Università degli studi di Napoli Federico II

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

TESI DI LAUREA

Reflective Middleware e Pervasive Computing: Un Approccio Basato su Plug-ins

Relatori:

Ch.mo Prof. Antonio d'Acierno

Ch.mo Prof. Giuseppe De Pietro

Candidato:

Mario Ciampi

Matr. 041/2768

Correlatore:

Ing. Giuliano Gugliara

ANNO ACCADEMICO 2002-2003

Indice

Prefazione

La diffusione ormai notevole dei dispositivi mobili ha portato alla determinazione di un nuovo modello di computazione, denominato *ubiquitous computing*, secondo il quale quest'ultima deve avvenire in ogni luogo, in ogni istante e con ogni tipo di dispositivo dotato di capacità di calcolo.

Per far ciò, un ambiente popolato da più dispositivi eterogenei deve essere capace di fornire ad un utente poco esperto, che utilizza un qualsiasi device, servizi ed informazioni che i dispositivi elettronici gli mettono a disposizione. Un ambiente di questo tipo prende il nome di ambiente *pervasivo*.

L'utente non deve preoccuparsi di connettersi al sistema, né di configurare il suo device; inoltre, deve avere la possibilità di offrire servizi che altri utenti potranno utilizzare. Dal punto di vista dell'infrastruttura software, una delle chiavi per rendere possibile una continua computazione è garantire un'interoperabilità senza limiti con i dispositivi dell'ambiente pervasivo. In tale ambiente l'eterogeneità è indispensabile poiché differenti piattaforme hardware e software e sistemi operativi devono coesistere.

L'utilizzo dei *middleware* nello scenario dell'ubiquitous computing semplifica la costruzione e la comunicazione dei componenti basati su applicazioni distribuite, ma non ha la flessibilità adeguata alle specifiche. Anche se alcune piattaforme middleware convenzionali si adattano manualmente ad ogni dispositivo particolare, non c'è la possibilità di cambiamenti dinamici a tempo di esecuzione.

Il *reflective middleware* introduce questa flessibilità e dinamicità, sfruttando un protocollo meta-object che permette di ridisegnare,

anche a tempo di esecuzione, l'architettura del motore di un middleware (come politiche, meccanismi o strategie); è così che diverse piattaforme middleware su differenti sistemi possono interoperare in uno scenario di ubiquitous computing.

Molti degli attuali reflective middleware richiedono un insieme di requisiti hardware e software ai mainframe e desktop, che Personal Digital Assistant (PDA), PocketPC o piccoli dispositivi elettronici non possono soddisfare.

Il lavoro di ricerca per inseguire le esigenze di raggiungere in un ambiente pervasivo dispositivi con poche risorse, e di non utilizzare soluzioni proprietarie, ha condotto alla sperimentazione di un nuovo prodotto: UIC. L'*Universally Interoperable Core* (UIC) è un particolare reflective middleware, utilizzato per il progetto Gaia di ubiquitous computing presso l'University of Illinois, che offre un meccanismo di interazione tra device, middleware, sistemi operativi, reti e protocolli di trasporto attraverso l'utilizzo delle personality.

In pratica, in UIC, sono definite le classi che realizzano i meta-object per la definizione di aspetti essenziali della comunicazione (livello UIC-Core) e classi derivate che definiscono il comportamento di componenti di una specifica piattaforma middleware (livello Specialization).

A partire da UIC è stato possibile realizzare un reflective middleware composto da più componenti, i plug-ins, che permettono di comunicare con diversi tipi di middleware, quali CORBA e Java RMI, in modo del tutto trasparente al client. Il cuore del middleware è un modulo che, mettendo a disposizione del client, attraverso un'interfaccia, un insieme di funzionalità astratte, carica il giusto plug-in che permette ad un programma client di interagire utilizzando un qualsiasi protocollo di comunicazione.

Questo modulo del middleware provvederà a scaricare il plug-in che non serve più, garantendo una gestione delle risorse efficiente, necessaria per i piccoli dispositivi.

Per poter mettere in luce le potenzialità di questo scenario si è implementato un'*esempio d'uso*: un navigatore per visualizzare immagini. Si ha così la possibilità di offrire un servizio ad un utente che può usare anche piccoli device senza che il client sia limitato a far riferimento a server che comunicano con lo stesso middleware.

Questo caso d'uso può essere provato in uno scenario costituito da più dispositivi eterogenei che comunicano tra loro utilizzando differenti protocolli di comunicazione.

Capitolo 1
Ubiquitous e pervasive computing

1.1 Introduzione

Negli ultimi anni si sta assistendo ad un'affermazione sempre più marcata di dispositivi di calcolo dalle dimensioni ridotte e soprattutto mobili, dovuta alle evoluzioni tecnologiche sia nel campo dei microprocessori che in quello delle reti di comunicazione, accompagnata, peraltro, dallo sviluppo di nuovi standard basati su tecnologie di comunicazione wireless.

Sulla base di questi presupposti un numero sempre più elevato di utenti interagiranno tra loro tramite dispositivi mobili dalle differenti caratteristiche sia hardware che software; una tendenza, questa, che sta conducendo all'affermazione di un nuovo modello di calcolo denominato *ubiquitous computing* [1,2].

Il concetto di ubiquitous computing implica da un lato la capacità da parte del dispositivo di ottenere informazioni dall'ambiente nel quale è "calato" e di utilizzare queste ultime per costruire dinamicamente un modello di interazione adeguato all'ambiente stesso, dall'altro la capacità da parte dell'ambiente di accettare nuovi utenti e di riconfigurarsi ed adattarsi dinamicamente alle caratteristiche hardware e software di questi ultimi.

L'obiettivo principale dell'ubiquitous computing, dunque, è spingere gli utenti ad eseguire una continua computazione, ovunque, in ogni istante e con qualunque dispositivo dotato di capacità di calcolo.

Questo scenario introduce un nuovo modello di computazione, il cosiddetto *pervasive computing*, che può essere definito come la possibilità di ottenere e gestire informazioni in maniera semplice, efficiente e rapida, indipendentemente dalla locazione fisica e per mezzo di strumenti semplici quali PDA, cellulari o console ludiche oltre ovviamente che per mezzo degli office-pc, che interagiscono tra loro per utilizzare i servizi che l'ambiente gli mette a disposizione.

1.2 L'evoluzione storica verso il modello del pervasive computing

Nel 1991 Mark Weiser, capoufficio del dipartimento tecnologico del Centro di Ricerca di Palo Alto (PARC) della Xerox, mostrò una visione del computing che si opponeva all'ubiquità dei personal computer. "Le tecnologie più profonde sono quelle che scompaiono, che saranno nascoste nella fabbrica della vita giornaliera finché sarà indistinguibile dalla vita stessa" [3], scrisse. "Le macchine entreranno nell'ambiente umano senza che l'uomo si sforzi più di entrare in quello delle macchine, l'uso del computer dovrà essere ristoratore quanto una passeggiata nei boschi".

I significativi sviluppi hardware, come la diffusione dei sensori, le comunicazioni wireless e la rete globale, hanno reso la visione di Weiser una strada percorribile concretamente, anche se i computer sono ancora visti come macchine su cui girano programmi in un ambiente virtuale nel quale entriamo per realizzare un compito e usciamo quando abbiamo finito.

Il pervasive computing presume una visione completamente differente: un dispositivo rappresenta un ingresso in uno spazio di applicazione dati, non un deposito di software personalizzati che un utente deve gestire; un'applicazione è il mezzo con il quale un uten-

te realizza un compito e non del software scritto per sfruttare le capacità di un dispositivo; un ambiente per il computing è uno spazio fisico di informazioni avanzate, non un ambiente virtuale che esiste per memorizzare e far funzionare il software [2].

La necessità dell'informazione percettiva sull'ambiente differenzia il pervasive computing dal traditional computing. I dispositivi percettivi danno ai sistemi pervasivi delle informazioni, come le locazioni delle persone e dei device, e questi possono usarle per interagire in modo più naturale con gli utenti.

Il pervasive computing matura la propria evoluzione a partire dal lavoro che incominciò nella metà del 1970, quando il primo PC portò il computer più vicino alle persone. Nella visione di Weiser, comunque, l'idea di creare un computer personale è tecnologicamente mal posta, infatti essa mantiene il computing separato dalla nostra vita odierna.

Sebbene il PC non abbia distribuito il completo potenziale di information technology agli utenti, certamente ha avviato il primo passo verso la creazione di computer popolari, anche se non pervasivi. Inoltre, è stato anche un fattore strumentale per la crescita fenomenale dei componenti hardware e per lo sviluppo di interfacce utente grafiche.

Poi con l'avvento del networking, il personal computing è evoluto verso il *distributed computing*. Quest'ultimo ha contrassegnato il passo successivo verso il pervasive computing attraverso l'introduzione dell'accesso a informazioni remote e con la comunicazione fault tolerant, ad alta disponibilità e con sicurezza [1].

Sebbene il World Wide Web non sia stato designato per essere un'infrastruttura per il distributed computing, la sua ubiquità a livello rete l'ha fatto diventare una scelta attraente per la sperimentazione di concetti di distributed computing. Esso ha anche creato una cultura che è sostanzialmente più disponibile allo sviluppo di am-

bienti di pervasive computing rispetto alla cultura che esisteva ai tempi in cui Weiser scrisse la sua visione.

Tutto ciò ha dimostrato che è possibile distribuire il computing senza la perdita della scalabilità. I semplici meccanismi che permettono di collegare risorse hanno fornito un mezzo per integrare le basi dell'informazione distribuita in una singola struttura.

Ancora più importante, il Web ha aperto la strada verso la creazione di un'infrastruttura di informazioni e comunicazioni sempre più caratterizzate dall'ubiquità. Molti utenti ora riferiscono i loro punti di presenza nel mondo digitale, tipicamente tramite homepage, portali o indirizzi e-mail. Il computer che usano per accedere a questi "siti" è diventato largamente irrilevante. Sebbene il Web non pervade il mondo reale di entità fisiche, è tuttavia un potenziale punto di partenza per il pervasive computing.

Consideriamo infine il *mobile computing*: esso è emerso dall'integrazione della tecnologia cellulare con il Web [4]. Sia la dimensione che il prezzo dei dispositivi mobili sta diminuendo ogni giorno, pertanto questi ultimi potrebbero eventualmente supportare la visione di Weiser riguardo il pervasive computing con dispositivi di scala "pollice", facilmente disponibili agli utenti in qualsiasi ambiente umano. I sistemi di telefonia cellulare che separano il telefono cellulare dalla scheda Subscriber Identity Module (SIM) approssimano questo modello di operazione. Gli abbonati possono inserire la loro scheda SIM e automaticamente usare il telefono cellulare, eseguendo e ricevendo chiamate come se fosse il proprio.

Gli utenti possono già accedere allo stesso punto nel Web con parecchi differenti dispositivi – PC, telefono cellulare, Personal Digital Assistant (PDA) e così via. Le schede SIM dimostrano anche che il sistema terminale sta diventando meno importante rispetto all'accesso al mondo digitale. In questo senso, siamo sulla buona via riguardo al concetto di "scomparsa" dei computer, facendo sì che agli utenti non importi ciò che c'è dietro di loro.

Lo scopo "anytime anywhere" del mobile computing è essenzialmente un approccio reattivo all'accesso dell'informazione, ma prepara la strada per realizzare l'obiettivo del pervasive computing, "all the time everywhere".

Come mostra la Figura 1.1, il pervasive computing è un insieme contenente il mobile computing. In più a quest'ultimo, i sistemi pervasivi richiedono il supporto per l'interoperabilità, la scalabilità, la velocità e l'invisibilità per assicurare che gli utenti abbiano accesso trasparente al computing all'occorrenza.

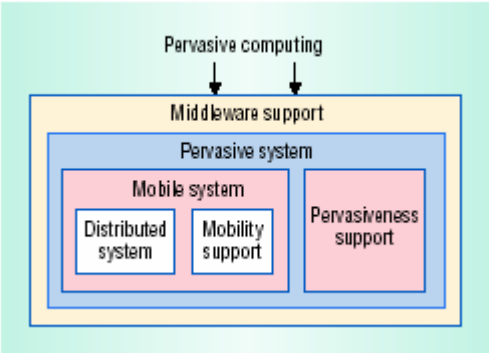

Figura 1.1 Sottoinsiemi del pervasive computing

1.3 Modello del pervasive computing

I progressi tecnologici necessari a costruire un ambiente che realizzi pervasive computing ricadono in quattro vaste aree: i dispositivi, il networking, il middleware e le applicazioni. La Figura 1.2 mostra le loro relazioni.

1.3.1 I dispositivi

Un ambiente intelligente dovrebbe contenere molti tipi di dispositivi differenti:

- dispositivi di input tradizionali, come mouse o tastiere, e dispositivi di output, come altoparlanti o LED;
- dispositivi mobili wireless, come PDA, telefoni cellulari, palmari e così via;
- dispositivi smart, come apparecchi intelligenti o biosensori.

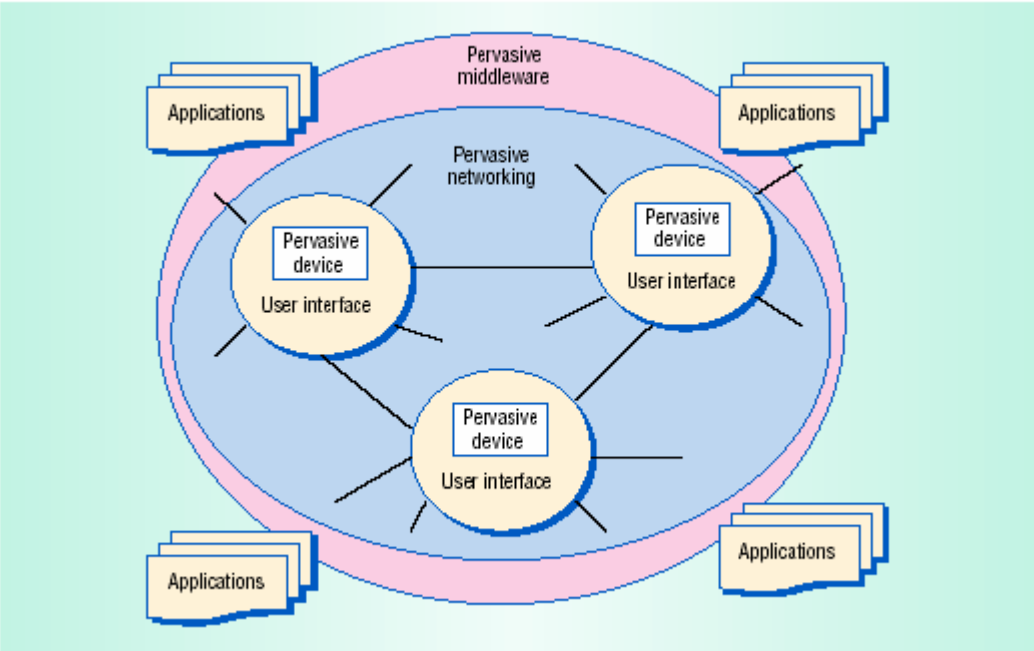

Figura 1.2 Piattaforma del pervasive computing

Idealmente, il pervasive computing dovrebbe comprendere ogni dispositivo che ha incorporata un'intelligenza attiva e passiva. I sensori dovrebbero automaticamente prelevare l'informazione, trasferirla e prendere azioni basate su di essa; essi rappresentano un importante sottoinsieme dei dispositivi pervasivi. Ad esempio i sensori basati sul Global Positioning System forniscono la locazione dei

dati che un dispositivo può tradurre in una rappresentazione interna di latitudine, longitudine ed altezza.

1.3.2 Il networking

E′ supposto che il numero di dispositivi pervasivi si moltiplichi rapidamente nei prossimi anni. Secondo una stima dell′IDC, una società di analisi del mercato, alla fine del 2003, il numero di dispositivi pervasivi dovrebbe aver superato il numero di abitanti a livello mondiale.

A causa di questa proliferazione, molte tecnologie correnti devono essere rinnovate. Oltre ad estendere l′infrastruttura backbone, le reti globali come Internet devono anche modificare le applicazioni esistenti per integrare questi dispositivi pervasivi nei sistemi sociali esistenti [4].

1.3.3 Il middleware

Come il distributed computing ed il mobile computing, il pervasive computing richiede un involucro, il middleware, per interfacciarsi tra il networking e le applicazioni utente che girano su dispositivi pervasivi.

Come mostra la Figura 1.2, questo middleware pervasivo mediera le interazioni con il networking kernel sulle applicazioni utente e terrà gli utenti immersi nello spazio del pervasive computing. Il middleware consisterà per la maggior parte di firmware e pacchetti software che saranno eseguiti nel modello client-server o peer-to-peer.

Le interfacce utente sono un altro aspetto del middleware. I Web browser standard rappresentano l'estremità alta della sofisticazione delle interfacce: essi usano molto più colore, grafica e controlli che gli utenti tipicamente richiedono su dispositivi pervasivi, mentre il mobile computing ha già introdotto i microbrowser.

1.3.4 Applicazioni

Il pervasive computing è più basato sull'ambiente rispetto al mobile computing o alle applicazioni Web-based. Questo significa che le applicazioni porteranno gli aspetti sul middleware e sul networking ad una grande estensione.

In ogni caso, le applicazioni pervasive dovrebbero scomparire dall'ambiente.

In base a quanto detto, con la netta diminuzione delle dimensioni e del prezzo, i dispositivi mobili possono essere validi candidati per supportare la visione di Weiser che ipotizzava un 'aumento graduale dei dispositivi di calcolo pervasivi prontamente disponibili agli utenti in ogni ambiente umano'.

Oggi, il livello di sviluppo raggiunto dalla tecnologia nelle reti di comunicazione e nel campo dei dispositivi mobili pone le basi per una effettiva realizzazione dell'ubiquitous computing attraverso gli ambienti pervasivi.

1.4 Gli ambienti pervasivi

Un ambiente pervasivo deve essere in grado di interagire con una vasta tipologia di dispositivi, che potranno essere consumatori

e/o fornitori di servizi, e che devono avere la capacità di integrarsi tra loro e l'ambiente in maniera altamente trasparente.

Supercalcolatori, workstation, palmari, Personal Digital Assistant, telefoni cellulari e semplici smartcard saranno i principali dispositivi utilizzati all'interno di questi ambienti.

Una simile eterogeneità richiede un elevato grado di adattabilità dei servizi, ossia l'ambiente deve poter riconoscere le caratteristiche hardware e software del particolare dispositivo in base alle quali adattare le funzionalità erogate.

Inoltre, i dispositivi devono integrarsi nell'ambiente ed interagire spontaneamente con esso. Questo requisito richiede meccanismi di configurazione automatica grazie ai quali l'ambiente è in grado di riconoscere l'ingresso e l'uscita dei dispositivi. Al nuovo dispositivo in ingresso l'ambiente deve rilasciare dinamicamente i parametri di configurazione, eliminando ogni operazione manuale a carico dell'utente [5]. Si ottiene in tal modo la spontaneità di interazione che, però, solleva problemi di sicurezza, per cui l'accesso non deve essere indiscriminato, ma devono essere implementati meccanismi di autenticazione ed autorizzazione.

Viceversa, l'evento sollevato dall'uscita di un dispositivo non può essere gestito in modo classico; nascono, infatti, problematiche relative all'affidabilità dei servizi. L'elevata mobilità dei dispositivi può essere la causa di una disconnessione dall'ambiente solo temporanea al termine della quale il servizio dovrà essere ripristinato. L'ambiente deve prevedere un meccanismo che individui il tipo di disconnessione, per evitare un erroneo rilascio delle risorse allocate e permettere che l'utente rientrato riprenda l'elaborazione che era stata sospesa.

E' evidente che per le specifiche caratteristiche di un simile modello, le metodologie classiche di progetto e sviluppo sembrano non adeguate all'implementazione di ambienti e servizi di ubiquitous

computing, proprio perché non pensate per far fronte alle problematiche evidenziate in precedenza.

In particolare, la realizzazione di servizi ed in generale di applicazioni per ambienti pervasivi non può prescindere dalle caratteristiche implicite degli ambienti stessi.

Questo significa principalmente progettare e sviluppare applicazioni sulla base di nuovi requisiti impliciti, finora non considerati, o considerati in maniera diversa, nel caso di applicazioni destinate agli ambienti distribuiti classici.

Di conseguenza, la progettazione e lo sviluppo di applicazioni e servizi per tali ambienti non può essere condotta efficientemente attraverso le classiche metodologie di sviluppo software che non sono state pensate per affrontare le emergenti problematiche dell'ubiquitous computing. In particolare, nuove metodologie, nuovi modelli e nuovi patterns architetturali devono essere sviluppati in modo da ottenere strumenti efficienti per la progettazione e lo sviluppo di componenti software con elevato grado di adattabilità, scalabilità, auto-configurabilità e capaci di interagire dinamicamente e spontaneamente.

1.4.1 Caratteristiche e problematiche

Un ambiente pervasivo è capace di integrare dispositivi altamente eterogenei tra loro. Al suo interno supercalcolatori, tradizionali dispositivi di input e output, laptop, PocketPC, PDA, telefoni cellulari, sistemi embedded e smartcard devono interagire senza risentire il gap hardware e software che li diversifica.

L'estrema eterogeneità dei dispositivi hardware e dei componenti software coinvolti in progetti di pervasive computing richiede un elevato grado di scalabilità e di adattabilità dei servizi forniti dall'ambiente. In particolare, uno stesso servizio deve poter essere

usufruito da dispositivi dalle caratteristiche hardware (intese come disponibilità di risorse) molto diverse tra di loro. Inoltre, tali dispositivi sono tipicamente equipaggiati con software di base diversi e supportano componenti software che utilizzano paradigmi di comunicazione differenti, come ad esempio la tecnologia wireless Bluetooth o lo standard Ethernet per la comunicazione senza fili IEEE 802.11b [4].

Questo comporta notevoli problemi di interoperabilità tra i diversi dispositivi e tra i dispositivi e l'ambiente, che non potranno essere risolti dalle infrastrutture middleware classiche perché esse hanno due importanti limiti: un'infrastruttura "pesante" che non si adatta alla carenza di risorse di molte classi di dispositivi mobili come PDA, telefoni cellulari, smartcard, sistemi embedded ed altri, e le difficoltà ad interagire con altri middleware, se non tramite soluzioni di "bridging" particolarmente complicate e personalizzate.

Ancora, la trasparenza delle interazioni è sicuramente uno degli obiettivi più ambiziosi e allo stesso tempo più rilevanti del pervasive computing. Infatti, la caratteristica di mobilità della maggior parte dei dispositivi pervasivi introduce un elevato numero di eventi sollevati dall'ingresso e dall'uscita dello stesso dispositivo nell'arco di un'unica interazione. Far gravare sull'utente le operazioni di configurazione ogni qualvolta un suo spostamento fisico causa l'uscita e il rientro del dispositivo dall'ambiente sarebbe una scelta non performante.

Raggiungere tale obiettivo presuppone la presenza di meccanismi di auto-configurazione all'interno dell'ambiente pervasivo, tramite i quali è l'ambiente stesso a riconoscere l'ingresso di un nuovo dispositivo e ad inviargli i parametri di configurazione necessari per accedere all'infrastruttura di rete.

Purtroppo, le caratteristiche di auto-configurazione e di interazione spontanea tra i dispositivi e l'ambiente fanno nascere problemi di sicurezza. La possibilità di accedere spontaneamente

all'ambiente, e quindi alle risorse in esso contenute, deve essere monitorata.

Negli ambienti distribuiti classici sono state proposte specifiche per lo sviluppo di meccanismi di accesso sicuro, ma ancora una volta le caratteristiche degli ambienti di tipo pervasivo richiedono soluzioni specifiche.

Per evitare un accesso indiscriminato alle risorse e ai servizi sono necessari meccanismi di autenticazione degli utenti, nonché una politica di certificazione dei diritti [6]. Attraverso la certificazione dei diritti è possibile accordare ad un utente autenticato l'accesso ad una o a più classi di servizi. Tale classificazione permetterà di far risiedere nello stesso ambiente più tipologie di servizi, ognuna delle quali potrà essere acceduta con un differente livello di sicurezza.

In aggiunta agli aspetti precedentemente evidenziati, si deve tener conto di altri due fattori che influenzano la caratterizzazione del modello ubiquitous.

I dispositivi pervasivi interagiscono con l'ambiente richiedendo e/o offrendo servizi. Al suo interno l'ambiente conterrà una serie di funzionalità e risorse che possono essere messe a disposizione e che devono essere note agli utenti. Inoltre, la capacità dei dispositivi di essere anche fornitori di servizi comporta un aggiornamento dinamico, da parte dell'ambiente, dell'insieme dei servizi presenti al suo interno.

Per tale motivo, gli ambienti pervasivi devono essere dotati di servizi di resource discovery, affinché un dispositivo entrante possa ottenere informazioni relative ai servizi ed alle risorse disponibili nell'ambiente.

L'ultimo requisito implicito ancora non evidenziato è quello dell'affidabilità dei servizi offerti. In un ambiente pervasivo l'evento di rilascio improvviso del servizio non può essere considerato una condizione eccezionale o un fallimento dell'operazione al quale far fronte in modo classico. Infatti, se una disconnessione di un disposi-

tivo client in un ambiente classico deve essere trattata come un fallimento/guasto del dispositivo, in un ambiente di tipo pervasivo rappresenta un'operazione ordinaria.

L'intrinseca prerogativa, in base alla quale l'ingresso e l'uscita del dispositivo non devono influire sulla regolare erogazione del servizio, comporta una gestione specifica dell'interruzione del servizio stesso. In particolare, sono necessarie politiche per la gestione dei particolari eventi che possono causare l'uscita di un dispositivo, al fine di determinare, con una ragionevole probabilità di successo, il tipo di disconnessione in corso, ottenendo così servizi di fault tolerance adeguati.

Riepilogando, in un ambiente che supporta un modello di calcolo pervasivo si possono individuare le seguenti caratteristiche intrinseche:

- eterogeneità dei dispositivi hardware e software coinvolti;
- scalabilità e adattabilità dei servizi ai differenti dispositivi;
- spontaneità di interazione tra i dispositivi e l'ambiente.

Da queste caratteristiche scaturiscono una serie di problematiche, quali:

- interoperabilità e integrazione tra i diversi dispositivi;
- sicurezza degli accessi;
- affidabilità dei servizi;
- necessità di servizi di resource discovery.

1.4.2 Servizi di base

Le considerazioni che sono emerse in relazione alle caratteristiche intrinseche di un modello di ubiquitous computing hanno evi-

denziato la presenza di requisiti comuni a tutte le classi di applicazione che si potranno sviluppare e/o installare in un ambiente pervasivo.

Per tale motivo, la possibilità di realizzare efficientemente applicazioni per ambienti di pervasive computing presuppone l'esistenza nell'ambiente di una serie di servizi di base comuni a tutte le possibili tipologie di applicazioni.

In definitiva, l'ambiente deve essere in grado di riconoscere l'ingresso di un nuovo utente e fornirgli automaticamente i parametri di accesso all'infrastruttura di rete. Deve prevenire l'accesso indebito alle risorse ed ai servizi di rete da parte di utenti non autorizzati. L'insieme di risorse e servizi deve essere aggiornato dinamicamente con l'ingresso e l'uscita dall'ambiente dei vari dispositivi. Inoltre, lo scenario pervasivo deve essere dotato di meccanismi che consentano ai dispositivi l'uscita in qualsiasi momento, e senza che questa provochi danni all'ambiente o agli altri utilizzatori, dando la possibilità, a tali dispositivi, di riprendere l'elaborazione, in un momento successivo, a partire dallo stato precedente l'interruzione.

A supporto di queste esigenze possono essere identificati i seguenti servizi di base [7], comuni a tutte le applicazioni presenti all'interno di un ambiente di pervasive computing:

- Dynamic Location Discovery

Implementa la trasparenza delle interazioni. Esso permette all'ambiente il riconoscimento del dispositivo in ingresso e al dispositivo di ottenere l'accesso ai servizi di rete. In tal modo le interazioni tra i dispositivi e l'ambiente avverranno spontaneamente e senza il controllo o la configurazione manuale da parte dell'utente. Il servizio rileva l'evento sollevato da un dispositivo entrante e gli fornisce dinamicamente i parametri di configurazione per accedere alla rete. Il nuovo dispositivo configurato viene registrato per memorizzare la sua presenza all'interno dell'ambiente. Inoltre, nel caso

in cui esso sia un fornitore di servizi, si deve provvedere ad aggiornare l'elenco dei servizi e delle risorse disponibili.

- Dynamic Services e Resources Discovery

  Forniscono agli utilizzatori di dispositivi mobili l'accesso alla directory dei servizi e delle risorse disponibili nell'ambiente. La directory verrà aggiornata dinamicamente dall'ambiente nel caso in cui il dispositivo entrante sia a sua volta fornitore di servizi o risorse.

- Secure Access

  Permette l'accesso ai servizi ed alle risorse condivise ai soli utenti autorizzati. In ambienti pervasivi l'accesso in sicurezza ai servizi ed ai dati rappresenta una problematica di cruciale importanza in quanto le caratteristiche di tali ambienti, in particolare la capacità dei dispositivi di autoconfigurarsi e quindi di accedere automaticamente ai servizi, contrastano con i requisiti di privacy e di integrità [8] che sono tipici delle applicazioni distribuite. Il problema dell'accesso in sicurezza, ad ogni modo, non si limita all'autenticazione dell'utente, ma include anche la certificazione dei diritti. In questo modo, sarà possibile non solo autenticare gli utenti per l'accesso all'ambiente, ma si potrà anche accordare l'accesso a classi di servizi e di risorse differenti in funzione del profilo e dei diritti dell'utente stesso.

- Fault tolerant services

  Questa classe di servizi permette l'implementazione di caratteristiche di affidabilità e di tolleranza ai guasti nell'ambiente. In particolare, nell'ambiente si dovranno realizzare meccanismi attraverso i quali le disconnessioni da parte di un dispositivo uscente non vengano confuse con fallimenti del dispositivo stesso come invece accade negli ambienti classici per applicazioni distribuite. Questo si-

gnifica che da un lato devono essere rilasciate le risorse in precedenza allocate al dispositivo uscito dall'ambiente, dall'altro si dovrà salvare lo stato dell'elaborazione parziale in modo tale da consentire al dispositivo di riprendere in un secondo momento l'elaborazione possibilmente anche in un nuovo ambiente.

1.5 La sicurezza nell'era del pervasive computing

Il nuovo scenario del pervasive computing richiede un modello di sicurezza adeguato che tenga conto dei nuovi media e dei rischi associati; ad esempio il proliferare di device palmari, oltre a portare vantaggi noti ed intuibili, aumenta il rischio del furto o della perdita degli stessi e dunque vanno previste procedure rapide e semplici per disattivarne i codici d'accesso.

1.5.1 Evoluzione del modello di sicurezza

Prima dello sviluppo delle architetture distribuite e del Web, il modello di sicurezza ritenuto valido si basava su quattro pilastri: protezione perimetrale (intesa come isolamento degli asset), segretezza sulle tecnologie ed i meccanismi di sicurezza adottata (security by obscurity), integrità e tracciabilità delle operazioni svolte. Il modello proposto è dunque quello di una fortezza che garantisce sicurezza attraverso la costituzione di un perimetro fortificato intorno alle risorse da difendere, risorse che inoltre sono continuamente "osservate" da sentinelle sugli spalti.

Tale approccio continua anche nell'era di Internet con la proposta di un sistema di sicurezza organizzato dietro il firewall, un vero

e proprio ponte levatoio che dovrebbe far passare solo gli amici e respingere gli sconosciuti.

Internet ha prodotto però una discontinuità nel modello di sicurezza IT classico; già con l'avvento dei sistemi distribuiti e client-server i sistemi di sicurezza visti come "fortezza" sono ridimensionati, perché si indebolisce sensibilmente la componente di "isolamento". Il Web mette in discussione anche la separazione netta tra chi sta dentro e chi sta fuori la rete in quanto prevede la presenza di utenti "anonimi", cioè non precedentemente riconosciuti e registrati. Anche la barriera del knowledge (segretezza), intesa come conoscenza delle caratteristiche tecnologiche, viene a cadere in quanto i protocolli proprietari di telecomunicazioni (SNA, DECNet, eccetera) sono stati sostituiti da protocolli standard e "liberi" come il TCP/IP di cui sono note le caratteristiche (e le debolezze) mentre le architetture Web si basano su componenti software di largo consumo (i cosiddetti COTS, Commercial Off-The-Shelf), se non addirittura di pubblico dominio.

Oggi il modello di sicurezza IT proposto per le architetture Web più avanzate, come il B2B, si basa sull'utilizzo di reti private virtuali (VPN) che "disaccoppiano" la rete aziendale da quella pubblica utilizzando dei tunnel protetti e privati, isolati mediante strumenti crittografici; tali "percorsi sicuri" dovrebbero attraversare la prateria insicura di Internet, infestata da banditi e predoni, contando sull'inespugnabilità dei meccanismi di difesa (a volte anche attivi) posti intorno alla carovana in marcia.

Tale modello "rafforzato" soffre però di alcune debolezze:

1. presuppone che le minacce siano tutte fuori alla rete aziendale (cioè esterne al convoglio);
2. ritiene di poter identificare e tenere sotto controllo tutte le componenti della carovana, o detto in altri termini, ritiene che il co-

dice eseguito all'interno della rete aziendale sia sicuro e di origine "certificata".

Quest'ultimo punto, in particolare, è oggi una criticità reale a causa del cosiddetto "mobile code", cioè dei programmi che sono caricati dall'esterno e mandati in esecuzione in locale: esempi noti sono le applet, gli script, gli ActiveX, i plug-in, le macro nelle varie suite Office, eccetera.

Riguardo al mobile code, la crittografia, attraverso la firma digitale, può solo "certificare" l'origine del codice ma non l'effetto che produrrà; il rischio è di trovarsi di fronte a "Cavalli di Troia" che dietro una falsa apparenza celino forme di attacco insidiose agli asset informatici [9].

1.5.2 Ubiquitous security

Nel giugno 2000, la società Ovum [10] ha presentato un rapporto dal titolo "e-Business Security: New Directions and Successful Strategies", che invita le aziende a modificare radicalmente i loro modelli di sicurezza per adeguarli alle nuove opportunità offerte dall'e-Business.

La soluzione proposta è un nuovo approccio definito *ubiquitous security*, che prevede di applicare le misure di sicurezza in modo flessibile per ciascuna componente del modello di e-Business; tale flessibilità andrebbe collegata al livello di "trust" accordato all'utente ed al suo device di accesso, quale che esso sia.

1.6 I principali progetti di ubiquitous computing nel mondo

Sono numerosi i progetti, recentemente attivati da centri di ricerca ed industrie, che mirano alla realizzazione di ambienti di pervasive computing. L'obiettivo comune di questi progetti è il raggiungimento di un ambiente di calcolo 'invisibile', tuttavia ogni approccio si basa su scelte progettuali e tecnologie notevolmente diverse tra loro.

Di seguito vengono riportati alcuni dei principali progetti di ubiquitous computing in ambito internazionale, ognuno dei quali sviluppa ambienti che forniscono una tipologia di servizi differenti.

1.6.1 Portolano

E' un progetto attivo presso l'Università di Washington e riguarda lo sviluppo di ambienti nei quali l'utilizzo dei calcolatori diventa sempre più trasparente. Il progetto enfatizza l'invisibilità e basa la propria elaborazione sull'intenzione dell'utente, la quale è dedotta attraverso le sue azioni nell'ambiente e le sue interazioni con altri oggetti. I dispositivi impiegati nel progetto sono altamente ottimizzati per un particolare compito, così da inserirsi nell'ambiente e richiedere una minima conoscenza tecnica al dispositivo utente. In sintesi, Portolano [13] propone una infrastruttura basata sugli agenti mobili. Uno smistamento automatico dei dati li fa 'migrare' tra le varie applicazioni dell'ambiente assecondando le esigenze degli utenti.

1.6.2 Gaia

E′ un progetto attivo presso l′University of Illinois a Urbana-Champaign. L′obiettivo di Gaia [14] è quello di progettare ed implementare un sistema middleware che gestisca le risorse contenute in un Active Space. L′idea è quella di sviluppare un sistema in grado di localizzare i dispositivi all′interno di uno spazio fisico, rilevare quando i nuovi dispositivi entrano spontaneamente in esso, ed adattare il contenuto e il formato delle informazioni alle risorse del dispositivo di output. In definitiva, Gaia convoglia le funzionalità di un sistema classico in uno spazio fisico, nel quale oltre a supportare le funzionalità classiche, come eventi, segnali, file system, sicurezza, vengono aggiunti i requisiti impliciti di un ambiente intelligente. Attualmente è stato sviluppato un prototipo, costituito da una sala conferenza attrezzata con tecnologie che permettono la realizzazione, al suo interno, di un Active Space.

1.6.3 Palio

Palio (Personalized Access to Local Information and services for tourists) è un progetto europeo a cui hanno partecipato, tra gli altri, l′Università di Siena e l′Istituto di Fisica Applicata "Nello Carrara".

L′obiettivo del progetto è quello di fornire nuovi servizi turistici, direttamente disponibili a turisti e cittadini, sviluppando ed implementando sistemi con interfacce user-friendly e personalizzate. Il sistema Palio punta all′integrazione delle differenti tecnologie di comunicazione (wireless e wired), per offrire servizi sia attraverso terminali fissi che mobili, prevedendo una localizzazione dei dispositivi client per aggiornare dinamicamente le informazioni in base alla posizione dell′utente. Punta, inoltre, ad adattare il contenuto delle

informazione in base al dispositivo client e ad assicurare la scalabilità e l'interoperabilità dei differenti servizi erogati.

1.6.4 Wilma

Il progetto Wilma (Wireless Internet and Location Management Architecture) [15], sponsorizzato dalla Provincia Autonoma di Trento e alla cui realizzazione hanno partecipato il Centro per la Ricerca Scientifica e Tecnologica e l'Università di Trento, studia soluzioni per integrare diversi protocolli e mezzi di accesso senza fili per supportare i sistemi informatici dipendenti dal contesto (posizione, tempo, attività, storia precedente). Le attività di ricerca in corso mirano a ridurre la complessità sperimentata dall'utente nell'interagire con i vari sistemi. I problemi di ricerca comprendono la determinazione del mezzo ottimale di accesso alla rete senza fili per contesti specifici, la migrazione delle informazioni tra ambienti diversi, la determinazione e l'uso della posizione dell'utente, l'anticipazione delle richieste in base a quanto avvenuto in precedenza e l'adattamento alle necessità dell'utente senza il suo intervento diretto. Inoltre Wilma si propone di valutare i risultati in un "Laboratorio Aperto" composto da varie zone, coperte da un servizio di rete senza fili, collegate da un'infrastruttura fissa a larga banda.

Capitolo 2
Middleware

2.1 Introduzione

Il termine middleware apparve per la prima volta alla fine degli anni ′80 per descrivere la gestione software delle connessioni in rete, ma incominciò a diffondersi a metà degli anni ′90, di pari passo con l′evoluzione della tecnologia sulla rete. Da allora il middleware si è evoluto verso un insieme più ricco di paradigmi e servizi offerti per aiutare a costruire applicazioni distribuite in modo più semplice e più gestibile. Il termine è stato associato da molti professionisti principalmente con i database relazionali fino agli inizi degli anni ′90, ma già pochi anni dopo questa concezione fu persa [16,17].

Concetti simili al middleware odierno sono stati i sistemi operativi di rete, i sistemi operativi distribuiti e gli ambienti di calcolo distribuito [18].

Al giorno d′oggi il middleware è visto come una classe di tecnologie software designate per aiutare a gestire la complessità e l′eterogeneità inerenti i sistemi distribuiti. Esso è definito come uno strato software tra il sistema operativo ed i programmi applicativi che fornisce un′astrazione comune di programmazione attraverso i sistemi distribuiti.

Lo standard ISO OSI (International Standards Organization Open System Interconnection), nonostante non abbia trovato implementazioni pratiche di particolare successo, grazie alla separazione dei ruoli introdotta dalla sua architettura protocollare suddivisa a livelli

(Applicazione, Presentazione, Sessione, Trasporto, Rete, Data Link, Fisico), è pur sempre un valido modello che consente di comprendere meglio la collocazione dei vari elementi oggetto della discussione.

Nel caso in cui un middleware venga implementato come un protocollo (ovvero un insieme di regole e comandi che permettono a due entità remote di scambiarsi informazioni in maniera efficiente e trasparente rispetto alle proprie caratteristiche hardware e software), viene spontaneo identificare la posizione del middleware nello stack OSI.

Middleware come CORBA e DCOM [19] vengono necessariamente posizionati nel livello Applicazione, ovvero definiti come protocolli che supportano gli applicativi veri e propri, standardizzando il formato dei messaggi e comandi che questi sfruttano per comunicare da host a host. Alcuni servizi di rete standard utilizzano protocolli applicativi specifici come HTTP (HyperText Transfer Protocol), FTP (File Transfer Protocol) e SMTP (Simple Mail Transfer Protocol); ma per la creazione di applicazioni distribuite non standard o comunque più complesse sono impiegati protocolli di middleware come CORBA.

Indipendentemente dalla collocazione protocollare nello standard OSI, si può pensare ad un Middleware Layer come a tutto ciò che si interpone tra il sistema operativo (quindi software di basso livello che controlla direttamente l'hardware) e i software applicativi di alto livello; in questo modo si garantisce l'indipendenza tra questi due livelli.

2.2 Cos'è un middleware?

2.2.1 Specifiche e proprietà

Un protocollo di middleware fornisce un ben preciso modello di programmazione per la generazione di applicazioni distribuite, sollevando il programmatore dalla necessità di occuparsi di tutti i dettagli di basso livello della comunicazione e dello scambio di messaggi tra processi distribuiti, come è possibile osservare dalla Figura 2.1.

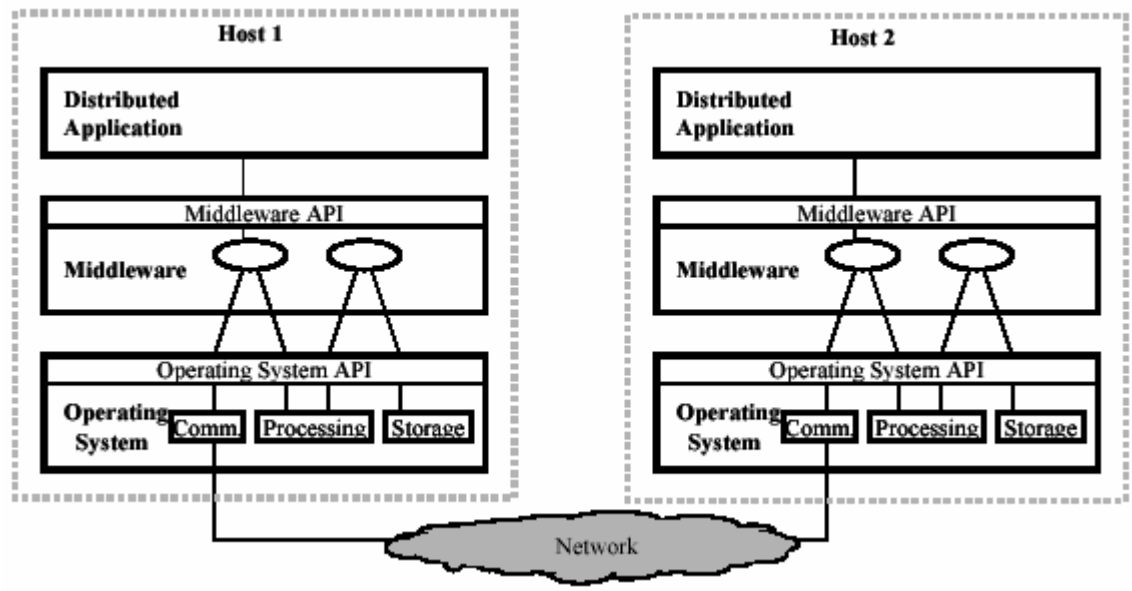

Figura 2.1 Collocazione del middleware all'interno dell'host

In questo modo, il middleware fornisce un blocco per costruire applicazioni di più alto livello sottoforma di API (Application Programming Interface), così come le socket sono messe a disposizione dal sistema operativo.

Le principali specifiche e proprietà che il middleware deve fornire sono:

- *marshalling* ed *unmarshalling*: i dati prodotti dalle applicazioni devono essere formattati per garantire l'indipendenza hardware e software dell'host di provenienza: il marshalling, effettuato in fase di trasmissione, prevede la suddivisione dei dati in più messaggi, la codifica in caratteri universali ASCII, la conversione little-endian in big-endian, ecc.; l'unmarshalling, effettuato in fase di ricezione, opera esattamente al contrario riportando i dati in uno stato comprensibile a quel particolare host;
- *data representation* e *codification*: per poter eseguire il marshalling/unmarshalling, il protocollo di middleware deve definire tutti i tipi di dati che le applicazioni possono utilizzare per comunicare (es. string, float, enumerated, ecc.); in tal modo il programmatore non dovrà conoscere i dettagli dei processi di comunicazione a patto di lavorare, in fase di programmazione, coi tipi messi a disposizione dal protocollo che sta impiegando;
- *invocazioni remote*: i moderni middleware, con la nascita della programmazione a oggetti distribuiti, devono permettere l'invocazione remota di processi e metodi per garantire un'interazione efficiente tra le applicazioni;
- garanzia di totale *trasparenza rispetto ai protocolli OSI di livello equivalente o inferiore* (TCP/UDP, IP, PPP/Ethernet, ecc.);
- garanzia della *location transparency*: gli applicativi che costituiscono il sistema distribuito devono poter comunicare senza preoccuparsi della loro posizione fisica né dei loro indirizzi logici; sarà infatti il Middleware Layer ad occuparsi dei dettagli di basso livello;
- totale *trasparenza di utilizzo per l'utente finale*: l'utente dell'applicazione programmata col supporto middleware non deve "accorgersi" della presenza dello stesso e quindi non

deve occuparsi di coordinarne il funzionamento, in quanto sarà del tutto integrato nel codice dell'applicazione.

Si osservi inoltre che in generale la visione logica del middleware che permette la comunicazione di un programma client con uno server è diversa da quella fisica, come si può osservare dalla Figura 2.2. Infatti è possibile immaginare il middleware come ad una piattaforma che è interposta tra la macchina client e quella server; in verità esso non è un blocco monolitico, ma è suddiviso in due parti, collocate sulle due macchine differenti. La comunicazione tra il client ed il server, dunque, avverrà grazie all'interazione dei due frammenti di middleware.

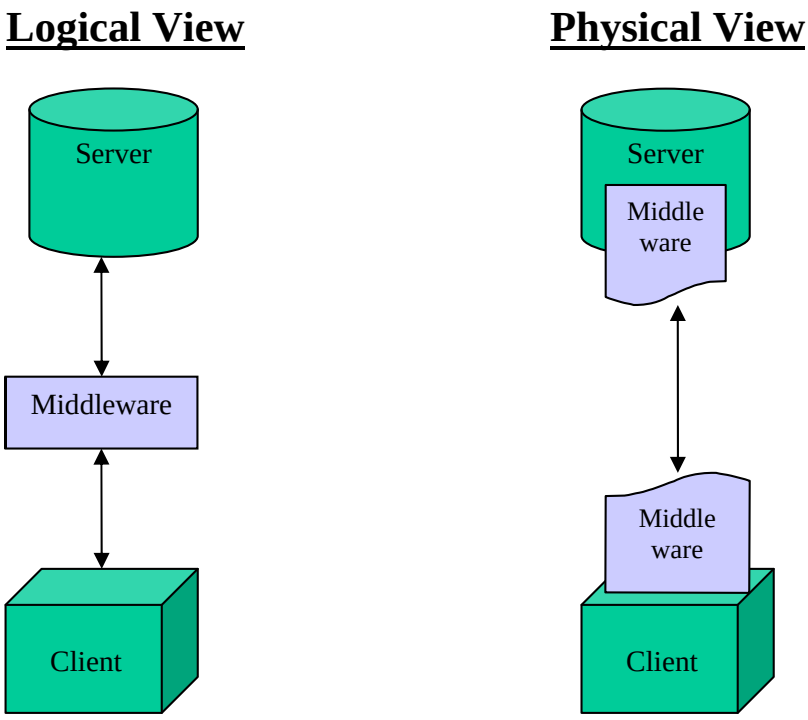

Figura 2.2 Visioni logica e fisica del middleware

2.2.2 Categorie di middleware

Esistono vari tipi di middleware che sono stati sviluppati. Questi variano tra di loro in base alle astrazioni di programmazione che forniscono ed ai tipi di eterogeneità che mettono a disposizione.

Tuple distribuite

Un database relazionale distribuito offre l'astrazione delle tuple distribuite; questo è il tipo di middleware più sviluppato al giorno d'oggi. Il suo linguaggio SQL (Structured Query Language) permette ai programmatori di manipolare insiemi di queste tuple (un database) in un linguaggio English-like con una semantica intuitiva e con rigorosi principi matematici basati sulla teoria degli insiemi e sul calcolo predicativo. I database relazionali distribuiti offrono anche l'astrazione di una transazione. Essi tipicamente offrono l'eterogeneità attraverso i linguaggi di programmazione, anche se limitata alle implementazioni del fornitore. I Transaction Processing Monitor (TPM) sono comunemente usati per la gestione delle risorse end-to-end delle query dei client, specialmente per la gestione dei processi lato server e la gestione di transazioni multi-database.

RPC e RMI

I concetti di RPC e RMI sono nati e si sono evoluti per permettere, in generale, la comunicazione tra applicazioni remote attraverso paradigmi ben precisi; non sono di per sé protocolli o software, ma semplici specifiche o paradigmi che vengono poi impiegati all'interno dei middleware e protocolli veri e propri come ad esempio CORBA.

La Remote Procedure Call (RPC), ideata verso la metà degli anni '80, nasce in un ambiente informatico in cui la programmazione a oggetti non esiste ancora: quando si parla di comunicazione remota si intende sempre la comunicazione tra processi, ovvero i singoli

componenti di un software più grande che vengono gestiti singolarmente dal sistema operativo. Le RPC non fanno altro che permettere l'invocazione e l'attivazione di un processo su un host remoto.

Applicato in un rapporto client-server, il processo client richiede un servizio risiedente sul processo server inviando la richiesta *Request()* tramite il Communication Module integrato nel processo, mentre il server risponde con una *Reply()* tramite il suo Communication Module. Il meccanismo estremamente semplice ha, però, possibilità limitate nello scambio effettivo di dati e nell'interazione tra le applicazioni remote.

Le Request Method Invocation (RMI) nascono come evoluzione delle RPC in concomitanza con l'espansione e la sempre maggiore diffusione del modello di programmazione ad oggetti: si sente la necessità di stabilire una comunicazione tra host remoti di più alto livello, non più basata sui singoli processi bensì sui metodi di un oggetto.

L'idea principale è molto semplice: se una singola applicazione funziona mediante l'invocazione locale dei propri metodi interni (local method invocation), sarà possibile controllare tale applicazione facendo sì che essa esponga alcuni dei suoi metodi verso l'esterno, tramite un'interfaccia, in modo tale che altre applicazioni remote possano invocarli e ottenere dei parametri di ritorno. L'interazione tra applicazioni diventa in questo modo estremamente più forte e omogenea, aprendo le porte alla creazione di interi sistemi distribuiti.

In aggiunta, i vari protocolli/software che impiegano RMI includono generalmente dei sistemi di fault tolerance, per garantire nel miglior modo possibile l'efficacia e l'efficienza dell'invocazione remota.

Middleware orientato ai messaggi

Il middleware orientato ai messaggi (MOM, Message-Oriented Middleware) fornisce l'astrazione di una coda di messaggi a cui possono accedere processi attraverso la rete. Esso è la generalizzazione di un affermato costrutto del sistema operativo: la mailbox. Questo middleware è molto flessibile, in quanto può essere configurato con la topologia dei programmi che depositano e prelevano messaggi da una data coda. Molti prodotti MOM inoltre offrono code con persistenza, replicazione o prestazioni real-time.

2.2.3 Il modello ad oggetti distribuiti

Unendo i concetti del modello della programmazione object-oriented e di RMI, è stato possibile stabilire un'importante innovazione: la gestione attiva e l'interazione tra oggetti dislocati nello spazio su differenti elaboratori.

Tutto questo avviene attraverso l'astrazione completa di tutti i dettagli di comunicazione di più basso livello (indirizzamento porti, pacchetti, ecc.) grazie all'utilizzo e all'integrazione, in fase di programmazione, di specifici middleware. I vantaggi in termini di produttività, velocità e semplicità di sviluppo del software sono notevoli.

Le specifiche primarie che un sistema a oggetti distribuiti deve rispettare sono:

- la comunicazione deve avvenire attraverso protocolli di livello applicativo dedicati (ad esempio CORBA e DCOM) che implementino a loro volta il paradigma RPC o RMI;

- la possibilità di istanziare oggetti in locazioni diverse: un'applicazione deve poter creare, secondo necessità, oggetti remoti;
- l'opportunità di interagire con gli oggetti remoti, sfruttandone metodi e attributi come se fossero locali (soddisfacendo la specifica di location transparency del middleware);
- referenziare ogni oggetto, a livello applicativo, attraverso una stringa alfanumerica univoca per tutto il sistema distribuito, chiamata *OBJID* (Object Identificator); essa è generalmente lunga fino a 1024 caratteri e contiene un gran numero di informazioni relative all'oggetto. In questo modo la struttura interna del middleware ha le opportune conoscenze per operare il marshalling/unmarshalling, le invocazioni remote, ecc. La stringa contiene in particolare dati riguardanti il sistema operativo utilizzato, le caratteristiche hardware, l'indirizzo IP dell'host, i porti attivi ed altro. Ad ogni oggetto istanziato su un host viene quindi attribuito il proprio identificativo, una sorta di targhetta con cui verrà referenziato all'interno della rete;
- l'eliminazione e/o la migrazione degli oggetti insieme al loro codice (Object Migration), che si rivela particolarmente utile per la creazione di sistemi temporanei; ad esempio può essere necessario che un server "installi" del software sui client che si connettono.

2.2.4 Interfaccia

Le colonne portanti di ogni sistema a oggetti distribuito sono le interfacce, su cui si basano a loro volta le RMI.

In fase di programmazione la creazione dell'interfaccia rappresenta un aspetto estremamente delicato da cui dipende il corretto funzionamento del sistema.

Coerentemente al modello di programmazione ad oggetti, l'interfaccia non è altro che la componente di un'applicazione che espone i metodi dell'applicazione stessa che possono essere chiamati dall'esterno come da specifiche RMI.

Il nome dei metodi, il tipo, l'ordine ed il numero dei singoli parametri di ingresso/uscita devono essere espressi pubblicamente, dato che l'interfaccia mostra soltanto ciò che l'applicazione concede all'esterno, mascherando l'implementazione ed eventuali attributi o metodi privati.

Inoltre, mentre prima le interfacce venivano definite nelle singole applicazioni per la comunicazione tra più applicazioni in locale, con lo sviluppo della programmazione ad oggetti distribuiti queste si sono evolute anch'esse, costituendo in tal modo le cosiddette *Remote Interface*.

Le Remote Interface sono in realtà parte integrante dei protocolli di middleware, e devono essere definite in fase di programmazione: ciascun oggetto distribuito è referenziato attraverso la propria OBJID e possiede una Remote Interface con la descrizione dei metodi che intende esporre. L'implementazione effettiva dei metodi si trova nella classe da cui deriva l'oggetto.

Il linguaggio di programmazione dell'interfaccia deve essere indipendente dagli ambienti di sviluppo, quali C++ o Java, e legato esclusivamente al middleware impiegato. Quest'ultimo è incluso nel progetto come una normale libreria e le varie interfacce sono descritte in file separati.

Alcuni esempi pratici di linguaggi per l'implementazione delle interfacce sono:

- Sun XDR, vecchio standard utilizzato per le RPC, ormai obsoleto;

- DCOM IDL, basato su DCE (Distributed Computer Environment) ed utilizzato per le RPC, impiegato nelle tecnologie Microsoft COM e OLE;
- CORBA IDL, utilizzato per le RMI, uno dei più moderni e versatili;
- Java RMI, utilizzato per le RMI, con funzionalità analoghe a quelle di CORBA IDL.

2.3 Uno standard per i middleware: CORBA

CORBA (Common Object Request Broker Architecture) [20] non è un software, ma un insieme di specifiche proprietarie della OMG [21], nate nel 1991 con lo scopo di definire un nuovo protocollo di comunicazione per applicazioni molto versatile e multipiattaforma.

L'OMG (Object Management Group) è un'associazione indipendente costituitasi nel 1989 dalla fusione di 11 compagnie con obiettivi no-profit; il suo impegno principale è stato quello di produrre e fornire specifiche di ogni tipo per lo sviluppo del software, portabili e del tutto indipendenti dalle software house proprietarie.

CORBA, nelle sue varie implementazioni, è uno dei migliori esempi di middleware che si possano studiare: sostanzialmente, si tratta di un insieme di specifiche aperte per un'architettura e un'infrastruttura utile a far comunicare gli elaboratori connessi ad una rete. Utilizzando un protocollo standardizzato dalla stessa OMG, IIOP (Internet Inter-ORB Protocol), un programma basato su CORBA prodotto da qualunque software house e che gira su una qualunque macchina, sistema operativo e rete, CORBA può interoperare con qualunque altra macchina, programma e rete che sfrutta il medesimo standard. Scalabilità e portabilità sono quindi due delle caratteristiche vincenti di questo middleware.

CORBA si rivela inoltre utile in molte situazioni; grazie alla sua proprietà di essere facilmente integrabile su qualsiasi macchina, viene ogni anno scelto da un gran numero di ditte.

Impiegato su server, CORBA supporta una grande quantità di client, garantendo sempre un'ottima efficienza e buone prestazioni grazie ai suoi avanzati sistemi di fault tolerance e recupero degli errori; attualmente è presente sulle macchine di molti famosi siti Web di livello mondiale.

2.3.1 L'ORB

Il componente più importante dell'intera architettura CORBA è l'ORB (Object Request Broker). Esso è il nucleo operativo, il vero e proprio cuore delle specifiche CORBA, e rappresenta lo strato comune a tutti i componenti CORBA del sistema distribuito, attraverso il quale essi comunicano e si scambiano informazioni; è cioè l'unico responsabile della comunicazione remota tra gli elaboratori, del marshalling e dell'unmarshalling dei dati e della location transparency.

Si tratta in ogni caso di un oggetto molto complesso, tanto che gran parte delle specifiche di CORBA si concentrano proprio in questo componente: possiamo dire che quando una software house implementa le specifiche, in effetti produce un nuovo ORB.

Proveremo a delineare qui le caratteristiche principali del componente, considerando il classico paradigma di un'architettura client-server, dove per client intendiamo un componente qualsiasi dell'applicazione che intenda far riferimento ai metodi di un oggetto distribuito, e server il componente che espone i metodi attraverso l'interfaccia.

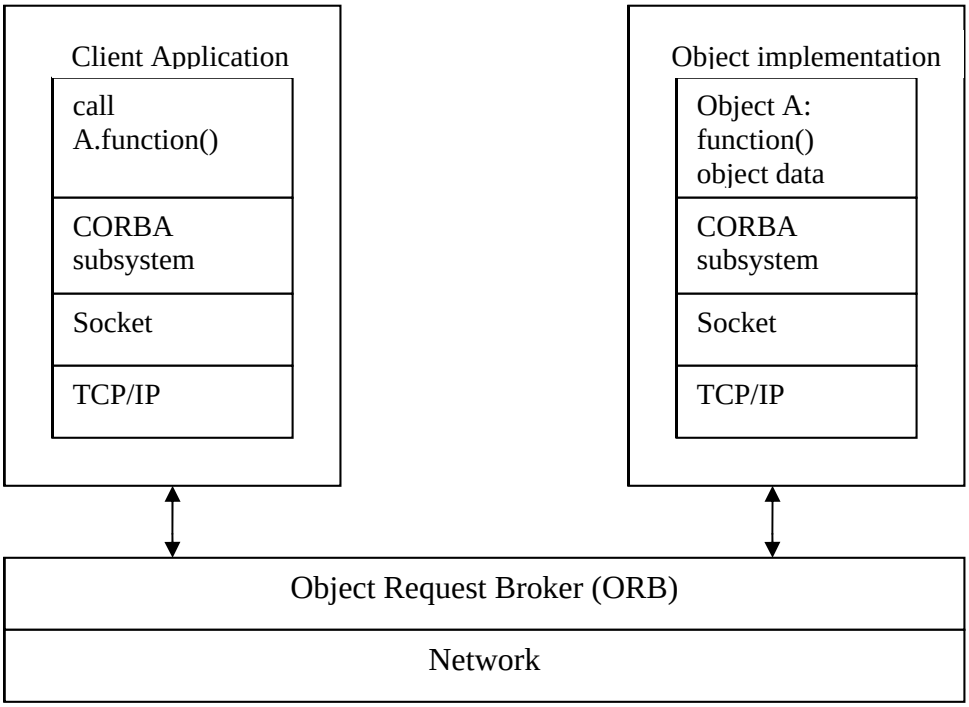

Figura 2.3 Object Request Broker

Come si vede dalla Figura 2.3, in CORBA, l'Object Request Broker si occupa di tutti i dettagli di basso livello che riguardano il routing delle richieste da un client all'oggetto, e del routing della risposta alla giusta destinazione (mantenendo la location transparency di cui abbiamo parlato precedentemente).

L'ORB, inoltre, è responsabile della gestione della Interface Repository (IR), un database distribuito standardizzato dalla stessa OMG contenente tutte le interfacce definite per quel determinato sistema, al quale tutti gli oggetti possono accedere per ottenere un riferimento ad una particolare interfaccia.

Lato client, l'ORB fornisce numerosi servizi: si occupa di fornire, come detto, le interfacce dall'IR, gestisce inoltre le invocazioni dei metodi e opera la conversione degli Object Reference (OBJID) da Session Format a Stringify Format.

Questa conversione è estremamente importante nel processo di comunicazione: un Object Reference in Session Format è una sorta

di "gettone elettronico" che viene associato ad ogni invocazione di un oggetto, permettendo all'ORB l'invocazione della giusta istanza.

In questa forma non c'è però alcuna informazione utile per il client che sta effettivamente invocando un metodo appartenente all'oggetto remoto. Quindi, quando è necessario informare un client dell'entità di un certo oggetto nel sistema distribuito, gli viene passata un'altra versione dell'OBJID, la versione Stringify Format, chiamata in genere IOR (Interoperable Object Reference), che, oltre a contenere informazioni utili per il client, può essere memorizzata, stampata, visualizzata, eccetera.

In pratica si tratta di una stringa di caratteri esadecimali prodotti dall'ORB che tutti i client connessi all'ORB stesso devono possedere.

Lato server, invece, l'ORB offre alcune funzionalità supplementari. Ad esempio, si occupa della gestione degli oggetti; infatti, come da specifiche CORBA, è necessario che ad ogni OBJID corrisponda un oggetto valido e attivo; in realtà è possibile che il codice dell'oggetto sia effettivamente inattivo. Questo perché, nel tentativo di sprecare meno risorse possibili, l'ORB lato server potrebbe aver disattivato tutti gli oggetti che in quell'istante non sono utilizzati.

2.3.2 Stub e skeleton

In Figura 2.4 è rappresentato in maniera estremamente semplificato il processo di invocazione remota peer-to-peer. Come si può osservare, esistono un oggetto *Object* che contiene le implementazioni (nascoste) dei propri metodi e referenziato con la propria OBJID, ed un client che invoca metodi remoti.

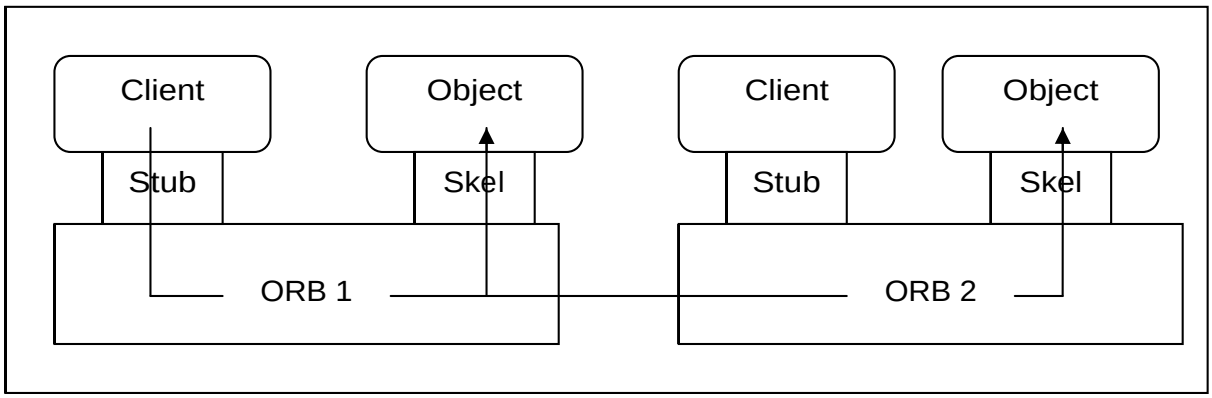

Figura 2.4 Interoperabilità tra gli ORB

L'oggetto possiede un'interfaccia, scritta secondo il linguaggio IDL (Interface Definition Language), la cui compilazione produce due moduli chiamati *stub* e *skeleton*; mentre il primo è strutturato per il client, il secondo lo è per l'oggetto. Questi due moduli funzionano in maniera molto simile ai proxy tra client e server; in particolare, il client e l'oggetto fanno riferimento a questi ogni volta che viene invocato un metodo dell'interfaccia.

L'idea principale di CORBA è proprio questa: ogni volta che il client intende utilizzare l'oggetto interagirà con lo stub, che a sua volta invocherà lo skeleton, che fungerà infine da chiamante per il metodo dell'oggetto. Il client non sa dunque dove l'oggetto sia istanziato: esso potrà trovarsi in locale sulla medesima macchina (nell'esempio: il client richiede un metodo dell'oggetto residente su ORB 1) oppure potrà risiedere su un altro host (ORB 2), nel qual caso la comunicazione tra gli ORB viene gestita dal middleware attraverso il protocollo IIOP in maniera del tutto trasparente al client, garantendo così la location transparency del sistema.

Ciascun client possiede dunque tanti stub quanti sono gli oggetti con cui intende interagire.

Ovviamente, per poter invocare un metodo dell'oggetto il client deve conoscere il relativo OBJID, che costituisce dunque l'URI (Universal Resource Identifier) dell'oggetto.

In verità gli stub sono moduli che permettono ad una applicazione client di invocare metodi remoti solo in maniera statica. Non è dunque possibile, attraverso il meccanismo degli stub, invocare metodi il cui nome è noto solo a run-time. Questo problema è risolto con l'utilizzo delle DII (Dynamic Invocation Interface), impiegate per le invocazioni di metodi in maniera dinamica, ovvero per le invocazioni relative a oggetti di cui non si conosce l'esistenza in fase di programmazione. In effetti, deve comunque esser stato previsto un sistema per ottenere l'interfaccia dell'oggetto, dopodiché, tramite le DII, viene creato un particolare oggetto CORBA che si occupa esclusivamente delle invocazioni dei metodi remoti. Il mashalling/unmarshalling dei dati è però stavolta affidato all'ORB e chiaramente non allo stub. Analogamente, per la pubblicazione di servizi dinamici lato server esiste il Dynamic Skeleton Interface.

Infine, attraverso il componente ORB Interface, è possibile, sia per il client che per l'oggetto, accedere all'ORB quando ciò non avviene per richieste di invocazione (stub e skeleton) e di attivazione dei metodi.

2.3.3 OA e POA

L'Object Adapter (OA), ovvero l'elemento più importante della struttura lato server dell'architettura CORBA, possiede un ruolo intermedio nel percorso di invocazione del metodo: quando infatti il client inoltra un'invocazione ad un metodo remoto, questa, prima di essere passata allo skeleton, viene elaborata dall'OA; dopodiché quest'ultimo invocherà lo skeleton, che spacchetterà i parametri e li fornirà all'implementazione stessa. A questo punto l'implementazione esegue il metodo e ritorna dei parametri al client.

Il ritorno dei parametri passa sempre dallo skeleton che gestisce anche le eventuali condizioni di eccezione.

In pratica, il compito principale dell'Object Adapter riguarda la salvaguardia delle risorse lato server e dunque la scalabilità del sistema; è compito dell'OA, infatti, attivare o disattivare gli oggetti e i loro relativi metodi sull'ORB, a seconda dello stato di carico del sistema stesso e delle disponibilità hardware e di rete. Anche se a livello di architettura l'Object Adapter è un componente separato dall'ORB, e l'ORB può teoricamente implementare più tipi di OA, questo non è un modulo effettivamente separabile. Questo perché le interfacce che connettono OA e ORB sono proprietarie; non c'è quindi motivo logico per cui un produttore che progetta un nuovo ORB debba poi vendere separatamente l'OA, che non potrebbe funzionare con nessun altro ORB sul mercato.

Molto spesso l'OA è però sostituito dal POA (Portable Object Adapter) di più nuova concezione, più essenziale e molto più leggero, studiato per funzionare su sistemi limitati come sistemi embedded, palmari, PDA, ecc. Questo è essenziale per questi dispositivi, infatti l'Object Adapter rappresenta l'elemento più pesante ed esigente in termini di risorse dell'intera architettura CORBA.

Il POA mappa gli oggetti astratti CORBA sulla loro effettiva implementazione, detta *servant*, definendo le politiche di gestione degli oggetti appartenenti al suo ambito (come ad esempio la durata della vita di un oggetto oppure le sue modalità di attivazione).

Svincolando l'oggetto CORBA dalla sua implementazione, viene assicurata, in modo del tutto trasparente al resto dell'applicazione distribuita, la sua portabilità (potendo realizzare più implementazioni dell'oggetto).

Osserviamo inoltre che l'Object Reference contiene informazioni sul nome del POA che gestisce il CORBA Object, il nome dell'ORB che riceve le invocazioni al CORBA Object e l'ID del CORBA Object.

Quest'ultimo è l'identificativo del CORBA Object che è unico rispetto al POA che lo gestisce. Inoltre l'oggetto si dice attivo se il POA ha effettuato il mapping con il rispettivo servant, come mostrato in Figura 2.5.

Le varie classi ed i metodi che costituiscono l'oggetto hanno degli ID contenuti in tabelle appartenenti ad un database gestito dal POA, chiamato Implementation Repository.

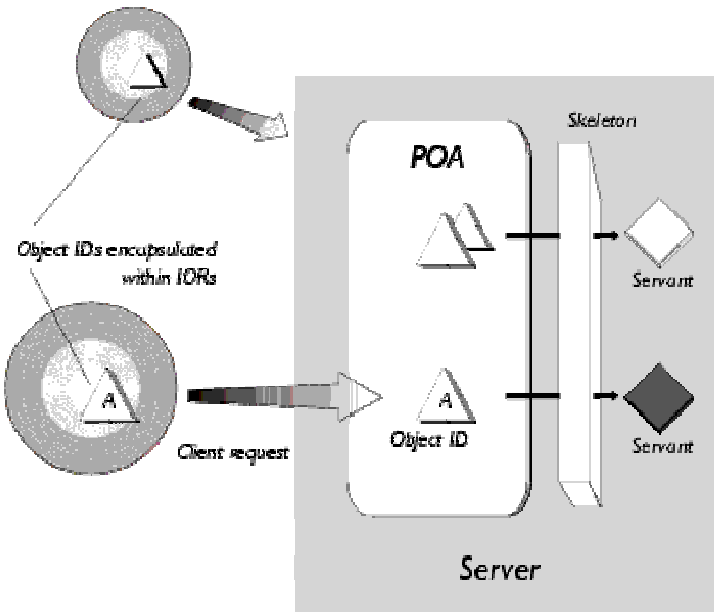

Figura 2.5 Il POA mappa i CORBA Object sulle loro implementazioni

2.3.4 GIOP

La soluzione proposta dall'OMG per soddisfare il requisito di interoperabilità tra oggetti residenti su ORB distinti si articola sulla specifica di due protocolli, uno generico e l'altro dipendente dal sistema di trasporto utilizzato, che risiedono al di sotto dell'ORB: GIOP (General Inter-ORB Protocol) e IIOP (Internet Inter-ORB Protocol) .
GIOP è un protocollo astratto che specifica un insieme di tipi di messaggi utilizzabili tra un client ed un server, una sintassi standard

nel trasferimento dati delle interfacce IDL ed un formato standard per ogni messaggio utilizzabile. GIOP fornisce quindi la sintassi per la comunicazione inter-ORB, ma non la semantica.

IIOP invece aggiunge le informazioni di cui ha bisogno un ORB, ovvero la semantica, per comunicare attraverso un meccanismo di trasporto con connessione affidabile come TCP. Da questa prima descrizione ne deriva che per ogni meccanismo di trasporto utilizzato si deve realizzare un particolare protocollo che connetta il livello di trasporto al protocollo astratto GIOP.

Il livello di trasporto su cui GIOP si basa per trasferire i messaggi deve garantire le seguenti proprietà:

- deve essere orientato alla connessione;
- deve essere affidabile;
- deve fornire l'astrazione del flusso di byte;
- deve notificare le perdite di connessione.

Si può notare dalla Figura 2.6 che queste assunzioni si adattano alle caratteristiche del protocollo TCP/IP.

GIOP utilizza un formato binario standard per la trasmissione dei tipi IDL chiamato *Common Data Representation* (CDR), che definisce l'ordinamento e l'allineamento dei byte per ogni tipo IDL. Le caratteristiche della codifica CDR permettono la massima efficienza nel marshalling e nell'unmarshalling, a prezzo di una dimensione dei messaggi non ottimale. Per compensare questo, la codifica non contiene l'identificazione dei tipi, ossia il ricevente vede il messaggio come un flusso di byte non formattato. Quindi la decodifica del messaggio richiede un accordo preventivo tra mittente e ricevente, che avviene attraverso gli stub e skeleton ottenuti dalla compilazione dell'interfaccia IDL.

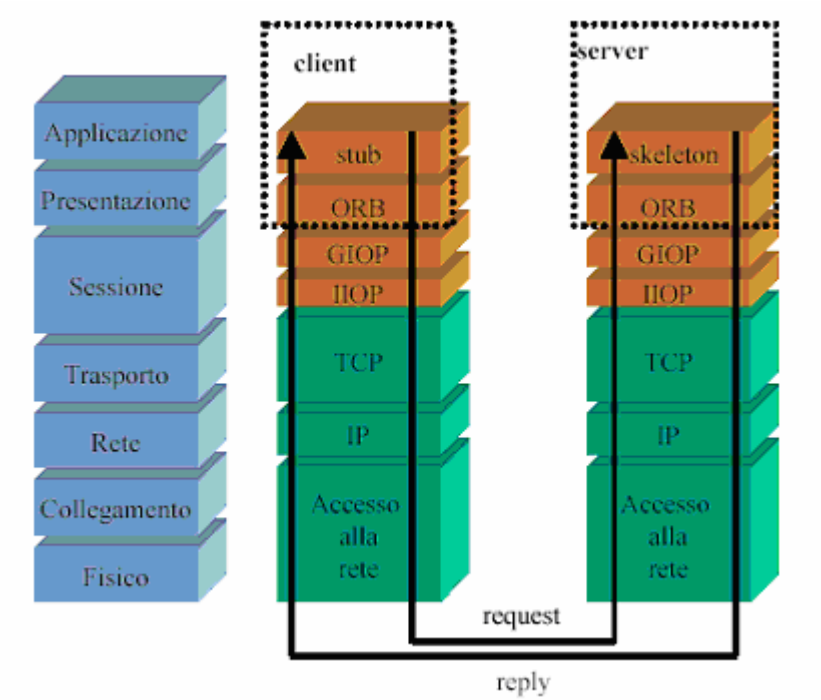

Figura 2.6 Confronto tra lo standard ISO/OSI e l'architettura CORBA

In GIOP sono definiti otto tipi di messaggi, come è possibile osservare dalla Figura 2.7.

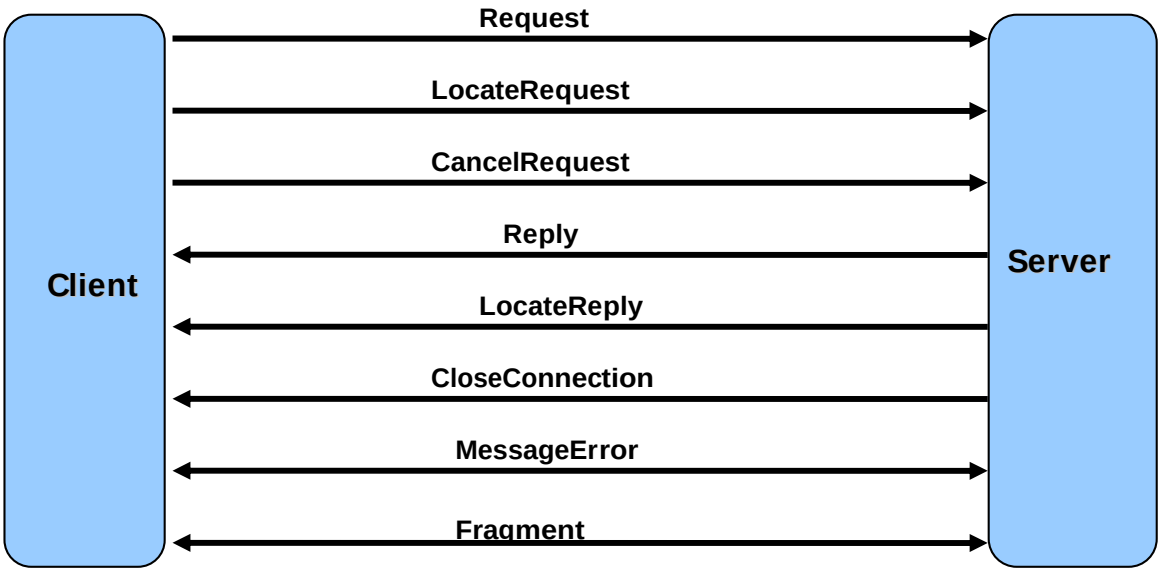

Figura 2.7 Messaggi GIOP

Come si vede dalla Figura 2.6, i messaggi più importanti per la comunicazione client/server, sono:

- *Request*, che è inviato sempre dal client al server ed è utilizzato per invocare un'operazione o accedere ad un attributo. I messaggi Request trasportano tutti i parametri necessari per l'invocazione;
- *Reply*, che è inviato sempre dal server al client e solo in riposta ad una precedente richiesta. Esso contiene i risultati dell'invocazione, cioè il valore di ritorno, i parametri di uscita ed eventuali eccezioni. Ogni messaggio GIOP è composto da un header (comune a tutti i tipi di messaggio) e un body (dipendente dal tipo di messaggio), come si può osservare dalla Figura 2.8.

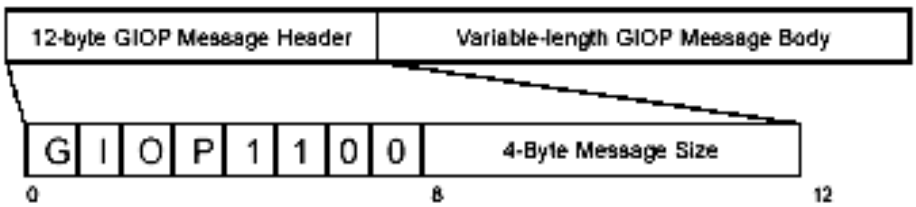

Figura 2.8 Header di un messaggio GIOP

In sostanza, l'header di un messaggio GIOP contiene l'indicazione del tipo del messaggio e la sua dimensione.

2.3.5 IIOP

Come visto, GIOP assume che sia stata stabilita una connessione tra client e server e si limita a stabilire il formato dei messaggi scambiati.

IIOP invece consiste nel mapping specifico di GIOP sul protocollo TCP/IP e, per quanto detto, consiste nella specifica delle informazioni di indirizzamento degli oggetti. Ma queste, come visto in pre-

cedenza, sono contenute all'interno dell'IOR dell'oggetto, quindi, in sostanza, IIOP deve semplicemente specificare come un IOR codifichi al suo interno le informazioni di indirizzamento TCP/IP.

Il significato dei campi dovrebbe risultare evidente: l'IOR contiene l'indirizzo IP e il numero del porto su cui è in ascolto il server in cui è registrato l'oggetto. Al momento dell'invocazione della richiesta, l'ORB del client stabilisce una connessione socket con il server attraverso questi dati e invia il messaggio GIOP *Request* contenente i dati della richiesta e l'object key dell'oggetto. Ricevuta la richiesta, il server decodifica l'object key determinando così l'oggetto che dovrà ricevere la richiesta.

A partire da IIOP 1.1, un IOR può contenere una serie di *tagged components*, che possono contenere informazioni di servizio. Ad esempio, gli ORB che supportano il Security Service OMG aggiungono a ogni IOR un componente che descrive la codifica utilizzata. Ogni tagged component è una struttura contenente due campi: il primo (tag) identifica il tipo del componente, mentre il secondo contiene i dati in forma di sequenza di byte.

La struttura dettagliata di un IOR è illustrata in Figura 2.9.

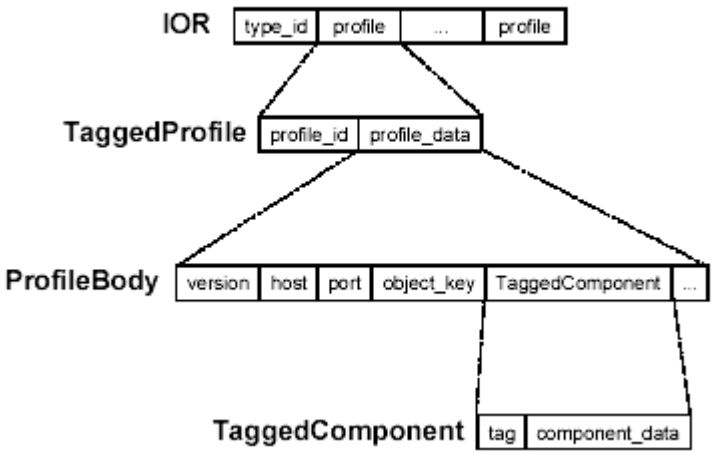

Figura 2.9 Alcuni dettagli di un IOR

Come si può osservare, un IOR può contenere più profili. Ogni profilo corrisponde in generale a un diverso protocollo. Questo fa sì che l'ORB possa scegliere dinamicamente il trasporto da utilizzare

per una richiesta con lo stesso IOR, rendendo così la comunicazione tra client e server effettivamente indipendente dal livello di trasporto.

In particolare, oltre ad IIOP, lo standard CORBA specifica attualmente il supporto ad un altro protocollo, il DCE Common Inter-ORB Protocol, che utilizza DCE-RPC come livello di trasporto sottostante; questo permette l'utilizzo di CORBA su protocolli proprietari alternativi al TCP/IP.

2.3.6 Il linguaggio IDL

Il linguaggio IDL (Interface Definition Language), ideato dall'OMG, è usato per descrivere le interfacce che gli oggetti client invocano e che le implementazioni degli oggetti mettono a disposizione. Una definizione di un'interfaccia IDL specifica completamente ogni parametro di un'operazione e fornisce l'informazione necessaria a sviluppare applicazioni client che usano le operazioni dell'interfaccia.

Le applicazioni client sono scritte in linguaggi per i quali sono stati definiti i mapping dalle dichiarazioni IDL. Come una dichiarazione IDL sia mappata in un costrutto del linguaggio client dipende dal particolare linguaggio. Ad esempio, un'eccezione IDL potrebbe essere mappata con una struttura in un linguaggio che è privo di eccezioni o in un'eccezione in un linguaggio che ne ha.
Le dichiarazioni IDL seguono le stesse regole lessicali delle istruzioni C++, sebbene siano state introdotte nuove parole chiave per il supporto dei concetti sulla distribuzione. Le dichiarazioni IDL forniscono anche un completo supporto per le caratteristiche standard di preprocessazione per C++.

La grammatica IDL è un sottoinsieme dell'ANSI C++, con costrutti addizionali per supportare il meccanismo dell'operazione di

invocazione. L'IDL è un linguaggio dichiarativo; inoltre, supporta la sintassi C++ per costanti, tipi ed operazioni di dichiarazione.

In particolare, i tipi di dato del linguaggio IDL si dividono in quelli base ed in quelli complessi e sono riportati in Tabella 2.1.

Tipi di dato base	Tipi di dato complessi
short	Object
long	struct
long long	union
unsigned short	enum
unsigned long	string
unsigned long long	wstring
float	sequence
double	array
long double	
char	
boolean	
octet	
wchar	
any	
fixed	

Tabella 2.1 Tipi di dato del linguaggio IDL

Si riporta infine la sintassi IDL con cui è possibile definire un'interfaccia:

```
module <identifier> {    /* define a naming context */
    <type declarations>;
    <constant declarations>;
    <exception declarations>;
    /* class definition */
    interface <identifier> [:<inheritance>]{
    <type declarations>;
    <constant declarations>;
    <attribute declarations>;
    <exception declarations>;
```

```
    /*method declaration */
    [<op_type>]<identifier>(<parameters>)
        [raises <exception>] [<context>];
    ...
    /* method declaration */
    [<op_type>]<identifier>(<parameters>)
        [raises <exception>] [<context>];
    }

    /* more interfaces here */

}
```

2.3.7 Classificazione degli oggetti

Come sappiamo, le applicazioni che impiegano CORBA comunicano attraverso l'ORB con oggetti distribuiti appartenenti ad altre applicazioni, svolgendo i loro normali compiti. Alcuni oggetti necessitano di essere istanziati, di ricevere notifiche, di distribuire le proprie Object Reference e di poter essere distrutti.

Naturalmente, tutte queste operazioni devono essere affidabili e sicure.

Per far ciò, l'OMG ha raccolto una serie di funzionalità usate dalla maggior parte delle applicazioni CORBA e le ha incluse in una serie di oggetti CORBA standard che le mettono a disposizione attraverso interfacce IDL standard. L'OMG fornisce le interfacce e rilascia le specifiche sul loro funzionamento: le ditte che producono le applicazioni le implementano direttamente nel loro codice.

Le specifiche per questa "raccolta" di oggetti standard sono denominate OMA (Object Management Architecture) e costituiscono una sezione a parte delle specifiche CORBA.

L'Object Management Architecture della OMG classifica gli oggetti in quattro principali categorie:

- *Application Objects*: sono gli oggetti "custom", ovvero quelli sviluppati dai singoli produttori per le loro applicazioni. Proprio perché non necessitano di alcuna standardizzazione, OMG non rilascia alcuna specifica riguardante questa tipologia di oggetti.
- *CORBA Services*: gli oggetti di questo tipo forniscono servizi di vitale importanza alle applicazioni CORBA, partendo da funzionalità base di basso livello per il controllo del sistema distribuito. Comprendono anche alcuni servizi di più alto livello come supporto per transazioni economiche e sicurezza in generale, ma la loro implementazione resta comunque molto vicina alla struttura dell'ORB.
- *Horizontal CORBA Facilities*: vengono classificate a metà tra le CORBA Services e gli Application Objects, descritti sopra, e forniscono fondamentalmente servizi non vitali alle applicazioni CORBA. A differenza delle Domain CORBA Facilities, descritte in seguito, questi oggetti sono particolarmente utili nell'ambiente business. Esistono quattro principali Horizontal CORBA Facilities: Printing Facility, Secure Time Facility, Internationalization Facility, Mobile Agent Facility. Questa è l'unica categoria di oggetti standardizzati a cui non è dedicata un intero gruppo di lavoro nella gerarchia OMG.
- *Domain CORBA Facilities*: costituiscono una delle categorie che godono di maggior attenzione da parte della stessa OMG; si tratta di interfacce IDL di pubblico dominio, migliorate continuamente dalla comunità con lo scopo di creare nuove interfacce e oggetti standard, sempre di pubblico dominio, più raffinati ed efficienti. Per coordinare il lavoro, OMG ha fondato nel 1996 la Domain Technology Committee.

Negli ultimi anni le implementazioni di CORBA sono state numerose in tutti i campi dell'informatica.

Dal momento in cui CORBA è un insieme di specifiche, sono state prodotte varie implementazioni del middleware più o meno complesse e pesanti, a seconda degli applicativi e dei software ospitanti.

CORBA viene normalmente impiegato su sistemi mainframe, oppure su semplici PC desktop, in piccole reti LAN (Local Area Network) o anche in grandi reti WAN (Wide Area Network).

Con l'evolversi della tecnologia disponibile e l'aumentare delle necessità delle prestazioni dei sistemi distribuiti su rete, si sono sviluppati alcuni sotto-rami ad hoc delle specifiche CORBA.

2.4 Un'altra piattaforma middleware: Java RMI

Java è un linguaggio completamente specificato [22]: nessuna parte è indefinita o dipendente dall'implementazione. L'obiettivo è chiaramente quello di avere un linguaggio realmente indipendente dalla piattaforma. Per ottenere un tale livello di portabilità, è necessario che ogni piattaforma possegga un interprete Java, meglio noto come *Java Virtual Machine* (JVM). Altri strumenti necessari ad una programmazione ad oggetti distribuiti sono l'API RMI [23] discussa in seguito ed i successivi elementi:

- il compilatore per applicazioni RMI: il tool *rmic* è un compilatore dedicato che, partendo direttamente dal bytecode ottenuto tramite la compilazione standard, permette di ottenere le due classi *stub* e *skeleton* proprie dell'architettura RMI;
- il gestore di oggetti RMI: *rmiregistry* è un programma che permette la gestione di oggetti remoti nell'ambito del meccanismo RMI. In pratica esso è un daemon, il cui compito è quello di

mettersi in attesa, su un porto di sistema, delle richieste di utilizzo degli oggetti da parte dei client.

2.4.1 L'ambiente Java

Java è un linguaggio di programmazione orientato agli oggetti di tipo interpretato o compilato "just in time" e nasce come risultato di un'opera di revisione e semplificazione del C++.

Essendo un linguaggio interpretato, un programma Java viene compilato in un codice intermedio, detto *bytecode*, uguale per tutte le piattaforme, ed eseguito da un qualunque sistema capace di interpretarlo. Un linguaggio di questo tipo, però, può penalizzare in maniera eccessiva le prestazioni. Questo problema può essere aggirato tramite il componente chiamato *Just In Time (JIT) Compiler*, che compila il codice intermedio in codice macchina eseguibile immediatamente prima della sua elaborazione. In pratica ciò avviene quando un metodo è invocato per la prima volta ed il codice nativo viene memorizzato in un opportuna Class Area; le successive chiamate a tale metodo eseguiranno il codice nativo del bytecode originario. Ciò consente, in alcuni casi, di velocizzare, di oltre venti volte, l'esecuzione dei programmi.

Un insieme di strumenti che consentono lo sviluppo di programmi Java è contenuto nel Java Development Kit (JDK).

Esso è di pubblico dominio e viene distribuito da diversi produttori, tra cui la Sun. Ne esistono varie versioni, per UNIX, per Windows 95/NT, per MacIntosh, ecc. Tutti i programmi di cui si compone dispongono di un'interfaccia a linea di comando. Di seguito ne sono elencati alcuni:

- il compilatore *javac*: la sua esecuzione su un file con estensione *java* produce un file *.class*, il bytecode, per ogni classe definita nei file sorgenti;
- l'interprete *java*: il compilatore fornito nel JDK, come detto in precedenza, non produce codice macchina, ma dei file in byte-code. Per eseguire quest'ultimi occorre utilizzare l'interprete tramite il comando *java* invocato sulla classe contenente il metodo che rappresenta il punto da cui ha inizio l'elaborazione dell'applicazione. La classe di avvio e tutte le altre classi eventualmente riferite da questa devono trovarsi in una delle directory specificate nella variabile di ambiente *CLASSPATH* o nella directory corrente;
- *Java Archive Tool*: il tool *jar* è utilizzato per creare e manipolare file compressi che possono, in tal modo, essere trasferiti sulla rete con una maggiore efficienza. Risulta particolarmente utile soprattutto nel caso di applet che devono venir scaricate da un server Web in un browser;
- certificati e firme digitali: *javakey* permette di generare certificati da utilizzare per la firma digitale delle applet. E' strettamente legato ai meccanismi di sicurezza implementati dal linguaggio che consentono l'esecuzione delle applet stesse solo nei casi in cui ne sia, in qualche modo, garantita la provenienza;
- il debugger *jdb*: permette di eseguire il debugging delle classi Java;
- il comando *javah*: consente di costruire, a partire da una classe Java, un file in C con estensione *h* contenente le dichiarazioni di una *struct* corrispondente alla classe stessa. Ciò permette alle applicazioni Java di interagire con codice scritto in C, realizzando i cosiddetti metodi *nativi*.

Java definisce un ambiente generale, la cosiddetta *Java Platform*, che consente di eseguire uno stesso applicativo, senza necessità di ricompilazione ed adattamento, sui diversi sistemi di calcolo in cui è disponibile. Si compone di due parti fondamentali:

1. la *Java Virtual Machine*;
2. la *Java Application Programming Interface* (Java API).

2.4.2 La Java Virtual Machine e le API Java

La Java Virtual Machine (JVM) è una macchina virtuale che esegue delle istruzioni di basso livello in formato bytecode. Tutte le JVM sono equivalenti dal punto di vista dell'esecuzione del bytecode, ma la realizzazione di ognuna di esse dipende dalla particolare piattaforma sottostante. La Figura 2.10 mostra una rappresentazione semplificata di una JVM.

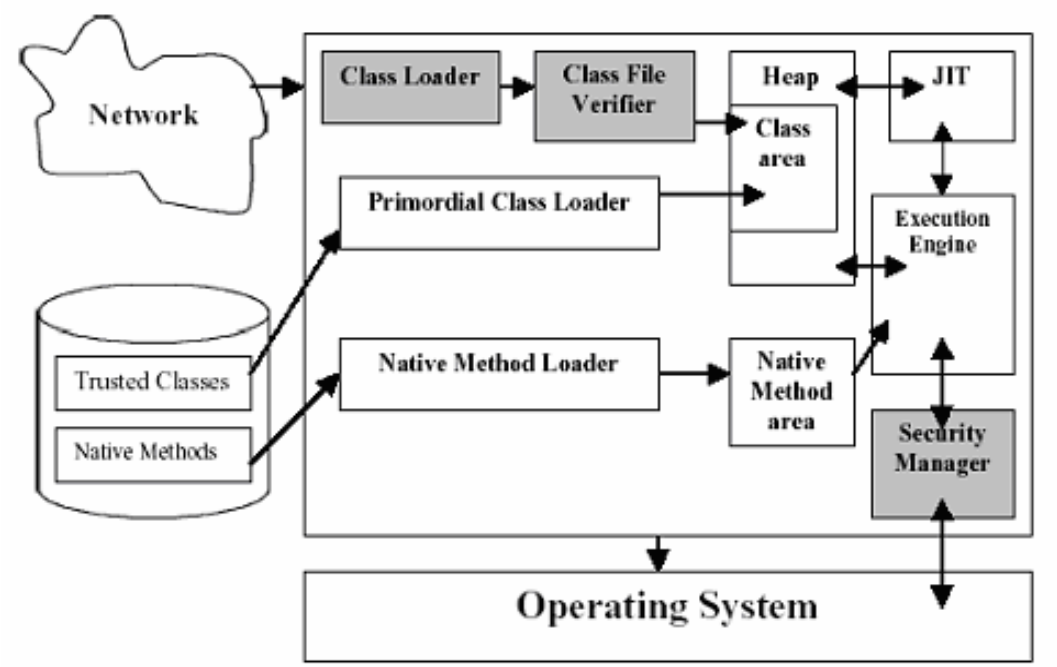

Figura 2.10 I componenti della Java Virtual Machine

In seguito sono brevemente analizzati gli elementi che costituiscono la Java Virtual Machine.

Class Loader

Prima che la JVM possa eseguire un programma Java occorre localizzare e caricare in memoria le classi di cui questo si compone.

Il Class Loader è costituito da diversi componenti, ognuno dei quali è responsabile della localizzazione ed il caricamento di un certo tipo di classe.

Le modalità con cui avviene il caricamento delle classi dipende dalla tipologia di queste. In particolare, le classi sono suddivise in tre categorie:

- classi appartenenti alla Java API: sono le classi fornite con la JVM che offrono le funzionalità per l'accesso alla rete, la gestione dei thread e la realizzazione di interfacce grafiche. Sono considerate sicure e, pertanto, non sono soggette allo stesso livello di controllo riservato alle classi acquisite da sorgenti esterne e possono, quindi, essere caricate in modo più diretto rispetto alle altre;
- classi installate nel file system locale: si tratta, anche in tal caso, di classi ritenute sicure, in quanto installate dall'utente che, presumibilmente, ne ha accettato i rischi connessi. Vengono, quindi, trattate come le classi presenti nella Java API;
- classi provenienti da altre sorgenti: all'interno di un browser rappresentano le classi di un'applet acquisita da un Web server remoto. Vengono considerate le meno affidabili, in quanto introdotte in un ambiente sicuro, quale quello della JVM, a partire da sorgenti potenzialmente ostili e spesso senza l'esplicito consenso da parte dell'utente. Per tale ragione, queste classi devono essere sottoposte ad un elevato grado di controllo prima di poter essere eseguite.

Class File Verifier

Alcuni file contenenti classi caricate dalla JVM provengono da sorgenti non sicure, per cui è necessario controllarli prima della loro esecuzione, al fine di verificarne l'integrità. Tale compito viene as-

solto dal Class Verifier, invocato dal Class Loader, che esegue una serie di test su quei file considerati potenzialmente nocivi.

Heap

E' un'area di memoria utilizzata dalla JVM per memorizzare oggetti Java durante l'esecuzione di un programma. Quando uno di questi non viene più utilizzato, un componente della JVM, il *garbage collector*, lo dealloca, liberando spazio che diviene disponibile per un eventuale riuso.

Class Area

E' il luogo in cui la JVM memorizza le informazioni relative ad una classe, quali, ad esempio, i metodi contenuti e le variabili statiche.

Spesso la Class Area è parte dello Heap, ma può anche far parte di una zona di memoria diversa. Ciò dipende dalla particolare implementazione della JVM.

Native Method Loader

Molte classi appartenenti alla Java API, quali, ad esempio, quelle che permettono la realizzazione di interfacce grafiche o l'accesso alla rete, richiedono implementazioni in codice nativo per poter utilizzare le funzioni del sottostante sistema operativo. Il Native Method Loader è responsabile della localizzazione e del caricamento di tali classi all'interno della JVM. Da notare il fatto che quest'ultima non è in grado di eseguire alcun tipo di controllo di sicurezza sul codice nativo.

Native Method Area

Rappresenta l'area di memoria in cui viene memorizzato il codice nativo, una volta caricato. Ciò ne consente un accesso più veloce, quando richiesto.

Security Manager

Una volta che il codice considerato non affidabile è stato controllato, questo è ancora soggetto ad ulteriori restrizioni. Il Security Manager permette di gestire tali vincoli. In un Web browser, quest'ultimo viene fornito dal produttore e rappresenta il componente della JVM che impedisce alle applet, per esempio, di leggere e scrivere sul file system locale, di stampare e così via.

Execution Engine

E' il cuore della JVM. Rappresenta il processore virtuale che esegue il bytecode.

Trusted Classes

Sono le classi fornite come parte dell'implementazione della JVM. Includono tutti i package che iniziano con "java."e "sun." oltre alle classi usate per implementare i componenti della Java API che dipendono dalla piattaforma. In genere, vengono poste nel file system, in un file compresso chiamato *classes.zip*.

Infine, come già detto, il secondo componente fondamentale della piattaforma Java è la *Java Application Programming Interface (API)*, un insieme di funzionalità di base offerte dal linguaggio agli applicativi ed eseguite dalla JVM al momento della richiesta. Ne sono state definite due tipologie, la Java Base API e la Java Embedded API. La prima fornisce un'insieme di funzionalità di base per l'esecuzione di applicazioni Java su computer, workstation e

network computer. A queste ne possono essere aggiunte altre presenti nella Standard Extension API. La seconda è stata definita per i prodotti dell'elettronica di consumo che richiedono minori risorse e funzionalità più specializzate rispetto ai computer.

2.4.3 Remote Method Invocation (RMI)

RMI è un API che fa parte della libreria standard del JDK 1.1. Esso consente di utilizzare un oggetto che si trova su una macchina remota come se fosse locale [24]. Grazie a tale meccanismo è possibile comandare una serie di oggetti in esecuzione su uno o più server remoti, dando luogo ad un'applicazione distribuita costituita da tante unità periferiche al servizio di una centrale.

La Figura 2.11 mostra la struttura tipica di un'applicazione che utilizza il protocollo RMI.

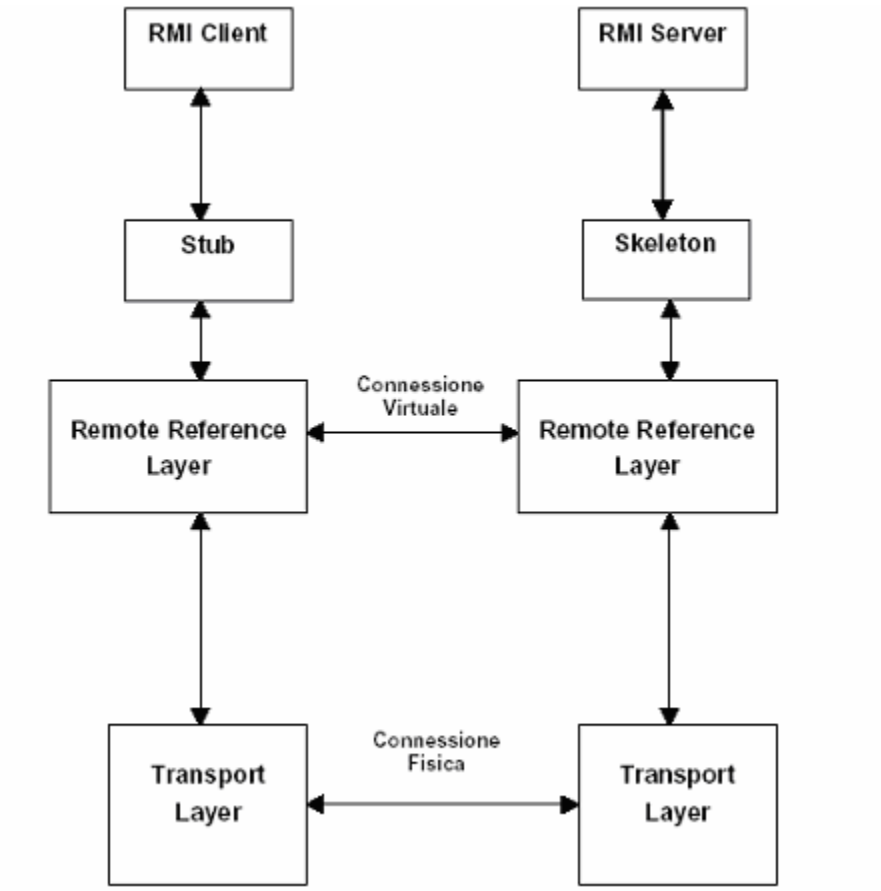

Figura 2.11 Protocollo RMI

Come si può vedere, abbiamo un server ed un client suddivisi in più livelli. Lo strato più alto è costituito, su entrambi i lati, dall'applicazione che viene eseguita sulla JVM. Al di sotto del livello applicazione, si trovano i due componenti fondamentali del meccanismo RMI, lo *stub* e lo *skeleton*.

Per capire il motivo di questa doppia presenza, è bene fare un passo indietro, riconsiderando il caso non distribuito. In tale circostanza, quando un oggetto desidera invocare un metodo di un altro oggetto, deve prima ricavarne un riferimento, istanziarlo direttamente o ricevendone l'indirizzo dall'esterno e, successivamente, eseguire un' istruzione del tipo:

```
nome_oggetto.nome_metodo(lista parametri)
```

Nel caso distribuito, invece, si deve ricavare il riferimento all'oggetto remoto ed invocarne i metodi a distanza.

La soluzione proposta da RMI consiste nello scaricare in locale un proxy dell'oggetto remoto e di considerarlo come se si trattasse a tutti gli effetti un elemento locale. E' la tecnologia RMI che si occupa in maniera del tutto trasparente di inoltrare le richieste di esecuzione dei vari metodi all'oggetto remoto residente sul server. In tal modo, il client ha l'impressione di maneggiare una risorsa locale, ma in realtà il codice viene eseguito dal processore residente sul server remoto.

I due oggetti che implementano questo meccanismo, il rappresentante locale (proxy) ed il reale oggetto remoto ad esso collegato, sono appunto lo stub e lo skeleton. Grazie a questo meccanismo il codice necessario ad invocare un oggetto remoto è praticamente identico al caso precedente, infatti basta scrivere:

```
nome_oggetto_remoto.nome_metodo(lista parametri)
```

L'equivalenza al caso non distribuito, in riferimento all'invocazione dei metodi, vale anche per il passaggio dei parametri. Ad un metodo remoto, infatti, può essere passata una variabile, un'istanza di un oggetto od una collezione di oggetti. Non esistono limiti da questo punto di vista, a patto che l'oggetto in questione sia serializzabile, ossia possa essere trasformato in un flusso sequenziale di informazioni trasferibili tramite uno stream.

Consideriamo ora gli altri strati presenti nello schema di Figura 2.11. I lati server e client sono collegati con il sottostante *Remote Reference Layer* (RRL), che a sua volta si appoggia al *Transport Layer* (TL). Il primo dei due ha il compito di instaurare un collegamento logico fra i due lati, codificare le richieste del client ed inviarle al server, decodificare le richieste ed inoltrarle allo skeleton. Ovviamente, nel caso in cui quest'ultimo fornisca dei risultati per il particolare tipo di servizio richiesto, il meccanismo di restituzione di tali valori avviene in maniera del tutto analoga, ma in senso opposto.

La gestione dei riferimenti ai vari oggetti e tutto quello che concerne le operazioni di basso livello avvengono, in parte, tramite l'RRL, ma soprattutto tramite il TL, in cui i dati vengono semplicemente visti come sequenze di byte da scrivere o leggere in certe locazioni di memoria. Quando il TL riceve una richiesta di connessione da parte del client, localizza il server relativo all'oggetto remoto richiesto. Successivamente viene eseguita una connessione per mezzo di una socket appositamente creata per il servizio. Una volta che la connessione è stabilita, il TL passa tale connessione al lato client del RRL ed aggiunge, in una tabella opportuna, un riferimento all'oggetto remoto. Solo dopo questa operazione il client risulta effettivamente connesso al server e lo stub è utilizzabile dal client.

Il TL è responsabile del controllo dello stato delle varie connessioni e della serializzazione dei parametri da passare ai metodi; quest'ultimo meccanismo è alla base delle trasmissioni dei dati fra client e server.

Se un periodo di tempo significativo passa senza che venga effettuato nessun riferimento alla connessione remota, si assume che tale collegamento non sia più necessario e, quindi, viene disattivato.

L'ultimo livello che, però, non viene incluso nel protocollo RMI è quello che riguarda la gestione della connessione attraverso le socket ed i protocolli TCP/IP.

Capitolo 3
Reflective middleware

3.1 Introduzione

Mark Weiser definisce l'ubiquitous computing con il concetto di "everything, everywhere, all the time computing" [3]. Dal punto di vista dell'infrastruttura software, una delle chiavi per rendere possibile un continuo computing è garantire un'interoperabilità senza limiti tra i dispositivi. L'utilizzo del middleware semplifica la costruzione e la comunicazione di componenti basati su applicazioni distribuite.

Per la natura degli scenari di ubiquitous computing, non è possibile prevedere una singola implementazione statica del middleware per l'intero ambiente. Si sono ormai diffuse differenti implementazioni di middleware e per soddisfare l'esigenza di eterogeneità dell'ambiente (coesistenza di piattaforme hardware, device, sistemi operativi e protocolli di comunicazione) è necessaria una struttura ancora più flessibile [25].

Il *reflective middleware* [26] introduce questa flessibilità risultando un elegante approccio per far fronte ai requisiti dell'ubiquitous computing. Esso sfrutta un protocollo meta-object [27], che permette di ridisegnare, anche a tempo di esecuzione, l'architettura del motore di un middleware (come politiche, meccanismi o strategie), facendo in modo che diversi tipi di piattaforme middleware su differenti sistemi possano interoperare in

uno scenario di ubiquitous computing.

3.2 Struttura di un reflective middleware

Il reflective middleware usa tecniche derivate dal *computational reflection* per aggiungere flessibilità al middleware.

Il computational reflection consiste nella possibilità da parte di un sistema software di effettuare delle computazioni (meta-computazioni) relativamente alle strategie elaborative del sistema stesso. Per questo un reflective middleware è un sistema software che incorpora strutture (rappresentazione interna) che realizzano alcuni aspetti del sistema stesso (rappresentazione esterna). Esso è logicamente strutturato in più livelli logici e ad ogni livello operano delle entità che definiscono il comportamento del sistema.

Il reflective middleware di Gregory Kiczales sfrutta un protocollo meta-object, combinando l'idea del computational reflection con la filosofia object-oriented. Il suo modello distingue gli oggetti di livello base, concernenti gli aspetti funzionali del sistema, dagli oggetti di medio livello, concernenti invece aspetti come politiche, meccanismi o strategie. Il livello base del reflective middleware indirizza le funzionalità dei programmi d'applicazione, mentre il livello medio disegna collezioni di componenti che compongono l'architettura interna della piattaforma middleware. Il reflective middleware permette dunque l'ispezione e la modifica di questi oggetti, abilitando cambiamenti nel comportamento del middleware.

A questo punto illustriamo un esempio significativo di reflective middleware, DynamicTAO, per poi mostrare la piattaforma che abbiamo scelto come punto di partenza per l'implementazione del nostro reflective middleware, ossia UIC.

3.3 Un primo esempio di reflective middleware: DynamicTAO

Con l'avvento del pervasive computing e dei dispositivi portatili, e con il miglioramento della portabilità della rete, è cominciato a diffondersi un ampio insieme di servizi, per utenti mobili, che soddisfino la caratteristica dell'ubiquità.

Poiché l'ubiquitous computing è caratterizzato da un costante cambiamento e un largo grado di dinamismo, gli utenti devono poter interagire con sistemi eterogenei usando vari tipi di dispositivi posti in locazioni diverse su reti e con qualità del servizio differenti.

A tutto questo va aggiunta la condivisione dello stesso sistema con moti altri utenti. In termini pratici esiste la difficoltà di concepire un fissato insieme di politiche e di meccanismi per l'allocazione di risorse e per la comunicazione e la sicurezza che si adattano alle differenze, ai cambiamenti e alla scalabilità. Il reflective middleware offre la flessibiltà in quest'ambiente dinamico.

3.3.1 Perché estendere TAO

DynamicTAO è un reflective ORB dello standard CORBA, costruito come un'estensione di TAO [28], che raccoglie gli enormi sforzi di Doug Schmidt e dei suoi colleghi per organizzare e modulare il middleware.

TAO è un ORB portabile, flessibile, estendibile e configurabile e usa la strategia di progettazione modulare per incapsulare i differenti aspetti del motore interno dell'ORB. Un *configuration file* specifica la strategia dell'ORB usata per implementare aspetti come la concorrenza, lo scheduling e la gestione delle connessioni. Al momento dello startup dell'ORB, il configuration file è analizzato e le

strategie selezionate sono caricate. TAO è principalmente progettato per applicazioni statiche *real-time*. Per questo, una volta che l'ORB è stato configurato, le sue strategie restano le stesse almeno finché non completa la sua esecuzione.

DynamicTAO [29] estende TAO per supportare la riconfigurazione *on-the-fly* assicurando che il motore dell'ORB resti consistente.

Ciò è reso possibile dalla struttura interna, che mantiene un'esplicita rappresentazione dei componenti interni dell'ORB e dell'interazione dinamica tra loro. Questo permette all'ORB di cambiare la specifica strategia senza far ripartire la sua esecuzione.

DynamicTAO è un reflective ORB che consente il controllo e la riconfigurazione del suo motore interno attraverso un'interfaccia che permette:

- il trasferimento di componenti lungo un sistema distribuito;
- di caricare e rilasciare moduli nell'ORB a run-time;
- il controllo e la modifica dello stato di configurazione dell'ORB.

L'infrastruttura può anche essere riconfigurata dinamicamente per applicazioni non-CORBA.

3.3.2 Architettura

Ogni processo che lancia DynamicTAO ORB contiene un componente configurator chiamato *DomainConfigurator*. Questo conosce le dipendenze tra un certo componente e gli altri componenti di sistema ed è responsabile di mantenere i riferimenti all'istanza dell'ORB e ai servants running di questo processo. In più, ogni istanza ORB contiene il *TAOConfigurator*, che gestisce i puntatori alle implementazioni delle strategie di DynamicTAO. Le specifiche implementazioni puntate sono resi disponibili all'ORB. La Figura 3.1 il-

lustra quanto detto per un processo contenente una singola istanza ORB.

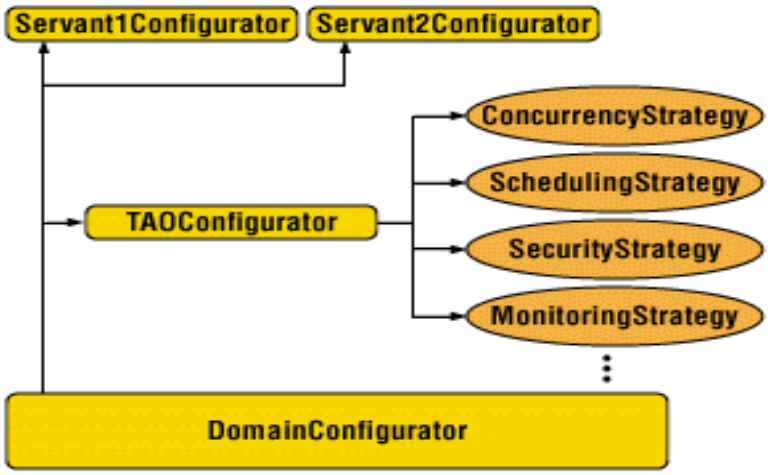

Figura 3.1 Configurazione di un processo con un'unica istanza ORB

Se necessario, la singola strategia può usare componenti configurator per memorizzare:

- la sua dipendenza all'istanza ORB;
- la dipendenza alle altre strategie;
- i riferimenti alle richieste del client che dipendono dalle strategie.

Con queste informazioni è possibile gestire la riconfigurazione in modo *consistente*.

I componenti che implementano le strategie sono dinamicamente caricabili con librerie così da poter essere collegati con il processo ORB a run-time; organizzati in categorie, rappresentano diffe-

renti aspetti del motore interno dell'ORB o differenti tipi di componenti servants.

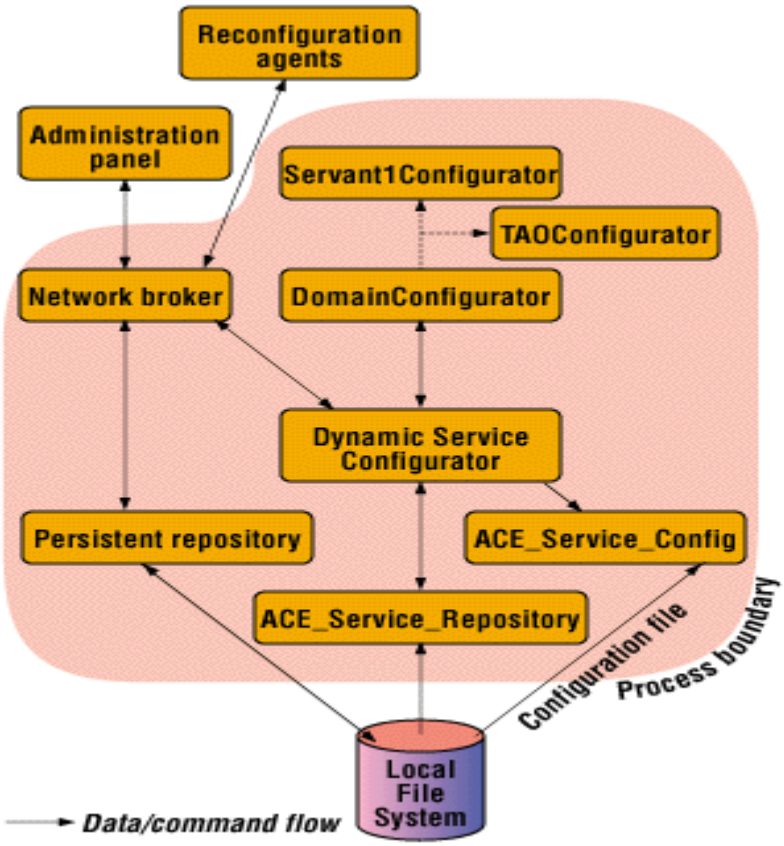

Figura 3.2 Componenti di DynamicTAO

La Figura 3.2 descrive la struttura architetturale di DynamicTAO e in seguito sono mostrati i componenti principali:

- *ACE_Service_Config*: responsabile del linkaggio dinamico dei servizi posti sul disco locale. Questi servizi sono inseriti nell'ACE_Service_Repository e usati da TAO per creare le strategie dell'ORB;

- *ACE_Service_Repository*: magazzino di memoria dei servizi. Può controllare la disponibilità di servizi e fare il *lookup* di un servizio particolare. I servizi compongono una strategia;
- *Default_Strategy_Factory*: fabbrica di oggetti 'strategia' per TAO. Si può distinguere in Default_Server_Strategy_Factory, se le strategie create sono usate dal lato server, e Default_Client_Strategy_Factory, nel caso siano utilizzate dal lato server;
- *Dynamic_Service_Configurator*: riceve richieste di riconfigurazione dal Net_Broker e delega l'esecuzione di alcune funzioni ai specifici componenti configurators (ad esempio, potrebbero essere un TAOConfigurator o uno specifico ServantConfigurator) e all'ACE_Service_Config (per l'inizializzazione di nuovi servizi e strategie e la loro inclusione nell'ACE_Service_Repository). Il Dynamic_Service_Config contiene il DomainConfigurator (mostrato in Figura 3.1) e fornisce operazioni comuni per la configurazione dinamica di componenti a run-time; inoltre riceve richieste sullo stato dei servizi immagazzinati. In questo caso, interroga l'ACE_Service_Repository e fornisce il risultato in un appropriato formato al Net_Broker;
- *Net_Broker*: riceve richieste di riconfigurazione e di lettura dello stato di configurazione da entità remote e le spedisce ai componenti locali. Solitamente, esso serve le richieste chiamando il Dynamic_Service_Config. Il Net_Broker è anche usato da entità remote per caricare nuove strategie e servizi sul file system locale;
- *Persistent Repository*: memorizza le differenti rappresentazioni (categorie) dell'ORB nel file system locale, offrendo metodi per la manipolazione (per esempio, la navigazione, la creazione o la cancellazione) di queste categorie e l'implementazione di ognuna di esse. Tali metodi sono attivati da richieste di riconfi-

gurazione provenienti dalla rete di comunicazione e smistati al *Network Broker*.

Ricapitolando, l'infrastruttura reflective middleware deve poter riconfigurare il suo motore interno per interoperare con differenti piattaforme middleware. Rispettando le opzioni di configurazione, le implementazioni esistenti, normalmente, permettono di modificare le politiche di concorrenza, le strategie di marshalling ed altri aspetti. Tuttavia, non è solo un problema di riconfigurare le politiche associate con altre prestabilite, ma in alcuni casi è importante modificare l'architettura interna dell'intero motore middleware. Quindi, attraverso la meta-interface del reflective middleware non solo deve essere possibile scegliere le implementazioni di strategie tra le categorie predefinite, ma si deve poter aggiungere o rimuovere tipi di categorie e riorganizzare l'architettura interna del motore del middleware. Questa proprietà permette la riconfigurazione del middleware dal punto di vista delle funzionalità richieste, facendo il possibile per rimuovere le funzionalità che a lungo non vengono utilizzate e addizionando le nuove richieste.

3.3.3 ORB consistency

L'obiettivo principale di un reflective middleware è avere un ORB core minimale in esecuzione senza interruzioni, mentre le strategie dell'ORB e dei servents sono caricate dinamicamente. Supportando una riconfigurazione on-the-fly, per l'ORB si introducono diverse importanti regole che non esistono con una configurazione in partenza statica. Il cambiamento di un'implementazione di una strategia a run-time implica un'attenta analisi. Ci sono dipendenze con le implementazioni caricate della stessa categoria utilizzate anche su

altri ORB e dipendenze con i client che stanno usando la vecchia implementazione.

Lo stato iniziale di una nuova strategia potrebbe dipendere dallo stato finale di una vecchia strategia che sta per essere rimpiazzata.

Non si può dunque eseguire alcuna operazione di riconfigurazione in maniera immediata perché la loro esecuzione potrebbe lasciare lo stato dell'ORB inconsistente. Ad esempio, cambiare una strategia con un'altra e cancellare una strategia sono due delle operazioni più conflittuali. Gli step da seguire sono due. Primo, bisogna determinare che l'implementazione della nuova strategia è compatibile con le restanti implementazioni analizzando le altre strategie in esecuzione sullo stesso ORB e, nello stesso tempo, le strategie della stessa categoria sugli altri ORB. Infatti, mentre cambiamo l'implementazione su un certo ORB, potrebbe essere richiesto il cambiamento dell'implementazione su più ORB remoti (per esempio, se le implementazioni caricate sono parti di un protocollo distribuito). Secondo, potrebbe essere necessario trasferire lo *stato* dell'implementazione sostituita alla nuova implementazione che la rimpiazza. Infine, bisogna tenere in considerazione che cambiare una strategia d'implementazione può produrre effetti sulle richieste del client che sono in processo. Ciò dipende molto dalla categoria dell'implementazione. Nel caso delle strategie di marshalling ed unmarshalling, per esempio, il cambiamento ha un effetto immediato sulle richieste successive. Nel caso di una strategia di concorrenza, tuttavia, i cambiamenti sull'implementazione hanno effetti solo sui nuovi client [30].

3.4 Portare il reflective middleware nel palmo di una mano

3.4.1 I device nello scenario dell'ubiquitous computing

Nel considerare i dispositivi PDA, sensori, telefonini e apparecchiature elettroniche varie che possono trovarsi nello scenario dell'ubiquitous computing, si individuano delle specifiche che l'ambiente deve possedere:

- adattarsi a dispositivi con risorse limitate;
- eterogeneità;
- alto grado di dinamismo.

Le risorse limitate a disposizione variano da dispositivo a dispositivo. Mentre per dispositivi come PDA è facile accrescere la potenza offerta e la memoria disponibile, per altri dispositivi, come sensori e piccole apparecchiature elettroniche, non è possibile o è troppo costoso aggiungere risorse aggiuntive.

L'eterogeneità è indispensabile, poiché differenti piattaforme hardware, sistemi operativi e piattaforme software implicano cambiamenti in alcuni parametri come l'ordinamento dei byte, la lunghezza in byte dei tipi standard e i protocolli di comunicazione.

Il grado di dinamismo nell'ubiquitous computing non è immaginabile in un ambiente tradizionale di server e workstation. Un PDA, per esempio, interagisce con molti dispositivi e servizi in differenti locazioni e ciò implica l'evolversi di molti parametri a partire dal tipo di comunicazione di rete, protocolli RPC e politiche di sicurezza. Quindi, poiché non è possibile predire tutte le possibili combinazio-

ni, il software sul PDA deve adattarsi ai differenti scenari dinamicamente.

Tutte le specifiche previste devono influenzare il disegno di un'infrastruttura middleware richiesta per l'ubiquitous computing. Alcune piattaforme middleware convenzionali possono adattarsi manualmente ad ogni dispositivo particolare, ma chiaramente senza la possibilità di cambiamenti dinamici a tempo di esecuzione.

Il reflective middleware si accorda perfettamente ad un ambiente ubiquitous. E' possibile mantenere le applicazioni (base level) inalterate ed effettuare cambiamenti alla piattaforma middleware (meta-level), attraverso tools che modificano il comportamento della piattaforma. Tuttavia, le implementazioni esistenti della maggior parte dei reflective middleware non sono appropriate a piccoli dispositivi mobili in uno scenario di ubiquitous computing per tre principali problemi: misura, opzioni di configurazione e dipendenza da una singola piattaforma middleware. Il problema nella maggior parte dei casi è che l'architettura middleware è costruita utilizzando un approccio monolitico e questo impone requisiti di ampia memoria.

3.4.2 Limiti dei reflective middleware esistenti

La maggior parte degli attuali reflective middleware richiede un insieme di requisiti hardware e software appropriati per computer desktop e non per palmari, PocketPC o piccoli dispositivi elettronici [25]. Alcuni middleware come Java RMI e CORBA sono più adatti a dispositivi con ampie risorse e solo da poco si stanno sperimentando soluzioni che prevedono tools di configurazione che si adattano alla differente architettura di dispositivi con risorse limitate. Tuttavia questi tentativi rimangono confinati ai device che adottano l'opportuno middleware senza la flessibilità necessaria in un am-

biente ubiquitous dove è prevista una interazione spinta tra dispositivi molto differenti tra loro (per piattaforma, sistema di comunicazione, ecc.).

Un problema è che tipicamente le applicazioni richiedono solo un piccolo sottoinsieme di funzionalità del middleware, ma spesso si vuole forzare a memorizzare sul dispositivo librerie con tutte le funzionalità, anche quelle che rimangono inutilizzate. Nel caso CORBA, per esempio, un programma client che semplicemente invoca un metodo su un oggetto remoto richiede solo le funzionalità del lato client dell'ORB e una interfaccia per invocazioni dinamiche o statiche.

Sfortunatamente, più implementazioni ORB includono tutte le funzionalità in una singola libreria o al massimo separano le librerie per il lato client e server, ma senza alcuna possibilità di scegliere uno specifico sottoinsieme di queste funzionalità. La maggior parte delle implementazioni middleware esistenti sono state disegnate per assistere allo sviluppo di applicazioni su server per workstation e desktop. I modelli d'esecuzione e i requisiti di queste applicazioni sono normalmente ben conosciute e le piattaforme di communication middleware esportano le funzionalità adatte a questi scenari (per esempio, applicazioni bancarie e prenotazioni aeree).

3.4.3 Un esempio di utilizzo del reflective middleware

Per comprendere tutte le proprietà e i requisiti menzionati, consideriamo il seguente esempio. Un utente, portando con sé un handheld device (dispositivo piccolo tale da tenerlo in una mano e capace di connettersi in rete), entra in una stanza che contiene dispositivi che controllano le luci e la musica di sottofondo. Tutti i dispositivi sono connessi ad una rete wireless. L'utente può usare

l'handheld device per controllare le luci (spegnerle e accenderle o controllare il loro livello d'intensità), così come il volume della musica. Si assume la presenza di una piattaforma reflective middleware che implementa tutte le proprietà menzionate prima (adattamento alle dimensioni dei device, opzioni di configurazione per la flessibilità e piattaforme middleware indipendenti).

La luce e la musica possono essere controllate istanziando un oggetto che esporta le loro funzionalità sul lato server. Ipotizziamo che il controller della luce usi CORBA e che quello della musica usi SOAP [31]. A questo punto, abbiamo tutte le informazioni necessarie per configurare la piattaforma middleware che usa questi dispositivi; è importante decidere se la configurazione è statica o dinamica.

Nel caso di dispositivi di controllo, non è una funzionalità standard utilizzata dall'utente quella di effettuare cambiamenti sul controller, sebbene resta tra le opportunità offerte dallo scenario ubiquitous. Se, poi, i dispositivi di controllo non sono mobili ed è ben definito un modello di accesso, non c'è da aspettarsi cambiamenti, anche perché il tipo di rete che usano non cambierà. Quindi la miglior soluzione (dal punto di vista della semplicità e misura del codice) è quella di configurare staticamente la piattaforma middleware che va successivamente installata nel device. Per i requisiti previsti, la configurazione ottimale per il controller della luce potrebbe consistere in un oggetto lato server CORBA che semplicemente permette la registrazione di un oggetto senza implementare alcuna altra politica. Per il controller della musica, si configura staticamente la piattaforma middleware per esportare le funzionalità server-side SOAP.

L'handheld device necessita solo delle funzionalità client-side, poiché non riceve alcuna richiesta dagli altri due dispositivi. Tuttavia, l'handheld device ha bisogno di una configurazione che gli permette di interagire con oggetti CORBA e SOAP. Poiché il disposi-

tivo è mobile, non è possibile conoscere a priori i suoi requisiti, quindi non è appropriata una configurazione statica dell'infrastruttura middleware. Tale aspetto si risolve se la piattaforma middleware caricata sul dispositivo prevede dei tools che supportano la riconfigurazione dinamica.

Per completare lo scenario di ubiquitous computing bisogna ricordare che, quando l'handheld device entra nella stanza, esso non conosce l'esistenza dei controller e i loro requisiti; deve dunque esistere un discovery service che provvede ad informare il device di ciò che l'ambiente gli mette a disposizione (i dispositivi esistenti, i servizi offerti, le loro interfacce, le proprietà delle loro piattaforme middleware).

Le proprietà del middleware dei controller sono usate per riconfigurare il middleware dell'handheld device, permettendo l'interoperabilità.

In questo modo, le applicazioni eseguibili sull'handheld device usano le interfacce dei controller per creare richieste dinamicamente. La riconfigurazione dinamica, in questo caso, avviene per includere un meccanismo di invocazione dinamica di metodi per oggetti CORBA e SOAP.

3.5 La soluzione adottata: UIC

DynamicTAO ha una architettura molto versatile che ben si adatta alle esigenze di un ambiente ubiquitous; la sua possibilità di riconfigurare il motore interno di un ORB a run-time, di caricare e scaricare strategie e servizi per le esigenze delle applicazioni, mantenendo la consistenza dell'ORB, permette una buona interoperabilità tra i dispositivi. Resta comunque la difficoltà di raggiungere device di limitate risorse e così, nel 2001, è nato, come successore di DynamicTAO*,* un reflective ORB di terza generazione: *UIC*.

L'*Universally Interoperable Core* (UIC) [32] è stato progettato specificamente per dispositivi con scarse risorse, ai quali permettere l'interoperabilità bidirezionale con le piattaforme middleware esistenti, in configurazione statica o dinamica.

3.5.1 Architettura

UIC definisce uno scheletro, il core, basato su componenti astratti che incapsulano le seguenti funzionalità standard, che sono aspetti comuni della maggior parte degli oggetti intermediari richiesti da un ORB:

- rete e protocollo di trasporto;
- creazione della connessione;
- strategia di marshalling e demarshalling;
- protocollo d'invocazione dei metodi;
- metodi d'invio dello scheduling di messaggi;
- generazione di riferimento ad oggetti e analisi;
- registrazione degli oggetti e metodo di spedizione;
- interfaccia client;
- interfaccia server;
- interfaccia di oggetti e attributi;
- gestione della memoria;
- strategie di concorrenza.

In concreto, i componenti che specializzano questi componenti astratti costituiscono il livello Specialization. Core e Specialization sono caricati dinamicamente dai device che li utilizzano per implementare le caratteristiche delle particolari piattaforme middleware, delle reti e dei protocolli di comunicazione che gli stessi device dovranno utilizzare.

UIC supporta i piccoli dispositivi e permette di personalizzare il suo core (sia dinamicamente che staticamente) per dispositivi eterogenei ed ambienti. La possibile configurazione va dalla versione con minime funzionalità, memoria e risorse alla versione completa.

Quest'approccio differisce dalle esistenti soluzioni, dal momento in cui queste ultime non permettono alle piattaforme middleware di rivolgersi anche ai dispositivi handheld.

Due sono le principali linee guida della progettazione di UIC: la semplicità e il seguente concetto: *'What You Need Is What You Get'* (WYNIWYG).

La semplicità richiede che la decomposizione di UIC in moduli sia facile e chiara. I progetti modulari prevedono la separazione di concetti minimizzando le dipendenze, permettendo, così, un facile adattamento alle esigenze dei differenti ambienti e delle applicazioni eseguite.

Il WYNIWYG si realizza, in pratica, facendo in modo che la funzionalità richiesta sia presente come un'istanza di UIC. Tuttavia, se sono richieste più funzionalità, è possibile facilmente introdurre le nuove funzionalità staticamente (ricompilando l'istanza con le nuove classi che implementano le funzionalità) o dinamicamente (addizionando le nuove funzionalità a run-time). Allo stesso modo, quando alcune funzionalità non sono più richieste, esse possono essere rimosse.

3.5.2 Personality

UIC è un reflective middleware che deve essere specializzato per incontrare le richieste delle particolari applicazioni. Queste specializzazioni, dette personality, possono definire UIC in modo da farlo interagire con una specifica entità (ad esempio server CORBA o server Java RMI).

Le personality possono essere classificate come client-side, server-side o entrambi, secondo la funzionalità prevista. Una personality client-side prevede funzionalità per spedire richieste e ricevere risposte, mentre una personality server-side è capace di ricevere richieste e spedire risposte. In questo modo i dispositivi possono accedere a funzionalità remote ed esportare funzionalità.

UIC può essere classificato come *single-personality* e *multi-personality*. Una single-personality UIC è capace di interagire con una singola piattaforma middleware, mentre una multi-personality UIC può interagire con più di una piattaforma contemporaneamente. E′ da sottolineare che una multi-personality UIC non è equivalente ad un insieme di single-personality UIC. Con un insieme di single-personality UIC, infatti, l′applicazione deve decidere quale UIC personality usare in base all′oggetto con il quale si vuole interagire. Invece, con una multi-personality UIC, l′applicazione usa la stessa istanza UIC e la stessa interfaccia per invocare un metodo senza interessarsi del tipo dell′oggetto remoto. In questo caso UIC automaticamente sceglie l′esatta personality. Quanto osservato è realizzato grazie all′*UIC Multi-personality Core* analizzato in seguito.

L′infrastruttura dei componenti base di UIC facilita lo sviluppo di personality che semplicemente implementano un insieme di componenti. Per esempio, un programmatore di un′applicazione che richiede una strategia specializzata di marshalling o una particolare strategia di connessione deve semplicemente reimplementare questi componenti specifici e riusare il resto senza modifiche. L′interfaccia standard definita dai componenti UIC garantisce che i nuovi componenti potranno interoperare con quelli esistenti.

Vediamo un esempio, considerando il componente UIC responsabile della generazione dell′header di una richiesta d′invocazione e del controllo dell′header di risposta. Questa funzionalità è richiesta per ogni implementazione middleware e quindi è prevista una generica interfaccia astratta. Tale interfaccia definisce due metodi, *sen-*

*dRequest(Request *req)*, che riceve come parametro un oggetto richiesta, e *receiveReply(Reply *rep)*, che genera un oggetto contenente informazioni relativi alla risposta. La specializzazione di questo generico componente deve implementare questa generica interfaccia. I due metodi dell'oggetto che prima erano virtuali vengono implementati da una classe padre che incapsula le specifiche proprietà di un protocollo middleware.

Le personality che si producono possono risultare minimali soprattutto considerando la possibilità di realizzare istanze di UIC che posseggono solo le funzionalità necessarie, compatibili con le piattaforme middleware standard e le esigenze delle applicazioni che si vogliono utilizzare. Ciò soddisfa la specifica WYNIWYG, e allarga la possibilità ai device di piccole risorse di interoperare in un ambiente eterogeneo e ad alto grado di dinamismo, semplicemente caricando le personality adeguate. Infatti, le misure delle personality per CORBA client-side possono variare dai 18 KB per un dispositivo palmare CORBA ai 48.5 KB per un dispositivo Windows CE. I 18 KB per l'ORB CORBA personalizzano il Core UIC con componenti per stabilire connessione TCP, creare header di richiesta, codificare e decodificare i tipi base, convertire i riferimenti a stringhe in riferimenti ad oggetti e controllare gli header di risposta.

3.5.3 Configuration

UIC può essere configurato sia staticamente che dinamicamente.

Nella *configurazione statica*, le personality vengono costruite a tempo di compilazione assemblando staticamente tutti i componenti. Il risultato è il singolo componente (la personality) che non può essere riconfigurato automaticamente (la sola possibilità è di sostituire la personality). Il principale beneficio di questa configurazione è la misura della personality.

Nella *configurazione dinamica*, invece, le personality sono una collezione di librerie, che sono caricate dinamicamente, riconfigurabile a run-time. Il principale vantaggio di questa configurazione è la possibilità di modificare l'architettura delle personality dinamicamente senza alcun effetto sull'applicazione. Tuttavia questa soluzione richiede una maggiore capacità di memoria necessaria del dispositivo, perché bisogna installare dei tools per caricare e scaricare i componenti richiesti.

Esiste anche una *configurazione ibrida*, che combina i benefici delle precedenti configurazioni: i componenti che più ci interessano sono collegati insieme, mentre i componenti che pensiamo di modificare o scaricare sono costruiti come librerie scaricabili indipendentemente. Si può quindi ridurre la misura delle personality rispetto alla configurazione dinamica in quanto più componenti sono collegati insieme.

I componenti UIC non condividono lo stato di un oggetto che si trasmettono, ma lo incapsulano nell'oggetto stesso, passando, quindi, lo stato come un parametro. Questa tecnica rimuove le dipendenze e semplifica la *riconfigurazione*. I componenti semplicemente fanno riferimento ad altri componenti, di cui sono definite le funzionalità da un'interfaccia astratta. In questo modo, un componente di UIC che processa alcuni oggetti ed informazioni genera dei risultati che sono spediti ad altri componenti senza mantenere alcuno stato dei risultati generati.

3.5.4 Scalabilità ed eterogeneità

UIC è progettato per permettere l'integrazione di dispositivi handheld con limitate risorse, nel modello standard di oggetti distribuiti. Il risultato è un'infrastruttura omogenea per lo sviluppo di applicazioni e servizi che nasconde l'eterogeneità dei device. La maggior

parte delle soluzioni esistenti che integrano dispositivi handheld negli ambienti distribuiti è basata su proxy, che trasformano l'invocazione del metodo nativo in un formato proprietario che può essere usato dal dispositivo handheld. In UIC non è richiesto alcun proxy, poiché è permesso l'uso della piattaforma nativa. In questo modo, gli sviluppatori di servizi non hanno bisogno di provvedere ad un proxy per ogni tipo di device.

Uno degli aspetti più critici nella realizzazione di un ambiente ubiquitous è quello che costringe, per l'integrazione di tutti i dispositivi esistenti, ad adattare l'applicazione ad ogni nuovo device; con l'approccio UIC, l'applicazione resta realizzata secondo l'ambiente nativo, mentre il dispositivo che intende interagire con essa ha solo bisogno di caricare la personality adatta allo specifico ambiente dell'applicazione e del dispositivo stesso. Dal quel momento in poi, qualsiasi applicazione scritta in un ambiente con cui il dispositivo può interagire potrà essere utilizzata. Questa interazione avviene, purché venga realizzata un'interfaccia che definisce quali sono i metodi pubblici, secondo le specifiche del middleware su cui gira l'applicazione.

3.5.5 UIC Multi-personality Core

L'*UIC Multi-personality Core* è una specializzazione di UIC che prevede dei tools. Questi ultimi permettono di agire introducendo cambiamenti sull'architettura interna della Multi-personality Core.

La Figura 3.3 illustra l'architettura dell'UIC Multi-personality Core, che è divisa in due parti: *Core Configuration* e *Dynamic Configurator* (tools utili alla riconfigurazione dinamica). Il Core Configuration è la collezione dei componenti UIC che definiscono il comportamento del Multi-personality Core. Il Dynamic Configurator, invece, implementa il meccanismo reflective e memorizza le informa-

zioni sulla dipendenza tra i componenti. Come mostra la figura, il Dynamic Configurator è implementato come un servant object che gira sulla configurazione corrente della Multi-personality Core. Questo approccio introduce due vantaggi:

1. è possibile modificare dinamicamente il meccanismo reflective usando differenti istanze del Dynamic Configurator;
2. è possibile iscrivere, nello stesso servant Dynamic Configurator, più di una personality contemporaneamente permettendo l'esistenza di differenti tipi di client (per esempio CORBA e SOAP).

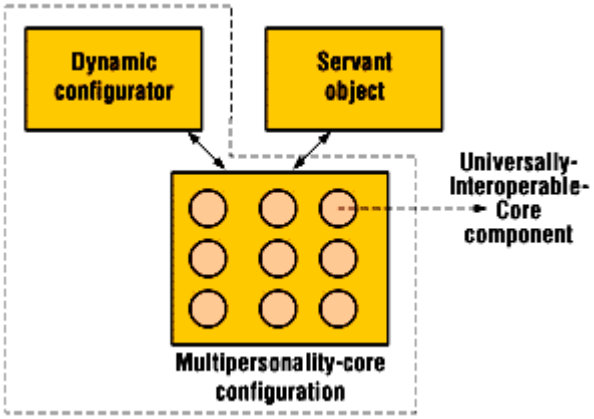

Figura 3.3 L'UIC Multi-personality Core

I tools che realizzano il Dynamic Configurator tengono conto delle dipendenze che esistono tra i componenti della Multi-personality Core (*Component Configurator pattern*) [33].

Se per esempio si vuole aggiungere la specializzazione SOAP alla Multi-personality, bisogna pensare di cambiare il comportamento dell'ORB per permettere l'interazione con gli oggetti SOAP, per esempio mappando opportunamente i tipi che si utilizzano.

3.5.6 Misure

In seguito sono presentate alcune misure di differenti personality e le performance di alcune di queste (personality CORBA per client e server).
Esistono personality che sono pienamente interoperabili con le esistenti implementazioni middleware. La personality CORBA, per esempio, permette di spedire richieste agli oggetti CORBA o ad oggetti implementati sul lato server con personality UIC CORBA. Inoltre, è possibile registrare oggetti UIC CORBA con l'esistente naming service CORBA.

UIC Personality	WindowsCE (SH3)	Windows 2000	PalmOS
HybridMulti-personality	39 KB	88 KB	N/A
CORBA Dynamic Server	28 KB	80 KB	N/A
SimpleTCP Dynamic Server	30 KB	84 KB	N/A
CORBA Static Client	29 KB	72 KB	18 KB
CORBA Static Server	35 KB	84 KB	N/A
CORBA Static Full (client + server)	48.5 KB	100 KB	31 KB

Tabella 3.1 Misure di personality

La Tabella 3.1 contiene le misure di alcune personality UIC costruite per differenti piattaforme. La *HybridMulti-personality* per-

mette di manipolare le personality (per esempio ClientCORBA, ServerCORBA e ClientJavaRMI) come componenti indipendenti. In ogni modo, è possibile aggiungere e rimuovere le personality dinamicamente, ma non riconfigurarle individualmente. Con l'HybridMulti-personality si registra un'implementazione servent object con differenti personality simultaneamente. La misura di questa personality include gli associati tools per il DynamicConfigurator.

CORBAStaticClient, *CORBAStaticServer* e *CORBAStaticFull* (client e server) sono implementazioni indipendenti che possono essere usati senza l'HybridMulti-personality. Tutte le personality CORBA descritte esportano un meccanismo di invocazione di metodi dinamici personalizzati, implementano un sottoinsieme di tipi di dati (tutti i tipi semplici, struct e qualche sequence) e segnalano le condizioni d'eccezione sollevate dal server, ma non le analizzano.

3.5.7 Overhead del reflective middleware UIC

La Figura 3.4 misura le performance in termini di numeri di richieste per secondi che vengono soddisfatte coinvolgendo gli oggetti client e server CORBA con la StaticFullPersonality UIC CORBA.

Il server implementa un metodo chiamato cubeIt, che riceve un intero long e ritorna il suo cubo; il client invoca l'operazione usando una propria interfaccia di invocazione dinamica implementata dalla personality CORBA e produce una statistica basata sulla media totale nel tempo del numero d'invocazioni remote. Il test consiste nel ripetere trecento richieste per dieci volte e il risultato finale è la media dei risultati intermedi di questi test. La socket di connessione è aperta nella prima richiesta e chiusa nell'ultima.

Allo scopo di prevedere un limite superiore per il massimo rate di richieste CORBA che potremmo permetterci, si è costruito un semplice programma che non usa CORBA ed un client che spedisce un

pacchetto di 52 byte ad un server attraverso una socket TCP/IP; appena il pacchetto arriva al server, questo lo legge (ma non lo processa) e risponde con un altro pacchetto di 28 byte che è letto infine dal client (entrambi i pacchetti spediti hanno la stessa misura nel test CORBA).

Questo semplice programma non prevede computazioni aggiuntive. Siccome né il client, né il server processano il pacchetto, abbiamo la misura del massimo rate di richieste che possono essere spedite usando le misure dell'header della personality CORBA. Il test con la personality spende un tempo per l'operazione che si aggiunge a quello della rete.

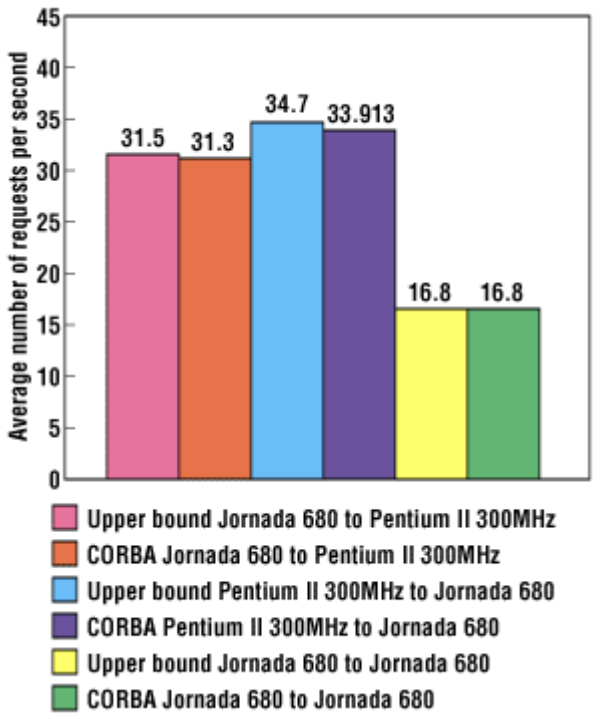

Figura 3.4 Overhead del reflective middleware UIC

La personality CORBA deve creare la richiesta header, fare il marshalling dei parametri del client, analizzare l'header lato server, fare il demarshalling dei parametri, trovare l'oggetto, chiamare il metodo, fare il marshalling dei parametri dei risultati, creare

l'header di risposta e fare il demarshalling dei parametri del risultato. La Figura 3.4 presenta tre scenari ed ognuno è diviso in due sottotest: il limite superiore ed il caso del test CORBA.

Nel primo scenario c'è un oggetto server che gira in Windows 2000 su Pentium II 300 MHz con 128 MB di RAM, con un client CE 2.11 Jornada 680 Handheld PC con processore SH3 32 bit a 133 MHz e con 15 MB RAM. Nel secondo scenario sono usati gli stessi dispositivi, ma l'oggetto client UIC CORBA gira su Windows 2000 ed il server su Jornada 680. Nel terzo scenario si usano due Handhelds Jornada 680, uno di questi ospita il client UIC CORBA e l'altro il server UIC CORBA. La rete in tutti gli scenari è una wireless RangeLAN2 con una massima banda teorica di 1.6 Mbps.

Il test dimostra che l'implementazione UIC CORBA non introduce ulteriori overhead e la media del numero dei pacchetti spediti in un secondo è fortemente influenzata dai ritardi di rete. Questo scenario non è realisticamente uno scenario di ubiquitous computing dove sono presenti un maggior numero di differenti dispositivi che condividono la stessa rete.

La Figura 3.4 evidenzia i seguenti aspetti:

1. la flessibilità prevista dal reflective middleware non si traduce necessariamente in un più lento tempo di esecuzione;
2. il reflective middleware fa il possibile per ottenere alta efficienza personalizzando l'implementazione dell'ORB che incontra i bisogni delle applicazioni.

Lo stesso test è stato provato su desktop Intel Pentium III con processore con frequenza 733 MHz, con 256 MB di RAM su S.O. Windows 2000. In questo test, sia il client che il server girano su due diversi desktop dello stesso tipo. In questo scenario la rete è Ethernet a 100 Mbps, che è condivisa da utenti non coinvolti nell'esperimento. I risultati sono descritti in Figura 3.5, che mostra

un'addizionale overhead del 4.4% nel test CORBA dovuto all'utilizzo delle funzionalità Remote Procedure Call (RPC).

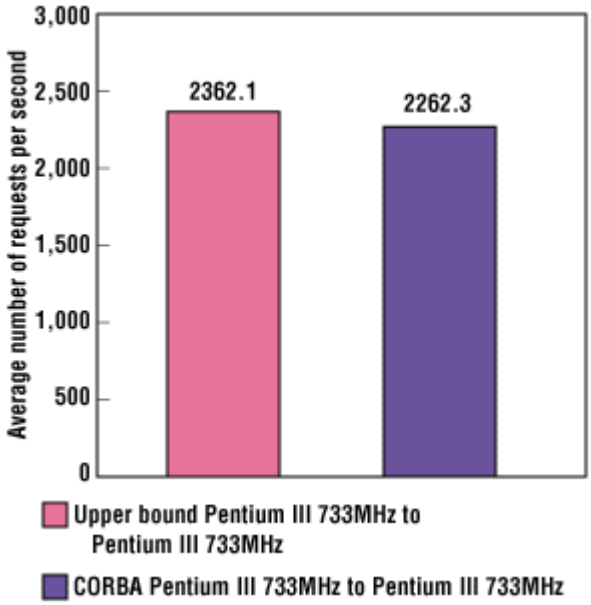

Figura 3.5 Un altro test per la misura dell'overhead

Capitolo 4
Un middleware a plug-ins

4.1 Introduzione

La proliferazione dei dispositivi portatili, accompagnata dalla diffusione delle reti wireless, offre nuove possibilità agli sviluppatori ed agli utenti di applicazioni e servizi. Con Internet, è possibile accedere a grandi collezioni di dati in ogni istante, ovunque e praticamente da ogni dispositivo.

Oltre al modello client-server, che sfrutta solo una parte della potenzialità di Internet, i modelli distribuiti peer-to-peer presentano tutte le entità partecipanti all'ambiente distribuito come client e server allo stesso tempo, fornendo quindi un modello interattivo bidirezionale. La chiave in questo paradigma è rappresentata dalle piattaforme middleware, che nascondono la complessità associata alla locazione dei servizi.

Il modello ad oggetti distribuiti è basato sul modello peer-to-peer ed estende il tradizionale modello di programmazione object-oriented, permettendo un'interazione con oggetti locali e remoti trasparentemente.

Esistono parecchi middleware che permettono lo sviluppo di applicazioni basate su oggetti distribuiti (come CORBA, Java RMI e DCOM). Però, la maggior parte delle implementazioni standard sono realizzate per desktop e non sono appropriate per ambienti eterogenei, popolati cioè da dispositivi differenti, come PDA, telefonini di terza generazione e PocketPC.

Quindi i programmatori che sviluppano applicazioni per tali dispositivi sono forzati ad implementare soluzioni personalizzate, perdendo l'abilità di interoperare con applicazioni distribuite che funzionano su Internet e di esportare le loro funzionalità per le altre applicazioni.

Si rende dunque necessario un middleware adattativo, in modo da supportare l'interoperabilità di client mobili con servizi eterogenei. Per far ciò, questo middleware deve essere multipiattaforma, ossia deve funzionare su più dispositivi con sistemi operativi diversi, e multilinguaggio, ovvero deve permette agli sviluppatori di programmi applicativi di poter usare qualunque linguaggio di programmazione. Usando questo approccio, il middleware dovrebbe alterare il suo comportamento dinamicamente per trovare i servizi mobili richiesti ed interagire con i servizi implementati da diversi tipi di middleware. Una tecnica adatta a sviluppare middleware con queste capacità è la *reflection*. Essa è un metodo di principio che supporta l'*introspection* e l'*adaptation* per produrre middleware configurabile e riconfigurabile.

Affinché un'applicazione funzioni dinamicamente usando implementazioni di middleware differenti deve essere programmata indipendentemente da loro. Quindi, è richiesta una definizione astratta delle funzionalità dei servizi, che possono divenire noti ad un'applicazione client attraverso un particolare servizio chiamato discovery service [34].

L'applicazione client mobile, che richiede un servizio conosciuto tramite quest'ultimo, può essere poi sviluppata usando questa definizione, detta interfaccia; in particolare, come linguaggio per la definizione di quest'ultima, è stato scelto l'IDL dell'OMG, dal momento che è un linguaggio standard e semplice, che permette di svincolare l'interfaccia di un servizio dalla sua effettiva implementazione. Una richiesta di un servizio astratto, poi, è mappata a run-time alla cor-

rispondente richiesta concreta appartenente alla particolare implementazione del middleware.

A partire da UIC (Universally Interoperable Core) è stato possibile realizzare un middleware composto da più componenti, in particolare da un modulo e da alcuni plug-ins. Grazie a questi un programmatore può comunicare con un generico middleware astraendosi da quest'ultimo, ossia può scrivere il suo programma che esegue invocazioni a metodi appartenenti ad oggetti remoti senza conoscere il particolare protocollo di comunicazione e dunque il particolare ORB (Object Request Broker).

Ciò avviene attraverso un'interfaccia che mette a disposizione del programmatore delle funzionalità astratte che vengono poi implementate dal plug-in del particolare middleware. Queste funzionalità sono quelle classiche di lookup all'oggetto remoto, marshalling e demarshalling dei parametri, invocazione di un metodo remoto, ecc.

Il middleware, dunque, si specializzerà a run-time sul particolare middleware con il quale si intende comunicare, caricando il giusto plug-in che incapsula le funzionalità implementate appunto per quel middleware. Si osservi come la gestione delle risorse sia dinamica, dal momento che i plug-ins saranno caricati e scaricati a run-time, allocando e deallocando, rispettivamente, spazio in memoria.

Un'altra possibilità che il middleware mette a disposizione consiste nell'integrazione di altri middleware presenti sul dispositivo. Se infatti quest'ultimo possiede delle librerie che permettono di comunicare con un middleware attraverso un particolare protocollo di comunicazione, è possibile implementare un plug-in che utilizzi tali librerie native, rendendo così possibile l'interazione con il middleware e assicurando flessibilità alla piattaforma.

Tutto ciò che il programmatore deve conoscere per comunicare con un particolare oggetto è il suo URI (Universal Resource Identifier), con il quale sarà possibile effettuare il lookup, e l'interfaccia

IDL del servizio, in modo da poter invocare correttamente il metodo remoto.

Attualmente il nostro middleware può comunicare con due diversi middleware molto usati: CORBA (Common Object Request Broker Architecture) e Java RMI (Request Method Invocation). In futuro è comunque prevista l'aggiunta di un terzo protocollo di comunicazione: SOAP (Simple Object Access Protocol).

4.2 Uno scenario

In Figura 4.1 è presentato un possibile scenario che illustra come è possibile sfruttare le potenzialità del nostro middleware. Esso è costituito da una varietà di dispositivi differenti che intendono comunicare con altri dispositivi che mettono a disposizione dei servizi.

In particolare, vi sono due PocketPC ed un laptop che comunicano attraverso una rete wireless con due server, uno CORBA su PocketPC ed uno RMI su laptop.

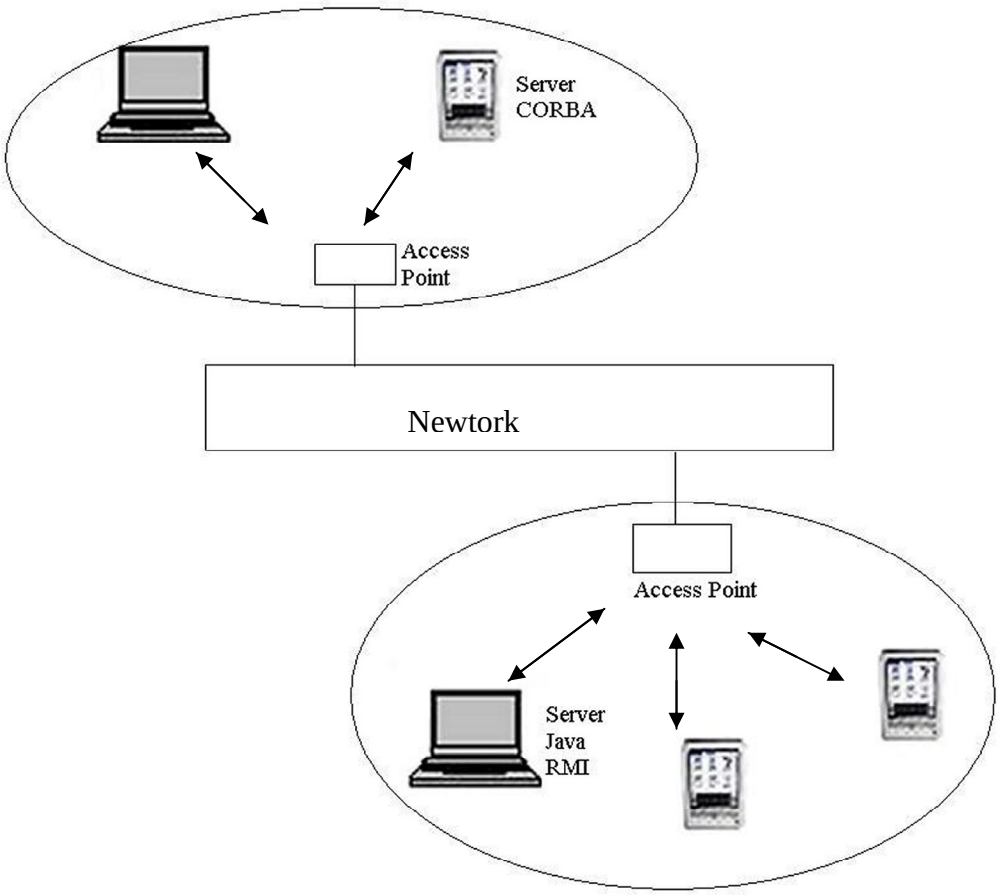

Figura 2.1 Un possibile scenario

Quella dei dispositivi non è comunque l'unico tipo di eterogeneità, dal momento in cui i servizi stessi sono messi a disposizione da middleware diversi che utilizzando anche protocolli di comunicazione differenti. In questo scenario, infatti, sono presenti due server diversi: uno CORBA ed uno RMI. Sono inoltre presenti dei dispositivi client che utilizzano i servizi messi a disposizione dai server sfruttando le potenzialità del nostro middleware, che provvederà a caricare i plug-ins corretti.

Dati dunque scenari di questo tipo, è necessario che le piattaforme middleware siano riconfigurabili per interagire con differenti tipi di middleware. Ciò permette lo sviluppo di applicazioni mobili indipendentemente dal fissato tipo di piattaforma, le cui proprietà non sono note al programmatore dell'applicazione a tempo di progetto.

4.3 Architettura del middleware

L'architettura del middleware realizzato è suddivisa in più componenti, come si può osservare dal Component Diagram di Figura 4.2 realizzato secondo lo standard UML (Unified Modeling Language) [35], un linguaggio descrittivo di progettazione definito dall'OMG (Object Management Group) [21], in particolare da un'interfaccia di comunicazione, da un modulo e dai cosiddetti Middleware Plug-ins, i quali possono a loro volta utilizzare alcune librerie.

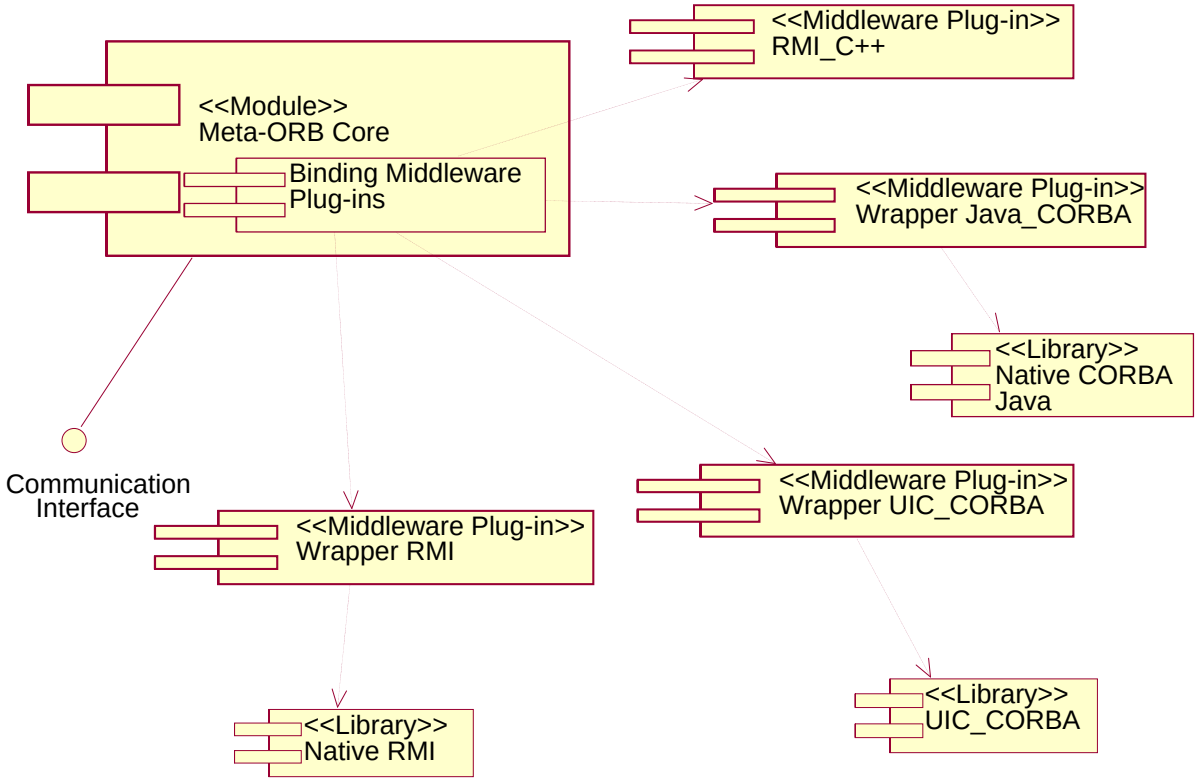

Figura 4.2 Component Diagram del middleware

4.3.1 L'interfaccia di comunicazione

Un programmatore usa le funzionalità che il middleware gli mette a disposizione attraverso la Communication Interface, che in pratica fornisce diversi metodi per la comunicazione con un generico middleware.

In particolare, conoscendo l'URI e l'IDL di un servizio, l'utente può invocare metodi appartenenti ad oggetti remoti senza sapere con quale linguaggio di programmazione questi siano stati istanziati e senza conoscere il paradigma di comunicazione.

In Figura 4.3 è riportata l'interfaccia; come vediamo essa consta di due funzionalità: una, chiamata *lookup*, che permette di reperire il riferimento di un oggetto dato il suo URI, ed un'altra, *sendRe-*

quest, che invece invia una richiesta di invocazione di un metodo remoto in base al riferimento dell'oggetto, al nome del metodo ed ai parametri di ingresso/uscita specificati.

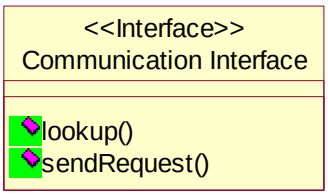

Figura 4.3 Interfaccia di comunicazione

4.3.2 Il modulo Meta-ORB Core

Il modulo Meta-ORB Core costituisce l'architettura che permette al programmatore di astrarsi dal particolare middleware con il quale vuole comunicare. Esso realizza varie funzionalità astratte, in particolare:

- *lookup*: fornendo in ingresso l'URI dell'oggetto remoto, è possibile reperire il riferimento a quest'ultimo, necessario per poter effettuare successivamente invocazioni ai suoi metodi;
- *invocazione di un metodo remoto:* con la conoscenza dell'IDL del servizio è possibile effettuare chiamate a funzioni appartenenti all'oggetto remoto;
- *marshalling* e *demarshalling*: i tipi dei parametri effettivi passati per l'invocazione remota di un metodo vengono mappati (marshalling) con i tipi propri del middleware riportati in Figura 4.4; avviene poi l'esatto contrario dopo la trasmissione dei parametri e del nome del metodo (demarshalling).

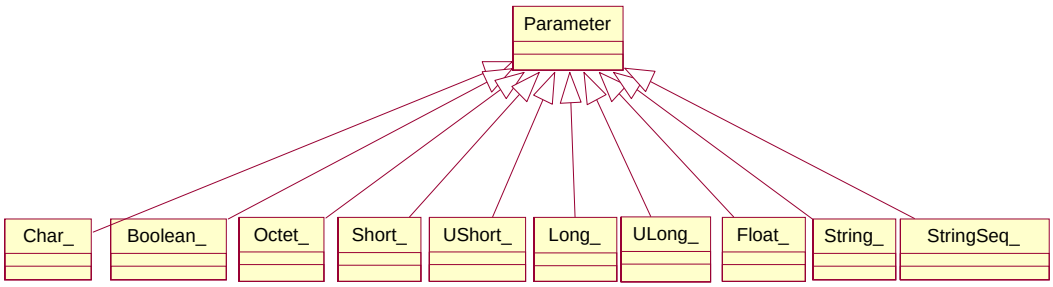

Figura 4.4 Tipi di dato del middleware

Il Meta-ORB Core realizza anche l'IDL to Language Mapping, ossia il mappaggio dei tipi dell'IDL con quelli del particolare linguaggio di programmazione utilizzato dal programmatore. Ciò è necessario dal momento in cui a quest'ultimo è richiesta solo la conoscenza dell'IDL del servizio e non del linguaggio di programmazione con il quale esso è stato implementato; tutto questo rende il middleware flessibile ed assicura una totale trasparenza al client di ciò che avviene all'interno di esso.

Dunque, quando un programma client invoca un metodo appartenente ad un oggetto istanziato da un server remoto, vengono svolti, in ordine temporale, i seguenti passi:

1. si effettua il lookup all'oggetto con la conoscenza del suo URI, ottenendo il riferimento ad esso;
2. con la conoscenza dell'interfaccia IDL del servizio, avviene l'invocazione del metodo remoto;
3. i tipi IDL dei parametri effettivi vengono mappati in quelli del tipico linguaggio di programmazione;
4. prima della trasmissione verso il server avviene il marshalling dei tipi dei parametri effettivi;
5. dopo la trasmissione, lato server, avviene il demarshalling dei tipi dei parametri;
6. dal server ha luogo il marshalling dei tipi dei parametri di ritorno;

7. si verifica, lato client, il demarhsalling dei tipi dei parametri di ritorno.

Un componente di fondamentale importanza del Meta-ORB Core è il Binding Middleware Plug-ins.

4.3.3 Il componente Binding Middleware Plug-ins

La funzione principale del componente Binding Middleware Plug-ins è quella di interoperare con servizi mobili eterogenei. Esso può essere configurato come un client IIOP per fare un certo numero di richieste IIOP o cambiare verso un'altra configurazione ed aspettare di ricevere eventi di interesse. Differenti paradigmi di middleware, sincroni o asincroni, possono essere incapsulati nel Binding Middleware Plug-ins se sono stati implementati.

Il Binding Middleware Plug-ins gestisce dunque il binding dei diversi componenti del middleware, i cosiddetti Middleware Plug-ins, in modo da caricare e scaricare dinamicamente a run-time i vari plug-ins, a secondo del middleware con cui comunicare.

In questo modo il middleware riesce a comunicare con altri middleware in modo del tutto trasparente al client, con flessibilità e leggerezza; infatti il Binding Middleware Plug-ins, all'atto di una richiesta di interazione con un oggetto remoto, caricherà il plug-in necessario a comunicare con il particolare middleware, che verrà poi deallocato quando non serve più, garantendo quindi una gestione delle risorse efficiente.

Dunque, il programmatore non deve conoscere con quale middleware dovrà interagire, né il linguaggio di programmazione con cui è stato istanziato l'oggetto remoto, dal momento che userà solo le funzionalità ad alto livello che poi saranno particolarizzate nello specifico plug-in caricato a run-time.

4.3.4 I Middleware Plug-ins

Un Middleware Plug-in è un componente software caricato a run-time che implementa le funzionalità astratte appartenenti al modulo Meta-ORB Core (lookup, marshalling, demarshalling, invocazione di un metodo remoto,…).

In particolare i plug-ins possono implementare queste funzionalità direttamente o appoggiarsi ad altre librerie, fungendo in tal modo da wrapper.

Attualmente sono esistenti un Middleware Plug-in che permette la comunicazione con CORBA che usa una libreria chiamata UIC_CORBA ed un Middleware Plug-in che invece implementa direttamente le funzionalità astratte e che permette la comunicazione con Java RMI, tra l'altro realizzato in C++.

Un'altra strada significativa è quella di usare librerie native di un particolare linguaggio di programmazione: è cioè possibile creare un Middleware Plug-in, che funga da wrapper, capace di utilizzare funzionalità proprie del tipico protocollo di comunicazione, come ad esempio Java RMI; basta creare metodi che invochino funzioni come il lookup e l'invocazione dinamica di un metodo già implementati nel package RMI della JDK. Questa possibilità è molto importante, dal momento in cui permette di colloquiare, ad esempio, con un server Java attraverso RMI in maniera molto efficiente, dal momento in cui il plug-in da caricare è molto piccolo. D'altra parte, questa scelta è possibile solo per sistemi che hanno la JVM installata e dunque non è fattibile per piccoli dispositivi portatili che invece ne sono privi. Lo stesso dicasi per quanto riguarda l'utilizzo delle librerie Java per CORBA.

4.4 Un gestore degli ORB

Le funzionalità fondamentali che il middleware mette a disposizione e che il programmatore deve usare sono due:

- *lookup*, che permette di ottenere il riferimento dell'oggetto remoto in base all'URI specificato;
- *sendRequest*, che invece invoca un metodo appartenente ad un oggetto remoto; i parametri sono, oltre al riferimento dell'oggetto remoto, il nome del metodo, i parametri di ingresso e quelli di uscita.

Dal momento in cui un generico programma client può fare un lookup ad un oggetto attraverso un qualsiasi ORB, questa funzionalità deve essere la più generica possibile, in modo che al client non sia richiesta la conoscenza del particolare ORB. Sarà poi la stessa funzionalità ad eseguire il lookup verso il giusto ORB.

L'individuazione del particolare ORB può avvenire eseguendo un parsing dell'URI, dal momento che la prima parte di esso è standard per i vari protocolli di comunicazione: ad esempio per quanto riguarda RMI esso incomincia con la stringa "*rmi://*", a cui segue l'indirizzo IP della macchina, il numero di porto del naming service (è stato usato *rmiregistry* proprio di Java, il cui numero di porto è 1099) ed il nome del servizio; per quanto riguarda invece IIOP, l'URI incomincia con "*IOR::*", che sta per Interoperable Object Reference, seguito da altri numerosi campi.

Inoltre si rende necessario registrare in qualche modo i riferimenti degli oggetti reperiti in una tabella dinamica, accompagnati dal tipo dell'ORB a cui essi fanno parte, in modo tale che, alla prossima eventuale invocazione di un metodo, possa essere facilmente individuato il giusto ORB a cui fare la richiesta. Quello che si viene dunque a creare è un *gestore degli ORB*.

Quanto detto è stato formalizzato nell'Use Case Diagram di Figura 4.6.

In particolare, dopo il lookup, avverrà un'*introspection*, consistente in un parsing dell'URI dell'oggetto, che permetterà l'individuazione dell'ORB con cui si dovrà comunicare; dopo aver reperito il riferimento dell'oggetto, questo verrà memorizzato con il tipo dell'ORB nella tabella attraverso il metodo *write_table*, a cui seguirà l'effettivo lookup verso l'oggetto remoto mediante il metodo *lookupORB*.

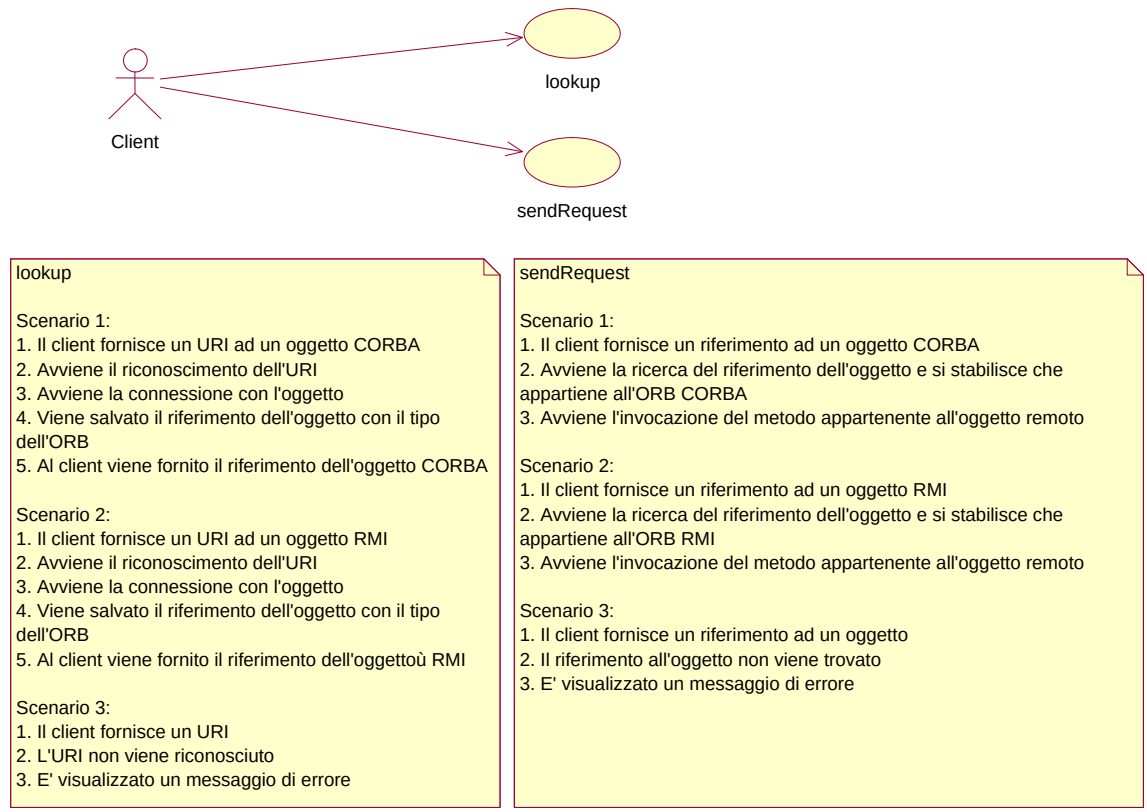

Figura 4.5 Use Case Diagram del gestore degli ORB

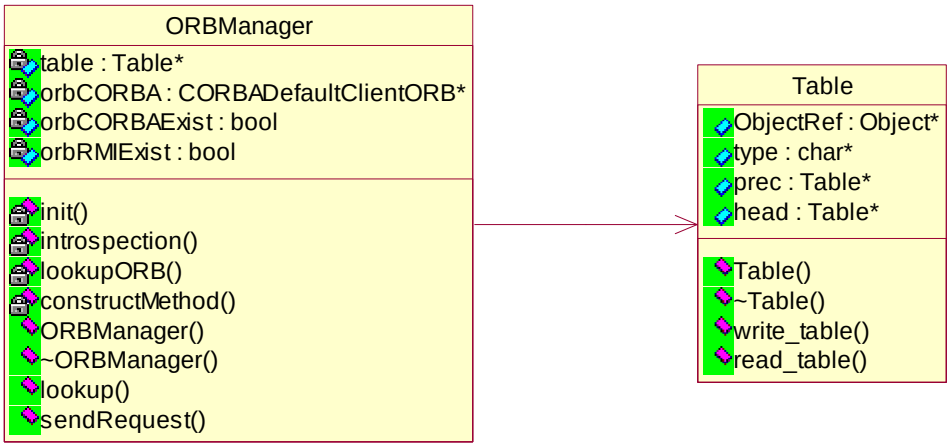

Figura 4.6 Class Diagram del gestore degli ORB

La tabella è gestita come una pila dinamica; la classe *Table* presenta dunque, oltre al riferimento dell'oggetto (*ObjectRef*) e al tipo dell'ORB (*type*), anche un puntatore all'elemento precedente (*prec*) ed uno alla testa della pila (*head*).

Quando verrà effettuata l'invocazione al metodo remoto, sarà innanzitutto compiuta una lettura nella tabella attraverso il metodo *read_table* per riconoscere il tipo dell'ORB in base al riferimento specificato, dopodiché sarà effettuato un mapping dei tipi IDL contenuti nella stringa costituente la firma del metodo in quelli del giusto linguaggio di programmazione attraverso il metodo *constructMethod*, che costruisce dunque il messaggio di richiesta. A questo punto avverrà l'effettiva richiesta che invocherà il metodo remoto.

Ovviamente, sia per quanto riguarda il lookup, che per l'invio della richiesta remota, sono previsti eventuali messaggi di errore nel caso in cui il sistema non dovesse funzionare correttamente. Ad esempio tipici errori sono:

- eseguire il lookup specificando un URI che non sia compliant con gli URI standard di Java RMI o di IIOP;
- errata inizializzazione di un ORB;
- sintassi di un metodo errata;

- invio di una richiesta di invocazione ad un metodo remoto specificando un riferimento ad un oggetto non presente nella tabella dinamica.

Si osservi che la classe *ORBManager* istanzierà il particolare ORB lato client al primo lookup verso un oggetto di quel middleware; esso poi non sarà deallocato immediatamente, in modo da non dover riallocare l'ORB ad una successiva eventuale richiesta. Gli attributi *orbCORBAExist* e *orbRMIExist* indicano se gli ORB sono stati già istanziati o meno.

Attualmente la classe *ORBManager* può istanziare due tipi di ORB, CORBA e Java RMI, anche se sono previste nuove aggiunte, come ad esempio SOAP.

L'andamento temporale delle varie operazioni che vengono effettuate tra il programma client e la classe *ORBManager*, che parte dalla prima richiesta di lookup ad un oggetto con la rispettiva allocazione dell'ORB CORBA o RMI e che è intervallato da richieste di invocazioni a metodi appartenenti ad oggetti remoti, è mostrato nel Sequence Diagram di Figura 4.7. Da esso si può osservare come la classe *ORBManager* intercetti le richieste fatte dal programma client e le rediriga verso il giusto ORB, individuato attraverso un'introspezione e l'utilizzo della tabella dinamica.

In questo modo è possibile comunicare con un particolare middleware in modo del tutto trasparente al client.

Figura 4.7 Sequence Diagram del gestore degli ORB

4.5 Costi: dimensioni e prestazioni

Attualmente i dispositivi mobili hanno un ammontare limitato di memoria di sistema, che può essere velocemente consumato da applicazioni utente; quindi è importante minimizzare lo spazio di memoria necessario a memorizzare l'implementazione del middleware. Utilizzando la reflection per "montare" e "smontare" i vari protocolli, è possibile utilizzare solo il numero minimo richiesto di componenti che devono essere memorizzati nel dispositivo, piuttosto che salvare le implementazioni complete multi-middleware.

In futuro memorizzare i componenti sui dispositivi sarà probabilmente più facile, dal momento in cui i dispositivi mobili avranno molta più capacità di memoria disponibile. Comunque, i componenti avranno ancora bisogno di essere trasmessi attraverso la rete (ad esempio, quando la piattaforma scopre di aver bisogno di componenti non correntemente installati sul dispositivo). Quindi, l'implementazione dei vari componenti dovrà essere ancora minimizzata.

La dimensione attuale dei vari componenti del middleware, ossia del modulo Meta-ORB Core e dei plug-ins per CORBA e per RMI, insieme alle librerie, sono riportati in Tabella 4.1; come si può notare, mentre per quanto riguarda i calcolatori desktop con sistema operativo Windows 2000 essi raggiungono anche l'ordine delle centinaia di KB, per quanto riguarda i dispositivi portatili con sistema operativo PocketPC 2002 questi diminuiscono più della metà, permettendo così un uso efficiente delle risorse.

	Windows 2000 su x86	**PocketPC 2002 su ARM**
Modulo Meta-ORB Core	42 KB	20 KB
Middleware Plug-in + Library UIC_CORBA	114 KB	34.5 KB
Middleware Plug-in + Library RMI_C++	154 KB	48.5 KB

Tabella 4.1 Dimensione dei componenti del middleware

Da un punto di vista prestazionale, sono stati effettuati dei test invocando dei metodi remoti più volte, sia utilizzando RMI che IIOP.

Ciò che viene alla luce, è che l'invocazione di un metodo avviene più velocemente nelle successive richieste, dato che i rispettivi plug-ins sono stati già caricati alla prima richiesta. Questo meccanismo permette dunque di risparmiare molto tempo, come si può osservare dalla Tabella 4.2 che mostra i risultati dei test realizzati, consistenti nell'invocare due volte un metodo appartenente ad un oggetto CORBA ed uno appartenente ad un oggetto RMI, con lo scopo di visualizzare una semplice stringa.

	Prima invocazione	**Seconda invocazione**
CORBA	3 ms	1 ms
RMI	5 ms	2 ms

Tabella 4.2 Invocazioni di metodi remoti

4.6 Un caso di studio

Il nostro middleware è stata testato, oltre che con applicazioni semplici come il passaggio di parametri dei tipi primitivi, anche con un caso di studio più complesso: un image server distribuito. In particolare, sono stati sviluppati tre principali componenti, due server image ed un client image, come mostrato nel Component Diagram di Figura 4.8.

Gli image server utilizzano piattaforme CORBA e RMI, anche se il server CORBA comunica utilizzando la libreria UIC_CORBA. Il client image è realizzato in C++ e può interagire con object server Java RMI e CORBA attraverso il modulo Meta-ORB Core, il quale, utilizzando il componente Binding Middleware Plug-ins, provvede a caricare il giusto Middleware Plug-in.

Al momento, il client deve conoscere gli esatti URI degli oggetti e la loro interfaccia IDL. Gli oggetti remoti su server espongono alcune funzionalità per navigare in un database statico e visualizzare le immagini in esso contenute.

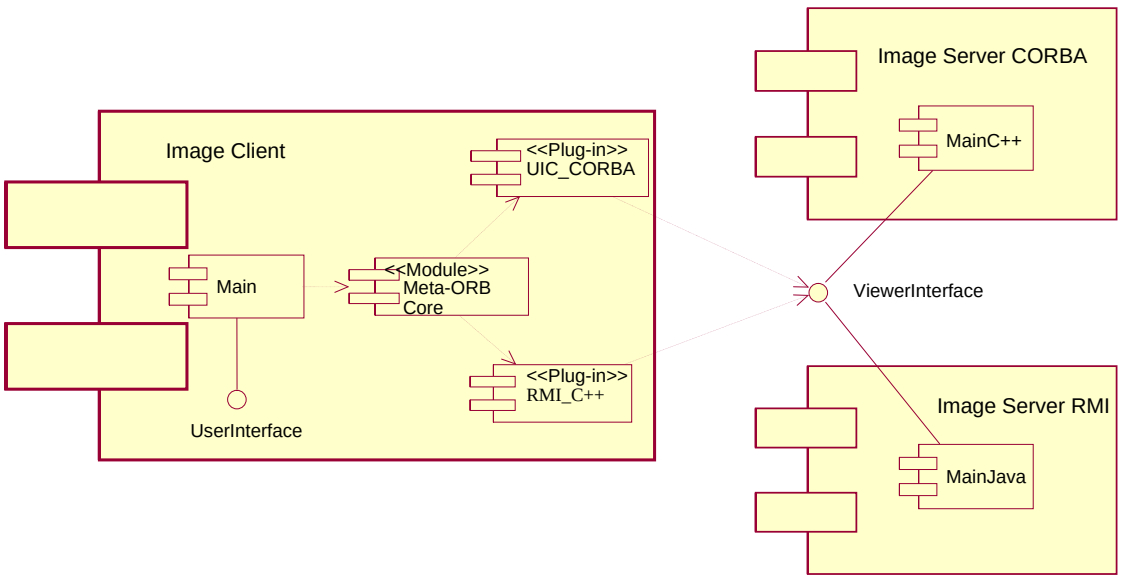

Figura 4.8 Il Component Diagram del caso di studio

4.6.1 requisiti

La possibilità da parte di un utente di poter visualizzare un'immagine scelta in una lista distribuita tra server sulle due piattaforme middleware CORBA e Java RMI rappresenta il requisito principale di questa specifica applicazione distribuita. Per rispettare un modello di comunicazione uniforme per il client, è opportuno che i metodi pubblicati nell'interfaccia dai due server siano gli stessi. Al contrario di quanto accade nel caso si utilizzi una sola piattaforma middleware, le interfacce potrebbero risultare diverse. Infatti, per il server Java si utilizza un'interfaccia Java, mentre per il server CORBA esiste l'interfaccia IDL. Se ci si uniforma sui parametri d'ingresso e d'uscita, si ha il limite imposto dalla piattaforma Java di poter avere come parametri di output solo il return type del metodo. Di conseguenza, la progettazione dell'image server distribuito seguirà questo vincolo.

In Figura 4.9 è riportato lo schema dell'Use Case del nostro prototipo.

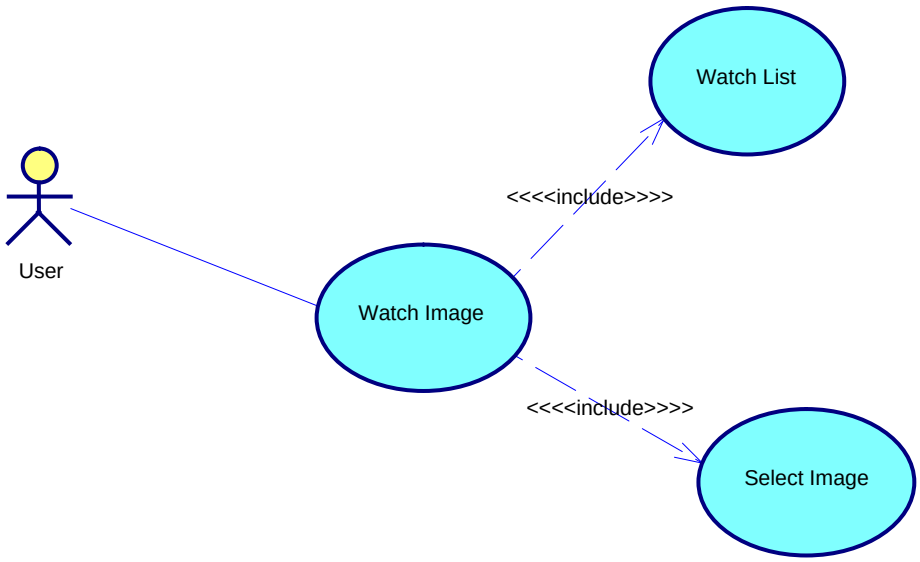

Figura 4.9 Use Case dell'image server distribuito

L'Use Case *Watch Image* è realizzato attraverso la visualizzazione della lista (*Watch List)* e la scelta dell'immagine (*Select Image*).

Watch List mette a disposizione dell'utente una lista di nomi delle immagini. Alla creazione di tale elenco possono essere inclusi nomi provenienti da eventuali sorgenti differenti.

La selezione dell'immagine da visualizzare appartenente alla lista di nomi avviene attraverso l'Use Case *Select Image*. La conferma alla selezione effettuata comporta la visualizzazione dell'immagine (*Watch Image*). E' chiaro che la selezione dell'immagine può avvenire solo a lista visualizzata. L'Use Case *Watch List* assume il significato di aggiornamento lista a partire dalla seconda volta che viene richiesta, così da poter effettuare la scelta anche tra eventuali nuove immagini messe a disposizione.

4.6.2 Class Diagram

In una prima astrazione ad alto livello si individuano due elementi fondamentali per la realizzazione dell'esempio d'uso che risponde ai requisiti suddetti: un gestore delle immagini che permette di assolvere ai casi d'uso *Watch Image* e *Watch List* e un navigatore per la gestione delle immagini che realizza l'Use Case *Select Image*. Il navigatore può essere collegato a più gestori d'immagini.

Prima di passare alla descrizione delle classi progettate per la realizzazione dell'image server distribuito, è necessario analizzare alcune caratteristiche della gestione dei tipi del middleware sviluppato.

Nell'architettura del middleware non si includono le regole di codifica per i tipi complessi di dati (per esempio struct, union o sequence, se non per le stringhe). La gestione di queste è lasciata alle applicazioni in accordo con i loro requisiti. Questo meccanismo può essere visto come una limitazione, ma la strategia di UIC e quindi

del nostro middleware prevede di salvare spazio importante. Lo standard ORB include per default meccanismi di marshal e demarshal di arbitrari tipi complessi, ma ciò contrasta con la filosofia di UIC del 'What You Need Is What You Get'. L'applicazione che ha bisogno di utilizzare questi meccanismi di codifica può scegliere di adattarsi ad utilizzare i tipi primitivi o utilizzare una libreria dinamica che realizza tale funzionalità. Essendo lo scopo di questo caso di studio quello di testare il middleware, è stato preferito utilizzare i tipi primitivi già codificati nell'architettura.

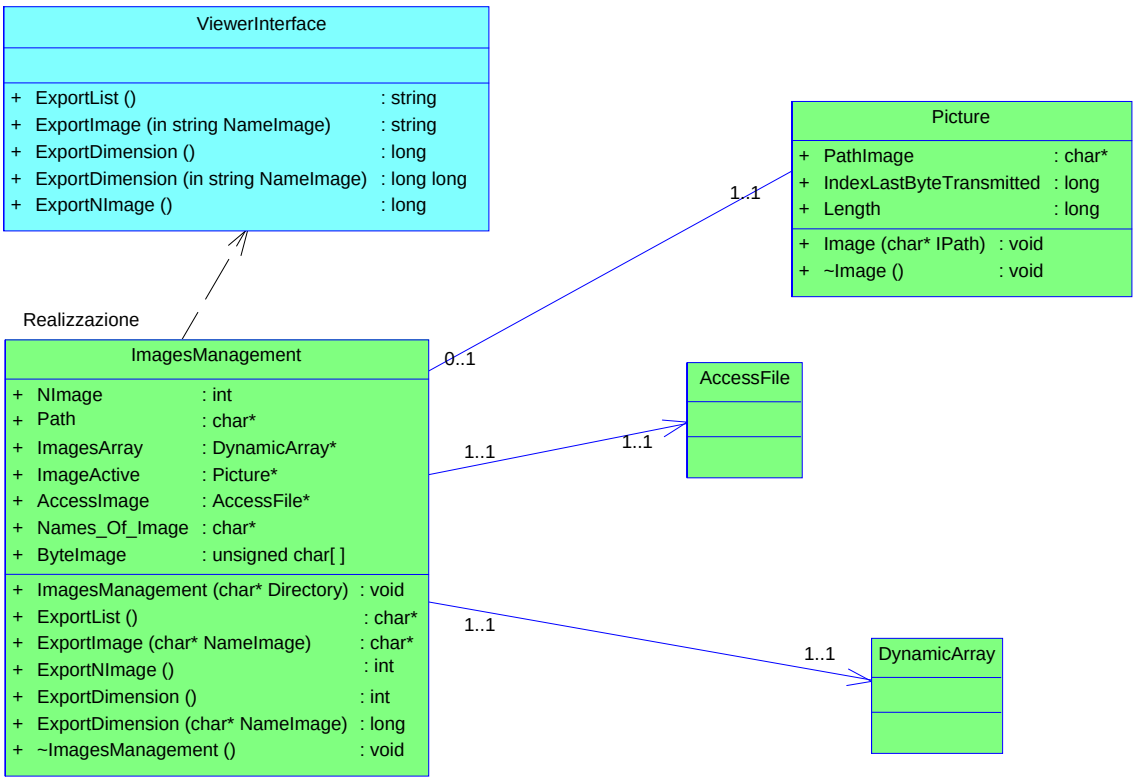

Figura 4.10 Class Diagram per la gestione delle immagini

Ciò ha comportato delle scelte implementative non convenzionali per un gestore d'immagini distribuito che si possono osservare nel Class Diagram in Figura 4.10. Questa scelta ha creato anche alcuni

problemi nella trasmissione dell'immagine dai server al client: infatti, per quanto riguarda CORBA, dal momento in cui il server è stato realizzato in C++, la stringa trasmessa terminava inizialmente al primo carattere nullo '\0'; anche per quanto riguarda RMI sorgevano dei problemi, dal momento in cui il server è stato realizzato in Java, che supporta UNICODE come codifica di rappresentazione dei caratteri in una stringa, ed il client in C++, che invece usa ASCII.

Questi problemi sono stati risolti facendo in modo che il client ricostruisca l'immagine correttamente, inserendo i vari terminatori nulli nelle corrette posizioni, nel caso di CORBA, e trasformando i byte tipo UNICODE in quelli tipo ASCII, nel caso di RMI.

Nell'interfaccia sono utilizzati solo tipi semplici come parametri di output ed input per i metodi definiti. Per la mancata realizzazione di una codifica dei tipi complessi verrà utilizzato il tipo *string* per trasmettere sia la sequenza di byte che quella di stringhe, attraverso i metodi *ExportList* ed *ExportImage*. La stringa di ritorno del metodo *ExportList* contiene la concatenazione di tutti i nomi di file d'immagini della lista gestita dalla classe *ImagesManagement*. Il metodo *ExportImage* restituisce una stringa nella quale ogni carattere rappresenta la conversione di un byte, appartenente al file immagine, nel tipo *char* (che comporrà la stringa). La sequenza di byte che compongono il file corrispondente al nome scelto dall'utente viene convertita in una sequenza di caratteri nel return type di *ExportImage*.

La classe *ImagesManagement* costituisce il cosiddetto *servant*, ovvero l'effettiva implementazione degli object server.

La classe *Picture* viene istanziata dal metodo *ExportImage* o *ExportDimension(NameImage)*. L'oggetto *Picture* gestisce e mantiene informazioni sull'immagine esportata e in uno scenario futuro potrebbe essere interrogato qualora si voglia prevedere funzionalità aggiuntive per l'utente, realizzando, magari, un'interfaccia anche per l'immagine.

Le classi *AccessFile* e *DynamicArray* non sono ulteriormente specificate in quanto rappresentano concetti sicuramente implementati da classi della libreria standard di un linguaggio di programmazione.

Tutte le classi dello schema in Figura 4.10 sono state realizzate in linguaggio Java e C++.

La gestione della navigazione delle immagini è realizzata attraverso le classi rappresentate nel Class Diagram di Figura 4.11.

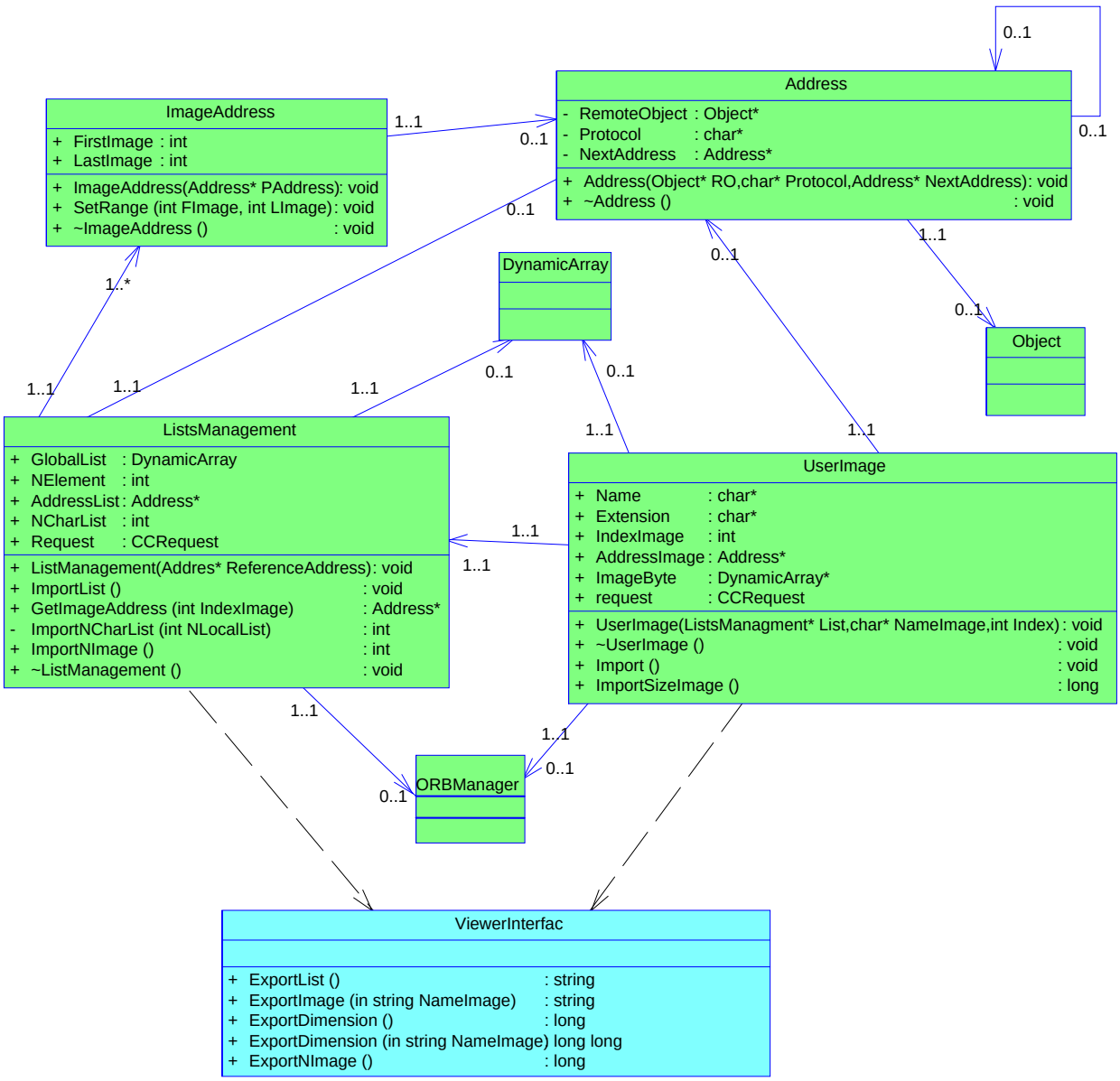

Figura 4.11 Class Diagram per la gestione della navigazione delle immagini

La classe *ListsManagement* gestisce l'insieme delle liste che offrono immagini all'utente: il metodo *ImportList* crea la lista globale

GlobalList che è un vettore dinamico di stringhe contenente i nomi dei file immagine disponibili all'utente. Questi file possono provenire da più directory, ognuna delle quali gestita dalla classe *ImagesManagement*.

Il metodo *ImportList* chiama tante volte il metodo *ExportList* dell'interfaccia quanti sono gli oggetti *ImagesManagement* che partecipano alla applicazione (nel caso specifico gli oggetti sono due: uno su server CORBA e l'altro su server RMI).

Le informazioni per accedere ad un oggetto *ImagesManagement* vengono prelevati dalla lista degli indirizzi *AddressList*. Questa lista è creata dal client e passata all'unico costruttore di *ListsManagement*.

La creazione di questa lista da parte del client avviene attraverso le fasi descritte nel Sequence Diagram di Figura 4.7. Se quindi il client interagisce con più object server, così come accade in questo applicativo, costruirà la sua lista di oggetti *Address*.

Come detto più volte, il client, per accedere al servizio, dovrà conoscere a priori l'URI e l'interfaccia IDL del servizio, necessari per raggiungere l'object server.

La classe *UserImage* gestisce l'immagine che l'utente desidera visualizzare. Il metodo *ImportImage* effettua la chiamata al metodo *ExportImage* dell'interfaccia. Per poter raggiungere l'oggetto *ImagesManagement* esatto, cioè quello che gestisce la directory contenente il file immagine richiesto dall'utente, si utilizzano le informazioni contenute nell'oggetto *Address* puntato da *AddressImage*, attributo della classe *UserImage*. Questo è settato opportunamente dal valore di ritorno del metodo *GetImageAddress* di *ListsManagement*. La classe *ImageAddress* serve a questo scopo.

La stringa ritornata dal metodo *ExportImage* va convertita carattere per carattere nella sequenza di byte che rappresenta l'immagine e che va a comporre l'attributo *ImageByte* di *UserImage*.

Della classe *ORBManager* si è già discusso, mentre la classe *Object* appartiene al middleware ed in essa vi è memorizzato l'object reference, insieme ad altre informazioni che lo riguardano.

4.6.3 Operare in uno scenario mobile

Per illustrare che il nostro middleware realizza la sua funzione primaria di interoperare con servizi eterogenei, consideriamo lo scenario di Figura 4.12.

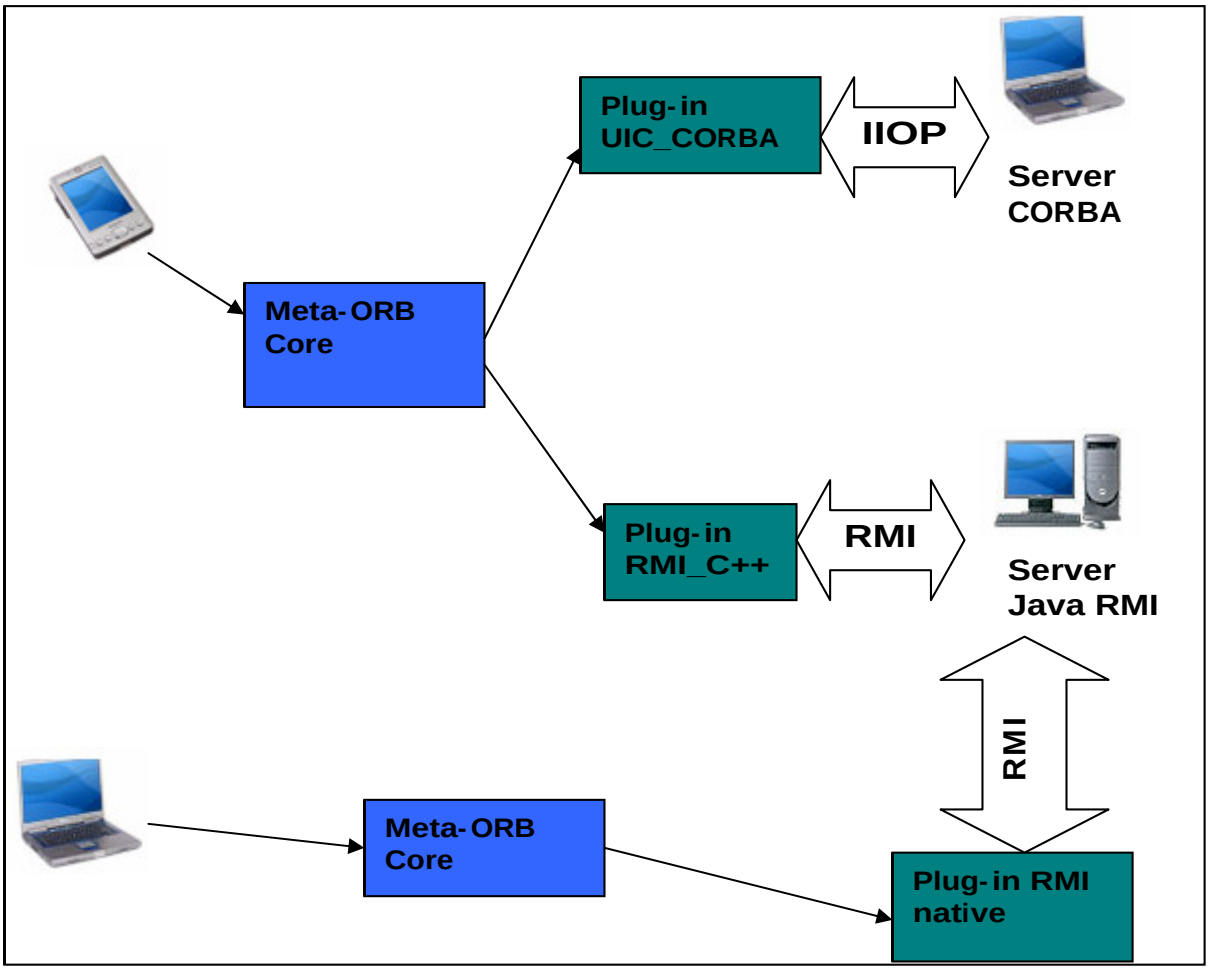

Figura 4.12 Uno scenario mobile

Il caso d'uso che abbiamo illustrato può essere provato in questo scenario, che include, lato client, un PocketPC con sistema operativo Microsoft PocketPC 2002 ed un laptop con sistema operativo Mi-

crosoft Windows 2000; i servizi CORBA, che chiaramente usano come protocollo di comunicazione IIOP, sono posti su un laptop, mentre quelli RMI sono implementati in Java e sono situati su un computer desktop.

I dispositivi client eseguiranno dunque un lookup ai servizi offerti dai server ottenendo i loro riferimenti e utilizzandoli poi per invocare i loro metodi con lo scopo di eseguire ad esempio il download delle immagini. Ciò avverrà in modo del tutto trasparente ai client, dal momento in cui sarà il Binding Middleware Plug-ins, contenuto nel modulo Meta-ORB Core, che adatterà dinamicamente il middleware.

Dalla figura si notano le potenzialità che il middleware ci mette a disposizione. Infatti essa mostra, tra le altre cose, un PocketPC che interagisce con il nostro middleware per comunicare con i server CORBA e Java RMI, in modo da utilizzare i servizi che questi ultimi gli mettono a disposizione. Il Meta-ORB Core, come si vede, caricherà opportunamente i giusti plug-ins, in particolare UIC_CORBA per quanto riguarda la comunicazione con il server CORBA, e RMI_C++ per quanto riguarda la comunicazione con il server Java. Quest'ultima scelta è forzata, dal momento in cui non esiste la JVM per un PocketPC. Riosservando la figura possiamo notare come un altro dispositivo client, il laptop, provvisto invece della JVM, possa interloquire, chiaramente attraverso il Meta-ORB Core, con il server Java utilizzando un plug-in che sfrutta una libreria nativa, che rende in tal modo la comunicazione più efficiente.

Questo esempio mette in luce come il middleware, oltre a garantire trasparenza al client di ciò che avviene ai livelli più bassi, in particolare non rendendo visibile il particolare protocollo di comunicazione che dovrà essere usato, si adatti anche all'ambiente che lo circonda, cercando di sfruttare al meglio le potenzialità che questo gli mette a disposizione.

Lo scenario mostrato indica dunque come sia possibile, conoscendo l'URI dei servizi e la loro interfaccia IDL, interoperare con loro attraverso differenti piattaforme adeguando appropriatamente il middleware.

Conclusioni e scenari futuri

In questa panoramica sullo scenario dell'ubiquitous computing e sulle problematiche da affrontare per la sua realizzazione, si intuisce quanto siano ambiziosi gli obiettivi da raggiungere per il soddisfacimento dei suoi requisiti.

Con lo sviluppo di un reflective middleware strutturato a plug-ins, si è concentrata l'attenzione sull'eterogeneità necessaria in un ambiente pervasivo. L'integrazione dei dispositivi con limitate risorse, l'interoperabilità tra essi su piattaforme middleware differenti (Java RMI e CORBA) in maniera trasparente ad un programma client sono traguardi tali da candidare il reflective middleware ed il suo approccio a supporto per l'eterogeneità di uno scenario ubiquitous.

Le successive linee di sviluppo sono innumerevoli, dall'integrazione di nuove piattaforme middleware, device e sistemi operativi con quale interoperare all'aggiunta di nuove funzionalità che un programmatore può utilizzare. Ad esempio, il prossimo passo potrebbe essere quello di aggiungere un plug-in che permetta di utilizzare il protocollo di comunicazione SOAP, rendendo in tal modo ancora più versatile il nostro middleware.

Maggior trasparenza si otterrebbe con l'utilizzo di un discovery service posizionato su un server dell'ambiente. Questo servizio base non solo si dovrebbe preoccupare di fornire ad un client l'URI del server object che vuole raggiungere, ma dovrebbe gestire una directory di servizi dinamicamente aggiornata da nuovi dispositivi che offrono servizi e da altri device che li ritirano uscendo dall'ambiente.

Se, invece, si è in qualche modo interessati al requisito di scalabilità ed a quello di adattabilità dei servizi ai differenti dispositivi, è necessario introdurre tecniche reflective per le varie applicazioni, affinché queste ultime si adattino alle risorse dei device (schermo a bassa risoluzione, tastiera ridotta, ecc.).

Restano, comunque, aperti numerosi campi di ricerca per il raggiungimento delle caratteristiche di ubiquitous computing.

Bibliografia

1. M. Satyanarayanan, *Pervasive Computing Vision and Challenges*, in the proc. of the IEEE Personal Comm. Vol. 6, n°8, August 2001, pp. 10-17.
2. G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman and D. Zukowski, *Challenges: An Application Model for Pervasive Computing*, in the proc. of the 6th ACM/IEEE Int. Conference on Mobile Computing and Networking, MOBICOM2000.
3. M. Weiser, *The Computer for the 21st Century*, Scientific Am., Sept., 1991, pp. 94-104; reprinted in IEEE Pervasive Computing, Jan.-Mar. 2002, pp. 19-25.
4. D. Saha, A. Mukherjee and S. Bandopadhyay, *Networking Infrastructure for Pervasive Computing: Enabling Technologies and Systems*, Kluwer Academic, 2002.
5. A. Misra, S. Das, A. McAuley and 3 K. Das, *Autoconfiguration, Registration, and Mobility Management for Pervasive Computing*, IEEE Personal Communications, Aug. 2001.
6. P. Eronen and P. Nikander, *Decentralized Jini Security*, Proc. Network and Distributed System Security 2001 (NDSS 2001), The Internet Soc., Reston, Va., 2001, pp. 161-172.
7. T. Kindberg and A. Fox, *System Software for Ubiquitous Computing*, Jan.-Mar. 2002 IEEE, pp.70-81.
8. M. Langheinrich, *Privacy by Design: Principles of Privacy-Aware Ubiquitous Systems*, Ubicomp 2001: Ubiquitous Computing, Lecture Notes in Computer Science, vol.2201, Spinger-Verlag, Berlin, 2001, pp. 273-291.

9. Lawrie Brown, *Mobile Code Security*, disponibile in http://www.adfa.edu.au/~lpb/papers/mcode96.html.
10. http://www.ovum.com/.
11. http://www.opsec.com.
12. http://www-3.ibm.com/pvc/tech/sap.shtml.
13. http://portolano.cs.washington.edu/.
14. M. Roman and R.H. Campbell, *Gaia: Enabling Active Spaces*, Proc. 9th ACM SIGOPS European Workshop, ACM Press, New York, 2000.
15. www.wilmaproject.org/objectives-it.html.
16. P. Bernstein, *Middleware: A Model for Distributed System Services*, Communications of the ACM, 39:2,February 1996, 86-98.
17. A. Campbell, G. Coulson and M. Kounavis, *Managing Complexity: Middleware Explained*, IT Professional,IEEE Computer Society, 1:5, September/October 1999, 22-28.
18. David E. Bakken, *Middleware*, disponibile in http://www.eecs.wsu.edu/~bakken/middleware-article-bakken.pdf.
19. N. Brown and C. Kindel, *Distributed Component Object Model Protocol DCOM/1.0*, 1998, http://www.microsoft.com/com (current 4 June 2001).
20. http://www.corba.org.
21. http://www.omg.org.
22. ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.pdf.
23. http://java.sun.com/products/jdk/rmi/index.html.
24. ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf.
25. G. Blair et al., *An Architecture for Next Generation Middleware*, Proc. Middleware 98, Lake District, England, Springer-Verlag, London, 1998.
26. F. Kon, G. Blair and R.H. Campbell, *Workshop on Reflective Middleware*, Proc. FIP/ACM Middleware2000, 2000.

27. G. Kiczales, J. des Rivières and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, Cambridge, Mass., 1991.
28. D. C. Schmidt and C. Cleeland, *Applying Patterns to Develop Extensible ORB Middleware*, IEEE Comm. Magazine, IEEE CS Press., Los Alamitos, Calif., vol. 37, no. 4, 1999, pp. 54–63.
29. http://choices.cs.uiuc.edu/2k/dynamicTAO.
30. M. Roman, F. Kon and R.H. Campbell, *Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case*, Proc. ICDCS99 Workshop on Middleware, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 122–127.
31. K. Scribner and M. C. Stiver, *Understanding SOAP: The Authoritative Solution*, SAMS, 2000.
32. http://www.ubi-core.com.
33. F. Kon and R.H. Campbell, *Dependence Management in Component-Based Distributed Systems*, IEEE Concurrency, vol. 8, no. 1, Jan.–Mar. 2000. pp. 26–36.
34. T. Kindberg and J. Barron, *A Web_Based Nomadic Computing System*, Computer Networks, vol. 35, no.4, Mar. 2001, pp. 443-456.
35. http://www.omg.org/uml/.

Ringraziamenti

Giunto al termine di questo lavoro e dei miei studi universitari, desidero esprimere la mia gratitudine a coloro che, in un modo o nell'altro, hanno contribuito alla loro realizzazione.

In primo luogo devo ringraziare il Prof. Antonio d'Acierno, che, disponibile come pochi, mi ha permesso di sviluppare questa tesi in maniera professionale e serena, grazie ai suoi suggerimenti ed alle sue esilaranti battute.

Un mio ringraziamento va anche alla struttura del CNR di San Giorgio a Cremano (NA). In questo ambiente devo ricordare il Prof. Giuseppe De Pietro ed in particolare l'Ing. Giuliano Gugliara, senza l'aiuto del quale questo lavoro sarebbe stato sicuramente più faticoso. Molte delle idee più avanzate sono venute da lui.

Desidero inoltre ringraziare i miei colleghi ed i miei amici che mi hanno sempre sostenuto ed hanno contribuito a migliorare la mia preparazione.

Un ringraziamento speciale va ad Emanuela, con la quale condivido la mia vita e le mie esperienze, per essermi stata vicina ed aver sempre creduto in me. Con lei ho trascorso la maggior parte del tempo dedicato agli studi che mi hanno permesso di superare molti esami di questo Ateneo, rendendoli in tal modo meno difficili. Grazie di cuore, anche alla sua famiglia.

Desidero infine ringraziare i miei genitori per avermi mantenuto agli studi durante questi anni e per aver avuto sempre fiducia in me. Da loro ho avuto l'educazione e l'affetto che ogni figlio desidererebbe avere. Vorrei ringraziare anche Giovanni, il miglior fratello che abbia mai conosciuto.