Sun Microsystem Laboratories, Mountain View, CA, USA

[PEI97] Holger Peine. “An Introduction to Mobile Programming and the Ara System”. Department of Computer Science, Kaiserslautern University, Germany. ZRI-Report 1/97.1

[RAS+97] Mudumbai Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. “Network-aware mobile programs” In Proceedings of the USENIX 1997 Annual Technical Conference, Anaheim, Cal., January 1997.

[SCF+96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein and Charles E. Youman. “Role-Based Access Control Models”. In IEEE Computer, Volume 29, Number 2, February 1996, pages 38-47.

[SCH95] Bruce Schneider. “Applied Cryptography” J. Wiley & Sons, 1995.

[SG90] James W. Stamos and David K. Gifford. “Remote evaluation”. ACM Transactions on Programming Languages and Systems, 12(4):537-565, Ottobre 1990.

[SG94] Abraham Silberschatz, Peter B. Galvin. “Operating System Concepts” . Addison Wesley 1994.

[VIG97] Giovanni Vigna. “Protecting Mobile Agents through Tracing”. Dip. Elettronica e Informazione, Politecnico di Milano. In Proceeding of the third Workshop on Mobile Object Systems, Finland, June 1997.

[VST97] Jan Vitek, Manuel Serrano e Dimitri Thanos. “Security and communication in Mobile Object Systems.” In Mobile Object Systems: Towards the Programmable Internet. Springer-Verlag, April 1997. Lecture Notes in Computer Science No.1222

[WHI94] Jim White. “Telescript Tecnology: The Foundation for Electronic Marketplace”. General Magic Inc. Mountain View (CA), USA , 1994.

[WLA+93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. “Efficient software-based fault isolation.” In Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 203-216, 1993.

[WWW+94] Jim Waldo, Geoff Wyant, Ann Wollrath, e Sam Kendal. “A note on distributed computing”. In Mobile Object Systems: Towards the Programmable Internet, pag. 49-64. Springer verlag, April 97. Lecture Notes in Computer Science No. 1222. Publicato anche come Sun Microsystems Laborators Technical Report TR-94-29.

5.2 Le chiamate principali del sistema

Di seguito riportiamo alcune delle chiamate a disposizione degli agenti ed altre che, pur riferendosi a membri o classi non pubbliche, permettono un analisi più approfondita del sistema.

5.2.1 La classe Agent

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

---

public abstract class **Agent**

extends Object

implements Serializable

Questa classe non e' derivata dalla classe Thread, e' semplicemente un oggetto passivo che viene eseguito dai workers. Un Agente puo' essere creato not Traceable, incapace di ricevere posta ma anche di essere ritrovato (non si tiene traccia della posizione) questa funzionalita' dovrebbe essere permessa solo per Agenti di Sistema che devono svolgere piccoli compiti di servizio.

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

Constructor Index

- **Agent**()

Method Index

- **addCredential**(Credential)
- **addPref**(GrantEntry)

  Aggiunge l'entry specificata tra le preferenze dell'agente

- **addPreferenze**(GrantEntry)
- **getID**()

  Restituisce l'ID dell'agente

- **giveMeCredential**()

  Se l'agente e' autorizzato per questa operazione, questo metodo aggiunge alle credenziali dell'agente quella del Place in cui si trova

- **go**(DomainName, String)

  Muove Agente in un altro Dominio.

- **go**(PlaceName, String)

  Muove l'agente specificato in un place dello stesso Dominio.

- **putArg**(Object)
- **run**()
- **setTraceable**(boolean)

Fields

Start

```
public String Start
```

Traceable

```
public boolean Traceable
```

Mail

```
public Mailbox Mail
```

preferenze

```
public ArrayList preferenze
```

Constructors

Agent

```
public Agent()
```

Methods

setTraceable

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

getID

```
public AgentID getID()
```

Restituisce l'ID dell'agente

putArg

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

run

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

setStart

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

go

```
public final void go(PlaceName p,
                     String metodo) throws CantGoException
```

Muove l'agente specificato in un place dello stesso Dominio. p e' il place di destinazione metodo e' il metodo che verra' eseguito una volta arrivato a destinazione

go

```
public final void go(DomainName d,
                     String metodo) throws CantGoException
```

Muove Agente in un altro Dominio. d e' il dominio di destinazione metodo e' il metodo che verra' eseguito una volta arrivato a destinazione

- **addPref**

```
public final void addPref(GrantEntry ge)
```

Aggiunge l'entry specificata tra le preferenze dell'agente

- **giveMeCredential**

```
public final void giveMeCredential()
```

Se l'agente e' autorizzato per questa operazione, questo metodo aggiunge alle credenziali dell'agente quella del Place in cui si trova

- **addCredential**

```
void addCredential(Credential cr)
```

- **addPreferenze**

```
void addPreferenze(GrantEntry ge)
```

5.2.2 La classe AgentID

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

---

public class **AgentID**

extends ObjectID

implements Serializable

Identificatore di agente ... caratterizzato dal nome del nodo di origine piu' il nome del dominio (utile per la gestione della consistenza della posizione)

Constructor Index

- **AgentID**(String, String, String, int)

L'ID di un agente e' formato da:

- code: codice identificativo unico all'interno del place di nascita
- Place: nome del place di origine
- Domain: nome del dominio
- Classe: nome della classe che implementa l'agente

Method Index

getAgentID(CodeSource)

Dato il suo CoseSource cs ritorna l'AgentID di un agente

getClasse()

Ritorna il nome della classe dell'agente

getCodeSource()

Ritorna il CodeSource dell'agente

getDomain()

Ritorna il dominio di origine dell'agente

getPlace()

Ritorna il nome del place di origine dell'agente

toString()

uguale(AgentID)

Verifica se 2 AgentID sono uguali

Constructors

AgentID

```
public AgentID(String domain,
               String place,
               String classe,
               int code)
```

L'ID di un agente e' formato da:

- code: codice identificativo unico all'interno del place di nascita

- Place: nome del place di origine
- Domain: nome del dominio
- Classe: nome della classe che implementa l'agente

Methods

getPlace

```
public String getPlace()
```

Ritorna il nome del place di origine dell'agente

getDomain

```
public String getDomain()
```

Ritorna il dominio di origine dell'agente

getClasse

```
public String getClasse()
```

Ritorna il nome della classe dell'agente

uguale

```
public boolean uguale(AgentID agid)
```

Verifica se 2 AgentID sono uguali

getAgentID

```
public static AgentID getAgentID(CodeSource cs)
```

Dato il suo CoseSource cs ritorna l'AgentID di un agente

getCodeSource

```
public CodeSource getCodeSource()
```

Ritorna il CodeSource dell'agente

toString

```
public String toString()
```

Overrides:

toString in class Object

5.2.3 La classe AgentSystem

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

---

public final class **AgentSystem**

extends Object

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

Field Index

- **Monitor**

Constructor Index

- **AgentSystem**()

Method Index

- **addPref**(AgentID, GrantEntry)
- **createAgent**(String, Object)

  Crea nuovo agente dalla classe cl ad associa args obj Restituisce AgentID dell'Agente
- **createAgent**(String, Object, boolean)

  Crea nuovo agente dalla classe cl ad associa args obj Restituisce AgentID dell'Agente potendo specificare se e' o no Traceable.
- **getCurrentLocation**()

  Ritorna la attuale posizione dell'Agente.
- **getNotTraceableAgentsNumber**()
- **getPersistentObject**(ObjectID)

Cerca nel DB oggetto persistente

- **getPlaceNumber**()
- **getTraceableAgentsNumber**()
- **giveCredential**(AgentID)
- **go**(AgentID, DomainName, String)

  Muove Agente identificato da AgentID aid.
- **go**(AgentID, PlaceName, String)

  Muove l'agente specificato in un place dello stesso dominio
- **isActive**(PlaceName)

  Verifica se una specifico nodo e' attivo.
- **kill**(AgentID)

  Uccide Agente caratterizzato da AgentID
- **removePersistentObject**(AgentID, ObjectID)

  Rimuove oggetto persistente dal DB
- **storePersistentObject**(AgentID, Object)

  Salva nel DB oggetto che deve rimanere persistente

Fields

Monitor

```
static Monitor Monitor
```

Constructors

AgentSystem

```
public AgentSystem()
```

Methods

go

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

Muove l'agente specificato in un place dello stesso dominio

go

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

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

kill

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

Uccide Agente caratterizzato da AgentID

createAgent

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

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

createAgent

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

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

storePersistentObject

```
public static ObjectID storePersistentObject(AgentID aid,
                                             Object o)
```

Salva nel DB oggetto che deve rimanere persistente

getPersistentObject

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

Cerca nel DB oggetto persistente

removePersistentObject

```
public static void removePersistentObject(AgentID aid,
                                          ObjectID oid)
```

Rimuove oggetto persistente dal DB

getCurrentLocation

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

Ritorna la attuale posizione dell'Agente.

isActive

```
public static boolean isActive(PlaceName p)
```

Verifica se una specifico nodo e' attivo.

getNotTraceableAgentsNumber

```
public static int getNotTraceableAgentsNumber()
```

getTraceableAgentsNumber

```
public static int getTraceableAgentsNumber()
```

getPlaceNumber

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

addPref

```
static void addPref(AgentID agid,
                    GrantEntry ge)
```

giveCredential

```
static void giveCredential(AgentID agid)
```

5.2.4 La classe AgentPolicy

```
java.lang.Object
    |
    +----java.security.Policy
            |
            +----AgentSystem.AgentPolicy
```

---

public class **AgentPolicy**

extends Policy

Field Index

- **agid_pk**
- **available_dp**
- **dp**
- **ks**
- **neg_voce**
- **nnvoci**
- **NomeFile**
- **nvoci**
- **permessi**
- **pref**
- **sem**
- **voce**

Constructor Index

- **AgentPolicy**()

Method Index

- **aggiungi**(PolicyEntry[])
- **evaluate**(CodeSource)

  evaluate restituisce la collezione di permessi associati ad un determinato codesource.

- **getAgid_pk**(AgentID)
- **giveCredential**(AgentID)

  Aggiunge all'agente di identificatore agid la credenziale del place

init(String, String)

Inizializza la politica:

- nomefile e' il nome del file che contiene la politica di place
- nomeKeystore e' il nome del file che memorizza il keystore delle chiavi utilizzate per il sistema delle credenziali

intersezione(Permissions, ArrayList)

pulisci(int, ArrayList)

refresh()

removeAgid_pk(AgentID)

removePref(CodeSource)

Rimuove dalla politica le prefernze dell'agente il cui codesource e' pref_cs.

setAgid_pk(AgentID, PublicKey[])

setPref(CodeSource, ArrayList)

Inserisce nella politica le preferenze pref_perm dell'agente il cui codesource e' pref_cs.

show()

show_voci()

togli(PolicyEntry)

Fields

voce

```
static PolicyEntry[] voce
```

permessi

```
static permessiEntry[] permessi
```

nvoci

```
static int nvoci
```

NomeFile

```
static String NomeFile
```
ks

```
public static KeyStore ks
```
neg_voce

```
static PolicyEntry[] neg_voce
```
nnvoci

```
static int nnvoci
```
dp

```
static boolean dp
```
available_dp

```
static boolean available_dp
```
sem

```
public static semaforo sem
```
pref

```
static Hashtable pref
```
agid_pk

```
static Hashtable agid_pk
```

Constructors

AgentPolicy

```
public AgentPolicy()
```

Methods

init

```
public static void init(String nomefile,
                        String nomeKeystore)
```

Inizializza la politica:

- nomefile e' il nome del file che contiene la politica di place
- nomeKeystore e' il nome del file che memorizza il keystore delle chiavi

utilizzate per il sistema delle credenziali

evaluate

```
public Permissions evaluate(CodeSource cs)
```

evaluate restituisce la collezione di permessi associati ad un determinato codesource. Se cs fa match con piu' voci della politica tutti i permessi delle varie voci vengono attribuiti.

Overrides:

evaluate in class Policy

refresh

```
public void refresh()
```

Overrides:

refresh in class Policy

aggiungi

```
static void aggiungi(PolicyEntry[] source)
```

togli

```
static void togli(PolicyEntry pe)
```

pulisci

```
static void pulisci(int nv,
                    ArrayList bad)
```

setPref

```
public static void setPref(CodeSource pref_cs,
                           ArrayList pref_perm)
```

Inserisce nella politica le preferenze pref_perm dell'agente il cui codesource e' pref_cs.

removePref

```
public static void removePref(CodeSource pref_cs)
```

Rimuove dalla politica le prefernze dell'agente il cui codesource e' pref_cs.

setAgid_pk

```
public static void setAgid_pk(AgentID agid,
                              PublicKey[] pk)
```

removeAgid_pk

```
public static void removeAgid_pk(AgentID agid)
```

getAgid_pk

```
public static PublicKey[] getAgid_pk(AgentID agid)
```

intersezione

```
public static Permissions intersezione(Permissions a1,
                                       ArrayList a2)
```

giveCredential

```
public static void giveCredential(AgentID agid)
```

Aggiunge all'agente di identificatore agid la credenziale del place

show

```
public static void show()
```

show_voci

```
public static void show_voci()
```

5.2.5 La classe TransCommand

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

---

class **TransCommand**

extends Command Classe comando, per GateWay ... trasporta qualsiasi cosa

---

Field Index

- **Agente**
- **AgID**

Constructor Index

TransCommand(Object, String, String, boolean)

Parametri:

- o e' l'oggetto trasportato, agente o comando
- d dominio di destinazione
- p place di destinazione
- a true se trasporto un agente

Method Index

exe()

E' il metodo che viene eseguito ad ogni passo per stabilire il prossimo passo, ossia per effettuare il routing.

exenodo()

E' il metodo che viene eseguito una volta che il TransCommand raggiunge il place di destinazione

Returned()

unsetGate()

Fields

Agente

```
boolean Agente
```

AgID

```
AgentID AgID
```

Constructors

TransCommand

```
TransCommand(Object o,
```

```
            String d,
            String p,
            boolean a)
```

Parametri:

- o e' l'oggetto trasportato, agente o comando
- d dominio di destinazione
- p place di destinazione
- a true se trasporto un agente

Methods

exe

```
public void exe()
```

E' il metodo che viene eseguito ad ogni passo per stabilire il prossimo passo, ossia per effettuare il routing.

Overrides:

exe in class Command

exenodo

```
void exenodo()
```

E' il metodo che viene eseguito una volta che il TransCommand raggiunge il place di destinazione

Returned

```
public void Returned()
```

Overrides:

Returned in class Command

unsetGate

```
void unsetGate()
```