INDICE

INTRODUZIONE

Le reti di calcolatori stanno riscuotendo un crescente interesse negli ultimi tempi soprattutto per merito di Internet e del Web. Cominciano ad essere disponibili applicazioni distribuite su larga scala e si sollevano importanti interessi economici, indirizzati ad esempio, verso il commercio elettronico, inteso come la possibilità di effettuare acquisti in linea tramite una rete di calcolatori. La sicurezza informatica diventa una condizione critica ed imprescindibile per il reale utilizzo di questi nuovi servizi.

Il forte incremento del numero di utenti interessati all'ampia disponibilità di servizi presenti e le risorse limitate del mezzo trasmissivo tendono a condurre verso la saturazione del traffico di rete. In questo contesto, le più complesse applicazioni distribuite mettono in evidenza i limiti di scalabilità e di flessibilità del classico modello di progettazione cliente/servitore.

L'attenzione del mondo della ricerca sui sistemi distribuiti si è rivolta dunque verso nuovi modelli, capaci di superare questi problemi: si è introdotto il concetto di mobilità del codice e tra i paradigmi di progettazione a codice mobile particolare interesse ha suscitato quello ad Agenti Mobili. Un agente mobile è un'entità computazionale che agisce per conto di chi lo ha messo in esecuzione, che possiede un proprio flusso di esecuzione e che ha la capacità di spostare se stesso, durante la sua esecuzione, da una locazione ad un'altra in una rete eterogenea. L'agente può muoversi autonomamente e spostarsi sui nodi che contengono le risorse con cui deve interagire per portare a termine il compito che assegnato.

Il modello ad agenti mobili può eseguire servizi in modo più flessibile ed efficiente rispetto al classico paradigma cliente/servitore, ma pone un particolare accento sui problemi di sicurezza. Infatti, espone il codice ai pericoli di un canale di comunicazione insicuro ed introduce la possibilità di mettere in esecuzione del codice non conosciuto, su un qualunque host, senza il controllo diretto dell'utente. Risulta necessario realizzare un framework di supporto per gli agenti mobili, in grado di offrire un adeguato livello di protezione delle risorse in caso di attacchi da parte di agenti non fidati, ma anche di garantire agli agenti di poter eseguire senza temere manomissioni o comportamenti illeciti da parte dell'ambiente ospitante.

Nel I capitolo, si affronta il problema sicurezza nella sua generalità ricordando i termini di maggior rilievo ed i meccanismi fondamentali. Si considera l'evoluzione della crittografia ed i suoi principali algoritmi, mettendone in rilievo debolezze e punti di forza. Nell'ultima parte si introducono i sistemi distribuiti nell'ottica di evidenziarne gli aspetti legati alla sicurezza.

Nel II capitolo, si introduce il concetto di mobilità del codice e si definiscono i vari paradigmi di progettazione a codice mobile. Si approfondisce il modello ad agenti mobili, delineando le caratteristiche chiave che un sistema realizzato con tale paradigma, deve possedere.

Nel III capitolo, si elencano le caratteristiche auspicabili in un linguaggio per la realizzazione di applicazioni sicure; si considera come riferimento il linguaggio Java, e se ne esaminano i costrutti ed i modelli di sicurezza. Infine si evidenziano i criteri che hanno portato all'implementazione degli algoritmi crittografici in Java.

Il IV capitolo descrive un supporto sicuro per lo sviluppo di applicazioni ad agenti: in particolare, si studiano i dettagli implementativi del modello di sicurezza del sistema SOMA, sviluppato presso il D.E.I.S., e già introdotto, nelle sue caratteristiche fondamentali, nel II capitolo. Il problema della

sicurezza distingue i due aspetti della protezione dell'ambiente e della protezione dell'agente. Un'attenzione particolare è dedicata all'analisi dei problemi connessi all'integrazione in SOMA di una adeguata infrastruttura per la gestione dei certificati a chiave pubblica (Public Key Infrastructure).

Cap.1 Modelli di sicurezza

1.1 Che cosa si intende per sicurezza

La definizione di sicurezza tratta dal dizionario Giunti [GIUNTI97] riporta: "*certezza, piena attendibilità, prevenzione, eliminazione parziale o totale di danni, pericoli e rischi ".* La sicurezza è una condizione da raggiungere e mantenere con strategie e strumenti anche molto diversi tra loro che dipendono dal particolare contesto considerato. Per esempio in ambiente militare, sicurezza può voler dire difesa da assalti di truppe nemiche, sospetto e diffidenza nei confronti di informatori, custodia di piani segreti, mantenimento di strutture belliche, costruzione di armi sempre più potenti. Invece in campo economico, potrebbe equivalere a salvaguardia di beni preziosi ed uso di brevetti. In ambito informatico, potrebbe esprimere la non manomissione di un calcolatore, la trasmissione protetta di dati, ecc. Dietro al termine sicurezza si celano diverse problematiche comuni a tutti i contesti: è necessario individuarle singolarmente, affrontarle e trovare soluzioni specifiche che garantiscano, nella loro globalità, il raggiungimento di un adeguato grado di sicurezza.

Per comprendere meglio il problema sicurezza, è necessario individuare sottoproblemi di più facile gestione (Figura 1-1). Strettamente correlati al termine sicurezza, troviamo i concetti di *salvaguardia* delle informazioni, e di *controllo* sull'utilizzo delle risorse.

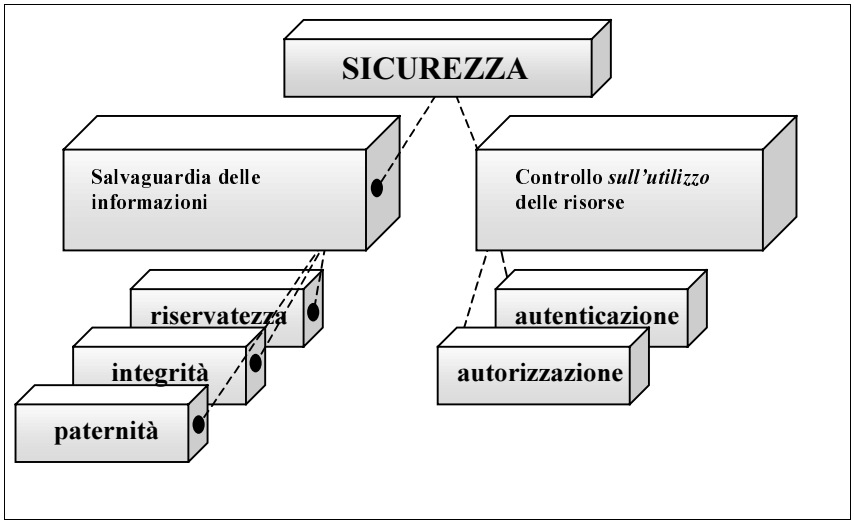

Figura 1-1 I problemi della sicurezza

Il concetto di *salvaguardia* delle informazioni evidenzia i seguenti problemi: impedire la diffusione non autorizzata di informazioni (riservatezza); garantire la non manomissione delle stesse (integrità); stabilire con assoluta certezza l'origine delle informazioni (paternità). Il secondo aspetto, il controllo sull'utilizzo *delle risorse,* è connesso a problemi di "autenticazione" e di "autorizzazione". In contesto informatico per risorse intendiamo, quelle di esecuzione (codice, threads, ecc.), di memorizzazione (file system) e di comunicazione (stream, socket, ecc.). L'autenticazione consente al sistema (informatico) di riconoscere un qualunque utente (o entità di esecuzione). L'autorizzazione comporta la necessità di attribuire ad un'identità un certo grado di fiducia ossia stabilire "chi" può fare "cosa".

Data l'importanza di questi termini nella prosecuzione del lavoro, ne riportiamo ora una descrizione più dettagliata.

1.1.1 Riservatezza

Le informazioni trasmesse su un canale qualunque (posta tradizionale, linea telefonica, rete di calcolatori) si dicono riservate se e solo se si garantisce che siano intellegibili esclusivamente dalle parti volutamente coinvolte.

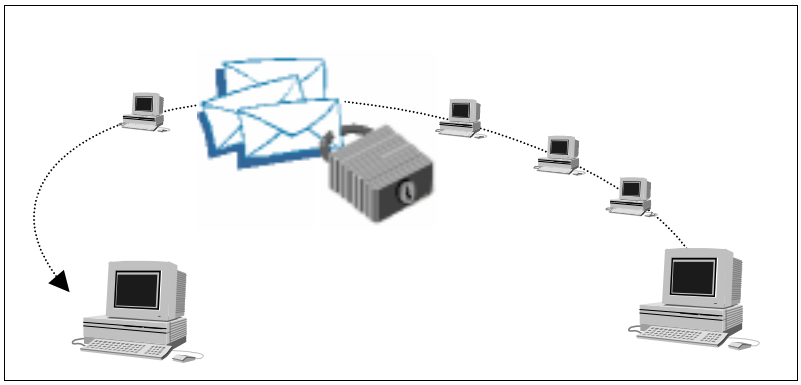

Figura 1-2 Comunicazione riservata

Per esempio il messaggio contenente l'ordine di attaccare Waterloo che Napoleone invia al suo Generale, non deve essere comprensibile da nessuno fuorché dai due interessati. Questo risultato può essere ottenuto mediante l'uso di inchiostro dalla composizione chimica particolare, tale da risultare normalmente trasparente, visibile solo per reazione con altre sostanze o per assorbimento di calore.

Considerando invece una rete telematica, riservatezza significa che le informazioni scambiate tra due nodi, anche molto distanti tra loro, devono essere comprensibili solamente ad essi. Così tutti i nodi intermedi coinvolti nella comunicazione, anche se si appropriassero dei messaggi transitanti, non riuscirebbero ad ottenere nulla da essi. Per risolvere questo problema si usano generalmente le tecniche della crittografia moderna (che affronteremo nei paragrafi successivi), in grado di fornire una sorta

di “lucchetto” elettronico che è possibile apribile con apposite “chiavi” digitali (Figura 1-2).

1.1.2 Integrità

Le informazioni ricevute da un qualunque canale si dicono integre se vi è la garanzia che nessuno ne abbia alterato il contenuto.
Per esempio se Napoleone invia un messaggio con l’ordine di attaccare Waterloo, il suo Generale deve ricevere il messaggio contenente proprio quell’ordine e non un invito alla ritirata.

In campo informatico, un messaggio si dice integro se durante una trasmissione, non subisce alcuna alterazione. Si pensi all’esempio dell’*offerta:* Bob ha intenzione di acquistare una specifica autovettura. Si collega ad Internet e chiede il preventivo a due autosaloni. L’autosalone “*Cheap*” risponde con la proposta più bassa: 10 milioni. L’altro autosalone “*Expensive”,* invia la propria offerta di 15 milioni. Ma “*Expencive*” riesce ad intercettare il messaggio dell’autosalone “*Cheap*”. Allora senza scrupoli lo manomette sostituendo 10 con 20 milioni. Bob dunque riceve entrambe le proposte e decide di rivolgersi all’autosalone “*Expensive*”, concludendo l’affare peggiore.

1.1.3 Paternità

In qualunque momento deve essere possibile determinare l’autore di una informazione senza che questi abbia alcuna possibilità di ripudiarla. Per esempio Napoleone non può negare di aver trasmesso il messaggio contenente l’ordine di attaccare Waterloo nel tentativo di evitare l’infamia della sconfitta. Ancora, si consideri la situazione seguente su Internet. Bob intende vendere un immobile.

Figura 1-3 Firma di un documento

Un acquirente, Alice, manda a Bob un messaggio in cui si impegna a pagare la somma di 100 milioni per l'immobile. Bob soddisfatto della proposta di Alice conclude la trattativa, rifiutando da quel momento in poi tutte le altre offerte. Se Bob non potrà dimostrare che sia stata proprio Alice a inviare quel messaggio, Alice, negando di aver mai stabilito la cifra intera, potrà versare a Bob una somma inferiore.

Un modo per attribuire paternità ad un documento è vincolarlo ad una qualche caratteristica, difficilmente imitabile, di chi lo ha prodotto. Basti pensare alla calligrafia con cui è scritto un testo. Il sistema più diffuso nel mondo reale, per stabilire la paternità di un documento, prevede l'utilizzo della firma umana (Figura 1-3).

Anche nel mondo digitale si è trovato il modo di apporre "firme" elettroniche alle informazioni, mediante l'uso della crittografia.

1.1.4 Autenticazione

Un qualunque sistema deve poter riconoscere le entità che vogliono accedere alle sue risorse. Viceversa, le entità devono identificare il sistema con cui interagire (mutua autenticazione). Per esempio si consideri la transazione elettronica, su Internet, del versamento di

una somma di denaro da una banca B1 ad una B2. Le due banche prima di effettuare lo scambio devono qualificarsi con un'elevata certezza. Infatti se un abile camuffatore assumesse l'identità di B1, potrebbe effettuare pagamenti con denaro proveniente da un conto non suo o inesistente, se invece assumesse l'identità di B2, potrebbe incrementare il proprio fondo.

Un sistema per realizzare un processo di identificazione può essere realizzato attraverso: il possesso di un oggetto, si pensi al distintivo di un agente di polizia; la conoscenza di un segreto, per esempio una frase od una parola (password); una caratteristica personale fisiologica, come l'impronta digitale oppure la retina.

1.1.5 Autorizzazione

Ogni entità deve assumere un ruolo all'interno del sistema in cui si trova. Il ruolo è definito da un particolare insieme di permessi. Con i permessi si intende esprimere quello che si può fare e quello che è invece proibito: è questo il compito dell'autorizzazione.

Considerando Internet, un utente paga una quota di denaro che gli consente di accedere ad un servizio: effettuare 10 consultazioni in una banca dati. La banca dati dovrà rifiutare l'undicesima richiesta di consultazione. Quindi strettamente legato all'autorizzazione troviamo il problema del "*controllo d'accesso"* che nella sua forma più generale può essere implementato da un meccanismo che si basa sull'utilizzo di una matrice (matrice di protezione) che stabilisce per ogni coppia Dominio-risorsa, il tipo di accesso consentito [Corr97].

1.2 La crittografia

La crittografia è la scienza che fornisce gli strumenti per rendere le informazioni incomprensibili a chiunque non sia autorizzato a consultarle. Ha origini molto antiche: vi sono tracce di applicazioni di crittografia risalenti agli egizi (che la usavano soprattutto per le comunicazioni belliche).

Vediamo i termini principali che saranno utilizzati in seguito per illustrare gli strumenti crittografici e le loro applicazioni:

- ***cifratura o crittazione****:* operazione utilizzata per nascondere le informazioni;
- ***cifrario****:* algoritmo usato per la cifratura;
- ***testo chiaro*** *("plaintext"):* informazione o messaggio da cifrare;
- ***crittogramma*** *("chiphertext"):* testo cifrato;
- ***chiave del cifrario****:* mezzo fondamentale per eseguire le operazioni di conversione dal testo chiaro a quello cifrato e/o viceversa;
- ***decrittazione****:* traduzione da testo cifrato a testo chiaro;
- ***crittosistema****:* ambiente in cui sono effettuate operazioni di crittazione e decrittazione;
- ***crittoanalisi****:* azione che mira a svelare il contenuto di un crittogramma senza conoscere la chiave del cifrario;
- ***crittologia****:* lo studio di crittografia e crittoanalisi.

La crittografia tradizionale mette a disposizione due grandi categorie di cifrari: *i cifrari di sostituzione ed i cifrari di trasposizione.* Entrambi operano una trasformazione semplice (per esempio una operazione di trasposizione o una di sostituzione, cfr. par. succ.). La crittografia moderna utilizza i suddetti cifrari, combinandoli per fornire *cifrari composti o di prodotto* che,

utilizzando una serie di trasformazioni semplici in cascata, risultano molto più complessi, quindi sicuri.

1.2.1 Cifrari di sostituzione

Un cifrario di sostituzione consente di ottenere il crittogramma mediante permutazioni sull'alfabeto del testo in chiaro. Vengono camuffati i simboli dell'alfabeto, che però conservano il loro esatto ordine. Tra i più antichi cifrari di sostituzione che conosciamo vi è quello di Cesare: si sostituisce ad ogni lettera quella che la segue di tre posti nell'alfabeto, cosi che la parola "CESARE" diventa "FHVDUH". In cifrari più generali, l'alfabeto viene traslato di "k" lettere invece che di tre. Ovviamente in essi la chiave di lettura è il numero k. Altri miglioramenti hanno portato a cifrari in cui è necessario stabilire una corrispondenza arbitraria fra i simboli del testo chiaro (come le 26 lettere dell'alfabeto) ed i simboli del testo cifrato, ad esempio secondo la corrispondenza di Figura 1-4, "cane" si trasforma in "rqfu".

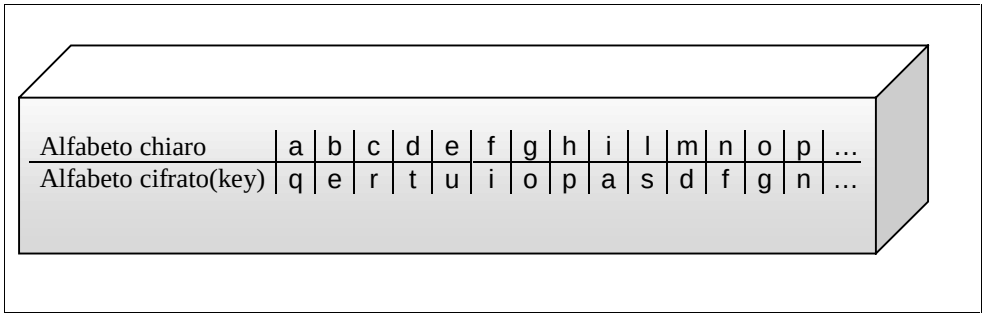

Figura 1-4 Tabella di corrispondenza

In tale sistema noto come *cifrario di sostituzione monoalfabetica*, la chiave è la tabella di corrispondenza delle 26 lettere dell'alfabeto completo. Sono possibili dunque 26! combinazioni. Tale cifrario potrebbe sembrare sicuro, ma la probabilità di decifrare il

crittogramma aumenta moltissimo se si conoscono le proprietà statistiche con cui il testo chiaro è stato scritto: è sufficiente sostituire le lettere, i digrammi, i trigrammi più ricorrenti in quel particolare linguaggio a quelle più ricorrenti nel testo cifrato.
Per appianare le differenze nelle frequenze delle lettere del testo cifrato si sono introdotti più alfabeti da usare a rotazione, ottenendo così un *cifrario di sostituzione polialfabetico* [Tei97]. Ne è un esempio il cifrario di Vigenére che consiste di una matrice quadrata contenente 26 alfabeti di Cesare. La prima riga chiamata riga "A" contiene l'alfabeto di uso comune; la seconda riga (riga "B") contiene l'alfabeto traslato e ruotato di una posizione (BCDE....XYZA) e così via fino all'ultima riga detta riga "Z" che contiene la sequenza ZABC....WXY. La chiave è una singola parola o frase che viene scritta ripetutamente sopra il testo da cifrare. Ad esempio, con la chiave "**lavorare**" si ottiene la tabella in Figura 1-5.

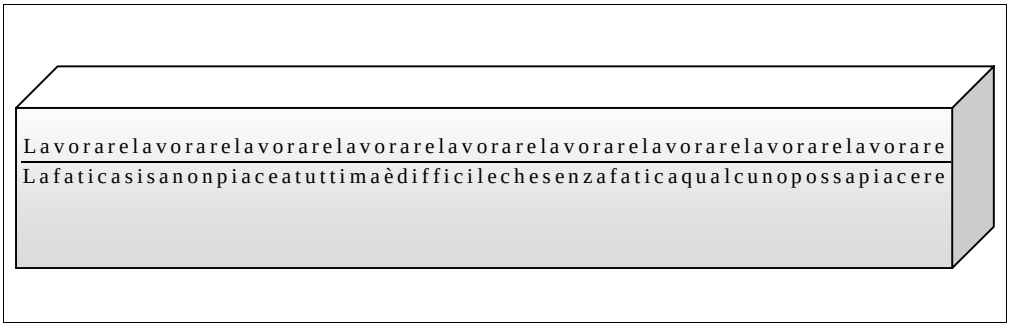

Figura 1-5 Cifrario di Vigenére

La lettera della chiave sopra la lettera del testo chiaro determina la riga e quindi l'alfabeto da usare per la cifratura. In questo modo una lettera del testo chiaro sarà rappresentata da lettere diverse nel testo cifrato, a seconda della sua posizione nel testo chiaro.

Anche questi cifrari possono essere facilmente svelati: se per ipotesi si conoscesse la lunghezza "k" della chiave, si potrebbe spezzare il testo cifrato in blocchi aventi quella lunghezza,

sovrapporli e ottenere così ogni colonna codificata secondo un cifrario monoalfabetico, facile da attaccare. Per conoscere il valore esatto di "k" si procede per tentativi fino a quando non si trova un valore per il quale le frequenze relative dei vari cifrari monoalfabetici sono uguali e approssimativamente corrispondenti a quelle del linguaggio con cui il messaggio è stato scritto.

1.2.2 Cifrari di trasposizione

I cifrari di trasposizione operano sull'ordine dei simboli del testo chiaro, senza alterarli. Considerando una "trasposizione colonnare", si agisce nel modo seguente: data come chiave una parola o frase che non contenga alcuna lettera ripetuta, dopo aver disposto il messaggio da cifrare in righe sovrapposte di lunghezza uguale a quella della chiave, si numerano le colonne così ottenute assegnando il numero 1 a quella posta sotto la lettera della chiave che è più vicina all'inizio dell'alfabeto e così via fino al numero che corrisponde alla lunghezza della chiave. Il testo cifrato viene ricavato leggendo le colonne, partendo da quella con il numero minimo.

In generale si può fissare un numero intero P (periodo della trasposizione) e scegliere una permutazione degli interi da uno a P. Per esempio se P=7, si può far corrispondere alla successione 1 2 3 4 5 6 7 quella permutata 4 6 3 5 7 1 2. Il testo viene considerato a blocchi di P byte ed i P byte di ciascun blocco vengono anagrammati in base alla permutazione. Per la decifrazione si usa la permutazione inversa. Anche un cifrario del genere può essere svelato procedendo per tentativi se il numero P non è molto grande.

1.2.3 Gli attacchi crittoanalitici

Esistono oggi numerose tecniche di crittoanalisi, citiamo quelle più diffuse in ordine di difficoltà crescente di determinazione della chiave di crittazione e quindi del messaggio in chiaro:

Attacco a solo testo cifrato
L'intruso non conosce nulla circa il contenuto del messaggio, e deve quindi lavorare soltanto sul ciphertext. In pratica esiste la possibilità di indovinare qualcosa circa il testo chiaro, e questo perchè, ad esempio, molti messaggi hanno testate di formato fisso, oppure iniziano in modo facilmente intuibile. Potrebbe anche essere facile indovinare una parola contenuta in un certo blocco di testo cifrato.
Attacco a testo chiaro conosciuto
L'attaccante conosce oppure può indovinare il testo chiaro corrispondente ad alcune parti di testo cifrato. Il suo compito è quello di decifrare il resto del ciphertext avvalendosi di queste informazioni.
Attacco a testo chiaro scelto
Il crittoanalista è in grado di ottenere qualsiasi testo lui voglia benchè crittato con la chiave sconosciuta. Il suo compito è quello di determinare proprio la chiave di crittazione.
Attacco "uomo in mezzo"
Questo tipo di attacco è efficace nei confronti delle comunicazioni di crittografia, come ad esempio nel caso dei protocolli per lo scambio di chiavi. Supponiamo che due corrispondenti stiano scambiandosi la chiave ed un intruso si pone nel mezzo della linea di comunicazione tra i due. Egli può eseguire uno scambio di chiavi separato con ciascuno dei due, dando ovviamente ad essi la netta impressione di aver scambiato la chiave tra loro e basta. In realtà invece potrà decrittare un messaggio proveniente da uno dei

due, leggerlo ed eventualmente modificarlo, e successivamente ricrittarlo ed inviarlo all'altro in maniera perfettamente trasparente.

1.2.4 La crittografia moderna

Un crittosistema è costituito da un algoritmo (cifrario) e da una chiave (chiave del cifrario). Oggi i cifrari sono basati ancora su operazioni di sostituzione e trasposizione ma non operano più solo singole trasformazioni, bensì applicano lunghe e miste sequenze di tali operazioni elementari. I primi crittosistemi davano molta rilevanza alla segretezza dell'algoritmo. Al contrario, oggi la maggior parte degli algoritmi è pubblica ed i motivi che inducono a questa scelta sono:

- la diffusione di un algoritmo crittografico consente un'analisi approfondita del suo funzionamento da parte dell'intera comunità scientifica. Aumentano così le probabilità di individuare tutti i suoi punti deboli sfuggiti per esempio in fase di progettazione;
- gli sforzi enormi che sono necessari per inventare e diffondere un nuovo metodo crittografico;
- la diffidenza verso un prodotto così importante nei compiti che deve svolgere: se un cifrario è di Dominio pubblico può essere controllato affinchè non possieda alcuna "backdoor", cioè un canale privilegiato per la lettura di tutte le informazioni crittate con tale algoritmo (un "passpartout").

La capacità di mantenere un segreto da parte di un algoritmo crittografico pubblico dunque si sposta unicamente sulla segretezza della chiave di cifratura. Quindi ogni decisione che coinvolge tali chiavi in modo diretto (scelta del formato) o indiretto (loro diffusione) si deve considerare estremamente pericolosa. Nei

prossimi paragrafi, illustrando maggiormente gli aspetti della crittografia moderna, vedremo come affrontare queste decisioni.

Tutti i metodi moderni fanno uso di chiavi per eseguire la crittazione e la decrittazione. Per alcuni algoritmi le due chiavi sono uguali, mentre per altri sono diverse. Questa distinzione dà origine a due categorie: quella degli algoritmi simmetrici (detti anche a *chiave simmetrica* o a *chiave segreta*) e quella degli algoritmi asimmetrici (detti anche a *chiave asimmetrica* o a *chiave pubblica*). Un'altra famiglia di algoritmi utilizzati dalla crittografia moderna è costituita dalle *funzioni hash*: funzioni capaci di comprimere un insieme di dati di input in un output di lunghezza fissa e generalmente di piccole dimensioni.

1.2.5 Algoritmi simmetrici

Gli algoritmi simmetrici sono quelli usati dalla crittografia classica: essi permettono al mittente e al destinatario di usare la medesima chiave per crittare e decrittare un messaggio (Figura 1-6).
Gli algoritmi simmetrici si suddividono in: *cifrari a flusso,* che possono crittare un solo bit di testo chiaro alla volta e *cifrari a blocco*, che considerano un certo numero di bit (tipicamente 64) e li cifrano come una singola unità. In genere il più usato è il codice a blocchi per le sue migliori prestazioni. Con questi algoritmi si possono affrontare i problemi di: riservatezza, integrità ed autenticazione.

Per ottenere *riservatezza* si cifra l'informazione, spesso solo il corpo del messaggio e non l'intestazione, trasmessa in chiaro per semplificare l'instradamento. Per avere *integrità* si usano tecniche combinate di cifratura e controlli algoritmici (tipo checksum, CRC, ecc.), ad esempio, cifrando il checksum del messaggio.

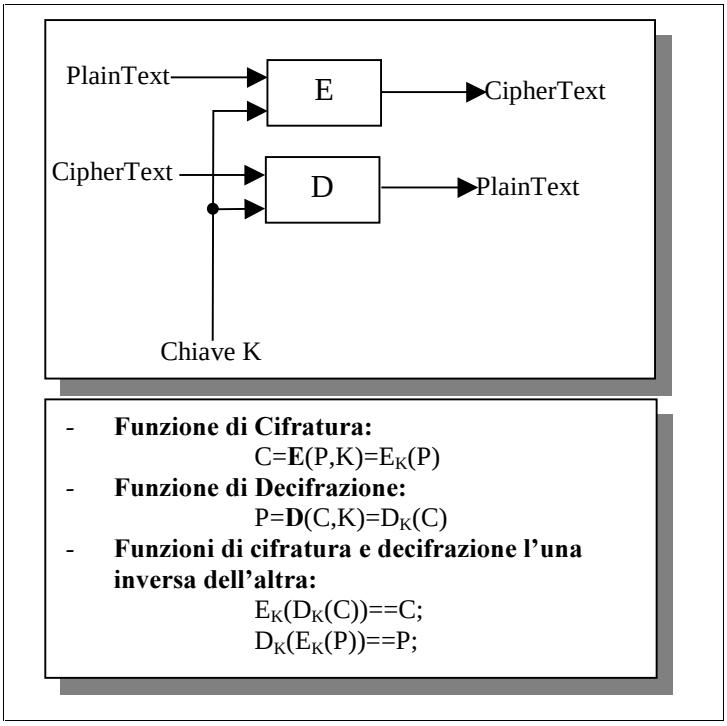

Figura 1-6 Algoritmi simmetrici

Mentre per l'*autenticazione*, si usano protocolli tali che gli algoritmi simmetrici consentano al vero mittente di includere nel messaggio informazioni che lo possano identificare con certezza (Figura 1-7).

La natura di questi sistemi mette in evidenza alcuni punti deboli:

- tutti i partecipanti alla comunicazione cifrata devono possedere una stessa chiave simmetrica. Bisogna risolvere il problema della consegna della chiave. Una prima soluzione è affidarla di persona a tutti, ma se ciò non è possibile la si dovrebbe trasmettere sullo stesso canale di comunicazione insicuro da cui ci si vuole proteggere: le chiavi così sarebbero intercettate ed usate per decifrare i messaggi altrui;

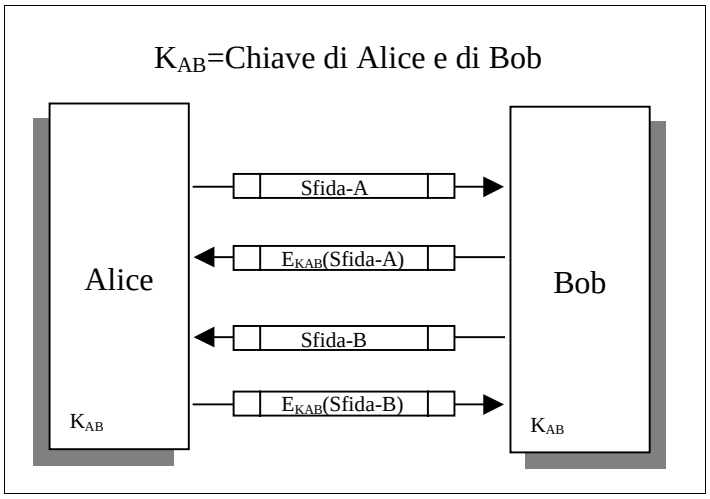

Figura 1-7 Protocollo di autenticazione

- poiché gli utenti condividono chiavi segrete, non si può facilmente risolvere il problema della *paternità* del messaggio. Se Bob ed Alice iniziano una comunicazione cifrata con una chiave, in caso di controversia, nessuno dei due è in grado di dimostrare ad una terza parte che l'altro sta mentendo.
- poiché serve una chiave diversa per ogni utente con cui si vuole intraprendere una comunicazione cifrata, il numero di chiavi da memorizzare può essere talmente grande da risultare ingestibile.

Diamo ora una descrizione più dettagliata del funzionamento di tali algoritmi, affrontando prima i quattro modelli più comuni di codice a blocchi:

***Electronic Codebook* (ECB)**
Questo metodo prevede che ogni blocco di testo sia cifrato sempre con la stessa chiave. È il sistema più semplice ma anche meno sicuro perché può fornire informazioni che facilitano gli attacchi

crittografici. Infatti se un messaggio contiene due blocchi uguali i due corrispondenti di testo cifrato saranno identici.

***Cipher Block Chaining* (CBC)**
È un modello simile al precedente anche se in questo caso i blocchi di testo cifrato sono "incatenati" ai propri predecessori per nascondere le parti che si ripetono all'interno del testo semplice che verrebbero altrimenti ripetute nel testo cifrato.
In particolare al blocco di testo cifrato precedente viene applicata l'operazione di XOR con il blocco corrente di testo semplice prima della normale crittazione con la chiave; il tutto viene successivamente cifrato. Il problema principale di tale schema è l'efficienza.

Cipher Feedback (CFB)
Come per il CBC i blocchi vengono "incatenati" fra loro ma questa volta dopo la normale crittazione, viene fatta la cifratura del blocco cifrato precedente e poi lo XOR con il testo chiaro, spezzettato in segmenti più piccoli. L'idea è quella di elaborare i dati non appena divengono disponibili invece di aspettare che un blocco sia del tutto completato (come invece avviene con il CBC).

***Output Feedback (OFB)*:**
Viene utilizzato affinché il procedimento di cifratura sia ancora più veloce rispetto al CFB. In questo metodo la connessione (detta *feedback*) avviene tra l'output del passo precedente e il blocco corrente.

Diamo una descrizione dettagliata del DES, uno dei principali algoritmi a chiave simmetrica, per capire la filosofia di tali cifrari.

1.2.5.1 DES

Il DES (Data Encryption Standard), adottato dal governo degli Stati Uniti nel 1977 come standard federale, deriva dall'algoritmo Lucifer inventato dall'IBM nei primi anni '70. Mentre Lucifer era ancora in via di sviluppo il National Bureau of Standard (NBS), diventato poi NIST, sollecitò l'industria americana alla creazione di un nuovo standard crittografico per la protezione di dati riservati ma non classificati come "segreti militari" o di "stato". Nel 1974 l'IBM propose un Lucifer modificato a cui fu dato il nome di DES. Nonostante la validità di questo software dovesse essere di 5-10 anni, ancor oggi è lo standard ufficiale.

Il DES è *un codice cifrato a blocchi* che può essere usato in tutti i modelli ECB, CBC , CFB e OFB. La chiave usata per crittografare è un blocco di 64 bit suddivisa in 8 sottoblocchi di 8 bits ciascuno; l'ultimo bit di ogni sottoblocco è di controllo, di conseguenza i bit liberi sono 56. Il testo da cifrare viene suddiviso in blocchi di 64 bit ciascuno e vengono cifrati uno dopo l'altro in successione con uguale procedimento se viene usato il modello ECB altrimenti vengono incatenati fra loro secondo i metodi CBC, CBF e OFB.

Ci sono diversi sistemi utilizzati per completare un messaggio, se esso non raggiunge la lunghezza desiderata di 64 bit (procedimento detto "pad"): un metodo aggiunge zeri fino alla lunghezza stabilita mentre un altro, se i dati sono binari, integra il blocco con bit che sono l'opposto degli ultimi bit del messaggio. Nel caso di dati ASCII si usano invece byte random specificando nell'ultimo byte il carattere ASCII corrispondente al numero di byte aggiunti. Infine un'ultima tecnica, in parte equivalente alla precedente, usa sempre bit casuali ma fornisce, negli ultimi tre bit, il numero di byte non padding cioè originali. In tal caso se il blocco è già di 64 bit si dovrà aggiungere un'altra stringa di 64 bit con 61

bit random e gli ultimi 3 nulli dato che indicano il numero di byte validi.

Durante la cifratura un blocco di testo normale viene per prima cosa trasposto (o permutato). Poi il blocco di 64 bit viene diviso in una metà destra e una metà sinistra di 32 bit. In seguito vengono applicate 16 passate (round) tramite una funzione che opera sia trasposizioni che sostituzioni ad ogni metà mediante sottochiavi. Durante ogni round l'output della metà sinistra diventa l'input della destra e viceversa. Dopo il completamento di tutti i round i due sottoblocchi vengono riuniti e il risultato permutato per invertire la trasposizione iniziale. Precisamente: indichiamo con T_i il risultato della i-esima iterazione, con S_i il semiblocco sinistro, con D_i il semiblocco destro e con K_i la sottochiave. In base all'algoritmo avremo che:

$T_i=S_iD_i$ blocco di testo chiaro (plaintext)
$S_i=D_{i-1}$
$D_i=S_{i-1}$ XOR $f(D_{i-1},\mathbf{K_i})$

Vediamo come opera la funzione "f":
il blocco D_{i-1} viene espanso da 32 bit a 48 con un modulo di espansione E. Indichiamo il blocco espanso con $E(D_{i-1})$, si calcola:

$E(D_{i-1})$ XOR $\mathbf{K_i}$;

il risultato precedente viene spezzato in 8 blocchi di 6 bit ciascuno: $B_1,B_2,...,B_8$ contenenti rispettivamente i bit da 1-6, da 7-12 ecc., ciascun blocchetto B viene usato come ingresso ad una funzione Z che restituisce stringhe di 4 bit indicate $Z(B_i)$.

La funzione Z opera in questo modo: preleva da ogni matrice fissata S-box i 4 bit del nuovo blocchetto $S_i=Z(B_i)$ posizionati in base alle righe e colonne specificate dai 6 bit del corrispondente B_i;

una volta concatenati gli 8 blocchetti $S_1,S_2,..,S_8$ verranno permutati ottendo:

$$P(S_1,S_2,..,S_8)=f\ (D_{i-1},\ \mathbf{K_i}).$$

La chiave è una stringa di 64 bit con 8 bit di controllo che vengono ignorati durante la cifratura/decifratura. Essa viene spezzata in due blocchi di 28 bit, supponiamo di chiamarli L_0 e R_0, dopodichè per 16 volte i semiblocchi vengono shiftati a sinistra ottenendo $L_1,R_1,L_2,R_2,...,L_{16},R_{16}$.
Quindi al primo round l'algoritmo utilizzerà la sottochiave $K_1=P(L_1R_1)$ dove P al solito indica una permutazione, al secondo $K_2=P(L_2R_2)$ e al 16° round $K_{16}=P(L_{16}L_{16})$ (si noti che tutte le operazioni effettuate producono sottochiavi K_i di 48 bit).

Per la decifratura il procedimento è lo stesso salvo che al 1° passo verrà utilizzata $K_{16}=P(L_{16}L_{16})$, al secondo $K_{15}=P(L_{15}L_{15})$, ecc. In termini di equazione avremo:

$T_i=S_iD_i$ blocco di testo cifrato (ciphertext)
$D_{i-1}=S_i$
$S_{i-1}=D_i$ XOR $f(S_i,\mathbf{K_i})$

1.2.6 Algoritmi asimmetrici

Gli algoritmi asimmetrici fanno uso di coppie univoche di chiavi complementari. Ogni utente possiede una chiave pubblica ed una chiave privata. La chiave pubblica può, essere conosciuta da tutti e viaggiare su canali insicuri, mentre la chiave privata è strettamente personale.

Il meccanismo fondamentale per il funzionamento di tali algoritmi è il seguente: *le due chiavi sono create in modo che il messaggio cifrato da una delle due possa essere decifrato solo e*

soltanto dall'altra. Consideriamo una coppia K formata da K_p, chiave pubblica, e K_s, chiave segreta; le due chiavi possono essere ricavate l'una dall'altra, ma l'operazione di derivare la chiave segreta da quella pubblica è troppo complessa per venire eseguita in pratica, anche su un calcolatore molto potente.

Per ottenere *riservatezza* è sufficiente cifrare un messaggio con la chiave pubblica del destinatario, che per poterlo leggere deve adoperare la propria chiave privata. Mentre per ovviare ai problemi di *paternità* è necessario cifrare un messaggio con la propria chiave privata: solamente la chiave pubblica del mittente (che tutti possiedono) lo può decifrare. Si realizza in questo modo la "firma elettronica" del messaggio. Se si cifra due volte un messaggio, una con la chiave privata del mittente ed una con la chiave pubblica del destinatario, il mittente ha la certezza che solo il destinatario riceva il messaggio ed il destinatario ha le garanzie di un messaggio firmato.

La crittografia asimmetrica risolve i problemi critici della crittografia simmetrica: si pensi alla facilità di gestione delle chiavi che non devono essere più scambiate segretamente ed al problema della paternità. Ma è caratterizzato da pessime prestazioni: per cifrare grandi volumi di dati, impiegherebbe un tempo molto superiore rispetto a quello necessario per un algoritmo simmetrico poichè richiede almeno il doppio di operazioni per ogni singola crittazione. Una soluzione molto valida in termini di rapporto prestazioni/sicurezza, consiste nell'utilizzare una tecnica mista: si sfrutta un algoritmo a chiave pubblica solo nella prima fase di negoziazione, in cui ci si scambia una chiave simmetrica, da usare poi per tutto il resto della comunicazione.

Gli algoritmi a chiave pubblica più noti sono il Diffie-Hellman e l'RSA [SCH95]. Diffie-Hellman (dal nome dei suoi inventori) è un algoritmo usato solamente per lo scambio delle chiavi. È considerato sicuro se le chiavi sono sufficientemente lunghe e sono usati generatori di chiavi appropriati. La sicurezza di Diffie-

Hellman si basa sulla difficoltà dei problemi logaritmici (che sono da considerarsi equivalenti a quelli di fattorizzazione sfruttati dall'RSA). Per capire il funzionamento degli algoritmi asimmetrici vediamo in dettaglio l'RSA, il primo cifrario completo di questo genere.

1.2.6.1 RSA

L'*RSA* (dagli autori Ron Rivest, Adi Shamir e Leonard Adleman) è nato dopo circa due anni dall'uscita dell'innovativo Diffie-Hellman. Questo tipo di cifrari si basa sul concetto di *funzioni unidirezionali*: si tratta di funzioni invertibili per le quali il calcolo della funzione diretta è facile, mentre quello della inversa è difficile per tutti coloro che non possiedono la chiave corretta. Un esempio di tale funzione è il logaritmo in base B dove l'inverso è appunto l'esponenziale. Se la base B è un numero ragionevolmente piccolo, il calcolo sia della funzione diretta che della inversa non risulta particolarmente difficile; diversamente, se il modulo o base fosse un grandissimo numero primo, la complessità di calcolo diverrebbe proibitiva.

La funzione unidirezionale che sta alla base di RSA viene costruita sfruttando la proprietà del prodotto di due numeri primi: è facile calcolarlo tra due numeri primi molto grandi, ma dato il loro prodotto, è difficilissimo risalire ai fattori che lo compongono.
Vediamo i passaggi che effettua il cifrario:

1) si trovano due numeri primi molto grandi P e Q, sia $N = P \cdot Q$ il loro prodotto;
2) si sceglie E (minore di N) non divisibile per $(P-1) \cdot (Q-1)$ cioè in modo che non abbiano fattori primi in comune; E deve essere dispari; $(P-1) \cdot (Q-1)$ non può essere primo perché è pari.
3) si calcola D, l'inverso di E, tale che

$$D \cdot E = 1 \textit{ modulo } (P-1) \cdot (Q-1);$$

4) il testo cifrato si ottiene con l'operazione:

$$C = (T^E) \textit{ modulo } N$$

dove T è il testo chiaro (intero positivo) e C il testo cifrato;

5) per ricavare il testo chiaro R, si deve effettuare:

$$R = (C^D) \textit{ modulo } N$$

dove C indica sempre il cipher-text.

La chiave pubblica è composta da due parti: il modulo N ed E mentre quella privata da N e D, perciò E è definito l'esponente pubblico e D quello privato. Non si conoscono metodi per calcolare D, P e Q dati $N=P \cdot Q$ ed E.

Vediamo una applicazione del cifrario: scelto N=33, abbiamo Q=11 e P=3; prendendo E=3 otteniamo D=7 infatti: $(P-1) \cdot (Q-1)=20$ perciò: 1 *modulo* 20 = 21 = $D \cdot E$; si elevano alla terza potenza i numeri da 0-8 e se ne esegue il modulo 33 (Figura 1-8).

plain-text	*cipher-text*	
0	0	
1	1	
2	8	
3	27	
4	31	($4^3 = 64$, 64 modulo 33 è uguale a **31**)
5	26	
6	18	
7	13	
8	17	

Figura 1-8 Applicazione di RSA

Per verificare che effettivamente questo cifrario funzioni decriptiamo i valori ottenuti al passo precedente: si eleva alla settima e si esegue il modulo 33 (Figura 1-9). La chiave è costituita

da un modulo e un esponente, quando si parla della sua dimensione si intende quella del modulo N. La scelta di N dipende dal bisogno di sicurezza. Più lungo è il modulo, maggiore è la sicurezza ma anche più lente sono le operazioni di cifratura.

cipher-text		*plain-text*
0	_____	0
1	_____	1
8	_____	2
27	_____	3
31	_____	4
26	_____	5
18	_____	6
13	_____	7 (13^7 = 62.748.517 modulo 33 fornisce proprio il valore **7**)
17	_____	8

Figura 1-9 Applicazione inversa di RSA

Infatti, raddoppiando la lunghezza della chiave si incrementa il numero di operazioni necessarie per la cifratura/decifratura con public-key di un fattore 4, e quelle eseguite con secret-key di un fattore 8. La ragione per cui la cifratura/decifratura con public-key richiede meno operazioni è che l'esponente pubblico E, può rimanere fisso anche se aumenta il modulo. Diversamente l'esponente segreto D deve incrementarsi proporzionalmente ad N. Si dovrà allora trovare un compromesso. Generalmente se la chiave è lunga 512 bit ogni fattore primo P e Q deve essere circa 256 bit.

Vediamo come utilizzare questo algoritmo per la riservatezza e la paternità:

- *riservatezza*: supponiamo che Alice voglia inviare un messaggio M a Bob. Alice crea il testo cifrato C applicando

$$C = (M^E) \textit{ modulo } N$$

dove E ed N costituiscono la chiave pubblica di Bob. Per decifrare, Bob calcola M applicando

$$M = (C^D) \textit{ modulo } N$$

con la sua chiave privata <N,D>, ottenendo così il messaggio originale.

- *paternità*: Alice vuole inviare un documento firmato a Bob, crea dunque una firma digitale S con questa operazione:

$$S = (M^D) \textit{ modulo } N$$

dove <N,D> è la sua chiave privata. Bob riceve tale messaggio e vuole verificare se davvero è stata Alice a scriverlo, quindi calcola:

$$(S^E) \textit{ modulo } N$$

(con <N, E> chiave pubblica di Alice), se il testo ottenuto è chiaro Bob potrà essere sicuro del mittente.

1.2.7 Funzioni hash (one way)

Una *funzione hash* trasforma un testo normale di lunghezza arbitraria in una stringa di lunghezza fissa generalmente più corta. Questa stringa rappresenta un "riassunto", "un'impronta digitale" del messaggio e viene definita valore hash o checksum crittografico. Noto il solo valore hash non è possibile ricostruire il messaggio originale (one-way).
Queste funzioni svolgono compiti molto importanti:

- *verifica dell'integrità*: l'integrità del messaggio è mantenuta se si verifica l'eguaglianza del suo valore hash prima e dopo la trasmissione;
- *sinteticità:* risulta più comodo manipolare l'impronta hash di un messaggio (di dimensione fissa per esempio 128 bit) piuttosto che il messaggio stesso di lunghezza maggiore ed impredicibile;
- *firma digitale*: per realizzarla è sufficiente allegare al messaggio il suo valore hash, operazione che se eseguita su un file di grosse dimensioni richiede calcoli meno lunghi e complessi rispetto ad altri sistemi crittografici.

Tra le funzioni hash più importanti troviamo: MD2, MD4, MD5 (MD sta per Message Digest) , SHA (Secure Hash Algorithm) e Tiger, che dal punto di vista dell'uso, si distinguono per il numero di bit del valore hash e per lo schema di padding. Vediamo, per chiarire con un esempio, come funzionano gli algoritmi hash.

1.2.7.1 MD2

MD2, inventato da Ron Rivest, produce un valore hash di 128 bit. Il suo funzionamento può essere scomposto in tre fasi successive:

1) si esegue il PAD, cioè si rende la dimensione del messaggio multipla di 16 byte aggiungendo i bit mancanti secondo una specifica convenzione;
2) si aggiunge una quantità di 16 byte chiamata checksum, in coda al messaggio. Tale quantità è già simile ad un valore hash, ma non sufficientemente sicura, per questo motivo si esegue anche la terza fase. I bit del checksum sono inizializzati a 0. Il messaggio è un multiplo di 16 byte, quindi è lungo k*16 byte. Il

processo di calcolo del checksum, che considera un byte per volta, richiede k*16 passi. Ad ogni passo si aggiorna un byte del checksum, in modo ciclico, così ogni bit sarà aggiornato k volte. Al passo n, viene considerato il byte (n mod 16), che indichiamo per semplicità byte n del checksum.

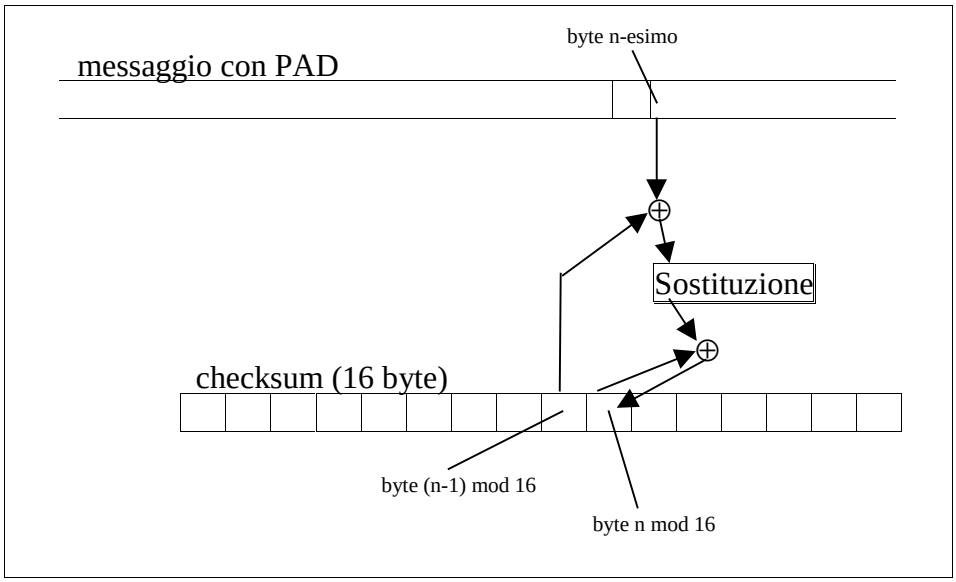

Figura 1-10 Calcolo del checksum

Il byte n del checksum dipende dal byte n del messaggio, dal byte (n-1) del checksum e dal valore precedente del byte n del checksum, secondo questa relazione: viene eseguito l'exor tra il byte n del messaggio ed il byte (n-1) del checksum, il byte risultante, che ha un valore compreso tra 0 e 255, viene usato per estrarre un altro byte da una tabella di sostituzione (per esempio quella contenente le cifre del pi-greco). Viene infine eseguito l'exor tra questo byte e il vecchio valore del byte n del checksum (Figura 1-10).

3) In questo ultimo passo avviene un calcolo simile al precedente. Il messaggio (con i bit di PAD ed il checksum in coda), viene elaborato a blocchi di 16 byte. Ogni volta viene costruita una quantità di 48 byte fatta dai 16 byte del valore corrente dell'hash, dai 16 byte del blocco di messaggio e dai 16 byte dell'exor dei due. La quantità di 48 byte viene modificata byte per byte in cicli di 18 passate (quindi vengono effettuati 48 x 18 passi). Della quantità così ottenuta si considerano i primi 16 byte come valore hash per lo stage successivo. Il valore hash è inizializzato con 16 byte a zero. È necessario conoscere il numero di ciclo perché viene sfruttato nell'elaborazione. Un byte fantasma "–1", appare prima del primo byte (byte 0) della quantità di 48 byte. All'inizio del ciclo 0, il byte –1 è settato a 0. Nel ciclo n, di ogni passata si prende il byte n-1, si esegue la stessa sostituzione fatta per il checksum e si calcola l'exor del risultato con il byte n. Dopo il ciclo 47 di ogni passata, il ciclo 48 setta il byte –1 al valore della somma modulo 256 del byte 47 ed il numero di ciclo. Dopo aver processato tutto il messaggio i 16 byte che vengono fuori dall'ultimo stage sono il valore hash.

1.2.8 Sicurezza degli algoritmi crittografici

Per quanto un algoritmo basato su una chiave possa essere complesso, esiste sempre un metodo in grado di invalidarne l'efficacia: l'attacco a *forza bruta*, che consiste nel provare la sequenza di tutte le chiavi possibili. Tuttavia tale tipo di attacco richiede un tempo di computazione crescente in modo esponenziale con la dimensione della chiave.

È dunque possibile proteggersi dimensionando adeguatamente le chiavi.

Infatti:

$$T_{max} = (\tau \cdot 2^n)/N$$

- T_{max} = tempo massimo richiesto per scoprire la chiave;
- τ = tempo necessario per una verifica;
- n = lunghezza della chiave;
- N = numero di calcolatori in parallelo;

Per esempio considerando il DES in cui la dimensione della chiave è di 56 bit, con un calcolatore che tenta 10^6 chiavi al secondo (1 MIPS), sono necessari T_{max}=2.000 anni. Se si considera una macchina a parallelismo massiccio, da 5,7M MIPS, si può avere un T_{max} di circa 3,5 ore. Mentre con una macchina da 57M MIPS T_{max}=21 minuti.

Per quanto riguarda invece la crittografia a chiave pubblica, la lunghezza delle chiavi è molto maggiore di quella delle chiavi simmetriche. *Qui il problema non consiste nell'indovinare la chiave giusta, bensì è quello di trovare la chiave segreta che si accoppia con quella pubblica*. Nel caso di RSA, ciò significa fattorizzare un numero intero molto grande, che ha due fattori primi altrettanto grandi.

In un sistema, per garantire un qualche livello di sicurezza, non sono sufficienti solamente un cifrario ed una chiave, serve anche *una adeguata infrastruttura capace di gestire i problemi legati all'amministrazione delle chiavi*:

- nella crittografia a chiave privata, ogni utente deve "ricordare" un elevato numero di chiavi, oppure ne deve memorizzare una sola, quella condivisa con un key distribution center (KDC), il quale avrà il compito di generare chiavi temporanee da consegnare agli utenti che intendono comunicare tra loro.

- nella crittografia a chiave pubblica, il problema maggiore è legato alla possibilità di sostituire la propria chiave pubblica con quella di un altro utente. Questa operazione si può fare alterando l'associazione tra la chiave pubblica ed il nome del proprietario. Per esempio Alice vuole mandare un messaggio a Bob, lo deve cifrare con la chiave pubblica di Bob, chiave che ha ottenuto da una infrastruttura di distribuzione sulla rete. Se Trudi è riuscita ad intercettare tale chiave e sostituirla con la propria (Figura 1-11), Bob non riceverà nulla di intellegibile, perché Alice cifrerà i messaggi con la chiave cui è associato il nome di Bob, ma che in realtà appartiene a Trudy.

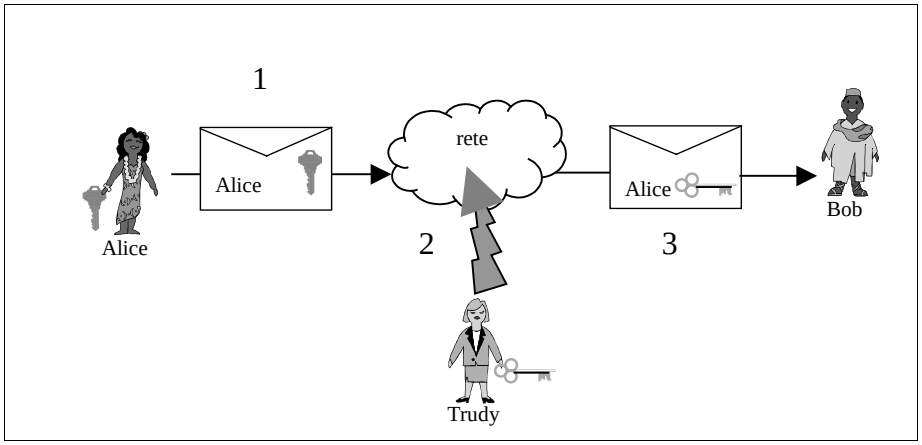

Figura 1-11 Sostituzione chiave pubblica

Solo Trudy, intercettando anche i messaggi, potrà leggerne il contenuto (Figura 1-12). Per evitare questo, le chiavi pubbliche sono inserite in "*certificati*" firmati, che sanciscono la corrispondenza tra chiave ed utente. Un certificato può essere self-signed ma in tale caso ha poco valore. I certificati dunque sono rilasciati e firmati da apposite autorità fidate, le Certificate Authority.

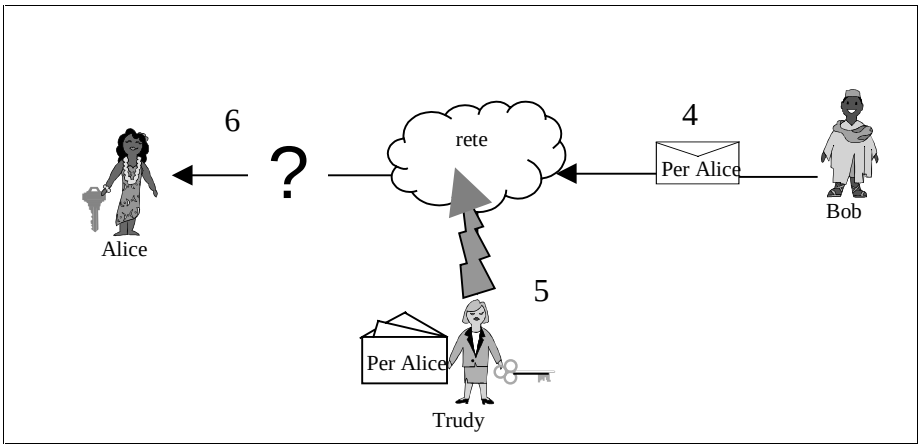

Figura 1-12 Lettura di messaggi altrui

Vedremo in maggior dettaglio questi problemi nel Paragrafo 1.3.1.

1.3 Sistemi distribuiti e sicurezza

Diversi anni fa, il problema della sicurezza riguardava contesti informatici ristretti e molto particolari. Allora venivano adottate soluzioni basate sui sistemi crittografici tradizionali e si collocavano i calcolatori in stanze accessibili solo a personale autorizzato. Questi due tipi di protezione erano sufficienti poiché il codice era perlopiù locale o comunque coinvolgeva poche macchine. Il concetto di sicurezza era dunque strettamente legato alla "chiusura" del sistema: l'hardware, i sistemi operativi, i protocolli per lo scambio di informazioni dovevano differire tantissimo da un sistema all'altro.

Oggi si parla sempre più di sistemi distribuiti, perché si è compreso il beneficio che può derivare dal loro uso. Essi consentono di condividere risorse costose, facilitano il lavoro di gruppo, forniscono una potenza di calcolo teoricamente illimitata, ecc. [Tan94].

Caratteristica fondamentale dei sistemi distribuiti è l'apertura: *"la capacità di adeguarsi alle condizioni di stato precedenti o successive alla messa in opera"* [Corr97].
L'apertura si può ottenere mediante l'adozione di standard di costruzione e di comunicazione: si veda ad esempio il modello OSI [Tan94]. Il sistema distribuito e aperto per eccellenza è Internet: rete nata per scopi militari, poi estesa agli ambienti scientifici di tutto il mondo ed infine a tutti gli utenti di calcolatori. Internet è la rete che ha suscitato i più grandi interessi economici, segnando il percorso verso il mercato globale. Ma Internet ha messo anche in evidenza come l'apertura esponga maggiormente ai problemi della sicurezza.

Un sistema aperto è in continua evoluzione, richiede dunque modelli di sicurezza snelli e facilmente configurabili. L'apertura è vantaggiosa se i tempi di risposta sono contenuti mentre la gestione della sicurezza può introdurre anche forti ritardi. Dunque spesso è necessario scendere a compromessi tra sicurezza e prestazioni.

Lo scambio di informazioni avviene su canali accessibili ad un numero potenzialmente infinito di utenti, accentuando i problemi di riservatezza, identità e paternità: occorre partire dal presupposto che ogni comunicazione avvenga con una controparte non fidata finché non si sia dimostrato il contrario.

Il numero potenzialmente infinito di utenti rende anche più probabile l'individuazione di sottili errori di programmazione, di implementazione di protocolli ecc. Questi errori possono essere sfruttati per procurare danni proporzionali al numero di sistemi interconnessi. L'interesse di fornire servizi retribuiti rende estremamente critici i problemi di autenticazione ed autorizzazione. Si pensi alle questioni legali legate al commercio elettronico: contratti, scambio di denaro, dichiarazioni, ecc. Gli strumenti crittografici moderni sono in grado, se correttamente usati, di fornire anche su sistemi aperti tutti i livelli di sicurezza desiderati.

Nel prossimo paragrafo vedremo come questi siano stati applicati in tradizionali sistemi *cliente/servitore.*

1.3.1 Modelli di sicurezza in sistemi cliente/servitore

Il primo paradigma di progettazione nato per rispondere alle esigenze di condivisione delle risorse di rete è quello "cliente/servitore": la macchina che possiede la risorsa (servitore) soddisfa le richieste d'uso che giungono dalle altre (clienti). Un esempio di applicazione cliente/servitore consiste in un demone di stampa (servitore) che risiede su una macchina collegata fisicamente ad una stampante, ed un insieme di processi di videoscrittura, attivi anche su macchine remote, che si rivolgono a tale demone per produrre documenti.

Seguendo questo paradigma è molto semplice sviluppare applicazioni distribuite che realizzano una suddivisione intuitiva di ruoli e compiti. In un sistema clente/servitore, il servitore deve poter identificare i propri clienti, e decidere se eseguire o meno il servizio richiesto, i clienti devono avere garanzie sul servitore e lo scambio di informazioni deve essere riservato. Mentre i problemi di riservatezza, paternità ed integrità, vengono risolti con l'adozione degli strumenti crittografici, una particolare attenzione va dedicata a come vengono affrontati e risolti i problemi di mutua autenticazione e autorizzazione.

1.3.2 Autenticazione

Per affrontare l'autenticazione si usano diversi protocolli, ricordiamo tra i più diffusi quelli basati su:

- password

- indirizzi di provenienza
- crittografia.

1.3.2.1 Autenticazione basata su password

Il servitore possiede un elenco di coppie <nome cliente, password>, quando un cliente accede al servizio esibisce la propria password, il servitore la confronta con tutte quelle presenti nell'elenco e, se trova una corrispondenza, è certo dell'identità. Questo sistema si fonda sull'assunzione che chiunque conosca la password giusta sia effettivamente chi dichiara di essere, per esempio tutti quelli che conoscono la password di Bob sono Bob. I limiti di tale concezione sono:

- la trasmissione in chiaro della password, che può essere facilmente intercettata da "origliatori" (entità perennemente in ascolto di ogni informazione che circola);
- la possibilità di "indovinare" una password che quindi deve essere scelta in un Dominio molto grande di valori possibili
- non vi sono garanzie sull'identità del servitore

1.3.2.2 Autenticazione basata sull'indirizzo di provenienza

Il servitore che possiede coppie <nome cliente, indirizzo di rete>, adempirà a tutte le richieste provenienti dalle locazioni nella sua lista, così l'identità di un cliente verrà stabilita in base al luogo di origine. Con questo sistema si evita la trasmissione di password, ma non ci si tutela dal problema dell'imitazione dell'indirizzo, che può essere altresì individuato da una tabella di routing e quindi inserito in messaggi appartenenti ad intrusi. Infatti un cliente può

facilmente truccare l'indirizzo IP dei propri messaggi. Anche con questo tipo di autenticazione non vi sono garanzie per il cliente.

Questi primi due sistemi possono essere adatti a reti locali, o proprietarie, cioè ove esista una adeguata protezione fisica del mezzo trasmissivo, e quindi non sia semplice interferire con il sistema di comunicazione. Ma non sono sufficienti in reti come Internet, in cui le caratteristiche di apertura minano fortemente la sicurezza: tutto viaggia in chiaro e può essere facilmente letto e alterato. E' allora necessario ricorrere ad una forma più forte di autenticazione.

1.3.2.3 Autenticazione basata sulla crittografia: gestione delle chiavi.

Le funzioni hash, la crittografia a chiave segreta e la crittografia a chiave pubblica possono essere utilizzate per mansioni di mutua autenticazione come già illustrato nei paragrafi precedenti. In questo paragrafo evidenziamo dunque gli aspetti molto importanti legati alla gestione delle chiavi.

Assumendo che la sicurezza di rete sia fondata sulla tecnologia a chiave segreta, se consideriamo una rete molto vasta con N nodi, ogni calcolatore dovrà autenticare tutti gli altri, quindi dovrà conoscere N-1 chiavi, una per ogni nodo della rete. Se si aggiunge un nuovo nodo, si dovrà generare un numero di chiavi sufficiente a condividere un segreto tra tale nodo ed ogni altro nodo connesso. Poi le chiavi dovranno essere distribuite a tutti. Questa soluzione risulta impraticabile se non su reti molto piccole. Dunque si ricorre ad un *intermediario fidato.* Un Key Distribution Center (KDC), cioè un nodo fidato che conosce le chiavi di tutti i nodi. Se viene aggiunto un nodo alla rete solamente questo ed il KDC si devono configurare con una nuova chiave. Se un nodo "A" vuole

comunicare con un nodo “B”, “A” deve chiedere al KDC (e ciò avviene in modo riservato poiché “A” ed il KDC condividono un segreto) una chiave con cui rivolgersi a “B”. Il KDC, identificato “A”, genera una nuova chiave di sessione (ticket) che trasmette ad “A” e “B” usando le rispettive chiavi segrete (Figura 1-13).

Finalmente i due possono avviare la comunicazione usando questo ticket per cifrare i messaggi. Kerberos [KauPS95] è il protocollo più famoso che fa uso di KDC.

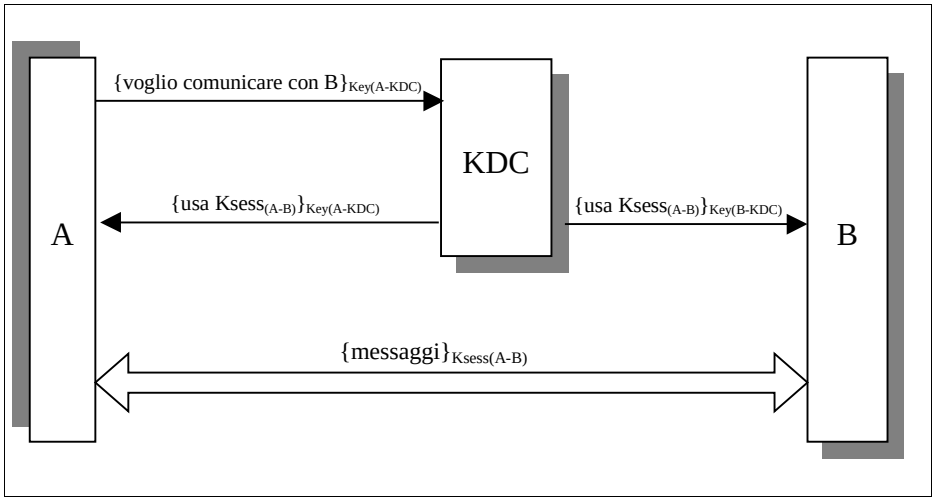

Figura 1-13 Interazione con il KDC

Vi sono tuttavia dei problemi connessi all’uso di KDC:

- il KDC ha informazioni sufficienti per impersonare chiunque;
- in caso di guasto del KDC, nessun nodo può più comunicare in modo sicuro;
- il KDC è un collo di bottiglia per le prestazioni, poiché tutti devono comunicare frequentemente con esso.

La distribuzione di chiavi è molto più semplice se la sicurezza di rete si basa su algoritmi a chiave pubblica: ogni nodo deve conoscere solamente la propria chiave privata e deve poter accedere a tutte le chiavi pubbliche presenti in una unica locazione. In tale circostanza occorre però tutelarsi da sostituzioni indebite di chiavi pubbliche di alcuni a favore di altri.

Anche qui una tipica soluzione consiste nell'avere un nodo fidato, in questo caso una Autorità di Certificazione (CA) che genera certificati cioè messaggi firmati che specificano un nome e la chiave pubblica corrispondente. Lo standard più diffuso per il formato dei certificati è l'X.509 [KauPS95]. La CA è l'equivalente a chiave pubblica del KDC e come per il KDC rappresenta un entità fidata che, se compromessa, può distruggere l'integrità di tutta la rete, introduce tuttavia dei vantaggi rispetto al KDC:

- può essere un dispositivo molto più semplice, quindi più sicuro;
- in caso di guasto non bloccherebbe tutta la rete, perché non interviene ad ogni inizio di comunicazione a differenza del KDC; quindi non rende necessaria alcuna replicazione;
- una CA, anche se compromessa, non può decifrare le comunicazioni che utilizzano certificati emessi prima del sabotaggio.

Il grosso problema delle CA è la revoca dei certificati: quando un certificato non è più valido, non è sufficiente cancellarlo dalla locazione unica in cui risiede la CA (come invece accade per i KDC); esso deve essere eliminato anche da tutti i nodi che ne hanno una copia. Questa operazione è talmente gravosa da inficiare tutti i vantaggi derivanti dall'uso dei certificati e pertanto non è realizzabile. In risposta a questo problema, ogni certificato possiede una data di espirazione. Ma tale soluzione è solo parziale,

perché non è detto che il momento della revoca coincida con la data di espirazione. La CA allora, con una certa periodicità, deve pubblicare *una Lista di Certificati Revocati* (CRL). Un qualunque utente, in caso di dubbio, è tenuto a controllare la CRL prima di usare ogni certificato in suo possesso. Con questa soluzione si riduce la finestra temporale di vulnerabilità, al periodo che intercorre tra due aggiornamenti successivi della CRL.

1.3.3 Autorizzazione

Nei sistemi cliente/servitore un modo per risolvere il problema dell'autorizzazione è l'uso di *Liste di Controllo d'Accesso* (ACL). Il servitore possiede una lista di utenti autorizzati per ogni servizio che può fornire. Ma se il numero di clienti cresce a dismisura, questo meccanismo diventa molto costoso da gestire. Si può allora introdurre il concetto di *gruppo*, che delinea una classe di individui autorizzati ad usufruire dello stesso servizio. Le ACL potranno contenere i nomi dei gruppi ed i clienti dovranno dimostrare la loro appartenenza ad essi (inserendo il nome del gruppo nel certificato, oppure usando un apposito ticket). L'uso di gruppi diventa ancor più flessibile, conveniente e scalabile se si considerano gruppi gerarchici.

Un secondo approccio utilizzato è costituito dall'uso di "capability". Ogni cliente possiede la lista dei servitori cui può accedere. In generale, la soluzione ad ACL si pone dalla parte della risorsa nel senso che per ogni risorsa vengono specificati i soggetti autorizzati all'accesso, viceversa la soluzione a Capability prevede che per ogni soggetto sia specificata la lista delle risorse cui può accedere. Per un confronto tra vantaggi e svantaggi sull'uso delle due soluzioni si veda [Corr97].

1.3.4 Sicurezza e mobilità del codice

Il modello cliente/servitore funziona perfettamente su reti locali o con poco traffico (che diversi anni fa rappresentavano la quasi totalità dei sistemi esistenti). Pone grossi problemi invece se si estende a reti con un altissimo numero di interconnessioni e con frequenti malfunzionamenti. Il problema principale è l'alto impiego di banda necessaria alle applicazioni cliente/servitore che richiedono il mantenimento di un canale permanente nel caso in cui la soluzione ad un problema richieda un alto numero di interazioni tra cliente e servitore. La comunicazione è infatti l'elemento cardine del modello: il cliente invia sulla rete una richiesta, il servitore esegue il servizio ed invia la risposta, il cliente valuta la risposta, se ha raggiunto solo un risultato parziale, invia una nuova richiesta e così via fino alla soluzione del problema.

Un altro limite del modello cliente/servitore è la sua limitata flessibilità. È necessario riscrivere il servitore ogni volta che si desidera ottenere un servizio, anche se simile a quello già offerto. Per questi motivi le applicazioni distribuite diventano componenti complessi, difficilmente gestibili ed aggiornabili.

Si sono cercate dunque nuove soluzioni, basate sulla *mobilità del codice*, ossia sulla possibilità di trasferire "by need" un flusso di esecuzione (con o senza i relativi dati) da una macchina locale ad una qualunque remota, in modo da realizzare applicazioni che eseguono localmente la maggior parte dei compiti e si spostano (impegnando i canali di comunicazione) solamente per trasferire i risultati ottenuti o per iniziare nuovi compiti. Una applicazione si può così comportare da cliente, quando richiede un servizio ad una risorsa locale, oppure da servitore, se si ferma su una macchina e mette a disposizione delle proprie risorse. In questo modo ogni macchina della rete diventa un possibile ambiente di esecuzione.

I sistemi in cui si utilizza il modello di programmazione a mobilità del codice soffrono degli stessi problemi di sicurezza presenti nei sistemi cliente/servitore e ne introducono degli altri: mentre nel modello cliente/servitore il codice può godere di un elevato grado di fiducia perché il servitore mette in esecuzione sempre e solo gli stessi algoritmi locali, questo nuovo paradigma è interamente basato sulla capacità del codice di muovere se stesso da un posto all'altro, esponendosi, durante tali spostamenti, a tutti gli attacchi che prima erano riservati principalmente ai messaggi e introducendo la possibilità di eseguire del codice non conosciuto su una qualunque macchina in modo non controllato dagli utenti della stessa.

Il linguaggio di programmazione utilizzato per produrre codice mobile assume dunque una grande rilevanza. Il "C" ad esempio fornisce gli strumenti per accedere alle risorse di più basso livello di un sistema quali registri e puntatori, quindi può consentire di assumere il controllo assoluto della macchina: il rischio di mettere in esecuzione un programma scritto in "C", proveniente da un punto qualunque della rete, quindi non fidato, potrebbe essere così alto da compromettere i molti benefici della computazione mobile.

Naturalmente se si adottassero tutte le misure di sicurezza evidenziate nei paragrafi precedenti (integrità, paternità ecc.), fidandosi di chi ha prodotto il software, non si dovrebbe temere nulla neppure dall'esecuzione di tali programmi (se non per errori di programmazione).

In realtà, le enormi latenze introdotte dalle reti geografiche ci costringono a rinunciare ad alcuni aspetti della sicurezza a favore dell'efficienza. Questo spinge a cercare soluzioni in nuove direzioni verso lo studio e la realizzazione di linguaggi, che consentano di realizzare applicazioni che anche se non fidate, non riescano comunque ad effettuare operazioni illecite. Vedremo nel Capitolo 3, quali caratteristiche dovrà possedere un linguaggio per

garantire sicurezza e mobilità. Nel Capitolo successivo, ci concentriamo maggiormente sui dettagli della mobilità.

Cap.2 Sistemi ad agenti mobili

Il primo passo necessario alla realizzazione di sistemi a mobilità del codice, richiede due tipi di analisi: una di basso livello, su quelli che sono i meccanismi della mobilità; ed una di più alto livello, che riguarda i paradigmi di progettazione del codice mobile.

2.1 La mobilità a livello di meccanismo

Le tecnologie che supportano la mobilità del codice presuppongono che lo *strato di rete* sia ben noto e visibile al programmatore. Questa caratteristica ha un forte impatto sulle caratteristiche architetturali dei sistemi per la mobilità.

L'elemento principale delle architetture per la mobilità è dunque il Computational Environment (CE): l'ambiente di esecuzione che assume l'identità della macchina host e che consente all'applicazione di rilocare dinamicamente i propri componenti su macchine diverse (Figura 2-1).

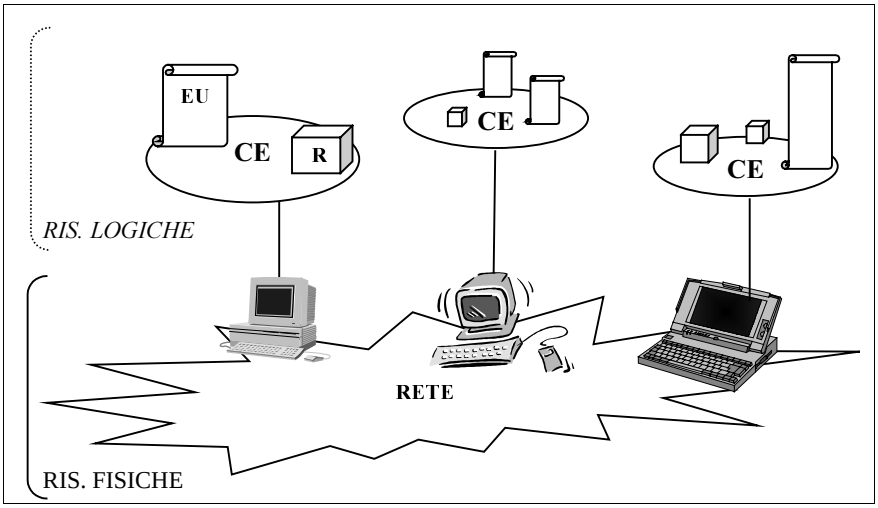

Figura 2-1 Architettura per la mobilità del codice

Il CE contiene le Unità di Esecuzione (EU) e le risorse. In questa visione, le EU sono flussi sequenziali di calcolo, ad esempio thread, mentre le risorse, sono le entità che possono essere condivise da più EU, come un file, una variabile di sistema, oppure, più in generale un oggetto (un elemento di un linguaggio ad oggetti). Una EU è composta da un *segmento di codice* che descrive il comportamento della computazione e da uno *stato*, che comprende lo spazio dei dati e lo stato di esecuzione (Figura 2-2). Lo spazio dei dati è l'insieme dei riferimenti alle risorse. Lo stato di esecuzione contiene invece i dati privati del controllo dell'esecuzione, come lo stack delle chiamate e l'instruction pointer.

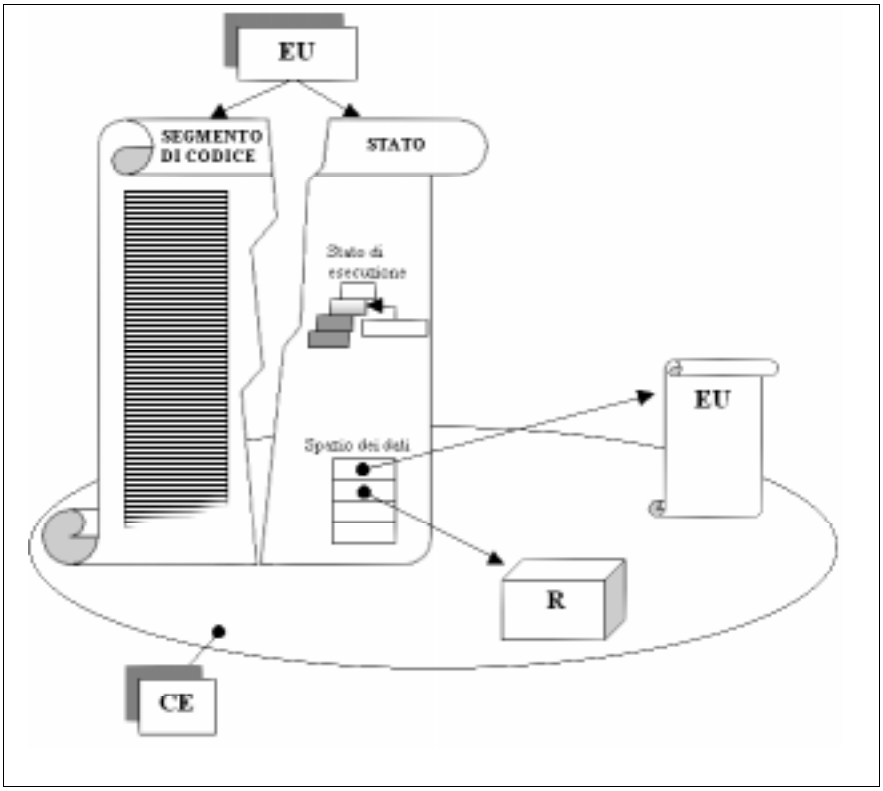

Figura 2-2 Elementi costitutivi di una EU

Nei sistemi a codice mobile, le EU possono essere rilocate su diversi CE. Il modo in cui avviene questo spostamento determina la seguente classificazione dei meccanismi per la mobilità (Figura 2-3).

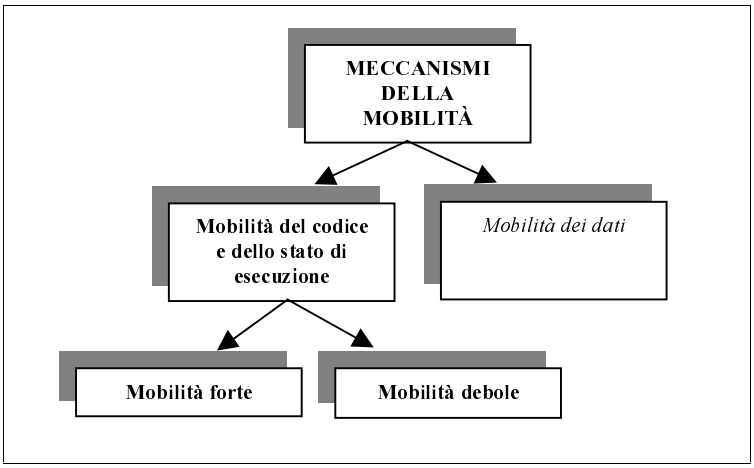

Figura 2-3 Tassonomia dei meccanismi per la mobilità

Vediamo questa suddivisione più in dettaglio.

2.1.1 Mobilità del codice e dello stato di esecuzione.

Considerando una EU è possibile distinguere due tipi di mobilità in funzione dello spostamento degli elementi che la costituiscono:

- *Mobilità forte*: è quella forma di mobilità che consente di migrare contemporaneamente sia lo stato di esecuzione, sia il codice della EU. È supportata da due meccanismi: la migrazione e la clonazione remota. Il meccanismo di *migrazione* sospende la EU, la trasmette al CE destinatario e la ripristina.
  Il meccanismo della clonazione remota, crea una copia della EU in un CE remoto. La differenza rispetto alla migrazione consiste nel fatto che l'EU di origine, non si scollega mai dal suo CE.

Sia la migrazione e che la clonazione possono essere di tipo proattivo cioè la EU decide autonomamente il momento e la destinazione della migrazione; oppure di tipo reattivo, la migrazione di una EU, viene provocata dall'interazione con un'altra EU con cui collabora a causa di qualche tipo di relazione, ad esempio una EU con il compito di instradare le altre EU.

- *Mobilità debole*: è la forma di mobilità che consente di *trasferire solo il segmento di codice* ad un diverso CE, può coinvolgere qualche dato di inizializzazione, ma lo stato di esecuzione non è assolutamente toccato dalla migrazione. I meccanismi che supportano la mobilità debole, possono collegare dinamicamente il codice trasferito ad un EU in esecuzione, oppure creare un nuovo EU. Questi meccanismi si possono distinguere secondo quattro criteri distinti: la direzione del trasferimento del codice; la natura del codice spostato; la sincronizzazione coinvolta ed il momento in cui viene eseguito il codice nel CE di destinazione.
  Direzione del trasferimento del codice: una EU può *caricare* una porzione di codice e lincarla od eseguirla dinamicamente, oppure può semplicemente *consegnarla* ad un altro CE.
  Natura del codice spostato: il codice da spostare può essere sia codice a se stante, sia un frammento di codice. Il codice a se stante può essere usato per istanziare una nuova EU nel CE destinazione, mentre un frammento deve essere inserito dinamicamente in un contesto di codice già in esecuzione.
  La sincronizzazione coinvolta: il trasferimento di codice può essere sincrono o asincrono, dipende se la richiesta del CE di destinazione sospende l'esecuzione oppure no.
  Il momento in cui viene eseguito il codice nel CE di destinazione: nei meccanismi asincroni, l'esecuzione del codice trasferito può avvenire immediatamente oppure può essere

ritardata. L'esecuzione può dunque avvenire solo se si verifica una data condizione (per esempio un evento di applicazione).

2.1.2 Mobilità dello spazio dei dati

Quando una EU migra verso un nuovo CE, il suo spazio dei dati (per esempio l'insieme dei collegamenti alle risorse accessibili), deve essere ripristinato. Questo può significare l'eliminazione di alcuni riferimenti, la creazione di altri, oppure anche la migrazione di particolari risorse. La scelta dipende dai vincoli che pone la natura della risorsa, dal tipo di riferimento possibile e dalla necessità d'uso dell'applicazione.

Identifichiamo una risorsa con una terna composta da: un identificatore unico, il valore della risorsa, ed il suo tipo. Il primo elemento che determina se una risorsa sia trasferibile o meno è proprio il suo tipo; per esempio una risorsa di tipo astratto, come un insieme di dati (ad esempio contenuti in un file), è presumibilmente trasferibile (tale decisione dipende dalle esigenze dell'applicazione, dalla convenienza in termini di prestazioni e dalla segretezza dei contenuti), mentre un tipo di risorsa fisica come una stampante non è ovviamente trasferibile.

Le risorse possono essere riferite dalle EU in tre modi differenti: per identificatore, per valore e per tipo. Il riferimento per identificatore si rende necessario quando una EU deve raggiungere una risorsa che non può essere sostituita da una equivalente, per esempio un particolare archivio anagrafico. Un riferimento per valore, implica che la risorsa originaria possa essere sostituita da una dello stesso tipo e valore, senza alcuna importanza per la sua identità, come, per esempio, una tabella di codici ASCII.

La forma più debole di riferimento è quella per tipo: la risorsa può essere sostituita in qualunque momento da una dello stesso tipo, non importa se di valore od identità differenti. È il caso, ad

esempio, delle risorse di sistema: memoria, schermo ecc. A seconda del tipo di risorsa e di riferimento, sono dunque possibili diversi meccanismi per la mobilità (Figura 2-4).

		Tipo di risorsa	
		Risorse trasferibili	**Risorse fisse**
Tipo di collegamento	**Rif. per Identificatore**	MIGRAZIONE RIF. RETE	RIF. RETE
	Rif. per valore	COPIA MIGRAZIONE RIF. RETE	RIF. RETE
	Rif. per tipo	RIPRISTINO RIF. RETE MIGRAZIONE COPIA	RIF. RETE

Figura 2-4 Classificazione meccanismi per la mobilità dei dati

Se consideriamo *un riferimento per Identificatore, nel caso di risorse trasferibili*, è possibile ricorrere alla migrazione della risorsa insieme alla EU (Figura 2-5).

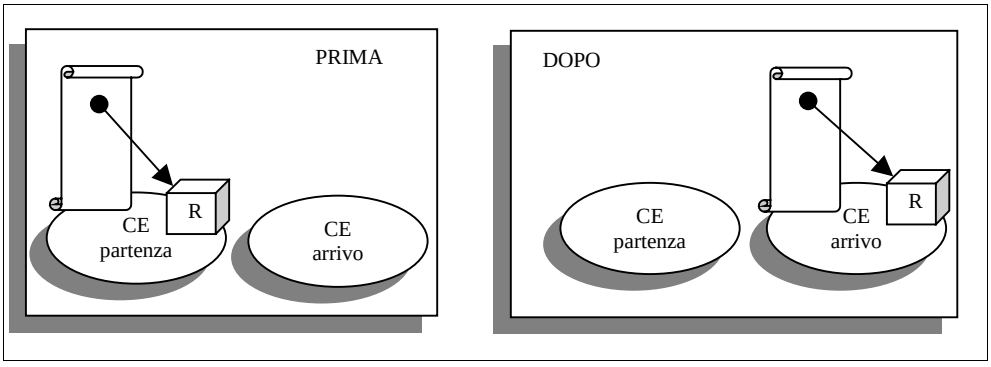

Figura 2-5 Migrazione della risorsa

Per ricorrere a questa soluzione, è necessario valutare i problemi legati alla condivisione di tale risorsa con altre EU ed il costo dello

spostamento, per esempio, in termini di impiego di banda e di tempo. Se si considera ancora *un riferimento per identificatore, ma con risorse fisse*, la soluzione consiste nell'adottare un riferimento attraverso la rete. Ossia è necessario un meccanismo che consenta di usare (leggere, modificare ecc.) la risorsa a distanza (Figura 2-6).

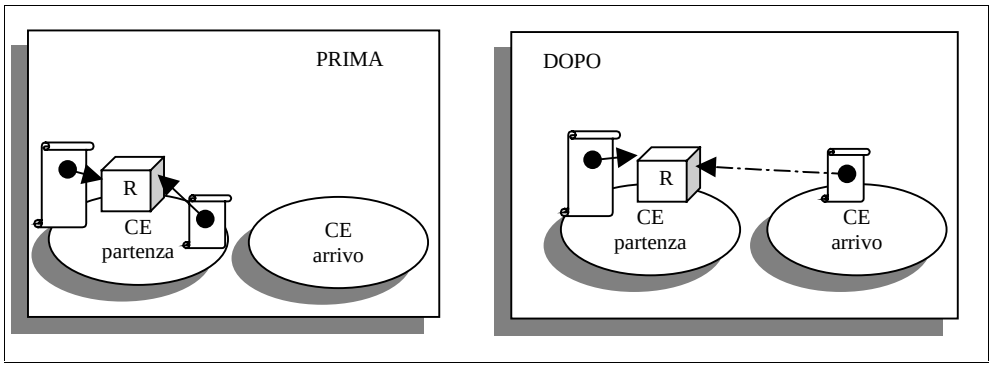

Figura 2-6 Riferimento attraverso la rete

Se si considera *un riferimento per valore, con risorse trasferibili*, il meccanismo più adatto è quello della copia (Figura 2-7). Si possono usare anche la migrazione ed il riferimento attraverso la rete. Mentre quest'ultimo riferimento è l'unico possibile se le risorse riferite per valore sono fisse.

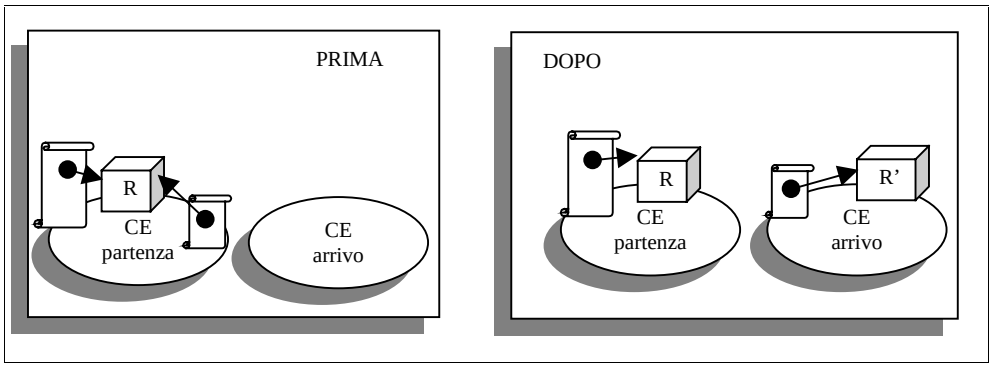

Figura 2-7 Copia della risorsa

Infine *con riferimenti per tipo* il meccanismo più appropriato è quello che consente il ripristino del riferimento ad una risorsa locale dello stesso tipo (Figura 2-8).

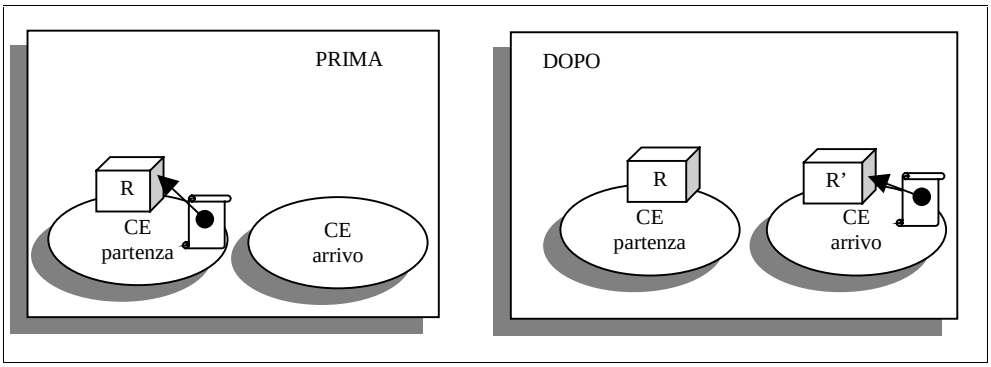

Figura 2-8 Ripristino del riferimento

Il meccanismo più semplice da realizzare, e quindi il più diffuso, prevede la rimozione del riferimento alla risorsa, nella speranza che questa non debba più servire. Ma quando la computazione ha la necessità di utilizzare una risorsa il cui collegamento è stato rimosso, viene lanciato un segnale di allarme (una eccezione) che segnala il fatto e dunque è necessario ripristinare il collegamento, seguendo una delle tecniche sopracitate.

2.2 La mobilità a livello di paradigma di progettazione

I paradigmi di progettazione a codice mobile, consentono di sfruttare i meccanismi della mobilità ed insieme a questi, portano alla realizzazione dei sistemi a mobilità del codice.
Analizziamo i tre paradigmi più significativi secondo la classificazione di [CarPV97].

2.2.1 Paradigma Remote Evaluation (REV)

In questo modello, una entità computazionale ha la conoscenza necessaria per eseguire un servizio, ma non possiede la risorsa che consente di svolgerlo. Però tale risorsa è collocata in una locazione remota. Allora l'entità computazionale invia a tale locazione la propria conoscenza sul servizio con la richiesta di svolgerlo. Nella locazione remota viene eseguito il servizio sulla risorsa seguendo la conoscenza acquisita. Una seconda comunicazione consente, infine, all'entità computazionale di ricevere i risultati.

Questo approccio è utilizzato, per esempio, dalle applicazioni che sfruttano il linguaggio Postcript [ADOBE95]: un processo di videoscrittura, residente su un certo host, invia alla macchina collegata con la stampante (risorsa), non solo il testo da stampare, ma anche le istruzioni (la conoscenza) necessarie a pilotare il processo di stampa.

2.2.2 Paradigma Code on Demand (COD)

In questo paradigma di progettazione si ha una situazione simmetrica a quella del REV. Una entità di esecuzione, residente in una certa locazione, possiede la risorsa da utilizzare, ma non conosce le informazioni necessarie al suo funzionamento. Queste informazioni sono invece presenti in una locazione remota. Dunque l'entità di esecuzione si rivolge alla locazione remota per avere la conoscenza desiderata. Dalla locazione remota viene spedita la conoscenza necessaria e finalmente l'entità di esecuzione può sfruttare la propria risorsa.

L'esempio di applicazione che segue questo modello è costituito dall'Applet Java. Piccole applicazioni che vengono

scaricate dalla rete dinamicamente su richiesta, e che consentono una elaborazione sulle risorse locali.

2.2.3 Paradigma ad Agenti Mobili

In questo paradigma una entità computazionale (l'Agente Mobile), inizia l'esecuzione in una locazione che mette a disposizione alcune risorse. Ad un certo punto, l'entità computazionale ha bisogno di altre risorse che si trovano in locazioni diverse. Allora, migra, verso tali locazioni portando con se la conoscenza di come operare con tali risorse ed i risultati intermedi che man mano ottiene. Questo paradigma è molto diverso dai due precedenti, perché intende per mobilità, lo spostamento dell'entità computazionale stessa, mentre nei due modelli precedenti la mobilità è espressa come spostamento di codice tra due distinte entità computazionali. Considerando questo paradigma molto importante e la massima espressione della mobilità, lo affrontiamo in dettaglio nei paragrafi successivi.

2.3 Agenti Mobili

Un Agente Mobile *è un'entità computazionale che agisce autonomamente in nome di una persona o di una organizzazione* [CheHK95]. Possiede un proprio flusso di esecuzione, è in grado di interagire con le risorse dell'ambiente in cui si trova e con altri agenti: in funzione di queste interazioni, può prendere alcune semplici decisioni che gli consentono di svolgere il compito per cui è stato creato. Non è vincolato al sistema in cui inizia l'esecuzione. Esso possiede la capacità di spostare se stesso da un luogo ad un altro di una rete eterogenea. L'agente con questa abilità, si muove sul sistema che contiene l'oggetto con cui vuole o deve interagire, ed usufruisce dei benefici dell'interazione locale.

Possiamo dunque considerare il ciclo di vita di un agente mobile composto dalle seguenti fasi: nascita, esecuzione, congelamento, terminazione (Figura 2-9). L'agente nasce, alterna fasi di esecuzione a fasi di congelamento (necessarie affinché l'agente possa essere trasferito da una macchina ad un'altra) e, raggiunto l'obiettivo prefissato, cessa la sua esistenza. L'agente viene messo in esecuzione dal "Principal" (per esempio un utente od un altro agente), per conto del quale agisce ("*on behalf of*"). L'unica preoccupazione del Principal è quella di attendere il risultato del compito affidato all'agente. L'agente inizia l'esecuzione sulla macchina in cui viene creato e può decidere di spostarsi in altre locazioni.

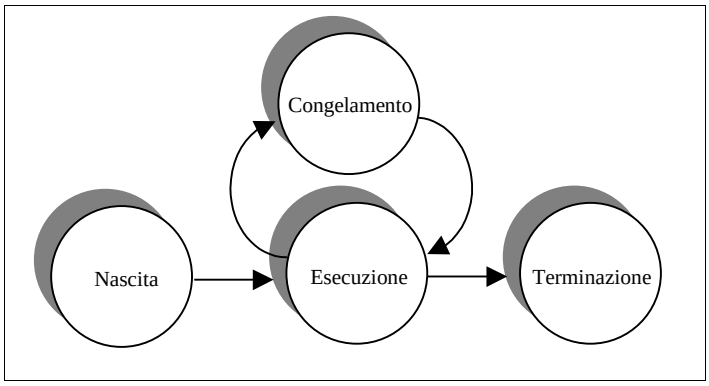

Figura 2-9 Ciclo di vita di un Agente Mobile

Dunque l'agente deve conoscere l'identità del sistema sottostante e dei sistemi che lo circondano, deve poter interagire con le altre entità, siano esse risorse od altri agenti, contenute nel sistema, deve essere in grado di attivare i meccanismi che lo congelano e lo trasportano nel luogo in cui ha deciso di andare.

In sostanza è necessario un "s*istema*" che fornisca all'agente tutti i meccanismi che gli consentano di esprimere pienamente le sue capacità: i meccanismi per la mobilità, i meccanismi per la comunicazione ed i meccanismi per la sicurezza. Vedremo le

caratteristiche che deve possedere tale sistema nei paragrafi successivi.

2.3.1 Vantaggi

Il paradigma ad Agenti Mobili, rappresenta un'idea innovativa ed un tentativo per porre soluzione ad alcuni gravi problemi delle reti distribuite su larga scala, si considerino, ad esempio, quelli dell'ampiezza di banda e delle connessioni intermittenti.

A parità di prestazioni, l'aumento del traffico di rete, comporta un crescente impiego di banda. Il mezzo trasmissivo su cui è costruito il canale di comunicazione, offre una banda limitata, ed ha dei costi di installazione e manutenzione molto alti (si pensi alla comunicazione via satellite). All'aumentare del traffico, è possibile raggiungere la saturazione del canale di comunicazione, cioè è possibile impiegare tutta la banda disponibile. Allora i messaggi trasmessi, vengono accodati in attesa che si liberi il canale, e si accumulano grandi ritardi sui tempi di consegna. Ne consegue un tracollo totale delle prestazioni. Dunque una soluzione, non molto economica, consiste nel cercare di aumentare sempre di più l'ampiezza di banda del mezzo trasmissivo, ed in questa direzione è orientato lo studio di nuove tecnologie; invece, un modo più economico, consiste nel cercare di ottimizzare la comunicazione per liberare il più possibile il canale.

È in questa direzione che si spinge l'utilizzo degli agenti mobili. Gli agenti mobili, consentono di minimizzare il numero di interazioni remote. Il Principal programma un agente per un determinato compito e lo mette in esecuzione. L'agente si sposta impiegando una prima volta il canale trasmissivo. Nel luogo remoto, da lui prescelto, grazie alla sua autonomia, l'agente è in grado di filtrare e raccogliere tutte le informazioni di cui necessita. Solo dopo aver concluso il compito per cui è stato creato, l'agente, può impegnare per la seconda volta il canale di comunicazione, e

portare il risultato del lavoro svolto alla locazione di partenza. Quindi una computazione, anche molto complessa, può essere portata a termine con due sole trasmissioni.

Vediamo ora il problema delle connessioni non affidabili. Si considerino due nodi perfettamente funzionanti, in uno è presente un servitore che offre un determinato servizio, nell'altro, si trova un cliente che vuole usufruire di tale servizio. In un tradizionale approccio cliente/servitore ogni messaggio scambiato porta ad un risultato intermedio che avvicina il cliente alla soluzione del suo problema. Ma se la connessione non è disponibile, pur essendo i nodi in grado di eseguire perfettamente, il percorso verso la soluzione del problema del cliente viene bloccato. Dunque il cliente è fortemente vincolato e ritardato dalla disponibilità del canale.

L'intermittenza del canale, non rappresenta un grosso ostacolo per l'agente mobile, il quale può impegnarlo solo per brevi momenti, poiché svolge il grosso della computazione a livello locale. Il cliente affida l'obiettivo all'agente, che si sposta sul nodo del servitore. L'agente interagisce con il servitore e può raggiungere l'obiettivo prefissato, mentre il canale di comunicazione non è disponibile.

2.3.2 Caratteristiche dei Sistemi ad Agenti Mobili

Un sistema ad agenti mobili deve realizzare l'astrazione del "*mondo degli agenti mobili": un insieme di luoghi distinti che contengono risorse (entità attive e/o passive), e agenti. Dove gli agenti dislocati nei diversi luoghi, possono spostarsi, interagire con le risorse, dialogare tra di loro. Un mondo dove tutto avviene in modo ordinato e sicuro.* Dunque un sistema ad agenti mobili deve essere scalabile. Non deve esistere un limite al numero di luoghi che si possono aggiungere al mondo. I meccanismi che

consentono la creazione dell'astrazione di "luogo" devono essere facilmente installabili e configurabili. È necessario adottare una forma di identificazione dei luoghi, efficiente e flessibile: utilizzando ad esempio un sistema di naming gerarchico.

Ancora, per aumentare la scalabilità del sistema, devono essere presenti meccanismi in grado di gestire e coordinare una qualche forma di replicazione delle risorse. Se una risorsa viene acceduta da un numero troppo grande di agenti, può non riuscire a soddisfare tutti, oppure può richiedere tempi d'accesso talmente elevati da risultare inutilizzabile. La replicazione, consentirebbe di evitare simili situazioni. Per una discussione dettagliata dei problemi di replicazione nei sistemi distribuiti si veda [CouDK94]. Il sistema deve adottare un sistema di nomi anche per gli agenti, che devono essere entità uniche all'interno del loro mondo e per le risorse.

Un'altra caratteristica che deve possedere il sistema ad agenti è l'apertura. Un sistema deve poter interagire anche con sistemi di natura diversa. Infatti la sede di tali sistemi è una rete di dimensione geografica, come Internet, una rete di reti eterogenee, dove chiunque può installare un proprio sistema ad agenti e rendere disponibili risorse preziose. Si devono dunque implementare dei servizi e delle interfacce di comunicazione ed interazione standard, si consideri ad esempio lo sforzo di standardizzazione in questa direzione di OMG [OMG94]. Un ostacolo all'apertura dei sistemi è la sicurezza. I diversi sistemi offrono diversi livelli di fiducia, per una discussione dettagliata sugli aspetti di sicurezza dei sistemi ad agenti mobili, si rimanda al cap. 4.

Strettamente legata alla caratteristica di apertura, troviamo la caratteristica di portabilità. Il sistema deve poter essere installato su una qualunque macchina in rete, indipendentemente dalle architetture hardware o software, affinché possa avere la massima diffusione. Infine è necessario che il sistema implementi uno dei meccanismi per la mobilità discussi nel primo paragrafo (un agente mobile è come una EU), uno o più meccanismi per la

comunicazione tra agenti, di cui vediamo alcuni cenni di seguito, e dei meccanismi per la sicurezza, argomento affrontato nel quarto capitolo.

2.3.3 Comunicazione

Distinguiamo principalmente due forme di comunicazione tra agenti mobili caratterizzate dal livello di reciprocità:

- *interazione stretta*: subentra quando è necessaria una forte collaborazione tra agenti. Può essere realizzata in generale attraverso l'uso di oggetti condivisi (*shared objects*) oppure, più in particolare, mediante l'invocazione diretta dei metodi del partner. Questo secondo modello di interazione stretta, prevede la conoscenza esplicita del tipo e numero degli argomenti dei metodi da invocare. Dunque, in prima analisi, può essere realizzata fornendo tali conoscenze agli agenti in fase di progettazione, quindi viene data una conoscenza statica, poco flessibile e limitata: è una soluzione molto conveniente se gli agenti che comunicano, risiedono sulla stessa macchina, altrimenti comporta l'uso di meccanismi come l'RMI [JAVASOFT] che sono inefficienti. Altrimenti è necessario un protocollo standard che consenta lo scambio di informazioni reciproche sui metodi ed i loro dettagli. Dunque la comunicazione può avvenire solo dopo una fase di negoziazione. Così si penalizzano le prestazioni, ma è l'unico modo per avere comunicazioni dinamiche ed aperte.

- *interazione lasca*: una forma di interazione meno forte, consiste nell'adottare un modello a scambio di messaggi, ("*message passing*"). Per utilizzare questo tipo di comunicazione è sufficiente conoscere l'identificatore unico dell'agente

destinatario. È il sistema sottostante che localizza il destinatario e recapita il messaggio (che può essere un dato od una informazione). L'interazione lasca è molto importante anche per la realizzazione di un'altra forma di comunicazione, quella Anonima: se due agenti non si conoscono, nel senso che non sanno i reciproci nomi, perché non hanno avuto origine nello stesso luogo o non sono "parenti", ma vogliono comunicare, è necessario un meccanismo che sopperisca a questa conoscenza. Dunque è possibile implementare una Blackboard, oppure uno spazio delle Tuple [CaGe89].

Gli agenti, per non contrastare i benefici che derivano dal loro utilizzo, devono cercare di utilizzare il meno possibile le forme di comunicazione remota e privilegiare le interazioni locali.

2.4 Un sistema ad agenti: SOMA

SOMA (Secure and Open Mobile Agent) è un sistema ad agenti mobili progettato presso il D.E.I.S. dell'Università di Bologna [SOMA98]. È costruito interamente in Java ed è stato realizzato cercando di rispettare tutti i punti, descritti nel paragrafo 2.3.2, che caratterizzano un buon sistema ad agenti mobili: analizziamo come sono affrontati nel sistema.

2.4.1 Scalabilità in SOMA

Un primo aspetto di scalabilità in SOMA è disponibile a livello di configurazione: gli ambienti di esecuzione degli agenti sono suddivisi in due livelli logici, e sono permessi inserimenti dinamici di nuove località, senza danneggiare la struttura globale creata in precedenza. I due livelli logici sono espressi in questi termini: il primo è il Place, che rappresenta l'ambiente di esecuzione degli

agenti, e il secondo è il Dominio, che raggruppa un insieme di Place con caratteristiche logiche o fisiche correlate. È possibile inserire nuovi Place e nuovi Domini, in ogni momento, sfruttando la gestione dinamica che permette di aggiornare le località del sistema. Se si presenta la necessità di far nascere un nuovo ambiente di esecuzione logicamente correlato ad alcuni già esistenti, allora è possibile inserire un nuovo Place all'interno di quel Dominio; se invece il nuovo ambiente (o i nuovi ambienti) sono completamente indipendenti da quelli già esistenti, è possibile creare un nuovo raggruppamento che costituisca un Dominio a se stante.

La scalabilità del sistema si rileva anche nell'aspetto della gestione utenti: è possibile concedere l'accesso al sistema a nuovi utenti, indipendentemente dal momento in cui si presenta questa necessità. Infatti, per ragioni di sicurezza, che saranno affrontate in dettaglio nel prossimo capitolo, solo gli utenti autorizzati possono interagire con il sistema; quando si presenta la necessità di inserire nuovi utenti tra quelli autorizzati è sufficiente interagire con l'interfaccia "Users Manager" per inserire i nuovi arrivati nel sistema.

Anche per la gestione degli agenti, in SOMA, si sono seguiti principi di scalabilità: il numero di agenti presenti nel sistema non è vincolato a decisioni prefissate e può aumentare in funzione delle esigenze (il limite è imposto solo dalle risorse hardware). Gli agenti sono identificati attraverso un sistema di naming gerarchico: rispettando i due livelli logici di Place e di Dominio, citati al punto precedente, l'identificatore di ogni agente è rappresentato in questo modo:

- numero intero unico all'interno del Place d'origine;
- nome del Place d'origine su cui ha avuto inizio l'esecuzione;
- nome del Dominio d'origine a cui appartiene il Place d'origine.

Nel sistema è possibile creare sempre nuovi agenti, indipendentemente da quanti già ne esistono: il loro identificare permette di differenziarli, senza ipotesi di località.

2.4.2 Portabilità e apertura in SOMA

L'ambiente di utilizzo degli agenti SOMA è un generico sistema distribuito, aperto ed eterogeneo. Nella progettazione del supporto è stata quindi prevista la possibilità di lavorare all'interno di una rete che collega macchine eterogenee: infatti, come si è accennato all'inizio, il sistema è realizzato completamente in Java, linguaggio particolarmente adatto per la mobilità (capitolo 2). Il byte code, cioè il codice intermedio, in cui sono compilati i programmi Java, consente di essere interpretato da ogni tipo di macchina (per cui esista un interprete) e questa è una condizione indispensabile per lavorare in sistemi eterogenei.

SOMA è anche un sistema aperto: consente l'interazione con agenti di altri sistemi che siano CORBA-compliant. Gli agenti SOMA possono svolgere il ruolo di servitori cioè possono offrire un servizio ad agenti di altri sistemi e, viceversa, hanno la capacità di richiedere un servizio, cioè di comportarsi come clienti di agenti non SOMA. In più, rientrano all'interno dello standard proposto dall'OMG: è consentito ad agenti di altri sistemi (anch'essi conformi allo standard) di spostarsi sul supporto di SOMA e accedere alle primitive messe a disposizione per i propri agenti, e viceversa, gli agenti SOMA possono migrare su altri sistemi.

2.4.3 La mobilità di tipo debole

La mobilità è una caratteristica molto importante per un agente, perché gli consente di continuare la sua esecuzione in un ambiente

diverso da quello attuale, per esempio in quello su cui risiede la risorsa con cui vuole interagire. Abbiamo classificato i tipi di mobilità in due categorie: mobilità forte e mobilità debole, secondo la definizione fornita in 2.1.1.

Il linguaggio Java, con cui è realizzato il supporto di SOMA, non consente all'utente di accedere alle informazioni dello stack e quindi non è possibile catturare lo stato di esecuzione di un agente: possiamo quindi affermare che Java supporta una mobilità di tipo debole. Questo significa che quando un agente decide di migrare su un ambiente remoto, arrivato a destinazione, la sua esecuzione non riprende esattamente dal punto successivo a quello in cui si era fermato, ma da un punto da decidersi a priori. In realtà, è possibile realizzare la mobilità forte anche con il linguaggio Java, catturando le informazioni presenti all'interno della JVM.

- Durante l'esecuzione dell'agente, si devono collezionare i dati necessari alla migrazione, che vengono poi spediti all'ambiente remoto su cui l'agente si sposta; arrivato a destinazione, l'esecuzione riprende da uno dei checkpoint inseriti dal Place mittente, per mantenere la consistenza. Lo svantaggio è che, per coordinare i due ambienti remoti, si deve progettare un post compilatore capace di inserire informazioni aggiuntive nel byte code da trasferire alla macchina destinataria.
- Un altro modo per realizzare la mobilità forte è quello di catturare dalla JVM tutte le informazioni necessarie per la migrazione, ma l'unico modo per farlo, è modificare la macchina virtuale stessa. L'inevitabile conseguenza di questa scelta consiste nella perdita della portabilità della JVM.

Nel sistema SOMA si è scelto di gestire un tipo di mobilità debole: il vantaggio è quello di mantenere la completa compatibilità con la JVM e quindi di sfruttare l'estrema portabilità di Java. La gestione di una mobilità debole è affrontata in SOMA, facendo scegliere

all'agente, prima della migrazione, il punto da cui ripartire nel Place di destinazione; quando un agente vuole spostarsi invoca il comando go(), offerto dal supporto SOMA, e specifica come argomento, il nome di un metodo da cui ricomincerà l'esecuzione (si veda il paragrafo 2.5.5). Questa caratteristica rende il codice di programmazione dell'agente molto strutturato, poiché suddivide i compiti da realizzare in funzione del Place su cui devono essere eseguiti (Figura 2-10). Maggiori dettagli sulla implementazione del comando go() sono forniti nel seguito.

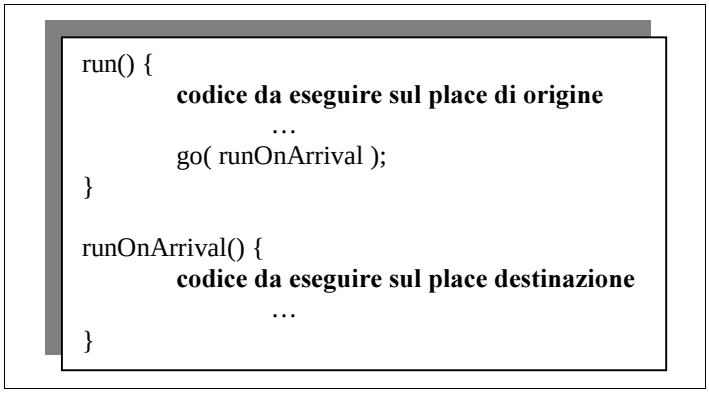

Figura 2-10 L'agente deve specificare il metodo da eseguire dopo la migrazione: si ottiene una programmazione strutturata.

2.4.4 La comunicazione

La comunicazione è un aspetto molto importante dei sistemi ad agenti mobili, perché spesso, per raggiungere l'obiettivo previsto dal Principal, è necessaria la cooperazione tra più agenti. I tipi di comunicazione previsti in SOMA sono tre:

- *condivisione di risorse*: questo tipo di comunicazione prevede un'interazione stretta tra gli agenti partecipanti, imponendo ad

essi di essere presenti contemporaneamente sullo stesso ambiente, ma non impone limiti sul numero di partecipanti.

- *scambio di messaggi*: questo tipo di comunicazione prevede un'interazione lasca tra gli agenti partecipanti. Ogni agente possiede una Mailbox, su cui può ricevere messaggi da tutti e da cui può spedire messaggi destinati solo agli agenti di cui conosce la relativa Mailbox. Lo scambio di messaggi si basa sul fatto che gli agenti sono sempre rintracciabili (a meno che espressamente non richiesto) dal Place su cui hanno iniziato la loro esecuzione (cioè quello che compare nell'identificatore, come spiegato in 2.4.1): è proprio questo Place che ha il compito di far arrivare i messaggi alla Mailbox dell'agente destinatario. La spedizione è asincrona: l'agente affida il messaggio al supporto e continua la sua esecuzione. La ricezione può essere bloccante o meno: infatti, l'agente ha la possibilità di verificare se nella Mailbox è presente un messaggio, oppure può scegliere di bloccarsi in attesa del prossimo.
- *comunicazione anonima*: la blackboard. Gli agenti possono depositare un messaggio nella blackbord e ad esso viene associata una stringa: la conoscenza di questa stringa consente di leggere il messaggio, senza nessuna conoscenza di che lo ha inserito.

2.5 Architettura del sistema SOMA

Il sistema è suddiviso su due livelli gerarchici, che rappresentano delle astrazioni di località, tra cui gli agenti possono spostarsi ed eseguire: il Dominio racchiude un insieme di Place, fisicamente o logicamente correlati, per esempio residenti in una stessa rete locale (LAN). Il Place rappresenta l'ambiente di esecuzione degli

agenti ed è implementato all'interno di una macchina, in cui possono essere presenti anche altri Place.
Tra tutti i Place di uno stesso Dominio, se ne sceglie uno, che deve svolgere i compiti di coordinamento interni al Dominio: questo Place è detto *Gateway*, o *Default Place*, ed ha un ruolo fondamentale nella gestione del sistema. Esso rappresenta il "passaggio" delle comunicazioni del Dominio con l'esterno: ogni tipo di interazione da svolgere con località esterne al Dominio è rivolta in direzione del gateway, che si comporta come intermediario tra gli ambienti interni al Dominio e quelli esterni.
Invece, le interazioni che si svolgono all'interno del Dominio avvengono in modo diretto, per agevolare la gestione intra Dominio, che dovrebbe essere più onerosa. Questi livelli logici in cui è suddiviso il sistema sono fondamentali per garantire una adeguata struttura di sicurezza, che sarà analizzata in dettaglio nel capitolo 4.

Come si è potuto capire, la topologia ha una forte rilevanza sulle interazioni che si possono svolgere all'interno del sistema. Per maggiore chiarezza, analizzeremo le fasi attraversate da un Place durante la sua attivazione e suddivideremo l'esame dell'implementazione del sistema, tra gestione interna al Dominio (intra Dominio) e gestione esterna al Dominio (inter Dominio). Ma prima di scendere nei dettagli implementativi di SOMA, studiamo come è realizzato il soggetto del sistema: l'agente mobile.

2.5.1 L'agente

L'agente è implementato come istanza di una classe astratta "Agent.java". All'interno di questa classe sono raggruppate le caratteristiche comuni di tutti gli agenti: l'identificatore, che permettere di distinguerli in modo univoco (paragrafo 2.4.1), la Mailbox, per gestire lo scambio di messaggi, il valore "Traceable",

che indica se devono essere rintracciabili, un campo, “Start”, in cui inserire il metodo da invocare dopo la migrazione e due strutture dati, le preferenze e le credenziali, utili per la gestione della sicurezza (cap. 4). Tutti gli oggetti che implementano gli agenti ereditano dalla classe Agent.java, in modo da ricevere direttamente tutte le caratteristiche appena citate.

L’agente è un oggetto passivo, e il suo flusso di esecuzione è guidato da un thread, detto *Worker*. Quando un agente nasce viene subito affidato ad un Worker che fa partire la sua esecuzione dal metodo specificato nel campo Start, che di default coincide con il metodo run(), e poi attende la sua terminazione. Nel momento in cui l'agente decide di spostarsi, il suo Worker è sostituito con un altro appositamente creato sull’ambiente di destinazione: anch’esso si limita a lanciare il metodo specificato come argomento nel comando go(), che l’agente trasporta all’interno del suo campo Start. Per inserire un agente nel sistema, l’utente può utilizzare il tool grafico “AgentLauncher”, che è analizzato in dettaglio nel paragrafo 4.2.2.1.

2.5.2 Attivazione di un Place

La topologia del sistema può essere modificata dinamicamente, attraverso il Tool di gestione dinamico. Quando si decide di inserire un Place nel sistema, questo non risulta utilizzabile fintanto che non si attiva. Infatti, non è detto che tutti gli ambienti previsti nel sistema siano attivi, ma possono anche momentaneamente non esserlo, per scelte di gestione, o per motivi tecnici, per esempio è caduto il nodo su cui è fisicamente implementato il Place.

L’attivazione di un ambiente si effettua dal Tool di gestione dinamica, citato sopra, e prevede che il Place esegua le fasi di attivazione descritte in Figura 2-11. Prima di descrivere in dettaglio le operazioni necessarie all’attivazione, si deve sottolineare che

ogni Place conosce l'identità del Gateway del suo Dominio, a cui deve rivolgersi per la gestione inter Dominio.

Figura 2-11 Attivazione di un Place

Il primo passo nel processo di attivazione è quello di collegarsi al Gateway, per ricevere le informazioni riguardanti la configurazione del Dominio. Infatti, nei compiti del Gateway, rientra anche quello di mantenere delle strutture dati aggiornate, da distribuire ai suoi Place. L'insieme delle informazioni del Gateway comprendono, non solo la configurazione del Dominio, ma anche le strutture dati che riguardano la gestione della sicurezza, descritta nel cap. 4. Infatti, nel secondo passo dell'attivazione è prevista la richiesta delle informazioni racchiuse in queste strutture di sicurezza. Una volta che il Place è a conoscenza di tutti i dati necessari, ricevuti

dal Gateway, può diventare parte integrante del Dominio a cui appartiene: quando ha terminato l'ultima fase di connessione con gli altri Place è pronto per ospitare l'esecuzione degli agenti. Il tipo di connessioni create nell'ultima fase, dipendono dalla posizione relativa degli altri Place: infatti è molto diversa la gestione interna al Dominio rispetto a quella esterna al Dominio.

2.5.3 Gestione intra Dominio

La gestione intra Dominio è agevolata dal fatto che i Place all'interno del Dominio sono tutti fisicamente collegati, in modo fisso: questi collegamenti consentono la comunicazione tra gli ambienti e lo spostamento degli agenti (Figura 2-12).

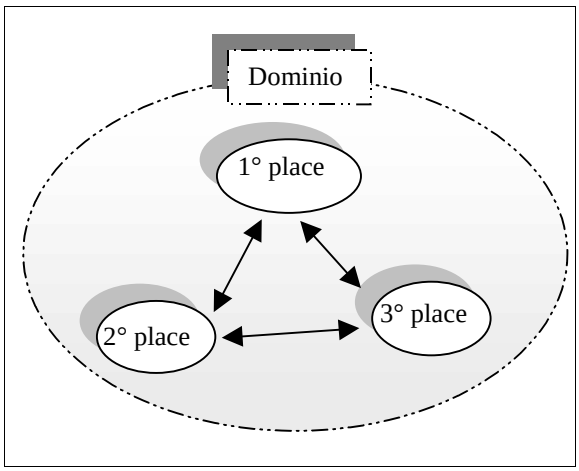

Figura 2-12 I Place interni al Dominio sono tutti collegati tra loro.

L'interazione tra i Place interni allo stesso Dominio avviene direttamente: per esempio, l'interazione può avvenire con una comunicazione diretta tra i Place interessati attraverso il collegamento sempre presente, tra i due ambienti.

In realtà il collegamento fisso tra i Place è implementato con due canali di comunicazione che svolgono due ruoli distinti: il primo serve per lo spostamento degli agenti, che approfondiremo nel paragrafo successivo, mentre il secondo è dedicato allo scambio di "Comandi". I Comandi sono oggetti che racchiudono il codice da eseguire su un Place remoto: sono usati dagli ambienti di esecuzione per sviluppare le fasi di coordinazione necessarie per offrire agli agenti mobili il supporto di esecuzione. Quando un Place riceve un Comando, esegue il suo metodo "exe()", che contiene il codice inserito dall'ambiente mittente per essere svolto nell'ambiente di destinazione. Un esempio di utilizzo dei Comandi, si ha nella fase di attivazione del Place: le informazioni riguardanti la configurazione delle località e gli strumenti di gestione della sicurezza sono richiesti al Gateway con dei comandi appropriati e l'intera fase di aggiornamento delle strutture interne del Place è anch'essa realizzata attraverso i Comandi.

2.5.4 Gestione inter Dominio

La gestione inter Dominio riguarda ogni tipo di interazione che coinvolge più di un Dominio. In realtà, la topologia del sistema è strutturata in modo da raggruppare ambienti correlati, tra i quali dovrebbe svolgersi il maggior numero di interazioni, rendendo minime le interazioni destinate all'esterno del raggruppamento stesso. Per questo motivo, si è scelto di realizzare collegamenti non fissi, come nella gestione intra Dominio, ma da realizzare solo su necessità, by need, per non mantenere collegamenti attivi poco sfruttati.
In più, la suddivisione del sistema nei due livelli logici consente di realizzare una struttura in cui il ruolo del Gateway sia di coordinamento anche per la gestione della sicurezza: ogni tipo di interazione (per esempio lo scambio di Comandi o la migrazione

degli agenti) proveniente dall'esterno del Dominio e destinata ad un Place interno, deve essere prima controllata dal Gateway. Il Dafault Place può quindi essere paragonato ad un cancello di accesso al Dominio (Figura 2-13).

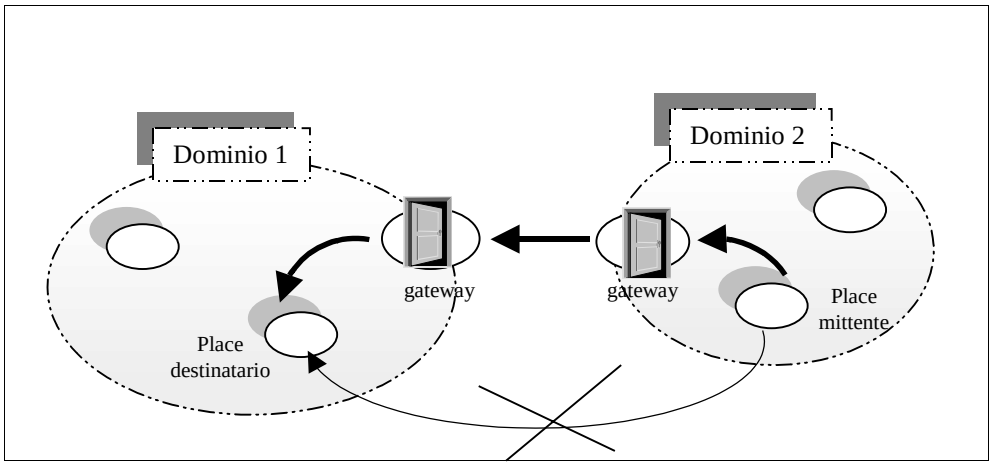

Figura 2-13 Gestione inter Dominio

Il Gateway deve intercettare tutti i tipi di messaggi destinati all'interno del Dominio e poi deve distribuirli ai Place a cui sono realmente destinati. Nella situazione opposta, deve raggruppare tutti i messaggi da inviare all'esterno del Dominio e occuparsi di consegnarli al Gateway del Dominio di destinazione. La migrazione degli agenti rispetta questa funzionalità, quindi un agente in arrivo in un nuovo Dominio deve prima passare sul Default Place, che poi lo consegna al Place di destinazione: al suo interno il Gateway contiene un meccanismo per la gestione dei nomi che gli permette di saper distinguere sempre il Place interno a cui consegnare l'agente (o in generale il messaggio). Il Gateway controlla anche se l'agente rispetta la politica implementata nel Dominio, prima di inserirlo nell'ambiente di esecuzione che desidera raggiungere. In conclusione possiamo paragonare il

Gateway a un proxy intelligente, che si inserisce come collegamento tra il suo Dominio e il mondo esterno.

Per aumentare la scalabilità del sistema, si è realizzata una coordinazione tra i Gateway, scegliendone uno per svolgere un controllo gerarchico sugli altri, che permetta di inserire con facilità nuovi Domini in modo dinamico. Questo è detto Supervisor, e riveste notevole importanza nella gestione dinamica della configurazione delle località e nella gestione utenti (cap. 4), poiché è l'unico a contenere le informazioni globali necessarie a tali gestioni in una struttura dati sempre aggiornata.

2.5.5 Mobilità

Il sistema SOMA è realizzato completamente in Java, che supporta la mobilità debole (paragrafo 2.4.3). La migrazione degli agenti è stata implementata attraverso la creazione del comando go(), che può essere invocato dagli agenti in un qualunque momento. L'effetto di questa invocazione è il seguente: l'esecuzione dell'agente è sospesa, l'oggetto in cui è implementato è serializzato e spedito all'ambiente di destinazione. Il comando go() prevede due argomenti: il primo rappresenta la località di destinazione, mentre il secondo, che è stato già introdotto nel paragrafo 2.4.3, specifica il nome del metodo da cui riprendere l'esecuzione.

La destinazione di uno spostamento può essere espressa in termini di nome del Place su cui spostarsi oppure di nome del Dominio da raggiungere. Il primo caso segnala al sistema la volontà dell'agente di effettuare una migrazione interna al Dominio, che rientra nella gestione intra Dominio. Il secondo caso, invece, specifica che il trasferimento coinvolge un Dominio diverso da quello attuale e quindi si impone il passaggio attraverso il Gateway. Infatti, come spiegato nella gestione inter Dominio, ogni

tipo di interazione (compresa la migrazione) con entità esterne al Dominio deve essere sottoposta al controllo del Default Place. Un agente che desidera cambiare il Dominio di esecuzione deve invocare il comando go(), specificando come destinazione il nome del Dominio da raggiungere: come conseguenza, l'agente è affidato dal Place attuale al suo Gateway, che lo consegna al Gateway del Dominio remoto. Nel nuovo Dominio poi l'agente può scegliere su quale Place riprendere la sua esecuzione. Se durante uno spostamento si verifica un problema che non consente all'agente di raggiungere il suo obiettivo, il sistema segnala all'agente stesso una situazione anomala attraverso l'eccezione "CantGoException". L'invocazione del comando go() può segnalare un evento imprevisto, come per esempio, il fatto che, in una transizione intra Dominio, il Place da raggiungere non sia attivo oppure che, durante una migrazione inter Dominio, si segnali la disattivazione del Gateway locale. Quando è sollevata l'eccezione CantGoException, è possibile eseguire delle azioni alternative che consentano all'agente di aggiornare le sue scelte: per fare questo è sufficiente gestire l'eccezione attraverso i meccanismi messi a disposizione da Java.

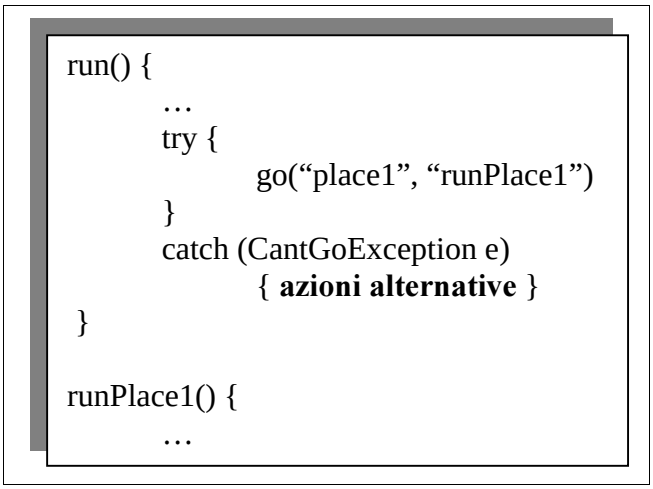

Figura 2-14 Gestione dell'eccezione CantGoException

Quando l'agente è arrivato a destinazione, si consulta il nome del metodo da cui ripartire, che è specificato come secondo argomento del comando go(). La mobilità debole di Java, impone di ricominciare l'esecuzione da un punto preciso e in SOMA si è scelto di farlo corrispondere con l'inizio di un metodo: come conseguenza di questa decisione, è bene che il comando go() risulti l'ultima istruzione di ogni metodo. Il flusso di esecuzione dell'agente è suddiviso in più metodi, su ciascuno sono raggruppate le istruzioni da svolgere su ambienti differenti, come è mostrato in Figura 2-14, rendendo la stesura del codice più lineare e maggiormente leggibile.

2.6 L'installazione di software in SOMA

Per completare il supporto del sistema SOMA è stato introdotto un servizio, offerto a livello applicativo, che consente l'installazione remota di software. Questo servizio è implementato nel pacchetto "SoftInstaller", nella directory di sistema. La struttura del servizio è suddivisa su quattro passi, di cui il primo svolge un ruolo di controllo di sicurezza e gli altri tre realizzano il servizio.

- La schermata iniziale serve per verificare se l'utente che ha richiesto di eseguire l'applicazione ne possiede il permesso (Figura 2-15).

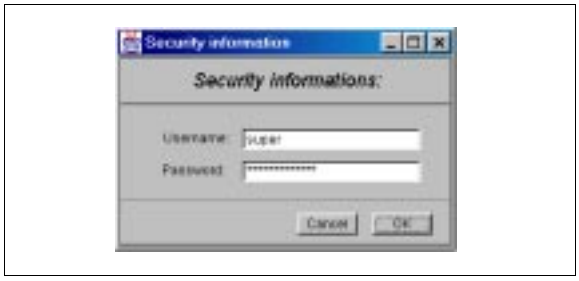

Figura 2-15 Schermata di login

- Nel primo passo (Figura 2-16), si devono fornire le informazioni sul pacchetto sorgente (Packet source), cioè si deve indicare l'ambiente di origine in cui risiede il software da installare. La scelta deve ricadere su un Place del sistema, che sia in quel momento attivo. L'utente deve selezionare dai menù a scelta multipla il nome del Dominio in cui risiede il Place da cui reperire il pacchetto software.

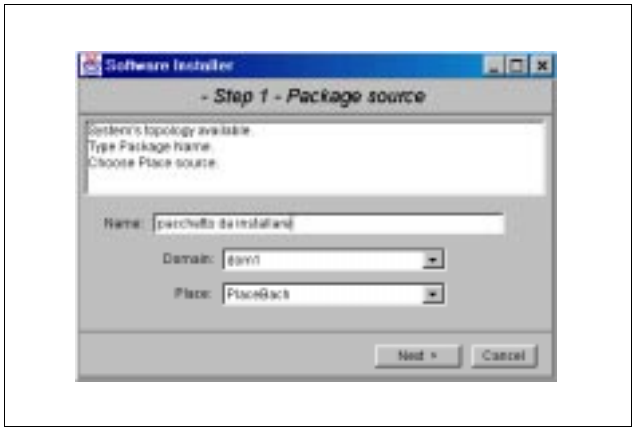

Figura 2-16 Prima schermata

Per visualizzare in tempo reale la configurazione del sistema, si sfrutta un agente ("PlaceList"), che ha il compito di navigare all'interno dei Gateway del sistema, per valutare quali Domini siano attivi e quali invece siano momentaneamente disattivi.
Nel campo "Nome" si deve inserire il nome del pacchetto, o meglio della directory, oggetto della installazione e si può procedere al passo successivo.

- Nel secondo passo (Figura 2-17) si devono indicare i Place ("Package Target Places") su cui installare il pacchetto, individuato al passo precedente. Nella colonna di sinistra sono elencati i Place disponibili (cioè attivi), rappresentati in un formato facilmente leggibile, che indica il nome del Place e il nome del Dominio a cui appartiene (per esempio, se il Place1

appartiene al Dom1, si rappresenta con Place1@Dom1). Nella colonna di destra, invece, sono elencati i Place su cui si desidera effettuare l'installazione (Target Places). L'utente deve decidere quali Place tra quelli disponibili "spostare" nella lista di quelli su cui installare il pacchetto software. I due pulsanti al centro consentono di traslare in un verso o nell'altro, il nome dei Place, tra le due colonne offrendo all'utente un meccanismo intuitivo, per la selezione dei Target Place.

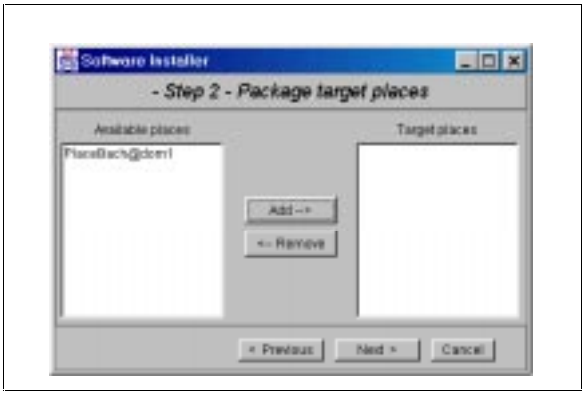

Figura 2-17 Seconda schermata

In caso di necessità è possibile ritornare al passo precedente, oppure, se la selezione dei Place è terminata si può raggiungere l'ultima fase dell'applicazione.

- Il terzo ed ultimo passo (Figura 2-18) è costituto da un report, che elenca i risultati ottenuti. Riporta la lista dei Place obiettivo segnalati nella fase precedente con a fianco l'andamento della operazione: "installazione eseguita" se è andato tutto a buon fine, e "installazione fallita" se si è presentato un qualche problema (per esempio se il Place obiettivo è risultato non raggiungibile).

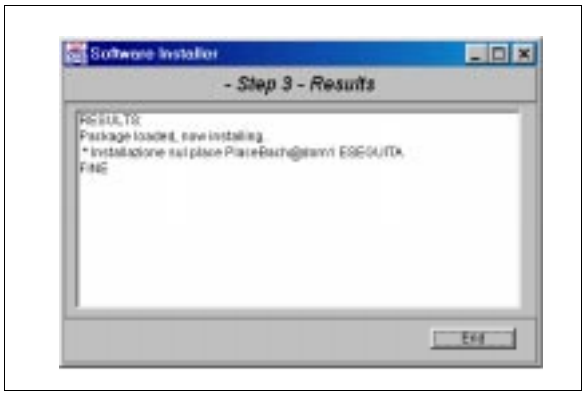

Figura 2-18 Terza schermata

L'applicazione si svolge attraverso la cooperazione di tre agenti: l'agente "Installer" è quello principale, che gestisce l'intefaccia grafica, lanciato dall'utente attraverso il tool "AgentLauncher"; il secondo agente è quello che esamina la configurazione del sistema, il "PlaceList", già citato nella descrizione del primo passo; il terzo è l'agente "installAgent", che svolge realmente il lavoro di installazione.

Quando l'utente accede all'ultimo passo, l'agente installatore è inviato prima sul Place sorgente e poi su tutti i Place della lista fornita dall'utente per effettuare l'installazione remota. Nella prima migrazione, raggiunge l'ambiente in cui risiede il pacchetto che deve copiare: per evitare di fargli trasportare una quantità enorme di byte, l'installatore compatta la directory sorgente (scendendo nei dettagli, invoca la *exec* del comando *jar* con i relativi parametri). Inserisce il pacchetto compattato in una sua struttura dati interna e si sposta sul primo Place della lista, per eseguirvi l'installazione. Quando è arrivato a destinazione, l'agente scompatta il pacchetto (eseguendo al contrario le operazioni precedenti) e, ad installazione completata, si sposta sull'ambiente successivo, fino all'esaurimento della lista.

Cap.3 La sicurezza in Java

3.1 Java come linguaggio ideale per la mobilità

3.1.1 Introduzione alle tecnologie a codice mobile

Per mobilità del codice si intende la possibilità di spostare da un nodo ad un altro dei componenti di un'applicazione, cioè *"la capacità di cambiare dinamicamente il collegamento tra frammenti di codice e la locazione in cui sono eseguiti"* [CarPV97].

Molto lavoro è stato fatto sulla mobilità, dalla ricerca nel campo dei sistemi operativi distribuiti. In tale area il problema principale è supportare la migrazione di processi attivi e di oggetti, a livello di sistema operativo. In particolare la migrazione di processi comporta il trasferimento di un processo di sistema operativo dalla macchina su cui sta eseguendo, ad un'altra. I meccanismi di migrazione gestiscono il legame tra i processi e i loro ambienti di esecuzione (per esempio file descriptor aperti e variabili d'ambiente) per consentire al processo di riprendere l'esecuzione nell'ambiente remoto. Lo scopo principale per cui si vogliono trasferire i processi a livello di sistema operativo è il "load balancing" cioè l'uso equo delle risorse dei nodi di una rete. Perciò la maggior parte delle soluzioni trovate in tale senso forniscono una migrazione trasparente dei processi.

Il programmatore non ha il controllo e neppure la visibilità di tutto il processo di migrazione. Un esempio di sistema che fornisce migrazione trasparente è COOL [LeaJP93] capace di muovere oggetti senza intervento o conoscenza dell'utente.

La migrazione di processi ed oggetti risolve problemi di host scarsamente accoppiati in sistemi distribuiti su piccola scala. Ma risulta insufficiente quando applicata a reti estese su larga scala. Tuttavia fornisce il punto di partenza per lo sviluppo di una nuova varietà di sistemi che implementano avanzate forme mobilità del codice. Questi sistemi introducono molte innovazioni rispetto agli approcci di migrazioni trasparenti.

La mobilità del codice è estesa su scala Internet: sono sistemi che operano in configurazioni su larga scala, ove le reti sono composte da host eterogenei, gestite da autorità differenti con diversi livelli di fiducia ed i collegamenti hanno diverse ampiezze di banda (connessioni senza cavo). Quindi, si considerano sistemi eterogenei, aperti e scalabili.

A livello di programmazione vi è visibilità della locazione: la locazione ha un impatto forte sulla progettazione e implementazione di applicazioni distribuite. Le applicazioni mobili sono consapevoli della locazione e possono prendere delle decisioni in base a tale conoscenza.

La mobilità è sotto controllo del programmatore: il programmatore ha a disposizione meccanismi ed astrazioni che abilitano la spedizione e il caricamento di frammenti di codice (o anche interi componenti) da, verso nodi remoti. Il supporto a tempo di esecuzione fornisce le funzionalità di base, ma non ha alcun controllo sulle politiche di migrazione.

Consideriamo ora gli aspetti salienti della mobilità per un linguaggio.

3.1.2 Le caratteristiche di un linguaggio per la mobilità

La natura dei sistemi aperti, distribuiti e multipiattaforma, che sono il campo d'azione per la mobilità, impone forti vincoli sulle caratteristiche che deve possedere un linguaggio per la mobilità. Di

questi vincoli, i più importanti sono portabilità e sicurezza. I vincoli di sicurezza, sono necessari ad assicurare che il codice proveniente da locazioni remote, eseguito localmente, senza alcun controllo diretto da parte dell'utente, non danneggi alcuna risorsa di sistema. Vedremo maggiori dettagli sulle caratteristiche di sicurezza necessarie ad un linguaggio per la mobilità nei paragrafi successivi, in riferimento a Java. Di seguito ci concentriamo maggiormente sulle caratteristiche più espressamente legate alla capacità di spostare del codice.

La piattaforma target è un computer su una rete, che può essere una LAN o Internet, quindi un insieme eterogeneno di macchine. Lo scopo è scrivere il codice mobile una volta sola e quindi eseguirlo su ogni macchina della rete, senza conoscere caratteristiche hardware o software di ognuna. In questo senso, è importante la scelta della giusta strategia di esecuzione: un linguaggio interpretato può supportare facilmente l'esecuzione in tale ambiente eterogeneo, se in ogni macchina coinvolta è presente un interprete. Invece un approccio compilato forzerebbe il supporto run-time della macchina mittente a conoscere la piattaforma della macchina destinatario per trasmettere un codice nativo appropriato e se la piattaforma di destinazione fosse sconosciuta si dovrebbero spedire tutti i possibili codici nativi al destinatario. La scelta di un linguaggio interpretato risulta dunque più adeguata in termini di portabilità, ma presenta il grosso svantaggio delle basse prestazioni legate ai tempi di interpretazione ed esecuzione.

Un buon compromesso consiste nell'adottare un approccio ibrido. Si compilano i programmi in un linguaggio intermedio che viene usato per la trasmissione e l'interpretazione. Con tale soluzione, il codice sorgente di alto livello, viene compilato in un codice intermedio portabile e di più basso livello, progettato per migliorare efficienza, sicurezza, trasmissione ed esecuzione.

Un ruolo importante nella mobilità è rivestito dalla visibilità e dalle regole di risoluzione dei nomi di un linguaggio. La visibilità

di un identificatore è data dal range di istruzioni da cui è conosciuto. Le regole per la risoluzione dei nomi determinano quale entità di elaborazione sia legata ad ogni identificatore in ogni punto di un dato programma. La maggior parte dei linguaggi utilizza regole di visibilità statica ciò significa che la visibilità del nome di una variabile è determinata dalla struttura lessicale del programma. La risoluzione dei nomi è un problema molto sentito dai linguaggi che supportano la mobilità. Infatti durante l'esecuzione di codice mobile, i nomi che appaiono in esso possono essere legati ad entità che potrebbero essere situate in ambienti computazionali remoti. La risoluzione dei nomi può essere eseguita automaticamente dal supporto run-time o si può fornire un aggancio al supporto run-time del linguaggio per consentire al programmatore di definire proprie regole di risoluzione.

Strettamente collegati al problema della risoluzione dei nomi, troviamo i problemi di collegamento dinamico: cioè come collegare del codice remoto ad un'applicazione, o come collegarsi alle risorse locali. Ad esempio, molti sistemi operativi supportano librerie collegate dinamicamente (DLL). L'idea alla base delle DLL è che tali librerie non debbano essere collegate al programma principale a tempo di compilazione, ma nel momento in cui vi si accede per la prima volta. Il collegamento dinamico di codice remoto estende, in modo naturale, la nozione di collegamento differito delle DLL, alle applicazioni di rete. La seconda forma di collegamento dinamico, quello alle risorse locali, implica che quando del codice si sposta da un ambiente di esecuzione ad un altro, deve essere in grado di accedere alle risorse (file o librerie per esempio) localizzate nell'ambiente di destinazione. Un linguaggio per la mobilità del codice deve superare tutti questi problemi.

3.1.3 Java e la mobilità

Vediamo quali sono le caratteristiche che rendono Java, linguaggio sviluppato nel 1991 nei laboratori di SUN Microsystems, estremamente interessante per la mobilità [ArnG96].

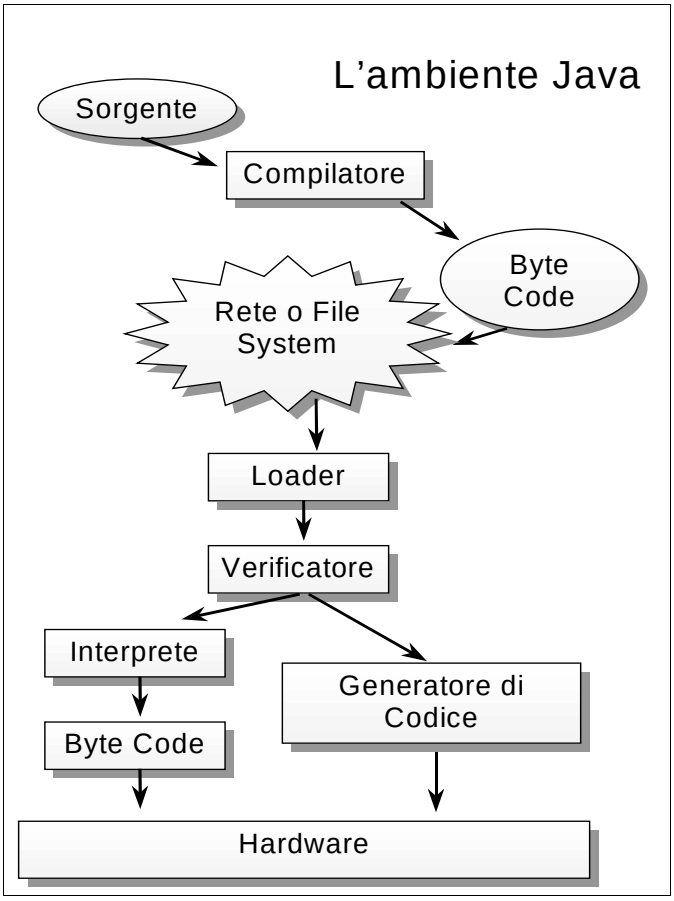

Figura 3-1 Architettura di Java

Java rispetta i due vincoli necessari ad un linguaggio per la mobilità, la portabilità e la sicurezza. La portabilità dei programmi Java è possibile perché il codice sorgente di una applicazione Java

viene compilato in una forma intermedia chiamata "bytecode" (Figura 3-1). Il bytecode è un flusso ottimizzato di istruzioni di basso livello, interpretate da un "processore virtuale", la Java Virtual Machine (JVM). È la JVM che dialoga direttamente con le risorse del sistema sottostante e dunque fornisce il livello d'astrazione necessario a rendere il linguaggio indipendente dalla piattaforma. Vi sono implementazioni di JVM per la maggior parte delle architetture esistenti. Java è un linguaggio ad oggetti che supporta la serializzazione. Un oggetto è l'astrazione che indica l'insieme dei dati e dei "metodi" (funzioni) usati per accedervi. La serializzazione è la trasformazione dell'immagine fisica dell'oggetto in una sequenza di byte. La deserializzazione è l'operazione inversa. Il meccanismo della serializzazione ed il confinamento offerto dagli oggetti, rendono semplice lo spostamento di codice e dati da una macchina ad un'altra. Il mittente trasforma un oggetto in una sequenza di byte (lo serializza) e trasmettere i byte su un canale di comunicazione (socket); il destinatario, completata la ricezione, opera la deserializzazione e si trova con codice e dati che tale oggetto racchiudeva.

Un'altra caratteristica importante per la costruzione di applicazioni mobili è la possibilità di collegamento dinamico alle risorse locali ed al codice remoto (cfr. par precedente). Il caricamento delle varie classi che compongono una applicazione Java avviene a tempo di esecuzione tramite il "class loader", un componente parte integrante della macchina virtuale Java. Le classi vengono caricate e collegate solo quando servono. Il class loader di default, carica le classi dal file system della macchina locale, ma è possibile scrivere un class loader personalizzato che implementi altre politiche, ad esempio, affinché le classi vengano recuperate da una rete eterogenea. Una propria applicazione Java può scaricare e collegare codice proveniente dalla rete ma la risoluzione dei nomi delle classi scaricate in modo dinamico deve essere programmata

esplicitamente. Java consente di fare questo attraverso la personalizzazione del class loader.

Ancora, Java offre molti strumenti semplici e potenti per la realizzazione di applicazioni distribuite in generale, in particolare per quelle a codice mobile. Sono presenti molte classi per la gestione uniforme dei canali di comunicazione e dell'I/O. Uniformità che si ottiene mediante l'astrazione di "stream": una interfaccia comune che nasconde l'implementazione dei canali di comunicazione. Lo stream può essere costruito su standard input (da consolle), output (su schermo) ed error, sulle socket, sui file, sulle stampanti, su tutti i dispositivi in grado di ricevere e trasmettere flussi di dati. Sono presenti le classi per il supporto dei thread, (processi leggeri che condividono lo spazio dei dati) e di tutti gli strumenti per la loro sincronizzazione (semafori) e coordinazione (segnali).

Java è ben integrabile con il web ed accessibile. Java consente di scrivere semplici applicazioni, le applet, che si integrano perfettamente con il linguaggio html [Kor98]. Per inserire un'applet in un documento html è sufficiente utilizzare un'istruzione apposita, il TAG APPLET. I maggiori browser disponibili sul mercato (gli interpreti del linguaggio html), sono stati estesi con una macchina virtuale Java. Questa JVM possiede un class loader che preleva automaticamente dalla rete le classi che costituiscono l'applet. Ogni volta che il browser accede ad una pagina che contiene il TAG APPLET, il class loader della JVM si collega alla locazione da cui proviene la pagina e preleva la classe che costituisce l'applet. Poi la JVM provvede all'esecuzione del codice. L'integrazione della JVM con i browser ha contribuito ad una ampia diffusione di Java sulla rete più importante per dimensioni: Internet.

In Java è stata studiata una robusta e flessibile architettura di sicurezza, sia a livello di linguaggio che a livello di componenti. Le prime applicazioni Java, le applet pongono seri problemi di

sicurezza, sono programmi che vengono eseguiti in modo automatico all'insaputa di chi accede alle pagine html. Per questo motivo ad esse, nella prima versione del linguaggio Java, era consentito l'uso di pochissime risorse, secondo il principio per cui una applicazione che può fare poco o nulla può fare limitati danni. Però una applicazione così concepita risultava poco utilizzabile. Dunque si è cercato di estendere le risorse usabili da un'applet secondo ben determinati modelli di sicurezza. Lo studio di sicurezza, concentrato sulle applet, è stato poi esteso alle applicazioni. Vedremo i modelli di sicurezza utilizzati in Java in dettaglio nei paragrafi successivi.

3.2 I costrutti di sicurezza del linguaggio Java

Analizziamo i costrutti che ci permettono di definire Java un linguaggio sicuro. In Java esiste un livello di accesso che controlla il riferimento ad ogni entità intesa come oggetto o elemento di dato primitivo. I livelli di accesso possono essere:

- *privato* (private): l'accesso all'entità è consentito solo al codice contenuto nella classe che la definisce.
- *standard* (default): l'accesso all'entità è consentito: al codice contenuto nella classe che la definisce; alle classi appartenenti allo stesso package.
- *protetto* (protected): l'accesso all'entità è consentito: al codice contenuto nella classe che la definisce; alle classi appartenenti allo stesso package; alle sottoclassi della classe che definisce l'entità.
- *pubblico* (public): l'accesso è consentito a tutte le classi.

Anche nel linguaggio C++, esistono simili regole di accesso, ma il supporto a tempo di esecuzione del C++ non interviene se, ad

esempio, si converte un riferimento ad una classe, in un puntatore arbitrario alla memoria che consente di accedere a tutte le locazioni, incluse quelle delle entità protette. Per evitare simili comportamenti, in Java non esiste l'aritmetica dei puntatori. Java non possiede la nozione di puntatore. In questo modo nessuna applicazione o applet java può accedere liberamente alla memoria. La gestione dei riferimenti avviene in modo trasparente. Quando un oggetto non è più referenziato non rimane a lungo in memoria, perché viene eliminato da una routine di "garbage collection" eseguita periodicamente dalla JVM.

Java possiede la clausola final. Le variabili dichiarate final sono trattate come costanti. Una volta inizializzate, diventano immutabili. Se si dichiara final un metodo, ereditando l'oggetto che contiene tale metodo, non è possibile sovrascriverlo. Ancora non è possibile ereditare da una classe se questa è final. Le stringhe in Java sono costanti.

Tutte le variabili istanziate sono automaticamente inizializzate. In questo modo si evita la lettura di una variabile non inizializzata, cioè si evita la lettura di locazioni casuali di memoria. Utilizzando molte variabili non inizializzate si potrebbe vedere un'intera area di memoria. I limiti di un array vengono controllati ad ogni accesso all'array stesso. Se così non fosse sarebbe possibile modificare il contenuto della memoria adiacente a quella allocata per l'array.

Gli oggetti non possono essere convertiti (cast) in modo arbitrario. Un oggetto può essere convertito solo in un altro oggetto appartenente alla sua superclasse o sottoclasse.

Un oggetto può essere serializzabile o meno. La serializzazione consente di scrivere un oggetto come una sequenza di byte. Se un oggetto viene serializzato, le regole che definiscono il controllo di accesso alle entità in esso contenute (private, public ecc.) non si riescono più a garantire. Per esempio la sequenza di byte può essere scritta su file, ed il file può essere letto da programmi non Java. La scelta se esporre un oggetto a questo tipo di attacco viene

lasciata all'utente, che quindi può specificare se un oggetto sia serializzabile o meno.

Lo scopo di tutti i vincoli precedenti è di proteggere la macchina di un utente da pezzi di codice "malvagi" e non viceversa. Infatti tutti questi accorgimenti non servirebbero a nulla se per esempio si utilizzasse un programma esterno alla macchina virtuale Java, per la scansione di tutta la memoria di un calcolatore (come un debugger). Il controllo sul rispetto delle regole precedenti avviene in tre momenti separati: a tempo di compilazione, a tempo di collegamento (cioè quando una classe viene caricata nella macchina virtuale) e a tempo di esecuzione.

3.2.1 Controlli a tempo di compilazione

Il compilatore Java è il primo componente che, ad esempio, si occupa del rispetto delle regole di sicurezza del linguaggio. Non è in grado però di intervenire sul rispetto dei limiti di un array e sulla conversione illegale tra classi non omogenee o sul corretto passaggio di un riferimento ad oggetto.

3.2.2 Controlli a tempo di collegamento

Il secondo componente che si occupa del rispetto delle regole del linguaggio è il "verificatore di bytecode". Dopo che il compilatore ha prodotto il bytecode, il programma è pronto per essere caricato nella macchina virtuale.

Lo scopo del verificatore di bytecode è di scoprire se il compilatore ha prodotto codice che contravviene alle regole di sicurezza del linguaggio. Infatti il compilatore potrebbe essere stato modificato per produrre bytecode illegale, o più semplicemente si potrebbe essere verificata una situazione come la seguente.

Consideriamo due classi, una classe A ed una classe B, in relazione di uso: B usa A per leggere il valore di una proprietà di A, per esempio un intero. La proprietà della classe A è definita pubblica (public int). Il codice sorgente della classe A viene modificato: la proprietà viene resa privata (private int). Si ricompila solo la classe A, il compilatore non segnala errori. Il risultato è che la classe B può continuare a leggere il valore dell'intero anche se questo è diventato privato. L'accesso illecito di B sulla proprietà di A viene denunciato dal verificatore di bytecode.

Il verificatore di bytecode (BV) è una parte interna della macchina virtuale Java, non possiede una interfaccia, non è dunque possibile accedervi. Il BV esamina automaticamente gli oggetti costruiti dal class loader. In particolare garantisce che:

- il file .class abbia il formato corretto: la definizione del formato dei file .class si può trovare nelle specifiche della macchina virtuale Java;
- non si erediti da classi final;
- ogni classe abbia una sola classe padre (l'ereditarietà multipla non è consentita per le classi);
- non ci siano conversioni illegali dei tipi di dato primitivi (per esempio da Oggetto a intero);
- non ci siano conversioni illegali tra oggetti;
- non si ecceda nello stack degli operandi: in Java ci sono due stack per ogni thread. Il primo è lo stack tradizionale contenente i record di attivazione dei metodi, lo stack dei dati. Il BV non può prevenire la saturazione di tale stack, si pensi ad una serie infinita di chiamate ricorsive. Mentre il secondo è lo stack degli operandi, necessario per ogni invocazione di un metodo. Lo stack degli operandi contiene i valori su cui opera il bytecode. È su di questo che il BV esegue i controlli.

3.2.3 Controlli a tempo di esecuzione

Analogamente al compilatore, anche il verificatore di bytecode non può garantire tutte le regole di sicurezza del linguaggio. È la macchina virtuale che si occupa dei controlli a tempo di esecuzione e più precisamente dell'accesso agli array e della conversione di oggetti. Si consideri, per esempio, il metodo in figura Figura 3-2.

```
void initArray(int a[], int nItems) {
        for(int i=0;i<nItems;i++) {
                a[i]=0;
        }
}
```

Figura 3-2 Inizializzazione di un array

Il verificatore di bytecode non può segnalare un errore perche array ed nItems sono passati come argomento al metodo. Solo un controllo a tempo di esecuzione può garantire che nItems non ecceda la dimensione dell'array. Se questo accade la JVM lancia l'eccezione *ArrayIndexOutOfBoundException.*

Nel caso in cui avvenga un cast tra oggetti che appartengono a classi scorrelate, viene sollevata l'eccezione ClassCastException. Ciò vale anche per le interfacce che sono considerate di tipo oggetto.

3.3 Il modello di sicurezza di Java

3.3.1 La Sandbox

L'idea alla base del modello di sicurezza di Java è il confinamento di un programma non fidato, per esempio proveniente dalla rete, in un'area ben delimitata detta Sandbox. In questa area sono messe a disposizione un numero minimo di risorse (CPU, schermo, mouse ed una parte di memoria), ma sufficienti ad un programma per funzionare. In funzione del livello di fiducia accordato al programma è possibile estendere il numero di risorse della Sandbox. Si può ottenere quindi una Sandbox che includa l'accesso ad un file del file system, oppure, in caso di totale fiducia, che garantisca l'accesso anche a tutte le risorse del sistema.

Il modello della Sandbox viene realizzato grazie a particolari meccanismi illustrati in Figura 3-3.

Il verificatore di bytecode (BV)
Abbiamo già visto di che cosa si occupi, nella figura si evidenzia come non tutte le classi siano soggette alla verifica del bytecode.

Il class loader
Nella versione Java 1.0, la variabile CLASSPATH è discriminante riguardo alla fiducia da garantire ad un programma: le classi presenti nel CLASSPATH sono considerate fidate, le altre sono soggette alle regole di una Sandbox dalle risorse minime: CPU, una porzione di memoria, schermo, mouse e socket con l'host di origine del codice. Anche per la versione Java 1.1 le classi nel CLASSPATH sono considerate fidate. La differenza che introduce è la possibilità di "firmare" le classi che provengono dalla rete,

dunque assegnare una Sandbox estesa in funzione del livello di fiducia che si vuole prestare ad una classe firmata.

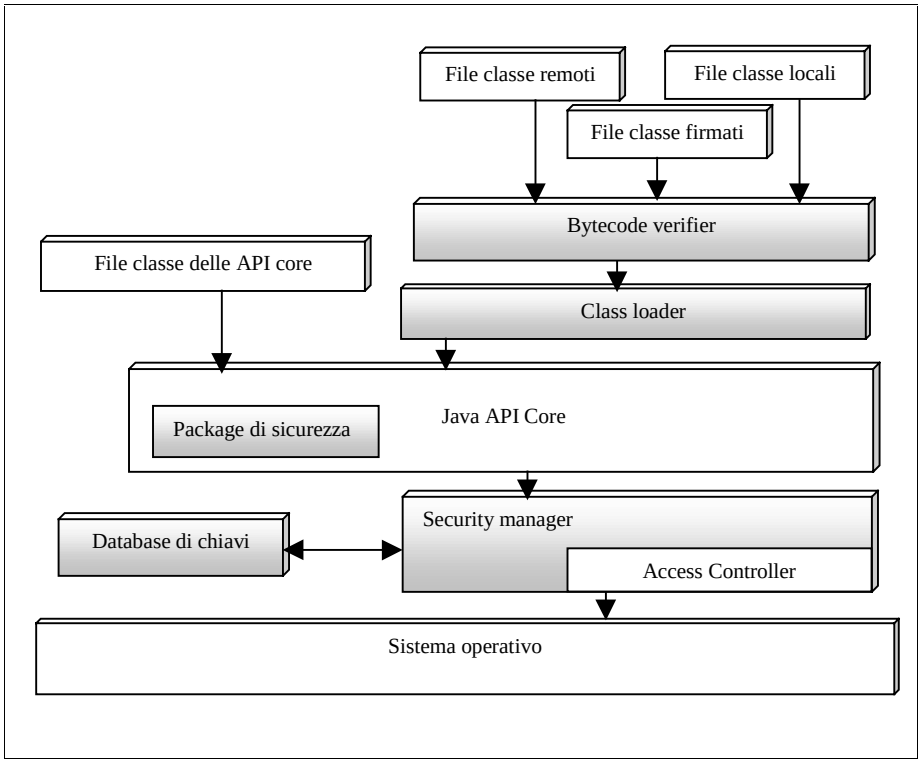

Figura 3-3 Architettura di sicurezza di Java 1.2

Infine nella versione Java 1.2, anche le classi presenti nel CLASSPATH possono essere soggette alle regole di una Sandbox che può essere estesa a piacere con una granularità molto fine [Oak98].

L'Access Controller

In Java 1.2 consente o previene la maggior parte degli accessi alle API del sistema operativo. L'intervento di questo componente introduce un grosso "overhead" sull'applicazione. Nelle versioni precedenti non è presente.

Il Security Manager
È l'interfaccia principale tra le API core ed il sistema operativo. Ha la responsabilità ultima per consentire o vietare l'accesso alle risorse. In Java 1.2 il Security Manager usa il controllore d'accesso per quasi tutte le decisioni. In Java 1.0 e 1.1 è il solo responsabile di tali decisioni.

Il package di sicurezza
È contenuto nel package "java.security". Fornisce molti strumenti per l'uso della crittografa, in particolare per:

- effettuare hash di messaggi;
- gestire chiavi e certificati;
- creare firme digitali;
- cifrare.

Tale package non è presente in Java 1.0.

Il database delle chiavi
È un insieme di chiavi crittografiche utilizzate dal Security Manager e dall'Access Controller per verificare le firme digitali che accompagnano una classe firmata. Può essere implementato da un file oppure da un database server.

3.3.2 Il Class Loader (CL)

Il Class Loader riveste un ruolo molto importante nella sicurezza di Java, per due motivi principali. Il primo è che si tratta dell'unico componente che conosce alcune informazione della classe che deve essere caricata nella Macchina Virtuale: il luogo d'origine e se la classe sia firmata o meno. Su queste informazioni si basa la politica

di sicurezza (chi può fare cosa) che il Security Manager e l'Access Controller dovranno far rispettare.

Il secondo riguarda lo spazio dei nomi di Java. Si consideri la situazione in cui vengono scaricate due classi, con lo stesso nome da due locazioni diverse della rete, dal medesimo CL. La seconda potrebbe sostituirsi alla prima compromettendo il comportamento dell'applicazione (costituita da più classi) in cui si inserisce. Non ci sono problemi a distinguere classi omonime con uno stesso CL, se ciascuna è firmata. Ma l'operazione di verifica della firma eseguita su ogni classe, con il solo scopo di separare i nomi, comporta una eccessiva diminuzione delle prestazioni. Dunque un buon compromesso, tra prestazioni e sicurezza della separazione dei nomi, consiste nell'associare alle classi scaricate da locazioni differenti diverse istanze di CL.

Vediamo in dettaglio come funziona il CL. Quando la macchina virtuale deve reperire una classe si rivolge al CL, il quale esegue le seguenti operazioni [Oak98]:

1. ricerca in memoria se la classe era stata caricata in precedenza, e se trova l'oggetto corrispondente, lo restituisce subito;
2. passo opzionale: consulta il Security Manager per sapere se il programma può accedere alla classe in questione. Se non è così lancia una SecurityException;
3. se non ha trovato l'oggetto al passo 1, consulta un CL interno per verificare se è una classe di sistema, e in tale caso la carica. Altrimenti cerca la classe nel CLASSPATH. La ricerca tra le classi di sistema viene eseguita sempre come prima operazione per evitare il pericolo delle sostituzioni;
4. passo opzionale: consulta il Security Manager per sapere se il programma può accedere alla classe in questione. Se non è così si lancia una SecurityException;
5. se non trova il file classe nel CLASSPATH, lo può cercare in luoghi differenti. Se lo trova lo legge e lo inserisce in un array

di byte, altrimenti scatta la “ClassNotFoundException”. Il modo in cui viene letta la classe (da file o da socket per esempio) e trasformata in un array di byte, differenzia i vari CL.

6. passa i byte di codice al Bytecode Verifier;

7. crea un oggetto di tipo “Class” dal bytecode, a cui è associato anche il nome del CL che lo ha caricato;
8. risolve l’oggetto, cioè trova e carica anche tutte le altre classi referenziate subito dall’oggetto stesso.

Tra i CL predefiniti, quello che ci interessa maggiormente è il Secure Class Loader. Introdotto con Java 1.2, il Secure Class Loader consente di associare un “Dominio di protezione” ad ogni classe che carica da un URL. I Domini di protezione sono gli elementi fondamentali su cui si basa l’Access Controller, li vedremo più dettagliatamente con questo componente.

3.3.3 Il Security Manager (SM)

Il Security Manager si occupa di arbitrare le richieste di uso delle cosidette risorse di sistema operativo: files, socket di rete, stampanti ecc. Lo scopo è quello di garantire l’accesso in funzione del livello di fiducia che l’utente ha nei confronti del codice. Nella macchina virtuale Java vi può essere soltanto una istanza per volta di SM. Nella versione Java 1.2 il SM affida le decisioni totalmente all’Access Controller. Quindi il SM a partire dalla versione 1.2 assume esclusivamente il ruolo di interfaccia alla sicurezza del sistema ed è mantenuto solo per compatibilità con il codice Java scritto con le versioni precedenti.

3.3.4 I concetti fondamentali del controllo d'accesso

Il controllo d'accesso si basa su quattro concetti fondamentali:

- *l'origine del codice* (CodeSource*)*: la locazione da cui si è ottenuta una classe;

- *il permesso* (permission): la richiesta di eseguire una particolare operazione;
- *la politica* (policy): l'insieme di tutti i permessi da garantire a tutti i CodeSource;
- *il Dominio di protezione* (protection domain): l'insieme un CodeSource e dei suoi permessi.

3.3.4.1 Il Code Source

Il "Code Source" è un semplice oggetto che racchiude l'URL da cui è stata caricata una classe e le chiavi eventualmente utilizzate per firmare tale classe. È necessario per individuare un Principal, un proprietario della classe.

3.3.4.2 I permessi

Il permesso rappresenta un'operazione particolare, ed è l'entità principale su cui opera l'Access Controller. Il permesso ha tre proprietà: nome, tipo ed azioni.

- *Il tipo:* consente di identificare a quale permesso ci si riferisce. Per esempio un tipo "*FilePermission"* indica che il permesso riguarda l'accesso ad un file;
- *Un nome:* indica l'oggetto specifico a cui è riferito il permesso.

Nel caso di “FilePermission” è il nome del file cui si vuole accedere. In generale i nomi dei permessi sono arbitrari.

- *Le azioni:*
  Alcuni permessi portano con sé una o più azioni che dipendono dalla semantica del permesso. Per esempio un oggetto “FilePermission” ha una lista di azioni che può includere: lettura, scrittura e cancellazione. Le azioni possono essere specificate anche da “wildcards”.

Le API di Java forniscono undici tipi di permessi standard: FilePermission, SocketPermission, PropertyPermission, RuntimePermission, AwtPermission, NetPermission, SecurityPermission, SerializablePermission, ReflectPermission, UnresolvedPermission, AllPermission. Una dettagliata descrizione di ciascuno di essi si trova in [JDK98].

Per esempio, la classe SecurityPermission contiene il permesso di utilizzare le classi del package “java.Security”: il campo nome è soggetto alla wildcard asterisco; mentre il campo azioni è vuoto. Un aspetto importante degli oggetti di tipo “permission” è la presenza del metodo “*implies()*”, con il quale è possibile determinare se o meno, un permesso può garantirne altri: il permesso di scrivere un particolare oggetto in un database implica anche un permesso di lettura nei confronti di tale oggetto.

I permessi possono essere aggregati in “collezioni”, a tal scopo si usa la classe “*PermissionCollection*”. In questo modo il CA può invocare il metodo “*implies()*” sui permessi di tutta una collezione. Per esempio, un utente ha dei permessi per leggere diverse directory. Quando l’utente accede ad un particolare file, il CA deve esaminare tutti i permessi dell’utente e, per garantire l’accesso, deve trovarne uno che consenta proprio quell’operazione. Aggregare tutti i permessi sui file in una singola collezione rende il compito del controllo di accesso più semplice e rapido. Lo scopo

principale delle collezioni è di aggregare soltanto oggetti permesso dello stesso tipo.

Le diverse "Permission Collection" vengono raccolte e raggruppate in modo arbitrario, nella classe "*Permissions".* Dunque la classe "*Permissions*" contiene insiemi eterogenei di collezioni e nelle collezioni sono racchiusi i singoli permessi.

3.3.4.3 La Politica

La politica di sicurezza è l'insieme globale dei permessi associati ai "CodeSource". La classe "Policy" svolge il ruolo di fornitore (provider) della politica attuale. Come per il Security Manager solo una istanza per volta della classe "*Policy*" può essere installata nella macchina virtuale Java. Ma a differenza del Security Manager, l'istanza attuale della politica può essere sostituita dinamicamente. La classe "*Policy*" consente di leggere una politica (metodo *getPolicy*), di installarla (metodo *setPolicy*) e di restituire tutti i permessi associati ad un dato "CodeSource" (metodo *evaluate*).

Sono necessari appositi permessi per eseguire i metodi appena citati. Emerge dunque un problema di inizializzazione: per dare un permesso è necessario avere una politica installata, ma per installarla è necessario averne il permesso. La soluzione è quella di leggere in un file di sistema (*java.security*) all'avvio della macchina virtuale Java, qual è il fornitore della politica: di default, la politica (intesa come associazione tra CodeSource e permessi) è espressa nel file "java.policy" (che può essere creato ed editato dal tool Java PolicyTool) e, per questo motivo, il fornitore di politica, che avevamo genericamente chiamato Policy, è la classe di sistema PolicyFile. In caso contrario, il fornitore deve essere una classe utente che eredita dalla classe astratta Policy e la politica può essere contenuta, ad esempio, su un server di rete.

3.3.4.4 Il Dominio di protezione

Un Dominio di protezione è il raggruppamento di un CodeSource e di tutti i suoi permessi. Cioè un Dominio di protezione rappresenta tutti i permessi che sono garantiti ad un particolare CodeSource. Per esempio, nella implementazione di default della politica (PolicyFile), un Dominio di protezione è una entry del file in cui è mappata la politica (java.policy). In Java 1.2 i Domini di protezione vengono creati su richiesta ("*on demand*"). Ogni classe della macchina virtuale, scaricata da un determinato URL, può e deve appartenere ad un solo Dominio di protezione: è il class loader che decide in quale Dominio inserirla, nel momento in cui la definisce.

Il supporto a tempo di esecuzione di Java mantiene il mappaggio tra il codice (classi ed oggetti) e i rispettivi Domini di protezione cioè i permessi. Esiste uno speciale Dominio di protezione: il Dominio di sistema, che consiste di codice di sistema che viene caricato dal "null class loader" (un class loader che carica classi solo dal disco locale, principalmente le classi localizzate sul CLASSPATH) che possiede privilegi speciali. È importante che tutte le risorse esterne protette come il filesystem, il supporto di rete, lo schermo e la tastiera siano direttamente accessibili solo dal codice di sistema.

3.3.5 La classe Access Controller (AC)

Per garantire l'accesso protetto alle risorse è necessario sapere chi fa la richiesta o per conto di chi questa avvenga; dunque l'Access Controller interviene durante il thread di esecuzione. Un thread di esecuzione può svolgersi interamente in un singolo Dominio di protezione oppure può coinvolgerne diversi.

Analizziamo ad esempio un thread che coinvolge due Domini: uno di applicazione e quello di sistema. Quando una applicazione vuole stampare un messaggio deve interagire con il Dominio di sistema che possiede l'unico punto d'accesso allo stream di output. In questo caso si deve evitare che Dominio dell'applicazione estenda i suoi permessi guadagnando tutti quelli aggiuntivi del Dominio di sistema coinvolto: potrebbero nascere seri problemi di sicurezza.

Nella situazione inversa consideriamo il Dominio di sistema che invoca un metodo del Dominio di applicazione come nel caso in cui il codice del sottosistema grafico (AWT) chiama il metodo "paint" di un'applet. È di nuovo cruciale che il Dominio di applicazione mantenga gli stessi diritti di accesso che possedeva prima dell'invocazione. *In generale, un Dominio meno potente non può guadagnare permessi aggiuntivi come risultato di una chiamata ad un Dominio più potente* [Gong98]. Questo è noto come il principio dei "*minimi privilegi"*: al thread in esecuzione vengono garantiti i diritti di accesso ottenuti come intersezione dei permessi di tutti i Domini di protezione che attraversa.

Ogni porzione di codice può interrogare il controllore d'accesso per sapere se un permesso gli è garantito: deve invocare il metodo "*checkPermission*" dell'Access Controller utilizzando come parametro l'oggetto, di tipo "*Permission*", che vuole ottenere. L'AC verifica se tutti i chiamanti del thread di esecuzione (per esempio tutte le classi nello stack delle chiamate) appartengono a Domini che garantiscono tale permesso e in caso positivo restituisce il controllo "silenziosamente". Altrimenti genera una eccezione "AccessControlException" in cui è motivata anche la ragione del diniego. Questo comportamento standard è il più sicuro ma in alcuni casi è limitante: un frammento di codice vorrebbe temporaneamente esercitare i suoi permessi, ma questi non sono direttamente disponibili al chiamante.

Per esempio, una porzione di codice di sistema può accedere a tutte le risorse, un'applet gli chiede di svolgere un compito. Il

limitato insieme di permessi di cui gode l'applet può impedire al codice di portare a termine l'esecuzione del suo compito. Per questi casi eccezionali l'Access Controller fornisce un costrutto: attraverso i metodi statici "beginPrivileged" ed "endPrivileged". Delimitare un pezzo di codice tra questi due metodi, significa fare ignorare al sistema runtime Java lo stato dei suoi chiamanti e significa che esso stesso si assume la responsabilità di esercitare i propri permessi.

Quindi per calcolare i permessi l'AC adotta le seguenti regole :

- il permesso di un thread di esecuzione è l'intersezione dei permessi di tutti i Domini di protezione attraversati dal thread stesso;
- quando del codice invoca la primitiva "beginPrivileged" , il supporto runtime di Java disabilita il meccanismo di limitazione dei permessi. Lo riabiliterà quando incontrerà la primitiva "endPrivileged".
- quando si crea un nuovo thread esso eredita dal thread genitore il contesto di sicurezza (per esempio l'insieme di Domini di protezione presenti nel padre nel momento in cui crea un figlio) per rispettare il principio dei minimi privilegi. Tale ereditarietà è transitiva.

Sono possibili due strategie per implementare tali regole d'accesso [Gong97]:

- nella strategia "*eager*", ogni volta che un thread entra in un nuovo Dominio di protezione o ne esce, l'insieme dei premessi effettivi viene aggiornato dinamicamente. Il beneficio è che il controllo sul permesso (innescato dalla "*checkpermission*") è semplificato e in molti casi più veloce. Lo svantaggio è che, poiché il "*checkpermission*" avviene molto meno frequentemente delle chiamate inter Dominio, una larga

percentuale di aggiornamenti di permessi possono essere uno sforzo inutile;

- nella strategia “lazy”, si esamina lo stato del thread (come riflesso dallo stack delle chiamate) soltanto quando si richiede un controllo di permesso (con il metodo *checkpermission*); non avviene un aggiornamento dinamico dei permessi ad ogni interazione interDominio. Un effetto negativo di questo approccio è un lieve degrado di prestazioni a tempo di controllo. Java 1.2 implementa questo tipo di valutazione in quanto più semplice da gestire.

3.3.6 Oggetti con guardia

Abbiamo visto quanto sia importante il contesto di sicurezza associato ad un thread di esecuzione per le decisioni che deve prendere l’AC (per garantire il principio del permesso minimo). Nella situazione in cui il fornitore di una risorsa non è nello stesso thread del consumatore di tale risorsa, e il thread consumatore non può fornire al thread fornitore le informazioni di contesto per il controllo d’accesso, perché tale contesto è protetto, oppure è troppo grande da passare o per altri motivi, l’AC non ha informazioni sufficienti per prendere alcuna decisione.

Allora per proteggere l’accesso alla risorsa si usa la classe “*GuardedObject*”. Il fornitore della risorsa può creare un *GuardedObject* che incapsuli la risorsa e fornirlo al consumatore. Per creare questo oggetto il fornitore specifica anche un oggetto “*Guard*” che conterrà i controlli necessari per proteggere la risorsa. Il consumatore può ottenere accesso alla risorsa solo se tali controlli saranno soddisfatti all’interno dell’oggetto “*Guard*”.
“Guard” è un’interfaccia (quindi ogni oggetto può divenire una guardia) che possiede un solo metodo, il “*checkGuard(object)*” ed è implementato nella classe Permission.

Per esempio se consideriamo la risorsa file "/a/b/c.txt", il fornitore deve creare un oggetto con guardia nel modo seguente:

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

Il consumatore deve ottenere l'oggetto g dal produttore e per leggere il file deve eseguire il codice:

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

Il metodo getObject(), dunque, invoca il metodo checkGuard() dell'oggetto "Guard" p che sarà fatto nel modo seguente:

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

si usa "*this*" come argomento perché p stesso, in questo caso è un permesso.

3.4 Java e la crittografia

3.4.1 La Java Cryptography Architecture

Abbiamo visto i meccanismi di Java che consentono di vincolare l'esecuzione di un frammento di codice nel rispetto delle politiche di sicurezza personalizzate. Vediamo ora come Java integri gli strumenti crittografici tradizionali, necessari affinché il lavoro dei componenti fondamentali (controllore d'accesso e class loader) possa essere consistente. Ad esempio, si pensi all'importanza della firma digitale di una classe: è proprio tramite questa, che il class

loader attribuisce alla classe, quello che deve essere il suo Dominio di protezione.

Java fornisce un insieme di API raccolte nel package java.security, che costituiscono l'Architettura Crittografica di Java (JCA). Tale architettura è stata progettata con i seguenti obiettivi:

- l'indipendenza dalla piattaforma e l'interoperabilità;
- l'indipendenza dagli algoritmi crittografici e l'estendibilità.

L'indipendenza dalla implementazione e l'indipendenza dall'algoritmo sono tra loro complementari: lo scopo è di rendere trasparenti all'utilizzatore i dettagli implementativi degli algoritmi e fornire con le API i *concetti* crittografici come la firma digitale ed il digest di messaggi. L'indipendenza dall'algoritmo crittografico è ottenuta attraverso la definizione di motori "*Engines*" crittografici (servizi) e la definizione delle classi che realizzano le funzionalità di questi motori, come ad esempio le classi "*MessageDigest*", "*Signature*" e "*KeyFactory*".

L'indipendenza dall'implementazione è raggiunta usando una architettura basata sul concetto di fornitore "*Provider*": un package che implementa uno o più servizi crittografici, contenente gli algoritmi per la firma digitale, gli algoritmi per il digest dei messaggi, per la conversione di chiavi ecc. Un programma può richiedere ad un "*Provider*" un oggetto (ad esempio di tipo *Signature*) che implementa un particolare servizio crittografico. Per avere una diversa implementazione è sufficiente rivolgersi ad un diverso "*Provider*". Il Provider può essere aggiunto in modo trasparente all'applicazione, ad esempio, quando è disponibile una versione più veloce e sicura.

Implementare l'interoperabilità significa che le varie implementazioni degli algoritmi crittografici possono lavorare

insieme: una chiave, una firma ecc. generate da una implementazione particolare devono poter essere usate anche da tutte le altre implementazioni. L'estendibilità degli algoritmi significa che si possono aggiungere facilmente nuovi algoritmi nelle classi motore.

Riportiamo le classi "*Engine*" più significative implementate in Java 1.2:

- Message Digest: usata per calcolare l'hash di un messaggio;
- Signature: usata per firmare dati e verificare le firme digitali;
- KeyPairGenerator: usata per generare una coppia di chiavi pubblica e privata di un algoritmo particolare;
- Keystore: usata per creare e gestire database di chiavi.
- Secure Random: usata per generare numeri pseudo casuali;

Una classe "*Engine*" fornisce l'interfaccia alle funzionalità di uno specifico tipo di servizio crittografico (indipendentemente da un particolare algoritmo utilizzato). Essa definisce l'"Application Programming Interface" (API), cioè i metodi che consentono all'applicazione di accedere ad un particolare servizio crittografico fornito. Le interfacce alle applicazioni fornite da una classe motore, sono implementate dalle "Service Provider Interface" (SPI), cioè per ogni classe motore esiste una corrispondente classe SPI astratta che definisce l'interfaccia che i "*Provider*" crittorafici devono implementare (Figura 3-4).

L'istanza di una classe motore "*oggetto API*" incapsula (come campo privato) una istanza della classe SPI corrispondente, "*oggetto SPI*". Tutti i metodi di un oggetto API sono dichiarati final e le loro implementazioni invocano i metodi SPI corrispondenti dell'oggetto incapsulato SPI. La chiamata al metodo statico "*getInstance*" produce la creazione di una istanza della classe "*Engine*" e della sua classe SPI corrispondente.

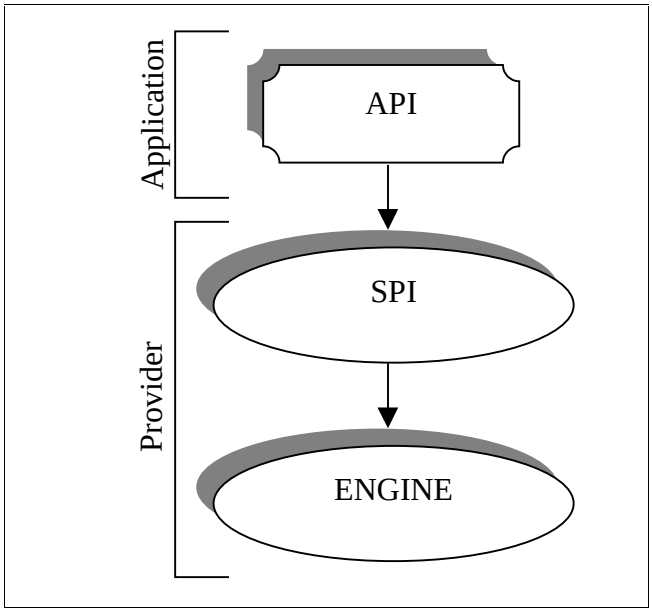

Figura 3-4 Le API sfruttano SPI (ed Engine) forniti dal Provider

Per fornire l'implementazione di un particolare tipo di servizio per uno specifico algoritmo, un "*Provider*" deve ereditare la classe SPI corrispondente e fornire l'implementazione di tutti i metodi astratti.

3.4.2 La Java Cryptography Extension

La Java Criptography Extension (JCE) estende le API JCA per includere operazioni di cifratura e scambio di chiavi.

JCE viene considerata una parte a se stante a causa dei problemi legali che impediscono di "esportare" al di fuori degli Stati Uniti d'America la tecnologia crittografica. Dunque la JCE di SUN può essere usata solo in America; in Europa o si implementa una propria JCE o si possono sfruttare quelle fornite da Cryptix [WEB2] oppure da IAIK [WEB3] ecc.
JCA e JCE insieme forniscono una API completa ed indipendente dalla piattaforma.

3.4.3 Uso delle classi crittografiche

Quando il codice di un'applicazione richiede un particolare algoritmo, viene restituito l'appropriato oggetto che lo implementa.

Esistono due modi per richiedere l'algoritmo: specificando solo il nome dell'algoritmo, o dichiarando il nome dell'algoritmo e il nome del package "Provider".

Se si specifica solo il nome dell'algoritmo, il sistema determina se vi sono implementazioni dell'algoritmo richiesto tra i "*Providers*" installati nel sistema e restituisce la prima implementazione che trova. Se sono dati entrambi i parametri, nome dell'algoritmo e del package, il sistema determina se esiste l'implementazione solo nel "*Provider*" esplicitamente richiesto, se non vi è, lancia una eccezione. Per utilizzare un oggetto che implementa un algoritmo crittografico in generale sono necessari tre passi:

1) *inizializzazione*: in questa fase si inserisce la chiave crittografica, che deve essere generata o reperita in precedenza;
2) *aggiornamento*: questa fase si inseriscono nell'oggetto i byte del dato su cui deve essere eseguita l'operazione crittografica;
3) *applicazione dell'algoritmo crittografico*.

Cap.4 La sicurezza in SOMA

Definire un modello di sicurezza per sistemi ad agenti mobili, significa: individuare quali sono i problemi di sicurezza legati al contesto degli agenti mobili; definire le strategie (politiche) che consentono di affrontare tali problemi; infine utilizzare e/o costruire gli strumenti (i meccanismi) che consentono di adottare tali strategie. Per la costruzione di un modello flessibile, è dunque molto importante la separazione tra "politiche" e "meccanismi": le politiche stabiliscono *cosa* è necessario fare, i meccanismi invece dicono *come* ciò si possa fare (Figura 4-1).

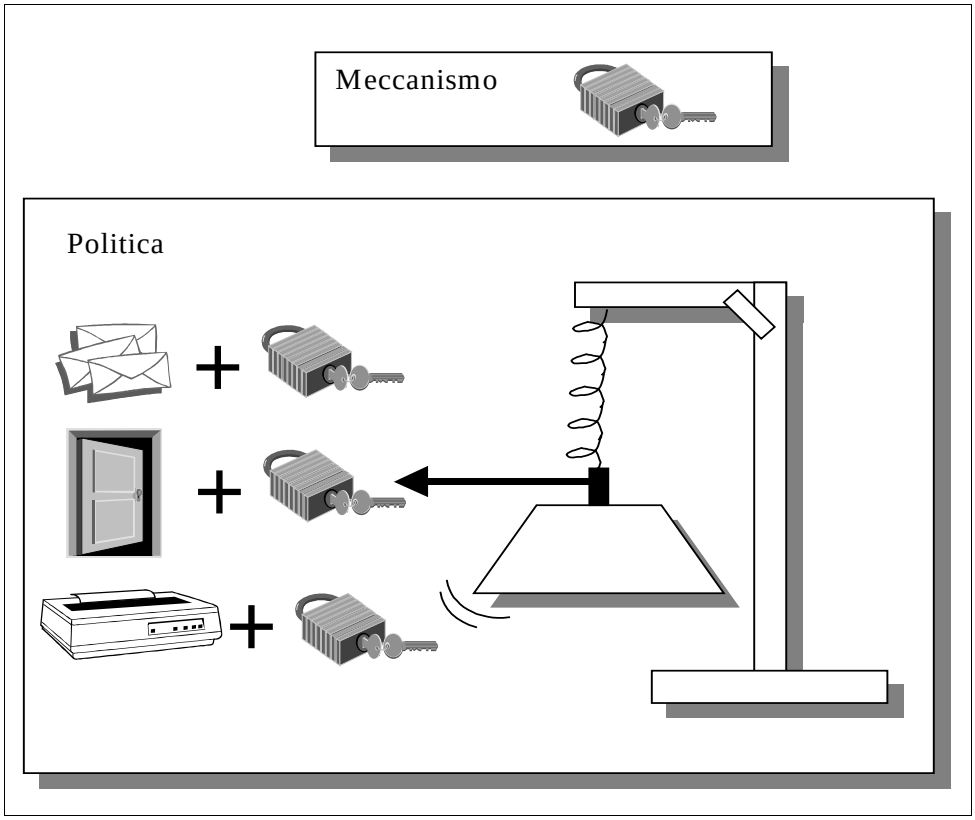

Figura 4-1 Separazione tra politiche e meccanismi

Il modello di sicurezza del sistema SOMA è implementato interamente in linguaggio Java. Alcuni componenti sono sfruttati nella loro forma originale (gli oggetti crittografici), altri sono adattati alle caratteristiche del sistema (ClassLoader ecc.). Seguendo la classificazione generale dei problemi di sicurezza di un sistema ad agenti mobili, ci occupiamo prima di come avvenga la protezione del CE. Successivamente affronteremo la protezione degli agenti. Infine, considereremo l'importante infrastruttura per la gestione delle chiavi.

4.1 I problemi in generale

L'analisi degli elementi costitutivi del mondo degli agenti mobili ci porta ad individuare due categorie principali di problemi di sicurezza. La prima riguarda la sicurezza dell'ambiente di esecuzione; la seconda categoria, invece, comprende i problemi sollevati dalla protezione dell'agente. Gli agenti e i CE si devono considerare con mutuo sospetto, fintantoché non si riesce a dimostrare un certo grado di fiducia reciproca. L'agente può cercare di impossessarsi o distruggere informazioni protette, oppure occupare eccessivamente o compromettere alcune risorse, con il solo scopo di ostacolare il regolare svolgimento di tutte le operazioni. Per esempio in sistemi ad agenti implementati in Java, l'invocazione, da parte di un agente, della chiamata di sistema "*exit()"* produce la terminazione della JVM e con essa di tutti i suoi servizi ed agenti contenuti.

Oppure un agente può creare danni accidentali a causa di errori di programmazione, ad esempio si pensi ad un agente che, per una gestione errata degli indici di alcuni cicli innestati che lo compongono, produce numerosi processi figli che eseguono inutili loop: essi sottraggono il tempo di CPU a tutte le altre computazioni. I CE, a loro volta, possono cercare di estrarre

informazioni preziose trasportate dagli agenti e condizionarne il comportamento modificandone i dati o aggiungendovi del codice. Si pensi ad un agente di viaggi che deve prenotare il volo più economico: in ogni CE che visita, legge le quotazioni dei voli e le memorizza in delle proprie variabili, in questo modo, riesce a confrontare tutti i prezzi che raccoglie. Un CE non fidato, potrebbe accedere al contenuto di tali variabili e modificarle adeguatamente per rendere la propria, l'offerta più conveniente.

4.1.1.1 Sicurezza dell'ambiente

Il CE deve essere in grado di autenticare il "Principal" dell'agente e di assegnare all'agente, in funzione di tale autenticazione, un certo grado di fiducia, cioè garantire all'agente i diritti di accesso ed uso di un limitato e predefinito numero di risorse. Quindi il CE, per evitare furti o danneggiamenti di informazioni riservate, deve poter prevenire ogni tentativo di violazione dei limiti d'accesso di un agente. I limiti di uso delle risorse, imposti agli agenti per evitare abusi, possono includere consumi massimi (tempo totale di CPU), e consumi massimi per unità di tempo (tempo totale di CPU per unità di tempo).

4.1.1.2 Sicurezza dell'agente

Un agente non deve poter interferire con altri agenti o sottrarre a questi risorse. Un CE non deve poter accedere alle informazioni trasportate da un agente senza che questo collabori, e non lo deve neppure poter alterare. Purtroppo, senza un supporto hardware adeguato, non è possibile evitare che un CE faccia quello che vuole su un agente. Quello che si può fare, è cercare di scoprire, in un CE di fiducia, le eventuali manomissioni subite da un agente quando

torna da un CE non fidato. Un modo per mantenere la riservatezza delle informazioni dell'agente è assicurarsi che l'agente non passi mai in un CE non fidato in una forma non cifrata: le informazioni che trasporta sono così insignificanti, senza la collaborazione di un CE fidato.

4.2 Protezione dell'ambiente

4.2.1 Politiche

Il primo passo per la protezione di un CE, richiede la definizione e progettazione di una politica di sicurezza. Il modello di sicurezza utilizzato per la protezione dei CE, mantiene una netta separazione tra politiche e meccanismi. In questo modo, è possibile proteggere facilmente nuove risorse che si aggiungono al sistema. In SOMA la politica di sicurezza è articolata su due livelli di astrazione: uno di Dominio ed uno di Place. Questo perché la politica deve essere un'entità snella, rapida da modificare consultare e aggiornare. Una decisione appartenente alla politica di Dominio viene riportata automaticamente su tutte le locazioni del Dominio di appartenenza, quindi è un mezzo, utile per adottare decisioni globali. Invece la maggior autonomia che deriva dalla possibilità di adottare decisioni a livello di Place, consente di ottenere una granularità molto fine sul controllo di ogni singola risorsa, sempre nel rispetto delle decisioni globali (di Dominio). Per rispettare questa organizzazione, le decisioni a livello di Place possono essere solamente più restrittive rispetto a quelle adottate a livello di Dominio.

La politica di Dominio viene gestita dal Gateway ed in esso memorizzata permanentemente. Ogni volta che si attiva un Place, questi si collega con il proprio Gateway per ricevere la politica di Dominio. Quindi, il Place interseca la politica appena ricevuta, con

quella locale, che possiede in modo permanente, ottenendo così la politica da adottare durante il suo tempo di vita.

Gli elementi costitutivi della politica, sono i permessi. In SOMA il permesso può avere un duplice significato: esistono permessi positivi, che specificano che cosa è possibile fare; e permessi negativi, che, dualmente, consentono di affermare quello che non si può fare.

4.2.1.1 Implementazione politiche

In SOMA si possono usare tutti i permessi definiti dalla architettura di sicurezza di Java, ad esempio quelli per l'accesso ai files, o per l'apertura di socket. Esistono anche nuovi permessi, che racchiudono le operazioni significative per l'interazione tra sistema ed agenti, ad esempio il PlaceAccessPermission, che ovviamente racchiude la possibilità di accedere ad un Place; oppure il ModifyPolicyPermission che consente la modifica della politica di un Place ecc.. Per l'elenco e la descrizione dettagliata dei permessi personalizzati vedere [Ten98].

In SOMA le politiche sono mappate in file di permessi, esistono due categorie di file: quelli che contengono la politica di Dominio, residenti sul Gateway; e quelli che contengono la politica di Place, che risiedono nel filesystem del Place. Esiste un tool grafico per la creazione e la gestione di tali files: l'AgentPolicyTool. La prima finestra dell'AgentPolicyTool consente di aprire, salvare e creare nuovi file; mette in evidenza il file su cui si sta lavorando ed elenca le "etichette" di tutte le entry di politica contenute nel file corrente (Figura 4-2). Le entry di politica, sono costituite da tali "etichette", e da un insieme di permessi; esse vengono costruite attraverso la seconda finestra dell'AgentPolicyTool (Figura 4-3).

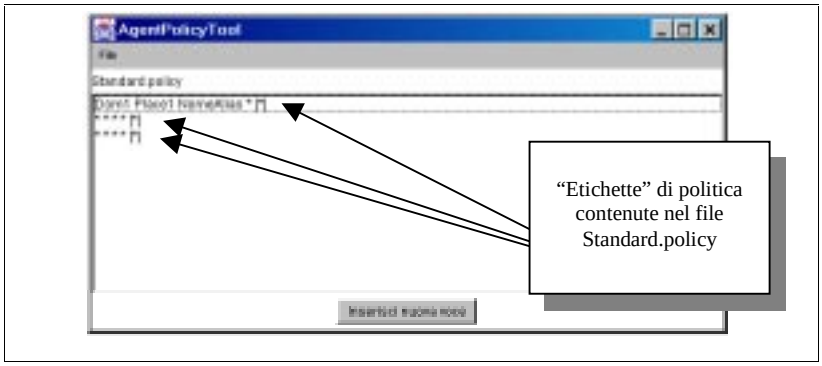

Figura 4-2 Prima finestra dell'AgentPolicyTool

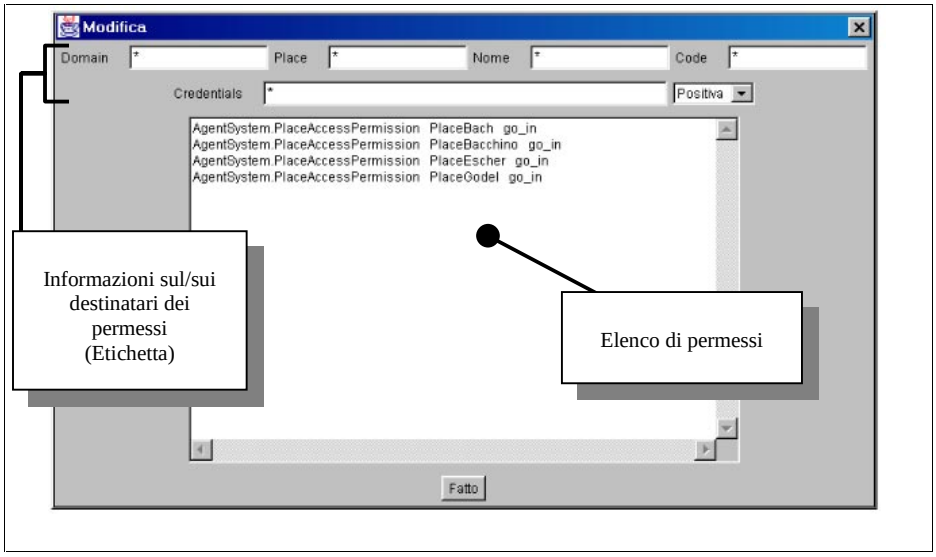

Figura 4-3 Seconda finestra dell'AgentPolicyTool

In questa finestra è possibile inserire il set di permessi in un apposito elenco, nel formato descritto nel cap. 2 sui permessi: cioè specificando i tre campi, tipo di permesso, nome e azioni, separati tra loro da spazi. Se si specificano permessi con un numero di campi inferiore, è necessario inserire la clausola "*null*". I campi di testo, che sono presenti nella parte superiore della finestra,

consentono di specificare i destinatari di tali permessi. È possibile dire il nome del Dominio, del Place, il nome della classe che implementa l'agente (Name), l'identificatore univoco all'interno del Place (Code) e l'alias di un utente (Credentials). È possibile utilizzare la wildcard "*" per indicare al posto di un attributo specifico, tutti i valori possibili.

Per esempio, se si inserisce l'asterisco nel campo Dominio, significa che i permessi sottoelencati, devono valere in tutti i Domini del sistema. La presenza dei nomi di Dominio e di Place è ovviamente necessaria per specificare le locazioni in cui valgono i permessi, il nome e l'identificatore univoco, invece, sono relativi alla classe che implementa l'agente. Il campo "*Credentials*", consente di specificare l'alias di un utente, è necessario per consentire la verifica della firma del Principal dell'agente e quindi associare all'agente i giusti permessi (vedremo nei paragrafi successivi come e perché ciò avvenga). Infine, troviamo il selettore per configurare il significato del set di permessi e cioè se tale politica esprima dei consensi o dei divieti.

4.2.2 Autenticazione

Il secondo passo verso la protezione dell'ambiente di esecuzione, (inteso ora come Place), prevede la necessità di autenticare gli agenti che vengono creati all'interno di un Place o che giungono da altri luoghi.

L'identificazione di un agente è composta da due parti: la prima consente di distinguere senza ambiguità un agente all'interno del sistema (cfr. paragrafo 2.4.1); la seconda è necessaria per identificare il Principal per cui l'agente esegue. Quest'ultima fase avviene grazie alla firma digitale del codice dell'agente, con la chiave privata del Principal che lo ha generato. Per la firma dell'agente si utilizza un algoritmo a chiave pubblica.

Quando un agente, arriva, o viene creato in un Place, la prima operazione di cui si occupano i meccanismi di sicurezza del Place, è la verifica della firma digitale. Tale operazione, richiede la conoscenza, da parte del Place della chiave pubblica del Principal che ha generato l'agente. Esiste un'infrastruttura per la gestione delle chiavi in SOMA che analizzeremo in dettaglio nel paragrafo 4.4. In realtà la firma digitale sopperisce anche al problema dell'interità dell'agente, ma affronteremo questo aspetto in dettaglio nel paragrafo 4.3, sulla protezione dell'agente.

È necessario fare una distinzione tra gli agenti che vengono creati localmente e quelli che giungono da altri Place. Nel caso degli agenti creati localmente, il sistema deve, in primo luogo, identificare l'utente che crea l'agente. Cioè il sistema deve poter riconoscere ad un utente il diritto di creare agenti firmati. Questo comportamento è strettamente connesso al problema della gestione delle chiavi crittografiche, lo affrontiamo nel paragrafo 4.4.5.2.

4.2.2.1 Implementazione: Launcher

Il primo meccanismo di autenticazione, che interviene alla creazione di un agente, consente di individuare gli utenti che possono creare agenti. Lo strumento che permette di mettere in esecuzione gli agenti in un punto qualunque del sistema e che svolge anche questa funzione di autenticazione, è il tool grafico Launcher. La prima operazione che compie il Launcher, è la connessione con il Supervisor (Figura 4-4), che ricordiamo essere l'unico luogo ove sono presenti le informazioni più aggiornate sul sistema.

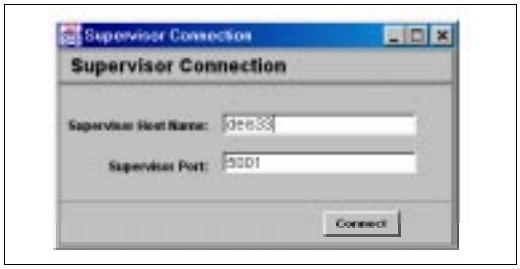

Figura 4-4 Connessione del Launcher al Supervisor

Il Launcher in questo modo, ottiene le informazioni riguardanti i Domini ed i Place attivi in tutto il sistema, e soprattutto, l'elenco degli utenti ammessi. Infatti nel passo successivo avviene la prima fase di identificazione: quella in cui l'utente è tenuto ad inserire una password, ed uno username (Figura 4.5).

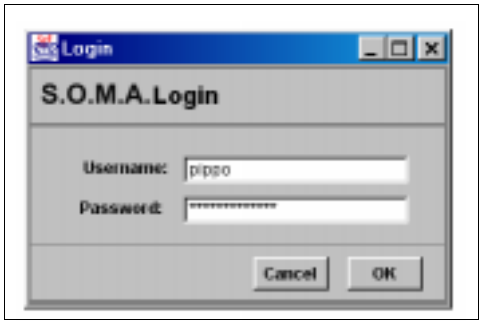

Figura 4.5 Schermata di accesso al sistema

Il Launcher, dopo aver verificato la password utente, consente di specificare la classe che implementa l'agente e di selezionare, da un elenco di Domini e relativi Place, l'ambiente ove iniziare l'esecuzione dell'agente (Figura 4.6).

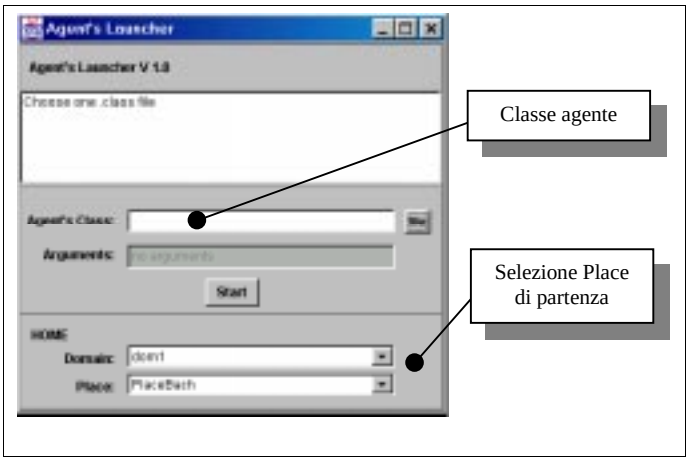

Figura 4.6 Schermata per l'avvio di un agente

Le informazioni di login, inserite nel Launcher, una volta verificate, vengono trasmesse al Place. Il Place, dopo aver creato l'agente, sfrutta tali informazioni per reperire la chiave privata dell'utente (vedere paragrafo gestione chiavi), che utilizza per firmare l'agente. A questo punto la distinzione tra agenti creati nel Place ed agenti che provengono da altri luoghi, non esiste più: entrambi, vengono sottoposti al meccanismo di verifica della firma, che consentirà poi di individuare un determinato Dominio di protezione (si veda il paragrafo successivo).

L'algoritmo utilizzato per apporre e verificare la firma digitale degli agenti è il DSA, insieme alla funzione hash SHA-1 [SCH95], (dallo standard NIST per la firme digitali), implementato dalle API JCA di Java. La firma avviene utilizzando un'istanza della classe signature, ottenuta dal provider SUN. È possibile sostituire l'algoritmo per la firma semplicemente inizializzando in modo differente tale oggetto (cfr. par. 3.4.1).

4.2.3 Autorizzazione

Il terzo passo verso la protezione dell'ambiente di esecuzione, consiste nell'associare all'agente, già identificato con un alto grado di certezza (al passo precedente), il suo insieme di permessi: questa associazione è controllata dinamicamente dell'Access Controller, che verifica se l'agente possiede il permesso adeguato per le operazioni che desidera effettuare. In SOMA, l'autorizzazione è realizzata sfruttando i meccanismi di sicurezza della macchina virtuale Java (cfr. par. 3.3.1). Quando un agente arriva in un Place, il Class Loader gli deve associare un adeguato Dominio di protezione: analizziamo, quindi in dettaglio, gli aspetti legati al Dominio di protezione e come è stato personalizzato il class loader sicuro.

4.2.3.1 Domini di protezione

Il componente che consente di associare all'agente il Dominio di Protezione, è il Class loader Sicuro. Il Class Loader Sicuro, individua, sulla base della politica contenuta nel Place, un insieme di permessi da attribuire all'agente. Il primo Dominio che associa all'agente è quello che deriva dalle credenziali del Principal che lo ha firmato e creato. In alcune circostanze, può essere necessaria l'estensione di tale Dominio: ad esempio, un agente che giunge in un nuovo Place, può non essere conosciuto e dunque, avere un set limitatissimo di permessi, come specificato nella politica per gli agenti non fidati. Ma se l'agente può dimostrare di aver eseguito, in passato, su Place fidati, tale set di permessi può essere espanso.

Ossia l'agente può mostrare delle credenziali aggiuntive e quindi fare aumentare il grado di fiducia nei suoi confronti. Perché ciò avvenga, l'agente deve richiedere ad un Place fidato su cui sta eseguendo, di essere firmato. In questo modo, quando si muove su

un altro Place, può dimostrare di aver eseguito in Place fidati, tramite il controllo delle firme che porta con se, e dunque può ottenere un maggiore grado di autonomia. Un problema può derivare dall'incapacità di differenziare i Domini di protezione di due o più agenti, perché provenienti dalla stessa locazione (Dominio e Place), firmati dal medesimo Principal, e con lo stesso codice. Per questo motivo si può inserire nello spazio dei dati dell'agente un particolare insieme di permessi: le Preferenze.
Il Dominio di Protezione di un agente viene definito ogni volta che l'agente cambia Place ed è calcolato attraverso l'intersezione di due insiemi di permessi:

1) l'insieme dei permessi ricavato dalle Credenziali, che deriva dall'intersezione di tutti i permessi dei firmatari dell'agente, in funzione della politica attuale del Place;
2) l'insieme dei permessi trasportati dall'agente, cioè le Preferenze.

Un ultimo aspetto, riguarda il Dominio di Protezione di un agente figlio, cioè di un agente generato da un altro agente. Per il principio dei minimi privilegi, l'agente figlio, non eredita tutte le credenziali acquisite dal padre. Gli vengono attribuite solamente le credenziali del Principal che ha generato il padre.

4.2.3.2 Il Class Loader Sicuro

In Java il SecureClassLoader attribuisce i Domini di Protezione alle classi, in funzione dell'oggetto *CodeSource*, associato ad ogni classe. Quindi, in un sistema ad agenti, è necessario inserire nel *CodeSource*, gli attributi di identificazione della classe che implementa l'agente. Gli elementi costitutivi del CodeSource sono: l'URL della classe e la lista di chiavi pubbliche utilizzate per la

verifica della eventuale firma della classe. In SOMA, l'URL, viene costruito nel modo seguente:

"http://NomeDominio/NomePlace/NomeClasse/Idnumerico;"

Mentre la lista è costituita dalle chiavi pubbliche necessarie per verificare le credenziali dell'agente. Per questioni di efficienza in SOMA viene implementato un meccanismo di *caching.* Il trasferimento di un agente viene spezzato in due momenti: nella prima trasmissione, si spostano la struttura e tutti i dati dell'istanza dell'agente (tramite la serializzazione dell'oggetto). Se nel filesystem (nella cache) della macchina destinazione è presente la classe che implementa l'agente, il processo di migrazione termina. Altrimenti avviene una seconda spedizione, in cui si invia il codice della classe stessa, che verrà anche memorizzato nella cache locale per velocizzare spostamenti futuri.

In particolare, nel Place di destinazione, dopo essere giunta la struttura dati di un agente, viene invocato il ClassLoader sicuro che cerca il codice dell'agente prima in memoria, per vedere se l'aveva caricato in precedenza, poi nel file system locale (nella cache) e, se non è presente neppure qui, si rivolge al Place di origine dell'agente, affinchè gli venga spedito.

Dopo aver reperito il codice dell'agente, il ClassLoaderSicuro, tramite il codeSource, verifica le credenziali e, insieme alle Preferenze, costruisce l'insieme di permessi che definiscono il Dominio di Protezione dell'agente, e lo memorizza in un'area accessibile al controllore d'accesso.

4.3 La protezione dell'agente

4.3.1 Protezione da altri agenti

In SOMA un agente può instaurare un rapporto stretto di collaborazione con altri agenti residenti sullo stesso Place. Un agente può richiedere al Place l'elenco degli altri agenti presenti (se ne ha il permesso) ed ottenere da tale elenco il riferimento all'agente con cui vuole iniziare l'interazione stretta. Questa interazione può consistere nell'invocazione reciproca di metodi (cfr. par. 2.3.3). Dunque è necessario che ogni singolo agente provveda alla protezione dei propri metodi, ad esempio, facendo un giusto uso delle clausole Public, Private, Protected, disponibili, in Java, ma anche in generale nei linguaggi di programmazione ad oggetti. Quindi il compito di garantire la protezione di un agente nei confronti di altri agenti, spetta al programmatore dell'agente, dal momento che il sistema controlla soltanto se un agente ha il permesso di iniziare un rapporto con altri agenti.

4.3.2 Protezione dall'ambiente

In SOMA, vengono protetti l'integrità del codice e la riservatezza dei dati di un agente quando questo è trasmesso su un canale di comunicazione. L'integrità si garantisce sottoponendo l'agente al processo della firma con la chiave privata del Principal creatore. Infatti l'algoritmo utilizzato per questo scopo, esegue anche l'hash dell'agente. Mentre la riservatezza si raggiunge sottoponendo l'agente a cifratura, prima che questi sia spedito sulla rete. In realtà, in SOMA, è possibile cifrare ogni tipo di oggetto scambiato tra due Places. Per questo scopo, viene utilizzato un apposito Comando

(cfr. par. 3.5.3) implementato dalla classe “TransCommand”. Questa classe incapsula l’oggetto da trasferire, sia esso un semplice messaggio, un altro comando, oppure un agente vero e proprio e si occupa di cifrare e decifrare l’oggetto che trasporta. Il TransCommand, cifra l’oggetto da spedire, con la chiave privata del Place mittente e con quella pubblica del Place destinatario. Quando il TransCommand, giunge a destinazione, decifra usando la chiave privata del Place di destinazione e la pubblica del mittente, l’oggetto da lui contenuto. L’utilizzo dell’algoritmo asimmetrico, risolve oltre al problema di riservatezza dell’oggetto che viaggia con il comando, anche il problema della paternità. L’algoritmo a chiave asimmetrica utilizzato dal TransCommand è l’RSA, implementato dal provider IAIK [WEB3].

Se tutte le informazioni scambiate tra due Place fossero solamente costituite da Agenti, non ci sarebbe il problema della paternità, essendo l’agente già firmato. Ma l’uso di questo tipo di algoritmo rende possibile risolvere il problema di paternità e riservatezza di ogni tipo di oggetto trasferito. Un lato negativo di questa soluzione, è costituito dalle basse prestazioni temporali che comporta. Infatti, le operazioni di cifratura e decifratura sono computazionalmente molto pesanti. Come proposta di miglioramento, si potrebbe adottare un algoritmo asimmetrico per negoziare la una chiave da utilizzare per cifrare successivamente tutti gli oggetti; cioè adottare un protocollo tipo SSL [FKK96].

Vediamo ora un altro aspetto, legato alla protezione dell’agente all’interno di un Place. Quando un agente entra in un Place, gli viene associato un componente, il Worker (cfr. par. 2.5.1), responsabile della messa in esecuzione del codice contenuto nella classe associata all’agente stesso. È molto difficile tutelare l’agente da un worker non fidato. Infatti il worker può creare danni, senza conoscere il contenuto dell’agente od alterarne l’integrità: per esempio mettendo in esecuzione più volte il metodo specificato dall’agente nell’istruzione go(). Oppure un worker alterato, può

decidere di non mettere in esecuzione nessun metodo dell'agente, quindi eliminarlo. Per evitare che queste situazioni compromettano l'esecuzione dell'agente, è necessario prendere atto del problema e cercare soluzioni a livello di applicazione.

4.4 La gestione delle chiavi

Abbiamo visto l'importante ruolo svolto dalle chiavi crittografiche in SOMA. Ricordiamo che le chiavi vengono usate in due algoritmi crittografici asimmetrici: uno per la firma digitale (DSA) ed uno per la cifratura (RSA). Questi due algoritmi consentono di risolvere i problemi di integrità, paternità e riservatezza degli agenti. In particolare, la soluzione al problema della paternità, permette ai Place di Autenticare i Principal, per cui gli agenti agiscono e conseguentemente, di esercitare i meccanismi di autorizzazione. Si è già discusso della vulnerabilità degli algoritmi crittografici ed in tale occasione si è messo in evidenza il fatto che una gestione non accorta delle chiavi può minare l'efficacia degli strumenti crittografici. Dunque in SOMA è stata progettata una infrastruttura che rappresenta un primo passo verso la soluzione dei problemi legati alla gestione delle chiavi.

4.4.1 Chiavi private

Quando un utente vuole lanciare un agente, deve firmarne il codice con la sua chiave privata (essendone il Principal). È bene che la chiave privata non sia nota ad altri che all'utente stesso. Molto spesso le chiavi private sono mantenute in forma cifrata all'interno dalla macchina su cui lavora l'utente. Per non vincolare l'utente ad interagire con il sistema da una sola macchina, la situazione più adeguata sarebbe quella di assegnare ad ogni utente un dispositivo

tipo *smart card* contenente la sua chiave privata, da inserire prima di accedere al sistema attraverso qualunque nodo abilitato. In questo modo, la macchina si copia temporaneamente la chiave privata e considera la smart card lo strumento per identificare l'utente. Una soluzione di questo tipo ha dei costi di realizzo molto alti, perché prevede l'inserimento di hardware dedicato.

Una soluzione di compromesso, tra il livello di sicurezza e il lato economico è quella di mantenere la chiave privata dell'utente, in forma cifrata, su tutti i nodi della rete, in cui sono mappati i Place che gli consentono di lanciare agenti SOMA. Quando l'utente vuole interagire con il sistema, deve inserire una password che svolge una duplice funzione: permette la sua identificazione e consente di decifrare la sua chiave privata.

4.4.2 Chiavi pubbliche

Consideriamo ora i problemi connessi alla gestione delle chiavi pubbliche. Il problema di alterazione della corrispondenza tra un Principal e la sua chiave pubblica (cfr. par. 1.2.8) è risolto attraverso l'uso di certificati firmati da un'autorità fidata. Ricordiamo che un certificato contiene l'associazione tra un Principal e la sua chiave pubblica e la firma dell'autorità fidata si pone come garanzia di tale corrispondenza. Dunque le chiavi pubbliche in SOMA, sono gestite da una "Certificate Autority" (cfr. par. 1.3.2.3) che emette e firma certificati per nuovi utenti e nuovi Place.

Dopo che il certificato è stato emesso e firmato, è necessario provvedere alla sua massima diffusione. In ogni Place del sistema devono essere presenti i certificati di tutti i Principal: cioè le chiavi pubbliche di tutti gli utenti e degli altri Place. Così il Place può verificare le credenziali di ogni agente e decifrare tutti gli oggetti che riceve e/o trasmette.

4.4.3 L'Amministratore di sistema

In SOMA esiste un utente privilegiato, che ha la possibilità di modificare dinamicamente la configurazione del sistema attraverso l'inserimento e l'attivazione di nuovi Domini, o di nuovi Place e l'aggiunta o l'eliminazione di account utente. Questo utente privilegiato è l'Amministratore di Sistema. L'account rilasciato al nuovo utente deve essere riconosciuto valido in ogni località del sistema, e quindi deve essere distribuito a tutti gli ambienti di esecuzione degli agenti. In tale contesto un ruolo fondamentale è svolto dal Supervisor, introdotto nel par. 2.5.4, che ha una funzione determinante non solo nella distribuzione, ma anche nella generazione, e nella revoca di chiavi.

4.4.4 Il Supervisor

Per gestire una distribuzione razionale, sull'intera configurazione di SOMA, dinamicamente in espansione, si è pensato di inserire i livelli logici di Place e di Dominio in una struttura gerarchica, coordinata da un elemento di livello superiore: il Supervisor. Il Supervisor mantiene le strutture dati contenenti le informazioni sulla topologia del sistema e tutte le coppie di chiavi dei Place e degli utenti in un database protetto. Egli svolge il ruolo di coordinatore dei Gateway. Con un apposito tool, il "*System Manager*", l'amministratore di sistema, può interagire con il Supervisor e modificare le strutture dati in esso contenute. Il Supervisor si occupa di propagare le modifiche apportate dall'amministratore a tutti i Gateway, e ciascun Gateway informa i propri Place (Figura 4-7).

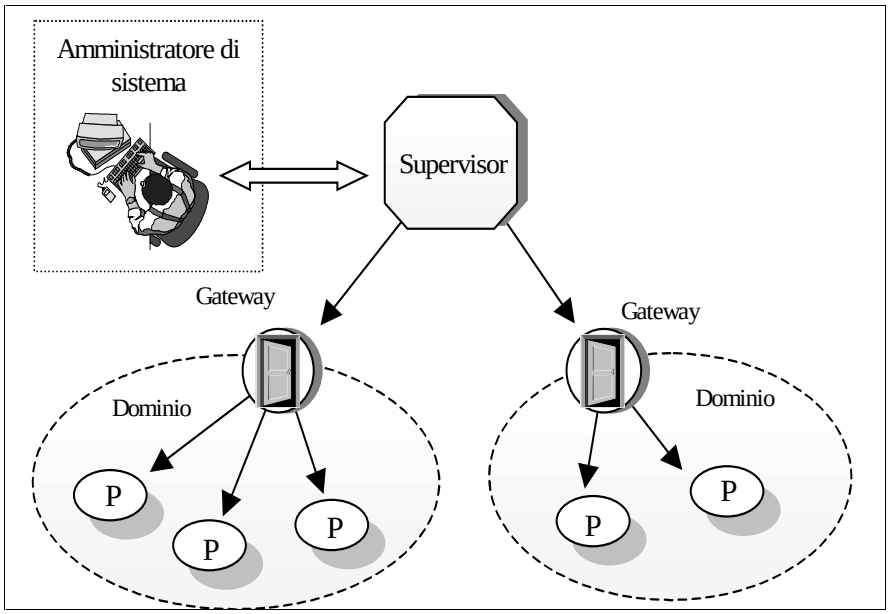

Figura 4-7 Tre livelli gerarchici

Vediamo ora più in dettaglio il tool System Manager.

4.4.5 Il System Manager

Il System Manager, è lo strumento che consente di modificare le strutture dati contenute nel Supervisor ed innescare i comandi in grado di propagare le modifiche in tutto il sistema (Figura 4**-8**). Come per il Launcher, anche per il System Manager, prima di poter interagire con lo strumento vero e proprio, è necessaria una fase preliminare in cui si devono specificare i dati relativi alla connessione con il Supervisor e i dati di account dell'utente (Figura 4**-9**). La schermata principale del System Manager, presenta la possibilità di utilizzare due tool differenti: l'Users Manager ed il Domains Manager.

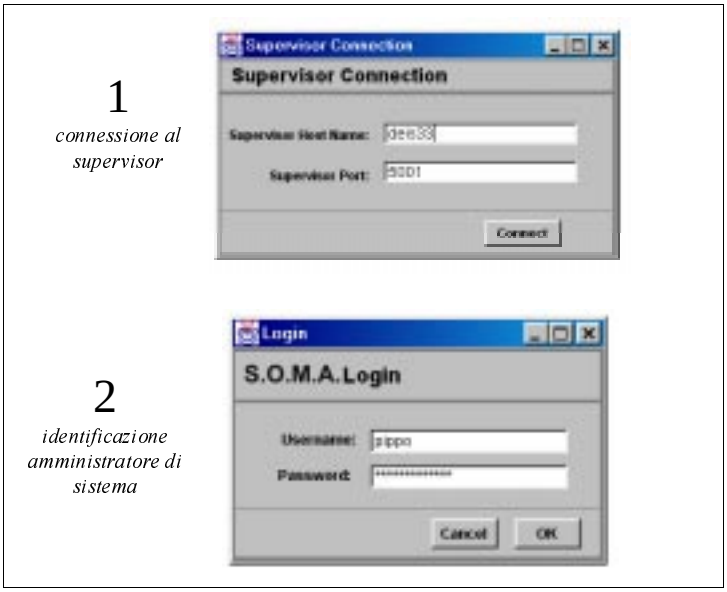

Figura 4-8 Fasi preliminare per avviare il System Manager

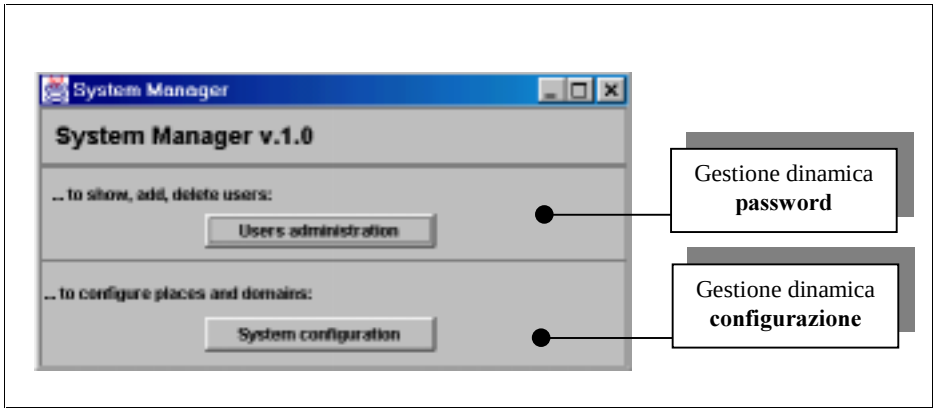

Figura 4-9 Schermata principale del System Manager

4.4.5.1 Domains Manager

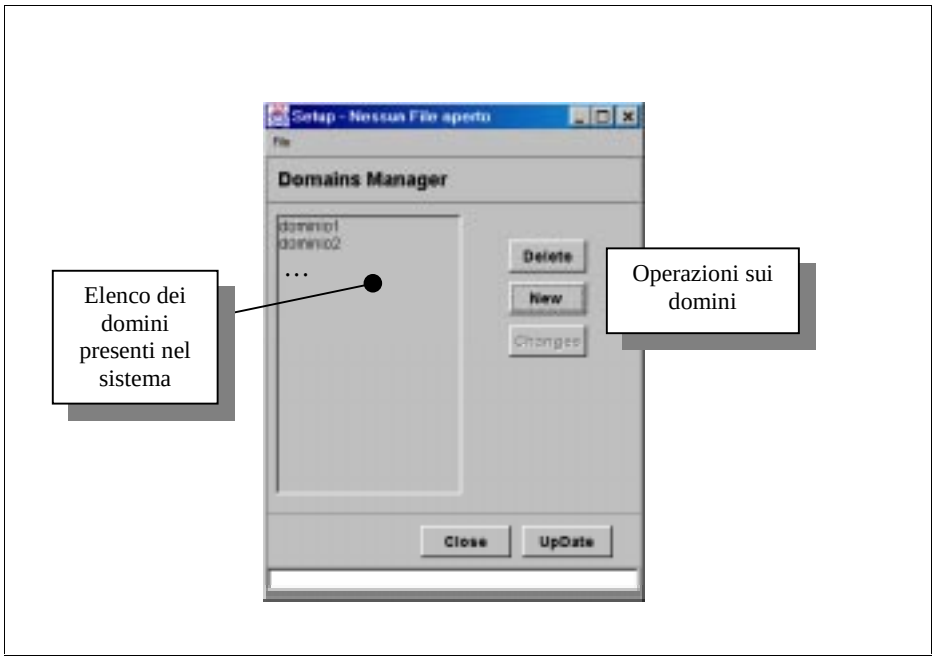

Figura 4-10 Domains Manager

Il Domains Manager consente di modificare e visualizzare l'intera struttura topologica di SOMA. Nella scherma principale (Figura 4-10), sono mostrati i Domini di cui è composto il sistema ed una serie di pulsanti per effettuare modifiche su tale elenco. I pulsanti "New" e "Delete", consentono rispettivamente di aggiungere o cancellare un Dominio. Con il pulsante "Changes", è possibile accedere ad una ulteriore schermata che consente di interagire con la configurazione dei Place del Dominio selezionato. Infine troviamo il pulsante "Update" che discutiamo dopo aver parlato della schermata relativa alla configurazione del Dominio. In tale finestra si possono notare tre sezioni principali (Figura 4-11). Nella prima è contenuto un elenco dettagliato dei Place, si noti la

presenza del flag “Status” che informa sullo stato di attivazione di un Place.

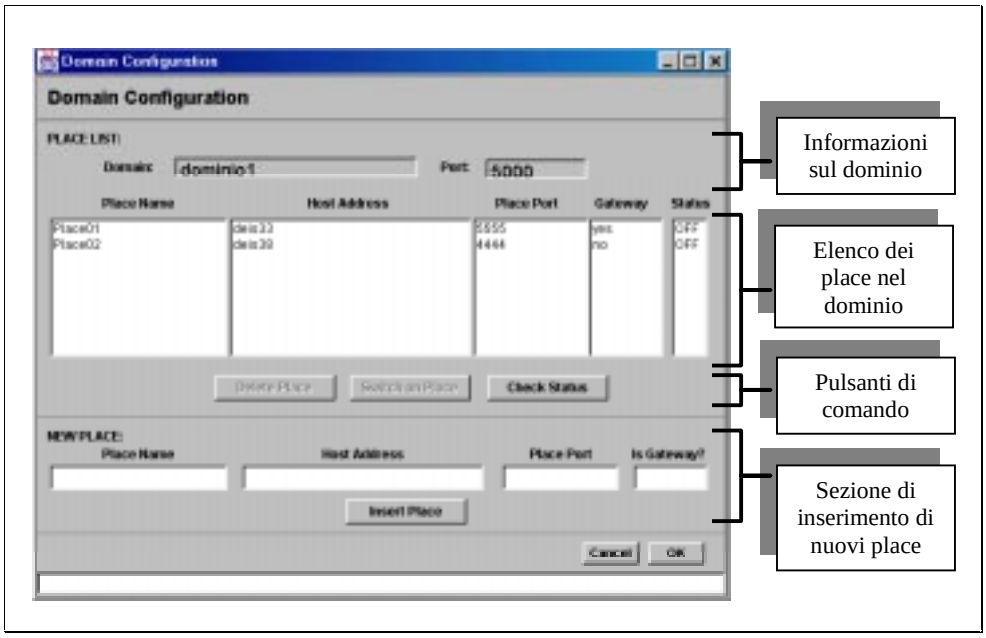

Figura 4-11 Schermata per la gestione di un Dominio

Nella seconda sezione compaiono tre pulsanti: “DeletePlace” che ovviamente consente di cancellare un Place dal Dominio; “Switch On Place” che permette di lanciare un Comando per l’attivazione remota del Place selezionato dall’elenco; “Check Status” che, infine, mette in esecuzione un comando che ha il compito di esaminare lo stato di attivazione dei Place e quindi effettuare il “refresh” dei flag di stato presenti nell’elenco. La terza ed ultima sezione, contiene i campi di testo per immettere un nuovo Place. Quando è terminata la configurazione del Dominio, premendo il tasto “OK” si torna alla schermata precedente, quella che visualizza la lista dei Domini. La configurazione, non è spedita al Supervisor fintantoche non viene premuto il tasto “Update”; con esso, si trasmette la configurazione opportunamente cambiata, al

Supervisor, il quale aggiorna in tempo reale le sue strutture dati che conservano tali informazioni (memorizzandole anche su disco) e si collega con i Gateway attivi in quel momento per comunicare anche a loro l'avvenuto cambiamento. Per quanto riguarda i Gateway non attivi, essi otterranno dal Supervisor la configurazione aggiornata quando verranno messi nuovamente in esecuzione: nella fase di attivazione è previsto il collegamento con il Supervisor per ottenere i dati di configurazione. Lo stesso processo avviene per l'aggiornamento dei Place da parte dei Gateway. Quindi una modifica alla configurazione generale del sistema, viene immediatamente recepita da tutti gli elementi attivi (Gateway e Place). Quelli non attivi la ottengono al momento del loro avvio. È importante osservare che quando viene aggiunto un nuovo Place in un Dominio, il Supervisor genera la coppia di chiavi pubblica e privata necessarie ad esso, e gliela trasmette. Si veda il par. 4.4.7, per una discussione più dettagliata su questi aspetti.

4.4.5.2 Users Manager

Vediamo nella Figura 4-12, la schermata principale dell'Users Manager. In essa è presente un display di output che visualizza i messaggi contestuali alle operazioni eseguite. In basso, si trovano i pulsanti: "Show Users", che consente di visualizzare nel display tutti gli utenti ammessi nel sistema; "Add User" che permette di aprire una nuova finestra in cui specificare i dettagli di un nuovo utente; infine "Delete User", che elimina l'utente selezionato dalla lista mostrata nel Display. Analizziamo l'uso ed il funzionamento di tale strumento in dettaglio nel contesto della discussione dei tre aspetti principali di gestione delle chiavi: generazione, distribuzione e revoca.

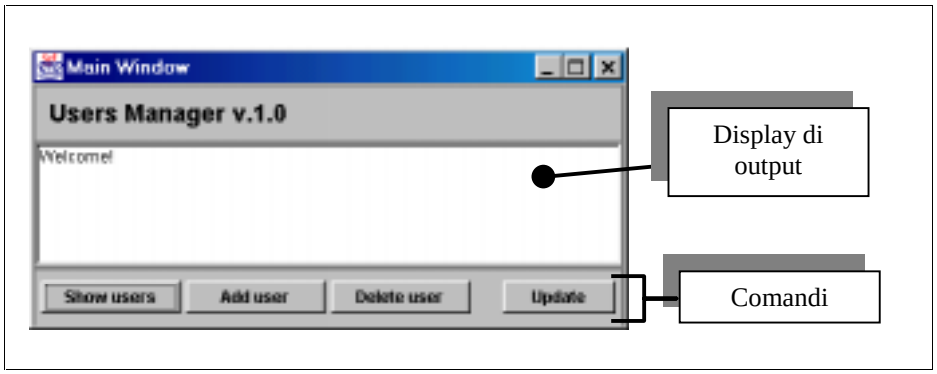

Figura 4-12 Schermata principale dell'Users Manager

4.4.6 Generazione e memorizzazione delle chiavi

Distinguiamo tra chiavi di utente e di Place perché esse vengono gestite in modo differente: la generazione delle prime, avviene in modo esplicito per ordine dell'amministratore di sistema; mentre le seconde sono create in modo automatico quando viene configurato il sistema.

4.4.6.1 Generazione chiavi utente

Quando un nuovo utente vuole accedere al sistema, cioè intende mettere in esecuzione degli agenti, la prima volta, deve rivolgersi all'amministratore, il quale gli attribuisce un account utente valido in tutto il sistema. Vediamo come avviene questa operazione. L'amministratore di sistema, utilizza la funzione "AddUser" dell'Users Manager. Questa consente di visualizzare una finestra in cui inserire i dati personali dell'utente: Nome e Cognome, indirizzo di e-mail, alias e password (Figura 4-13). Esiste una seconda casella per la verifica dell'immissione della password corretta.

Distinguiamo le informazioni inserite in informazioni personali ed informazioni di account: quelle personali, sono utilizzate per la generazione del certificato. Se tutti i campi sono compilati e le password fanno match, alla pressione del tasto “OK”, inizia la vera e propria generazione della coppia di chiavi e del certificato.

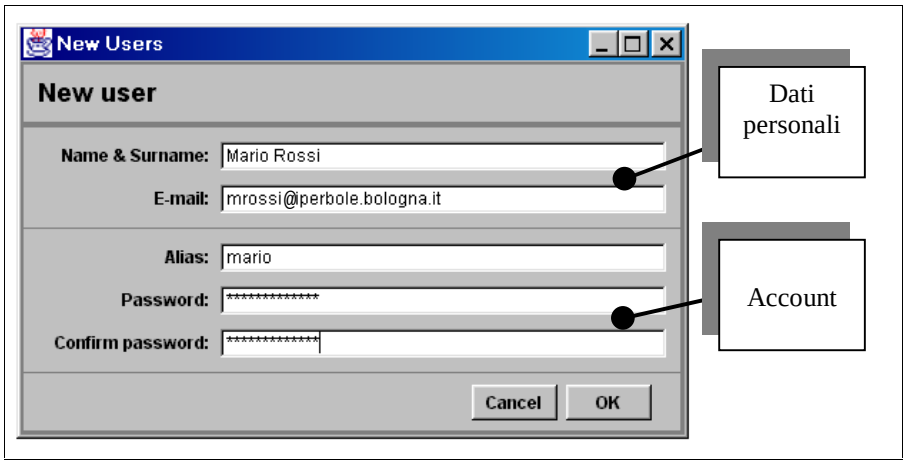

Figura 4-13 Schermata per l’immissione dei dati di un nuovo utente

Vediamo nella Figura 4-14, la sequenza di operazioni necessarie a tale generazione: viene aggiunto il provider IAIK, che fornisce gli oggetti crittografici necessari a generare la coppia di chiavi e il certificato in cui vengono inseriti i dati personali dell’utente e la sua chiave pubblica.

Successivamente, tale certificato viene firmato con la chiave privata del Supervisor ed inviato ad esso. Il passo ulteriore consiste nel memorizzare la chiave privata ed il certificato in un database protetto, il KeyStore. Questo database viene fornito dalle classi crittografiche JCA di SUN e verrà trattato in modo più approfondito nel par. 4.4.6.3. È il Supervisor che si occupa di memorizzare su un supporto permanente le chiavi private e i certificati e successivamente della loro distribuzione.

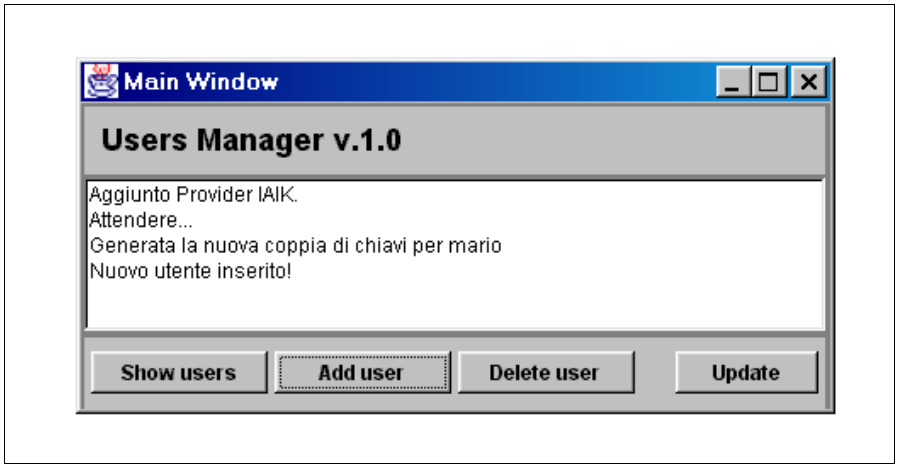

Figura 4-14 Generazione della coppia di chiavi utente

4.4.6.2 Generazione chiavi di Place

La generazione delle chiavi che verranno impiegate dai Place per trasferire oggetti cifrati con il TransCommand, avviene in modo trasparente all'amministratore di sistema. Abbiamo visto che l'utilizzo della funzione "Update" del tool Domains Manager, produce il trasferimento delle strutture dati contenenti la nuova configurazione del sistema al Supervisor. Quando il Supervisor riceve tale configurazione, controlla se in essa sono presenti nuovi Gateway e, se è così, si occupa della generazione delle coppie di chiavi pubblica e privata necessarie ai Place in cui risiedono tali Gateway. Quindi memorizza le coppie generate in un database, l'RSAKeyStore, differente dal KeyStore delle chiavi e dei certificati utente, perché ha il compito di contenere solamente chiavi utilizzate dall'algoritmo per la cifratura RSA. Anche questo database viene mappato in un file. Successivamente ogni Gateway, confronta l'elenco dei Place, che gli è viene inviato, con quello che

ha in memoria, se trova un Place nuovo, genera la coppia di chiavi per tale Place, e la memorizza in un RSAKeyStore locale. Quando il nuovo Place si attiverà, gli invierà la nuova coppia di chiavi.

Si è deciso di mantenere separati i database KeyStore e RSAKeyStore per agevolare futuri interventi, mirati ad ottenere una comunicazione cifrata più efficiente tra Place, ad esempio con l'adozione del protocollo SSL.

4.4.6.3 Il KeyStore

Il KeyStore è un oggetto crittografico messo a disposizione da Java 1.2. Rappresenta un database cifrato in cui i record possono essere di due tipi. Il primo consiste in chiavi singole: chiavi da utilizzare negli algoritmi simmetrici oppure chiavi private per cifrari asimmetrici. Il secondo tipo di entry è costituito da record composti da catene di certificati. La catena è necessaria per garantire la validità dell'associazione tra il Principal e la sua chiave pubblica. Se consideriamo una catena composta da due certificati, il primo è quello del Principal, firmato da un'autority fidata. Il secondo certificato, è quello che contiene la chiave pubblica di tale autority, ed è self-signed. Generalizzando, ogni certificato della catena potrebbe essere firmato dall'autority certificata in quello successivo, fino ad arrivare all'ultima autority, con un certificato self-signed. In questo modo è possibile organizzare gerarchicamente le autority. Quindi per verificare la correttezza di un certificato a capo di una catena, è necessario utilizzare la chiave contenuta nel secondo, che a sua volta deve essere verificato con quella nel terzo e così via fino all'ultimo, di cui si è certi poiché la firma deve essere nota. La firma dell'autority a capo della gerarchia, ultimo anello della catena, deve essere universalmente nota, diffusa anche con mezzi non informatici [KauPS95], in modo da poter sempre verificare la non manomissione dell'ultimo

certificato. Ovviamente maggiore è lunga la catena peggiori sono le prestazioni del processo di verifica.

La protezione offerta dal database KeyStore, è su due livelli: I record contenenti le chiavi singole sono protetti da una password individuale. Mentre l'accesso ai certificati non richiede alcuna password. È inoltre presente una seconda password che consente di verificare l'integrità dell'intero database.

Nella prima versione di Java 1.2 (beta 2), il KeyStore è fortemente improntato sulle strutture dati utilizzate dal package JCA di SUN. Quindi è in grado di contenere solamente certificati emessi da tale provider. Lo strumento messo a disposizione da Java per la generazione e gestione di certificati è il tool a linea di comando: keytool [JDK98]. Il KeyStore, dunque in tale versione può contenere solamente certificati generati con il keytool. Con l'ultima versione di Java 1.2, più precisamente con la relase beta 4, il KeyStore è stato trasformato in un database aperto. Questo significa che un diverso provider può personalizzare tale database per gestire i propri certificati.

In SOMA è utilizzata la versione di Java 1.2 beta 2, e si sfruttano i certificati prodotti da IAIK. Dunque per poter utilizzare il KeyStore di Java, si è eseguito un cast del formato del certificato proprietario di IAIK nel formato più generico "java.security.cert.Certificate". Questo cast è lecito poiché avviene tra classi parenti, ma comporta la perdita di alcuni metodi. Infine ricordiamo che il KeyStore non può essere, ovviamente, serializzato, questo particolare, insieme agli altri, ha condizionato l'architettura del sistema di distribuzione delle chiavi, che vedremo nel paragrafo successivo.

4.4.7 Distribuzione delle chiavi

Come nei paragrafi precedenti, distinguiamo qui, la discussione sulla distribuzione delle chiavi in due parti: distribuzione chiavi utente e distribuzione delle chiavi dei Place.

4.4.7.1 Distribuzione delle chiavi utente

L'amministratore di sistema, dopo aver generato un nuovo account, con il tool Users Manager, premendo in tasto "OK", invia queste informazioni al Supervisor, il quale deve trasmetterle ai Gateway, che, a loro volta, le propagano ai Place. L'architettura dell'infrastruttura di trasmissione è fondata sulla presenza di un demone, il "keystoreDEM". Il Supervisor e tutti i Gateway possiedono tale demone, egli è un servitore di KeyStore: rimane permanentemente in ascolto su un canale di comunicazione (socket), non appena giunge una richiesta (vedremo come, più avanti), risponde trasmettendo una copia del KeyStore mantenuto nella memoria locale. Questa architettura è mostrata in Figura 4-15.

Il demone presente nel Supervisor spedisce il KeyStore ai Gateway. I demoni sui Gateway, consegnano il KeyStore ai Place. In questo modo, quando un utente inserisce i propri dati di account tramite il Launcher, il Place, utilizza l'alias e la password dell'utente per estrarre la chiave privata dal database protetto. Il Place non deve possedere nessuna password per estrarre le chiavi pubbliche dai certificati

Abbiamo detto che i demoni di KeyStore, spediscono quando giunge loro una richiesta. Vediamo ora come viene sviluppata e da chi. Esiste in SOMA un Comando che svolge questo compito: il "KeystoreUpdateCommand"., per la verifica della firma degli agenti.

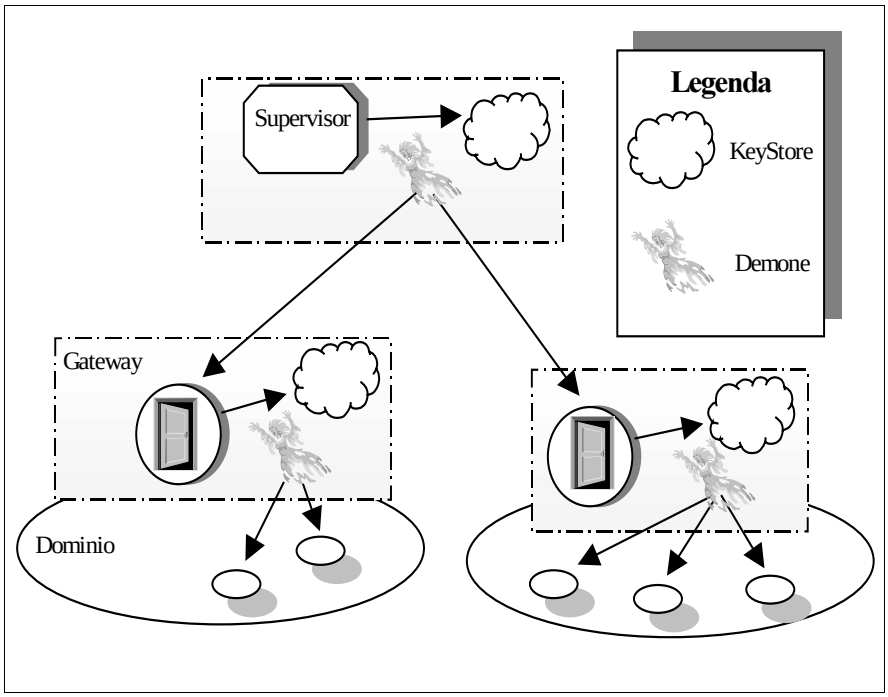

Figura 4-15 Architettura demoni keystoreDEM

Il Supervisor, quando vengono apportate modifiche dall'amministratore nel KeyStore locale, invia ai Gateway tale Comando, che contiene la sequenza di istruzioni per collegarsi al keystoreDEM del Supervisor ed aggiornare opportunamente la propria copia in memoria del KeyStore. Quindi il Gateway, terminato l'aggiornamento, invia lo stesso comando ai suoi Place, personalizzato (con parametri diversi) affinché si possano collegare al demone del Gateway. I Place ed i Gateway non attivi al momento dell'invio del Comando di aggiornamento del KeyStore, si collegano ai rispettivi demoni, durante la fase di inizializzazione.

4.4.7.2 Distribuzione delle chiavi dei Place

Mentre la distribuzione delle chiavi utente, avviene tramite il demone keystoreDEM, quella delle chiavi utilizzate dai Place per la cifratura, segue lo stesso percorso gerarchico che compie la struttura dati della configurazione. Per l'aggiornamento dei dati di configurazione, si utilizzano due Comandi:

- "ConfigurationUpdateCommand"
- "ConfigurationRequestCommand"

Vediamo cosa sono e in che ordine vengono impiegati. Quando viene premuto il tasto "Update" del Domains Manager, il Supervisor invia a tutti i Gateway attivi in quel momento il Comando "ConfigurationUpdateCommand", che contiene tutte le strutture dati aggiornate, compreso il database RSAKeystore e, nel metodo "exe", le istruzioni che li inseriscono nelle strutture dati locali. Si veda [Chi98] per una descrizione dettagliata sui Comandi. I Gateway, a loro volta aggiornano i rispettivi Place. Quando invece un Place è inattivo, nella fase di inizializzazione, invia il ConfigurationRequestCommand al proprio Gateway che gli risponde inviando il ConfigurationUpdateCommand.

4.4.8 Revoca delle chiavi

La revoca delle chiavi utente, avviene utilizzando il tool Users Manager (Figura 4-16). Quando l'amministratore di sistema deve eliminare un utente e le sue chiavi, lo seleziona dall'elenco utenti ammessi ed usa la funzione di cancellazione "Delete User" presente in tale tool.

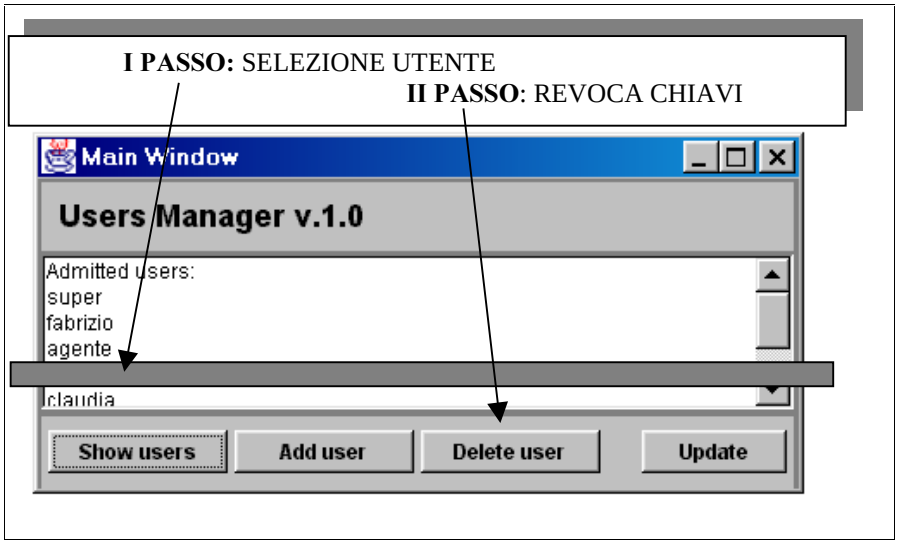

Figura 4-16 Eliminazione account utente

Premendo il tasto “Update” vengono cancellate la chiave privata ed il certificato associato dal KeyStore del Supervisor. Il Supervisor provvede a memorizzare la modifica in modo permanentemente, che poi si occupa di propagarla in tutto il sistema. Ciò avviene semplicemente inviando il Comando “KeystoreUpdateCommand” a tutti i Gateway, e Place, con lo stesso meccanismo di propagazione descritto nel par. 4.4.7.1.

Non è previsto alcun meccanismo per la revoca delle chiavi dei Place, poiché essi, essendo parte integrante del sistema, richiedono cambiamenti molto meno frequenti: tale problema non è tanto grave ed urgente quanto quello dell’allontanamento di un utente.

4.5 Un esempio di applicazione sicura

Vediamo in questo paragrafo, un esempio per la realizzazione di un'applicazione sicura. Consideriamo l'applicazione di installazione del software, descritta nel paragrafo 2.6 e vediamo quali sono state le scelte di sicurezza che ne hanno condizionato la progettazione. L'applicazione è costituita da tre agenti: un agente che si occupa dell'interfaccia grafica necessaria all'interazione con l'utente; un agente che si sposta nei vari Place per la raccolta del software da installare e la sua installazione; ed infine un agente di servizio, che recupera alcune informazioni sulla configurazione attuale del sistema.

Ad un certo punto l'agente che gestisce l'interfaccia grafica, deve generare altri agenti. È necessario attribuire a questi un Principal per garantire loro alcune credenziali. L'agente creatore non può ovviamente possedere la chiave del suo Principal, dunque una soluzione potrebbe essere quella di trasportare una chiave generica la cui firma garantisca sempre un insieme di permessi minimale, ma un agente con un simile grado di fiducia potrebbe fare talmente poco da risultare inutile per la quasi totalità delle applicazioni. Dunque, si deve ricorrere ad una soluzione progettuale: se un agente, deve creare un nuovo agente, deve tornare nel Place mappato nell'host in cui si è collegato il Principal, perché è l'unico Place a possedere la risorsa (display) che consente di interagire con esso e sospendersi finché il Principal stesso non ha immesso i propri dati di account, necessari al Place per reperire la chiave privata e firmare il nuovo agente.

Nel nostro esempio, questo comporta l'inserimento dei dati di account utente per due volte: la prima è richiesta dal Launcher, per lanciare l'agente fisso che gestisce l'interfaccia grafica. La seconda è necessaria perché l'agente fisso, possa generare gli altri due agenti firmati che dovranno muoversi nel sistema. Dunque il

Principal firma i tre agenti, la prima volta per imposizione del Launcher, le altre su esplicita richiesta del suo agente. Gli agenti così creati si possono spostare ed essere sottoposti al meccanismo di controllo di accesso, che verifica tutti i permessi necessari ad effettuare le operazioni desiderate.

CONCLUSIONI

I problemi di congestione del traffico di rete e di flessibilità dei servizi ci hanno spinto a cercare nuovi modelli di progettazione per le applicazioni su sistemi distribuiti, in grado di superare i limiti del classico paradigma cliente/servitore. Dall'analisi effettuata sulle caratteristiche di un sistema ad agenti mobili, possiamo considerare questo nuovo paradigma adatto per la realizzazione di applicazioni distribuite su larga scala, con un campo di azione eterogeneo, aperto e dinamico come Internet.

Per poter sfruttare il modello ad agenti mobili, è stato necessario realizzare un supporto con caratteristiche di apertura e scalabilità, in grado di superare i problemi di eterogeneità dei sistemi distribuiti e mantenere un adeguato livello di sicurezza. Le caratteristiche richieste dal supporto hanno portato all'adozione di Java come linguaggio di implementazione. La soluzione dei problemi di sicurezza è stata affrontata sfruttando le caratteristiche di Java, sia a livello di linguaggio che di strumenti crittografici offerti.

Il problema della sicurezza nel sistema ad agenti SOMA è stato affrontato secondo due punti di vista: la protezione dell'ambiente e la protezione dell'agente. Il supporto SOMA è stato suddiviso in due astrazioni di località, i Place e i Domini, che hanno permesso una naturale suddivisione delle scelte di politica da adottare nei diversi contesti.

La decisione di adottare modelli di autenticazione e autorizzazione flessibili e facilmente configurabili ha permesso agli

ambienti di esecuzione di associare ad ogni agente un preciso grado di fiducia, e ha consentito di effettuare controlli minuziosi sull'accesso ad ogni tipo di risorsa. Alla base dei meccanismi di sicurezza vi sono gli algoritmi crittografici: la loro forza è concentrata totalmente sulla complessità delle chiavi che giocano il ruolo fondamentale all'interno dell'algoritmo stesso. Nel progetto del supporto sicuro per agenti mobili, è stata dedicata una particolare cura al management dinamico, gestito dalla figura dell'Amministratore di Sistema, delle chiavi crittografiche, in particolare alla loro generazione, distribuzione e revoca. È stata realizzata una Certificate Autority che amministra i certificati dei nuovi utenti e memorizza le chiavi private in un database cifrato.

È stato inoltre implementato un meccanismo di controllo d'accesso che permette di interagire con il sistema solo agli utenti autorizzati che, dopo una fase di autenticazione, vengono associati in maniera non ripudiabile agli agenti che generano, diventando così responsabili, a tutti gli effetti, della loro esecuzione.

Infine è stata sviluppata un'applicazione per l'installazione di software, per mettere in rilievo, in un concreto esempio d'uso, come i meccanismi di sicurezza implementati nel sistema entrino in gioco e quali criteri debbano essere seguiti dal programmatore in fase progettuale.

Il lavoro svolto ha permesso di sfruttare la nuova tecnologia ad agenti mobili, come promettente soluzione in termini di efficienza e prestazioni, senza rinunciare ad un adeguato livello di sicurezza, (modificabile dal programmatore in relazione al grado di fiducia accordato al particolare contesto di esecuzione ed alla criticità dell'applicazione), sia per quanto riguarda la protezione dell'agente che delle risorse su cui l'agente esegue.

BIBLIOGRAFIA

[ADOBE95] Adobe System Incorporated, “PostScipt Language Reference Manual”, Addison Wesley, 1985.

[ARA98] Holger Peine. “Security Concepts and Implemenyation in the Ara Mobile Agent System”, IEEE Published in the Proceedings of WETICE’98. 17-19 June 1998.

[ArnG96] K.Arnold, J.Gosling. “The Java Programming Language”, Addison Wesley 1996, http://www.sun.com.

[Bog93] J.K. Boggs, “IBM Remote Job Entry Facility: Generalize Subsystem Remote Job entry Facility”, IBM Tecnical Disclosure Bullettin 752, IBM 1973.

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

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

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

[Chi98] Claudia Chiusoli, “Architetture dinamiche per ambienti ed applicazioni ad agenti mobili”, tesi di Laurea presso l’Università di Ingegneria Informatica di Bologna, 1998.

[Corr97] Prof. Antonio Corradi, “Corso di Reti di Calcolatori” presso Università di Ingegneria Informatica di Bologna, 1997.

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

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

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

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

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

[JDK98] “Java Development Kit” JDK Software Version 1.2 Beta 4. java.sun.com/products/jdk/1.2/docs/

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

[Kor98] Jukka Korpela “HTML 3.2 by Examples” October 1998 http://www.w3.org/MarkUp/Wilbur/.

[LeaJP93] R. Lea, C. Jacquemont and E. Pillevesse, “COOL: System Support for Distributed Object-Oriented programming”, Comm. Of ACM, vol. 36, n.9, pp. 37,46, 1993.

[Oak98] Scott Oaks. “Java Security” O’Reilly & Associated, Inc. May 1998.

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

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

[SOMA98] Sistema progettato nell’ambito del "Project Design Methodologies and Tools of High

Performance Systems for Distributed Applications" fondato dal Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST). Reperibile dal 1998 presso: http://www-lia.deis.unibo.it/Software/MA/.

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

[Tei97] Jeremy T. Teitelbaum "Honors Seminar in Cryptography" University of Illinois at Chicago Spring Semester, 1997 http://raphael.math.uic.edu/~jeremy/crypt/intro.html.

[Ten98] L. Tenti, tesi di Laurea presso l'Università di Ingegneria Informatica di Bologna, 1998.

[WEB1] www.java.sun.com.

[WEB2] www.systemics.com/docs/cryptix/

[WEB3] jcewww.iaik.tu-graz.ac.at/IAIK_JCE/jce.htm.