UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Un approccio innovativo per il *delivery* di servizi in infrastrutture di *nomadic computing*

RELATORE
Chiar.mo Prof. Ing. Stefano Russo

CORRELATORE
Ing. Domenico Cotroneo

CANDIDATO
Armando Migliaccio
Matricola 41/2784

ANNO ACCADEMICO 2002-2003

Un approccio innovativo per il *delivery* di servizi in infrastrutture di *nomadic computing*

Indice

Introduzione

Negli ultimi anni lo sviluppo delle applicazioni distribuite avviene sempre di più nell'ambito delle architetture orientate ai servizi (*Service Oriented Architecture*, SOA), basti pensare ad *Internet* ed i *web services*. In questo approccio il contesto elaborativo è fatto di tante entità che forniscono e richiedono servizi; un servizio può essere immaginato come un modulo *software* indipendente, sostituibile e riusabile per la gestione e la fornitura di una funzionalità secondo una ben definita interfaccia. Quando un cliente ha bisogno di un servizio, dapprima cerca (*discovery*) quello con le caratteristiche più adatte alle sue esigenze, e poi in seguito lo utilizza (*delivery*). Grazie ad una proprietà essenziale dei servizi, ovvero il grado di componibilità con altri servizi per crearne di nuovi, questo nuovo paradigma di elaborazione fornisce flessibilità e adattabilità alle moderne applicazioni.

D'altra parte, la strada alla mobilità dei dispositivi è aperta grazie alle tecnologie *wireless* su ampio e corto raggio (*UMTS* e *WI-FI*, ad esempio) e ai progressi tecnologici compiuti nell'ambito dell'elettronica *low-power*. Queste innovazioni hanno portato ad affiancare alle infrastrutture di rete classiche, nuove infrastrutture di rete ibride (*wired* e *wireless*) e totalmente *wireless* (senza alcun *core* di rete fisso) dette rispettivamente di *nomadic computing* e ad *hoc computing*.

Perciò, con il connubio tra le architetture SOA, gli apparati mobili dell'elettronica di consumo (*Personal Digital Assistant*, *smart phone*, …) e le nuove infrastrutture di rete, non è più difficile pensare a scenari in cui le persone hanno *wearable computer* per svolgere le loro attività quotidiane in assoluta libertà di movimento, oppure per utilizzare i servizi informatici offerti dalle strutture pubbliche o private con cui esse si trovano ad interagire (come *shopping mall*, aeroporti, musei, sedi lavorative, …).

Tuttavia, i tempi non sembrano essere ancora maturi per pensare ad uso delle risorse telematiche che sia pervasivo, ovvero dappertutto e in qualsiasi situazione. Infatti, gli ambienti mobili portano con sé numerose problematiche inesplorate nell'ambito dei sistemi tradizionali: a differenza delle reti *wired*, le connessioni delle reti *wireless* non sono affidabili, veloci e permanenti, ma possono soffrire di latenza ed intermittenze; i nodi di elaborazione non sono stabili e potenti come quelli fissi, ma possono essere mobili e poveri di risorse; le caratteristiche del contesto elaborativo sono estremamente variegate e mutano continuamente.

Con tali differenze, il *delivery* di un servizio informatico non può più affidarsi alle ipotesi su cui si fondano le soluzioni di *service delivery* per i sistemi distribuiti classici. E' necessario che nel progetto delle nuove architetture *service-oriented*, siano intrapresi nuovi approcci per far fronte alla elevata mobilità e dinamicità dei dispositivi e delle tecnologie *wireless*, integrati oppure no con gli elementi delle tecnologie tradizionali.

Questo lavoro di tesi cerca di dare alcune risposte proponendo Esperanto, una nuova piattaforma *middleware* orientata ai servizi per infrastrutture di *nomadic computing*, ovvero sistemi di elaborazione distribuita che rappresentano un buon compromesso tra i sistemi distribuiti tradizionali e i sistemi totalmente innovativi basati su infrastrutture di ad *hoc mobile networking*.

Il nostro interesse verso le infrastrutture di *nomadic computing* nasce osservando che nella maggioranza degli scenari oggi immaginabili è sempre possibile ricorrere all'utilizzo di una rete fissa (si pensi come prima ad un aeroporto, una galleria d'arte, un centro commerciale, la nostra casa...).

Inoltre, l'uso di una tale infrastruttura permette di "non spezzare il filo con il passato". Proporre, infatti, una nuova soluzione non è mai veramente efficace se non si cerca in qualche modo di portare in conto le esigenze delle soluzioni già esistenti.

Esperanto nasce perciò, sia con l'obiettivo di rispondere in maniera più diretta ai requisiti dell'elaborazione distribuita nei nuovi ambienti emergenti, sia con la

volontà di integrare le soluzioni attuali in un unico ambiente omogeneo dove tutte le applicazioni possono offrire e utilizzare servizi.

Come nelle *service oriented architecture*, anche in Esperanto è prevista la distinzione tra le fasi di *service discovery* e *service delivery*. In questa tesi saranno trattati solo gli aspetti relativi al *delivery* dei servizi; pertanto, sarà strutturata come segue:

Nel primo capitolo sono introdotti i concetti di *nomadic computing* e *service delivery*. In particolare si cerca di definire quali esigenze bisogna soddisfare quando si vuole sviluppare una soluzione di *service delivery* per ambienti di *nomadic computing*.

Nel secondo capitolo sono presentate alcune architetture orientate ai servizi e alcune soluzioni di *service delivery*, allo scopo di confrontarle anche alla luce degli elementi estratti dal primo capitolo. Uno studio approfondito delle soluzioni esistenti è naturalmente un passo obbligato per la progettazione di una nuova infrastruttura che miri a integrarle in un unico approccio.

Nel terzo capitolo è introdotta Esperanto, saranno illustrati il contesto in cui essa si cala, i requisiti e i vincoli di progetto nonché le proprietà desiderabili che dovrebbe avere. E' ampiamente illustrata la sua struttura, le sue caratteristiche e il comportamento dinamico degli elementi che in essa sono definiti.

Nel quarto capitolo sono affrontati con maggiore dettaglio gli aspetti di progetto più interessanti che la soluzione di *service delivery* Esperanto affronta per andare incontro alle esigenze degli ambienti di *nomadic computing*.

Nel quinto ed ultimo capitolo sono forniti alcuni aspetti implementativi e i dettagli del progetto di basso livello di parte degli elementi dell'architettura.

Ringraziamenti

Penso che l'unico modo per essere ascoltati sia quello di essere brevi.

Ho sempre immaginato che, della tesi, la pagina dei ringraziamenti fosse la cosa più noiosa da leggere. Forse è vero, forse no (la tesi stessa è forse più noiosa!) ma è bello, tra un ringraziamento ed un altro, esprimere un po' della propria personalità.

Non sono un bravo scrittore e soprattutto non sono bravo ad esprimere veramente quello che provo per le persone che mi sono vicine e che mi vogliono bene. Sarò banale, ma non m'importa, questa pagina è dedicata soprattutto alla mia famiglia: a Roberta, a mia madre, a mio padre, ai miei nonni. E' vero, in questi anni non mai stato molto sereno, ma l'affetto che provo per loro è grande, e qualsiasi sforzo per tirarlo fuori non sarà mai abbastanza. Grazie.

Ai miei amici, spero sinceri; anche loro talvolta hanno dovuto tollerare le mie tensioni e i miei malumori. Spero che il legame che ci stringe duri a lungo. Grazie.

A chi mi ha aiutato in questi anni di università, in questi ultimi mesi, con i consigli, con gli insegnamenti, con la compagnia; un grazie sincero a Tommaso, al mio correlatore Domenico, a Cristiano (il mio correlatore morale) e al mio relatore, il Prof. Stefano Russo.

Alla nostra sinergia, al nostro essere un tutt'uno. Se oggi siamo qui a scrivere queste pagine, è anche grazie a te. Rifare tutto quello che ho fatto in questi cinque anni senza il tuo aiuto, mi fa sembrare ogni scalino una montagna. Grazie Marcello.

Certo ancora lunga è la strada della vita, devo crescere, migliorare il mio modo di essere, affrontare mille sfide. Quest'anno è stato importante in tutti questi aspetti. Ed è anche grazie a te, Cristina.

Armando

Capitolo 1

Service Delivery in ambienti di *nomadic computing*

1.1 Introduzione

I sistemi di comunicazione e di elaborazione dell'informazione hanno subito negli ultimi anni una profonda evoluzione. Da quando un solo grande calcolatore (il temuto *mainframe*) era il protagonista assoluto nelle giornate di più persone, molta strada è stata fatta. Oggi, vedere una persona che disponga e usi normalmente un *personal computer* non è più una novità e immaginare un futuro dove molti dispositivi intelligenti saranno al servizio di ognuno di noi non sembra più essere fantascienza.

Questo percorso evolutivo dal *computing* del centro di calcolo, che passa per il *computing* personale fino a finire al *computing* ubiquo e pervasivo è stato delineato già da tempo da uno dei ricercatori più citati nell'ambito di questi argomenti, *Mark Weiser* [31] (la figura 1-1 illustra il suo modo di vedere tale evoluzione).

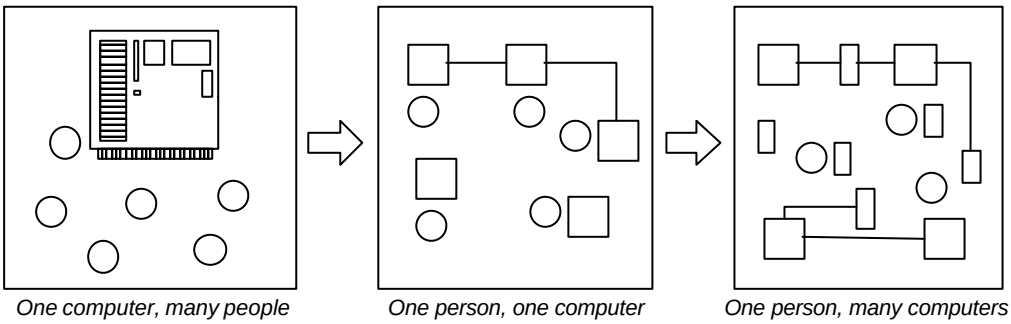

Figura 1-1: l'evoluzione dei sistemi di elaborazione e comunicazione.

I nuovi sistemi di elaborazione dell'informazione dovranno essere caratterizzati da un grado di mobilità, dinamicità e flessibilità sempre maggiore: un individuo ha bisogno di spostarsi portando con sé tutti gli strumenti che la tecnologia offre a

supporto del suo lavoro o dei suoi passatempi ludici. In questi movimenti, il contesto che ci circonda varia nelle risorse e nelle condizioni: i nostri dispositivi saranno veramente utili solo se sufficientemente flessibili da adattarsi a questa dinamicità. La loro specializzazione aiuterà poi l'integrazione negli ambienti di tutti i giorni (ufficio, casa,...), così da andare incontro alla "diffidenza" delle persone più restie alle innovazioni.

Tuttavia, il processo evolutivo dei sistemi distribuiti ha seguito due percorsi: il primo è quello tecnologico, ed è stato tracciato da Weiser; il secondo è quello metodologico, ed è stato tracciato dai principi dell'Ingegneria del *Software* con la nascita delle *Service Oriented Architecture* (definite rigorosamente da Brown in [53]).

Il concetto di SOA (*Service Oriented Architecture*) non è completamente nuovo, essa è essenzialmente una collezione di servizi interagenti, in cui ognuno fornisce una funzionalità ben definita. L'interazione può implicare un semplice scambio di dati o la coordinazione delle attività di più servizi. Secondo l'approccio *service-oriented*, le applicazioni distribuite sono in generale sviluppate come insiemi indipendenti di *service* (l'entità logica che definisce la specifica di una o più funzionalità), *service provider* (l'entità che implementa la specifica del servizio) e *service requestor* (l'entità che richiede un servizio ad uno specifico *provider*) [44]. In figura 4-2 è esemplificata l'interazione tra *service provider* e *service requestor* (detto anche *consumer*), ovvero come indicato in letteratura, l'operazione di *service delivery*.

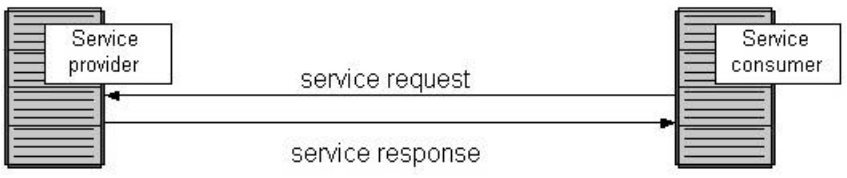

Figura 1-2: l'interazione tra *service provider* e *service consumer*.

Un nuovo compito va però assunto dalle SOA al fine di assecondare l'elevata dinamicità e mobilità invocata da Weiser: il supporto alla configurazione

spontanea delle applicazioni orientate ai servizi, come meccanismo reattivo alle mutevoli condizioni dell'ambiente [44].

A tale scopo, dietro questo processo evolutivo non può che esserci lo studio di nuovi paradigmi di interazione distribuita (o quanto meno la rivisitazione di quelli oramai assestati), dato che le ipotesi alla base dei sistemi classici perdono parte della loro validità quando applicate ai contesti del prossimo futuro (l'ambiente non è più statico, le connessioni tra i nodi di rete non sono più stabili e durature nel tempo ecc.).

Infatti, nei sistemi distribuiti tradizionali quando un processo richiede di interagire con un altro processo, al fine di ottenere un servizio da esso offerto, si affida da un lato alle buone capacità elaborative dei nodi di rete, dall'altro alla stabilità e alle buone prestazioni dei *link* dell'infrastruttura fisica. Pertanto, se una delle due parti non reagisce, qualche malfunzionamento molto probabilmente può essere accaduto.

I nuovi ambienti, invece, stravolgono queste caratteristiche: i nodi elaborativi (e con essi i servizi) possono facilmente disconnettersi e riconnettersi (per cui è errato assumere che la non disponibilità sia sintomo di malfunzionamento), comunicano tra di loro attraverso connessioni *wireless* (per natura intermittenti e inaffidabili) e sono poveri di risorse computazionali. In queste condizioni diventa difficile utilizzare un servizio: per fare in modo che il *service delivery* avvenga correttamente è necessario che le nuove soluzioni portino in conto delle nuove ipotesi introdotte da questi nuovi contesti.

Nel seguito descriveremo dapprima i concetti di base che caratterizzano il *distributed computing* emergente; successivamente formalizzeremo il concetto di *service delivery*, caratterizzandone requisiti e peculiarità.

1.2 *Distributed computing*

In via del tutto generale possiamo dire che un sistema distribuito consiste in una collezione di componenti dislocati su vari *computer* connessi attraverso una rete telematica [1]. Come già evidenziato, i sistemi distribuiti tradizionali sono evoluti negli anni verso nuovi sistemi di elaborazione e comunicazione: *mobile computing*, *pervasive computing*, *ubiquitous computing*, *nomadic computing*, *ad hoc computing*. Anche se alcune similitudini possono essere individuate, ognuno di essi ha un'identità ben definita se si guarda ai seguenti aspetti [1] [2]:

Device: per *device* si intende un nodo di rete dotato di capacità elaborativa; in base alle sue proprietà di mobilità spaziale si può classificare in:

- fisso: dispositivi generalmente potenti con grandi quantità di memoria e processori veloci;
- mobile: *device* tipicamente poveri di risorse e con problemi di alimentazione che implicano la loro limitata disponibilità nel tempo.

Embeddedness: per *embeddedness* si intende il grado di specificità del compito per cui viene progettato un *device* nonché il livello di integrazione raggiunto nel costruire il dispositivo che assolve il determinato compito. In base a questo criterio è possibile distinguere tra dispositivi:

- *general-purpose*: un dispositivo è *general-purpose* quando è progettato per non rispondere a compiti specifici (PDA, *laptop*, PC…);
- *special-purpose*: un dispositivo è *special-purpose* quando è progettato per rispondere a compiti specifici (microcontrollori, sensori, *badges* …).

Connessione di rete: con questo attributo ci riferiremo alla qualità del collegamento tra due o più *host* di rete; essa può essere:

- permanente: tipicamente una connessione di rete permanente è caratterizzata da un'elevata banda ed un basso *Bit Error Rate* (BER);
- intermittente: tipicamente una connessione di rete intermittente è caratterizzata da una modesta banda, spesso variabile, ed un BER influenzato fortemente da agenti esterni.

Contesto di esecuzione: in letteratura esistono diverse definizioni di contesto. In questo ambito intenderemo per contesto qualsiasi cosa che possa influenzare il comportamento di una applicazione, come risorse interne al dispositivo (stato della batteria, disponibilità della CPU, disponibilità della memoria, dimensione del *display*) o esterne ad esso (larghezza di banda, affidabilità e latenza del canale, prossimità degli altri dispositivi). Un contesto può essere:

- statico: se le risorse interne ed esterne variano poco o mai;
- dinamico: se le risorse interne ed esterne sono soggette a forti cambiamenti.

La definizione di contesto cui si è fatto riferimento è riportata in [1], [4].

Sulla base degli attributi elencati sopra i sistemi distribuiti possono essere classificati in sistemi di:

Traditional distributed computing: sono una collezione di *device* fissi e *general-purpose* connessi tra loro attraverso una connessione permanente le cui applicazioni vivono in un contesto di esecuzione statico.

Nomadic computing: i sistemi di *nomadic computing* rappresentano un compromesso tra i sistemi totalmente fissi e totalmente mobili; sono generalmente composti da un insieme di *device* mobili *general/special purpose* interconnessi ad un'infrastruttura *core* con nodi fissi (collegati tra loro attraverso una connessione permanente) *general-purpose*. Data la natura del sistema, le applicazioni vivono in un contesto di esecuzione che può essere talvolta statico e talvolta dinamico.

Ad hoc mobile computing: i sistemi distribuiti *ad hoc* sono costituiti da un insieme di dispositivi mobili, tipicamente *general-purpose*, connessi tra di loro attraverso un collegamento intermittente e senza alcuna infrastruttura fissa, dove le applicazioni vivono in un contesto di esecuzione fortemente dinamico. Gli scenari di ad *hoc computing* possono essere numerosi; si può pensare ad una scrivania con tanti dispositivi *wireless* che comunicano tra loro (stampante, pda, cellulare) o ad un campo di battaglia dove dei *tank* devono scambiarsi strategie di attacco.

Pervasive computing: si parla di *pervasive computing* quando il sistema distribuito è composto da un insieme di dispositivi fissi *special-purpose* (sensori, lettori di *badges*, microcontrollori, …) connessi tra loro attraverso un collegamento tipicamente permanente le cui applicazioni vivono in un contesto di esecuzione tipicamente statico soggetto solo talvolta a rapidi cambiamenti. Con *pervasive computing* in pratica si intende la crescente diffusione di dispositivi informatici intelligenti, facilmente accessibili e talvolta invisibili, il cui uso semplifica lo svolgimento dei normali compiti di tutti i giorni (si pensi alle idee e alle innovazioni per la domotica, la *domus robotica*). Tali dispositivi sono spesso progettati per estendere le applicazioni basate su standard aperti alla vita quotidiana e sono molto spesso incorporati nell'ambiente in cui operano (sensori di posizione, sensori di luminosità, …).

Ubiquitous computing: i sistemi di *ubiquitous computing* sono costituiti da un insieme di dispositivi mobili, per la maggior parte *special-purpose* connessi tra loro attraverso collegamenti intermittenti e le cui applicazioni vivono in un contesto tipicamente dinamico. I sistemi di *ubiquitous computing* nascono dall'evoluzione dei sistemi di *pervasive computing* data dall'introduzione della mobilità dei dispositivi (per questo spesso i due termini sono usati come sinonimi). Con l'*ubiquitous computing* i *device* faranno parte dei nostri naturali movimenti ed interazioni con i normali ambienti del prossimo futuro: tutto ciò che è possibile immaginare con i

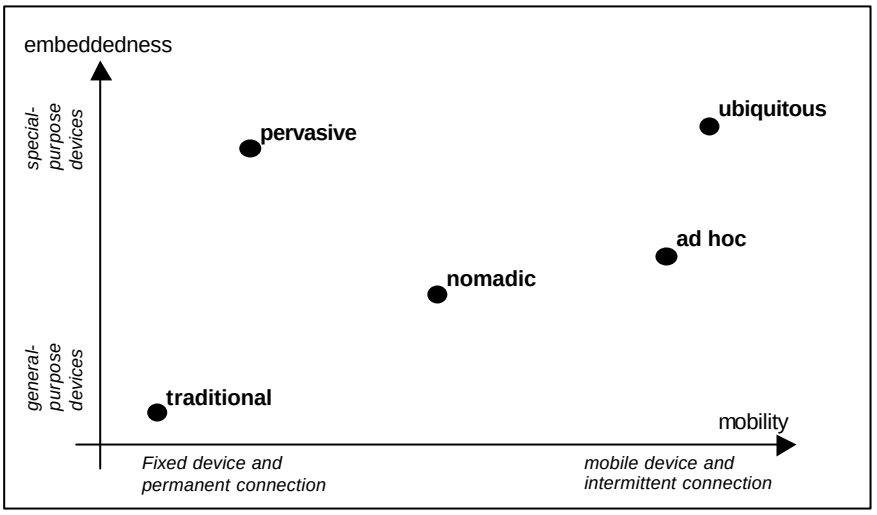

Figura 1-3: le varie forme di *distributed computing* caratterizzate dal punto di vista della mobilità e della specializzazione dei dispositivi.

computer e le telecomunicazioni, in movimento, fermi, in cielo, in terra o in mare può essere considerato *ubiquitous computing* [3].

Si osservi che *nomadic* e *ad hoc computing* possono essere visti come due forme diverse del concetto più generale di *mobile computing*. In realtà, come rappresentato graficamente in figura 1-4, "isole" di ad *hoc computing* possono coesistere nelle infrastrutture di *nomadic computing*.

Nella figura 1-3 si riporta una rappresentazione grafica della tassonomia presentata e tratta da [2]. La *mobility* è funzione della mobilità dei dispositivi, della tipologia di connessione di rete tra di essi e del contesto di esecuzione.

Come è facile intuire maggiore è il grado di mobilità e di *embeddedness*, più è sentita la necessità di fornire alle applicazioni di questi ambienti soluzioni efficaci per consentire agli utenti l'utilizzo dei servizi offerti dall'ambiente e dai dispositivi in esso presenti [3]. Infatti:

- L'elevata *embeddedness* implica (tipicamente) la povertà di risorse: non si può pensare più allo sviluppo di applicazioni/soluzioni che introducono un carico computazionale "pesante" in quanto la presenza di dispositivi "leggeri" è più la regola che l'eccezione.

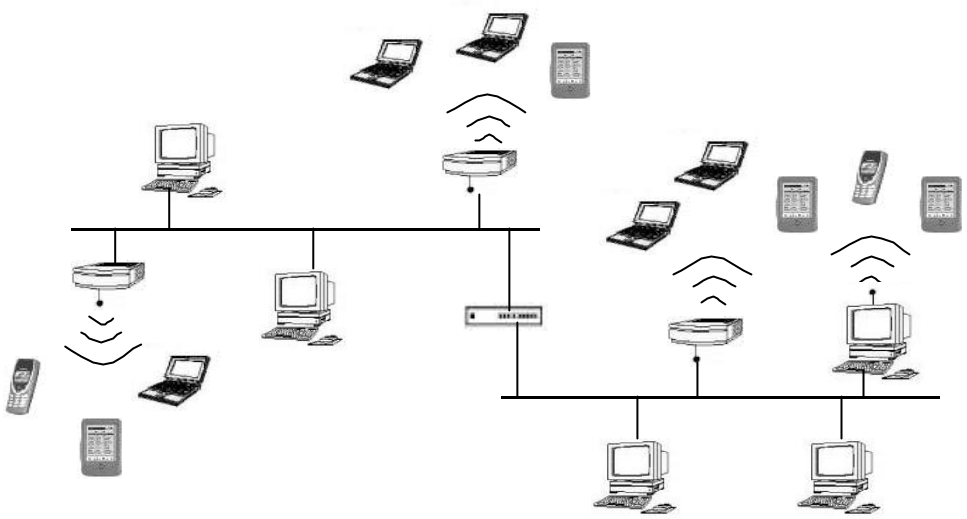

Figura 1-4: una tipica infrastruttura di *nomadic computing*.

- L'elevata mobilità dei dispositivi implica spesso che un servizio possa non essere disponibile ad un certo istante (perché la connessione di rete non è stabile, o perché il dispositivo sta migrando verso un altro ambiente). Ciò non deve comportare un malfunzionamento: bisogna pensare a soluzioni che tengano conto di queste eventualità.

Le tecniche di *service delivery* guardano in generale ai meccanismi da adottare per garantire l'interazione tra le applicazioni, tuttavia i principi, le tecnologie e i paradigmi che ad esse sono alla base possono variare da un sistema di elaborazione ad un altro. Nel seguito li affronteremo con riferimento al *nomadic computing*.

1.3 *Nomadic Computing*

1.3.1 Introduzione

Abbiamo detto che i sistemi di *nomadic computing* rappresentano un compromesso tra i sistemi totalmente fissi e totalmente mobili; rappresentano quindi a nostro avviso un passo intermedio per evolvere verso i sistemi di

elaborazione completamente mobili. Questa idea di "migrazione" dai sistemi fissi a quelli mobili è ben rappresentata dalla figura 1-4.

1.3.2 Perché il *nomadic computing*

L'attenzione di questa tesi è rivolta ai sistemi di *nomadic computing* perché rappresentano oggi un paradigma di comunicazione che meglio si adatta al nostro fare quotidiano: come indicato da Kleinrock, oggi viviamo in un "*disconnected world*" dove il più delle volte viaggiamo da una postazione fissa all'altra, ufficio, casa, aeroporti, hotel, portando con noi i nostri *computer* e i dispositivi di comunicazione. La combinazione del *mobile computing* supportato dalle telecomunicazioni emergenti, sta cambiando oggi il nostro modo di vedere l'elaborazione delle informazioni. Si riconosce, giorno dopo giorno, la necessità di accedere alle risorse anche quando si è in movimento e durante la migrazione da un contesto ad un altro. Ovvero, come riportato in [5], *anytime, anywhere access*.

D'altra parte, quando si pensa allo sviluppo di questi ambienti di elaborazione distribuita, le infrastrutture di rete oggi disponibili rappresentano il giusto compromesso tra carico computazionale e leggerezza dei dispositivi mobili.

Pertanto gli scenari di *nomadic computing* nell'ambito dell'*office automation*, nella domotica o in tutti quegli ambienti con nodi fissi in grado di offrire servizi e capaci di ospitare dispositivi mobili (aeroporti, gallerie d'arte, aerei, navi da crociera, strutture abitative…) diventeranno sempre più diffusi e costituiranno il passo da compiere per raggiungere l'obiettivo *all the time, everywhere access* dei futuri scenari di *ubiquitous* e *pervasive computing* [6].

1.3.3 *Middleware* per *nomadic computing*

I *middleware* rappresentano quella infrastruttura *software* tra i sistemi operativi e le applicazioni distribuite, nata per nascondere la complessità introdotta dalla rete (eterogeneità delle piattaforme, dei linguaggi di programmazione, dei sistemi

operativi, …). Nello sviluppo di *middleware* per sistemi distribuiti tradizionali è richiesto che i meccanismi di trasparenza introdotti rispondano a requisiti di qualità (non funzionali) come *scalability, heterogeneity, openness, fault-tolerance, resource sharing* ecc.

I sistemi di *nomadic computing* condividono gli stessi requisiti non funzionali dei sistemi distribuiti tradizionali, tuttavia alcuni di essi vanno rivisti in ragione delle variate circostanze in cui ci si trova ad operare.

La scalabilità del sistema è relativa all'abilità di servire un ampio numero di dispositivi mobili in modo efficiente. L'eterogeneità è complicata dalla presenza di differenti *link* di rete (*wireless*/*wired*), diverse tecnologie e dispositivi mobili possono coesistere nello stesso ambito di rete. La *fault tolerance* può dipendere dal tipo di applicazione, ma in generale le disconnessioni (nessuna copertura radio, l'alimentazione del dispositivo portatile si esaurisce, l'ambiente è rumoroso) potrebbero non essere un malfunzionamento o un'eccezione bensì un requisito funzionale.

Contrariamente ai requisiti non funzionali, le esigenze e le ipotesi alla base dello sviluppo dei *middleware* classici (i requisiti funzionali) cambiano quando si va incontro agli ambienti mobili.

I *middleware* per sistemi di *nomadic computing* devono introdurre un carico computazionale leggero, fornire un paradigma di comunicazione asincrono e assicurare la *context awareness* alle applicazioni.

I *middleware* per sistemi classici invece fanno l'esatto opposto e mal si adattano agli ambienti di *nomadic computing*. In particolare i punti di divergenza che spingono i ricercatori ad esplorare nuove soluzioni sono:

> Carico computazionale: il tipo di carico computazionale introdotto dai *middleware* classici è tipicamente *heavy-weight* (richiede una grossa quantità di risorse per fornire servizi alle applicazioni soprastanti), contrariamente al carico *light-weight* richiesto dai *middleware* per ambienti mobili, data la presenza di dispositivi con capacità limitate.

Paradigma di comunicazione: i *middleware* classici assumono, come base per il loro corretto funzionamento, la presenza di connessioni stabili e permanenti tra i nodi di rete e di conseguenza adottano di base un paradigma di comunicazione di tipo *sincrono,* in cui si richiede accoppiamento temporale tra *client* e *server* (ovvero *client* e *server* devono essere contemporaneamente presenti durante la loro interazione). Viceversa, negli ambienti in cui i dispositivi mobili possono volontariamente o meno connettersi al sistema, un paradigma di comunicazione sincrono deve inevitabilmente lasciare posto ad un paradigma di comunicazione *asincrono,* in cui non è richiesto che *client* e *server* siano connessi simultaneamente [1].

Adaptation delle applicazioni: l'*adaptation* consiste nell'insieme di strategie da applicare per adattare il comportamento delle applicazioni al contesto di esecuzione. Diversamente dai sistemi distribuiti tradizionali, i sistemi distribuiti per *nomadic computing* (e *mobile computing* in generale) vivono in un contesto di esecuzione estremamente dinamico: la larghezza di banda potrebbe non essere stabile, i servizi disponibili in un istante possono non esserlo un istante successivo, ecc. Per cui un approccio di adattamento *application aware* anziché uno *application transparent* può essere preferito; con il primo approccio le informazioni sul contesto di esecuzione (o parte di esso) e le decisioni da effettuare a valle dei suoi mutamenti, sono carico delle applicazioni. Con il secondo approccio, tutte le informazioni circa il contesto sono tenute nascoste all'interno del *middleware* e rese trasparenti alle applicazioni.

Proprio in virtù di quanto osservato, ci sono delle proposte per adattare i *middleware* per i sistemi distribuiti tradizionali agli ambienti di *mobile computing*. Tra i *middleware object-oriented,* è sicuramente rilevante la proposta dell'adattamento di CORBA (*Common Object Request Broker Architecture*) dell'OMG (*Object Management Group*) agli ambienti con accesso *wireless* e terminali mobili [33]. I principi di progetto basilari sono stati la trasparenza e la

semplicità degli ORB (intermediario di richiesta ad un oggetto, *Object Request Broker*) destinati ai dispositivi mobili. La trasparenza consiste nel nascondere agli ORB preesistenti, e quindi non mobili, la presenza dei meccanismi per la gestione della mobilità introdotti negli ORB per terminali mobili. In questo modo gli ORB non mobili non debbono implementare nuove funzionalità per interoperare con gli oggetti CORBA eseguiti sui terminali mobili. La semplicità è necessaria al fine di non sovraccaricare i dispositivi mobili, ed è ottenuta fornendo alle applicazioni CORBA un insieme minimo di funzionalità. La necessità di far interoperare in maniera trasparente gli oggetti "mobili" con quelli "fissi", lascia intuire come la specifica *wireless* CORBA sia rivolta ad ambienti di *nomadic computing*.

1.4 *Service Delivery*

1.4.1 Introduzione

Service Delivery è un termine usato per descrivere la fase durante la quale un componente (rappresentato da una dispositivo *hardware* o una risorsa *software*) ottiene un servizio da un altro componente (anch'esso sia *software* che *hardware*). Il servizio è scoperto durante la fase di *Service Discovery*, in cui il suo potenziale cliente diventa consapevole dell'ambiente a cui è connesso e scopre quali servizi sono in esso disponibili. Tipicamente è una caratteristica comune di tutte le SOA per ambienti mobili, distinguere la fase di *Service Delivery* da quella di *Service Discovery*. In questo modo si conferisce al progetto dei componenti *software* e *hardware* a supporto dello sviluppo di applicazioni per ambienti distribuiti maggiori caratteristiche di modularità e riusabilità.

1.4.2 Evoluzione del *Service Delivery*

Così come sono evoluti i sistemi di elaborazione dell'informazione (dai personali ai distribuiti), anche la definizione di *service delivery* è cambiata negli ultimi anni.

In ambito locale i processi *software* comunicano tra loro attraverso i tradizionali meccanismi di IPC – *InterProcess Communication* (*shared memory*, *mailboxes*, *pipes*, *files*); dunque l'interazione è risolta dal SO, che rappresenta per essi l'ambiente comune in cui evolvere. Con l'avvento del *networking*, il *distributed computing* tradizionale rappresenta la naturale evoluzione del *personal computing*. Tuttavia esso porta con sé problematiche non presenti nel *computing* locale: eterogeneità, latenza nelle risposte, inaffidabilità della comunicazione, individuazione delle risorse, …. Come già osservato, i *middleware* nascono proprio per far fronte a questi problemi e per garantire nel contempo i meccanismi di trasparenza (al linguaggio, alla locazione, alla piattaforma…) dei dettagli ad esso sottostanti. Con l'ulteriore evoluzione del *distributed computing* classico verso il *mobile computing*, nuove problematiche sono sorte: l'eterogeneità aumenta a causa della diversità dei dispositivi (le capacità di ogni *device*, in termini di potenza elaborativa, *power supply* e interfacce di I/O, cambiano in funzione dei compiti cui assolve) e delle infrastrutture di rete, l'inaffidabilità dei *link wireless* rende ancor più impredicibile l'esito delle comunicazioni, l'elevata dinamicità dell'ambiente complica la localizzazione delle risorse. I vecchi paradigmi di interazione in cui si assumeva che le entità interagenti:

- conoscessero la localizzazione reciproca,
- erano sempre attive tranne in condizioni di malfunzionamento, e conseguentemente che
- potevano interagire con un semplice protocollo di tipo richiesta/risposta (*request/response*)

devono quindi essere adattati per essere funzionali ai nuovi ambienti di *mobile computing* oppure lasciare spazio a nuove soluzioni pensate *ad hoc* perché:

- data la mobilità dei dispositivi, uno di essi può entrare o lasciare liberamente l'ambiente: avere consapevolezza di un riferimento mobile non sempre è la soluzione più adatta;

- date le caratteristiche delle tecnologie *wireless* e la "portatilità" dei dispositivi, essi possono essere momentaneamente non disponibili senza che ci sia un reale malfunzionamento: assumere che ci sia un guasto quando una risorsa non risponde non sempre è la scelta corretta;
- un protocollo di tipo richiesta/risposta come concepito nei *middleware* classici non è la soluzione più efficiente per le caratteristiche succitate.

Paradigmi di interazione in cui è previsto che:

- nella comunicazione non è necessario avere un riferimento diretto alla propria controparte, ma basta un riferimento ad uno spazio comune per lo scambio delle informazioni,
- l'interazione può avere successo anche quando le due parti non sono attive contemporaneamente e che
- preveda l'adozione di protocolli asincroni (tipo *one-way* e non *request/response*) anche per implementare protocolli più complessi

sembrano allora la soluzione più adatta quando bisogna progettare un'infrastruttura *software* per il supporto allo sviluppo di applicazioni distribuite in ambienti mobili.

Nel seguito descriveremo le caratteristiche fondamentali del processo di *delivery* guardando in particolare alle diverse tecniche di comunicazione esistenti.

1.4.3 Elementi fondamentali del *Service Delivery*

Le entità principalmente coinvolte in un processo di *service delivery* sono:

Client Agent: è il componente *software* o *device* che, una volta trovati nella rete i servizi di cui ha bisogno, ne richiede l'accesso per ottenerne i risultati. E' detto anche *service requestor*.

Service Agent: per dispositivi che forniscono servizi ad altri dispositivi, il *service agent* è un componente *software* che realizza il servizio

messo a disposizione, quando richiesto da un *client agent*. E' detto anche *service provider*.

Service Broker: è l'insieme dei componenti *software* che si pone da intermediario nella comunicazione e nella gestione di *client* e *service agent*, allo scopo di risolvere tutti i problemi connessi all'interazione tra queste entità (eterogeneità, inaffidabilità della comunicazione,…) e all'uso dei servizi (catalogazione, pubblicazione, rimozione,…). Spesso, un *service broker* permette l'interazione tra *client* e *server* collaborando con altri intermediari realizzati attraverso *proxy-pattern* [17]. In particolare i *proxy* (uno lato *client* ed uno lato *server* in un pattern di tipo *stub-skeleton*) devono nascondere tutti i dettagli di comunicazione, come ad esempio il *marshalling* e *unmarshalling* dei parametri, la costruzione delle richieste, l'invio dei messaggi, la ricezione delle risposte e la restituzione dei risultati. Questo approccio libera lo sviluppatore di curarsi dei dettagli implementativi di basso livello. Con la sua introduzione, il paradigma *client/server* diventa in realtà *client/broker/server*, dove la presenza del *broker* può essere più o meno consistente (*fat broker* o *thin broker*). Nel seguito sottintenderemo la sua presenza.

Evidentemente ogni *client agent*, prima di utilizzare un *service agent*, deve stabilire come accedervi. D'altra parte in ambienti dinamici ed eterogenei la conoscenza a priori delle modalità di accesso ad un servizio potrebbe precludere l'uso ad una parte di potenziali clienti. Inoltre, la fase di *service discovery*, in cui vengono offerti i meccanismi per la scoperta dei servizi dell'ambiente, può non essere sufficiente a stabilire le regole che *client agent* e *service agent* devono rispettare per poter interagire tra loro.

Come osservato in [15] [18] e come previsto ad esempio nella *Service Oriented Architecture* dei *web services* ([23], [24]), il processo di *service delivery* può dunque essere decomposto in due fasi:

Service Access Description: la fase in cui un *client agent*, una volta localizzato il servizio di cui ha bisogno, acquisisce tutte le informazioni (proprietà, capacità, modalità di accesso) necessarie per il suo utilizzo.

Interaction: la fase in cui *client* e *service agent* interagiscono, il primo per ottenere il servizio, il secondo per fornirlo.

Vediamo le due fasi con maggior dettaglio:

1.4.3.1 *SERVICE ACCESS DESCRIPTION*

La fase di *Service Access Description* può anche essere assente in quei casi in cui il *client* già conosce la modalità di accesso al servizio. Quando presente, le regole definite da un protocollo di *delivery* prevedono che la fase sia iniziata dal *client agent*, il quale collabora con il *service agent* (o con componenti intermediari della comunicazione come ad esempio il *service broker*) per negoziare l'accesso al servizio.

In generale un servizio è un componente complesso che può implementare una o più funzionalità ognuna delle quali è caratterizzata da una lista di parametri da specificare per l'accesso, delle proprietà che specificano la funzionalità offerta ed una lista di parametri che rappresentano il risultato della fruizione del servizio stesso. Una rappresentazione formale di queste informazioni è necessaria e può essere specificata in via del tutto generale attraverso un *record*, che indicheremo con *Service Access Description Record* (*SADR*). I campi del *SADR* contengono una serie di informazioni che specificano le proprietà ed i parametri delle funzionalità che rappresentano un servizio.

La fase di *service access description* consiste pertanto nello scambio di *service access description request* e *service access description reply* tra *client agent* e chi ha cura di catalogare i descrittori (il *service agent* stesso o il *service broker*, ad esempio): nella richiesta viene specificato il servizio di cui si vuole ottenere il descrittore, nella risposta, in condizioni normali, il descrittore è restituito.

Osserviamo, infine, che negli ambienti di *mobile computing* è importante associare al SADR il concetto di *temporal boundary*. Con il *temporal boundary,* il *SADR* specifica il servizio e le sue funzionalità anche dal punto di vista dei vincoli temporali che devono essere rispettati affinché si abbia una corretta fruizione nella successiva fase di *interaction*.

1.4.3.2 *INTERACTION*

Una volta che il *client agent* ha ottenuto il descrittore del servizio, conosce quali sono le funzionalità che esso offre e come potervi accedere. L'*interaction* ha luogo quando il cliente richiede l'esecuzione delle funzionalità (fornendo per questo gli eventuali dati di *input*) e/o il servizio fornisce gli eventuali dati di *output*. L'*interaction* consiste pertanto nello scambio di *service request* e *service response*. Essa può essere classificata in base ai seguenti criteri:

ACCOPPIAMENTO: rappresenta la forza del legame che c'è tra *client* e *service* nell'interazione. Esso si misura lungo le seguenti dimensioni [10]:

- *Accoppiamento nello spazio*: due componenti *software* sono spazialmente accoppiati se per interagire necessitano di conoscere il riferimento alla loro controparte (ovvero nelle *service request*/*service reply* si specificano i riferimenti rispettivamente a *service*/*client*). Viceversa, c'è disaccoppiamento spaziale se le entità interagenti non necessitano di tenere un riferimento l'una dell'altra.
- *Accoppiamento nel tempo*: due componenti *software* sono temporalmente accoppiati se per interagire necessitano di essere entrambi attivi (una *service request* è accettata dal *service provider* solo se esso è attivo, conseguentemente la elabora). Viceversa c'è disaccoppiamento temporale se le entità interagenti non necessariamente sono attive allo stesso tempo.
- *Accoppiamento nel sincronismo*: due componenti *software* sono accoppiati nel sincronismo, ovvero c'è interazione sincrona, quando

l'entità che chiede l'interazione (ad esempio il *client agent* con una *service request*) si sospende nella attesa della risposta della sua controparte. Viceversa, c'è disaccoppiamento nel sincronismo, in altre parole c'è interazione asincrona, quando l'entità che chiede l'interazione continua la sua elaborazione anche dopo la richiesta fatta alla sua controparte.

INIZIATIVA: l'iniziativa nella comunicazione tra *client* e *service* può essere intrapresa da entrambi i lati. Distinguiamo tra meccanismi [12] [16]:

- *Client pull o pull-based*: si parla di *delivery pull-based* quando l'iniziativa nell'interazione è presa dal *client agent* che invia richieste al *service agent*. Questo tipo di interazione si può ulteriormente distinguere in:
  - *One-way*: il *service agent* riceve dei dati, effettua delle elaborazioni e non invia alcun risultato al *client agent*, che è libero di proseguire nelle sue elaborazioni.
  - *Two-way o Request-Response*: il *service agent* riceve dei dati, effettua delle elaborazioni ed invia il risultato al *client agent*, che si era precedentemente posto in attesa all'atto della richiesta.
- *Server push o push-based*: il *delivery push-based* implica il trasferimento di informazioni al *client* senza che esso ne faccia esplicita richiesta. Questo tipo di interazione si può ulteriormente distinguere in:
  - *Notification*: il *service agent* invia dei dati verso i *client agent*; è l'operazione duale della *one-way*.
  - *Solicit-Response*: il *service agent* invia dei dati ed attende le correlate risposte dai *client agent*. Questa operazione è la duale della *request-response*.

A partire dalle modalità elementari di iniziativa elencate, è possibile immaginare interazioni più complesse. Ad esempio l'interazione *defferred synchronous* [1] consiste in una interazione *one-way* seguita da una *request-response*. Come effetto di ciò si ha appunto una sincronizzazione ritardata: il *client* invia la richiesta e continua nelle sue elaborazioni; successivamente è sua responsabilità andare a reperire in un secondo momento i risultati dell'elaborazione, sincronizzandosi con il *server* attraverso una *request-response.* Inoltre sono possibili delle soluzioni ibride, cioè sia *client-pull* che *server-push*, in cui tipicamente c'è la presenza di un'entità intermedia, nella quale i dati sono introdotti dal *server* attraverso delle operazioni tipo *push*, e consumati dai *client* con delle operazioni tipo *pull*. La figura 1.4 riassume gli approcci discussi con particolare attenzione ai paradigmi *client-pull* e *server-push*.

Infine si può aggiungere che entrambi gli approcci possono essere *event-driven* o *schedule-driven*: nell'interazione *event-driven* la richiesta dati (*pull*) o la trasmissione dati (*push*) è attivata da un evento, ad esempio l'azione di un utente (*pull*) oppure l'*update* di un dato (*push*); nell'interazione *schedule-driven* la richiesta dati (*pull*) o la trasmissione dati (*push*) è programmata secondo uno schema preconfigurato.

- *MOLTEPLICITÀ*: un meccanismo di interazione può essere classificato in base alla molteplicità delle entità che vi partecipano, ovvero possiamo distinguere tra interazioni del tipo [12]:
  - *1-to-1 o unicast*: in questo tipo di interazione i dati sono inviati da un

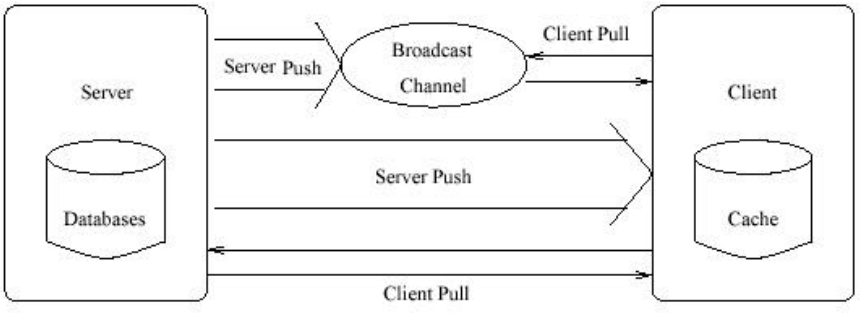

Figura 1-4: i paradigmi *client-pull* e *server-push* e ibridi.

service agent verso un *client agent* e viceversa; è la modalità di comunicazione tipica delle interazioni *pull-based*.

- *1-to-N*: questo tipo di interazione permette a più *client agent* di ricevere i dati inviati da un *service agent*. Si possono distinguere due tipi di comunicazione *1-to-N*: *multicast* e *broadcast*. Con la prima i dati sono inviati ad uno specifico sottoinsieme di *client agent* noto a priori. Con la seconda i dati sono inviati su di un mezzo sul quale sono in ascolto un numero di *client* non identificati e quindi non noti a priori. La comunicazione 1 ad N è tipica delle interazioni *push-based*.

1.4.4 Meccanismi di interazione

Secondo la struttura definita sopra, proviamo a descrivere alcuni dei possibili *meccanismi di interazione* [7] [10] [12] oggi esistenti:

- *Remote invocation*: l'invocazione remota rappresenta un'estensione in ambiente distribuito del meccanismo di chiamata a procedura (*call/return*) previsto dai normali linguaggi di programmazione. Questo tipo di interazione fu proposto per la prima volta nella forma di *Remote Procedure Call* (RPC) per linguaggi procedurali ed è stato poi esteso ai contesti *object-oriented* nella forma di invocazione a metodo remoto nei modelli ad oggetti distribuiti. L'invocazione remota nasconde gli intricati dettagli di rete, curandosi di tutti gli aspeti da risolvere quando si sviluppano applicazioni in ambito distribuito. Nel meccanismo di *remote invocation*, *client* e *service* sono fortemente accoppiati (nello spazio, nel tempo e nel sincronismo), e l'iniziativa è intrapresa dal cliente (*client pull*), tipicamente in maniera *event-driven*. La molteplicità è *1-to-1*.
- *Message oriented*: l'interazione orientata allo scambio di messaggi si basa sull'astrazione di una coda di messaggi tra processi interoperanti su rete, generalizzazione del noto concetto di *mailbox* tipico dei sistemi operativi. In questo modello la comunicazione non è di tipo *client-server* stretto, nel senso

che l'interazione non avviene per sua natura tra un'entità in attesa passiva di richieste e entità che intraprendono la comunicazione, ma è tra pari e prevalentemente asincrona. Ci sono diverse forme di interazione *message oriented* [9], che possono essere classificati in:

- *Message queueing*: questa forma di interazione è nota anche come *point-to-point messaging* e fornisce ai *client* l'astrazione di una coda di messaggi FIFO che può essere acceduta attraverso la rete. I *message sender* ripongono i messaggi nella coda; i *message receiver*, in gergo *recipient*, prelevano i messaggi dalla coda e vi ripongono eventualmente un messaggio che contiene il risultato della loro elaborazione. Nel meccanismo di *message queueing* le entità sono disaccoppiate (nello spazio, nel tempo e nel sincronismo - tuttavia possibile implementare interazioni con maggiori livelli di accoppiamento) e l'iniziativa è *pull-based*. Come suggerisce il nome (*point-to-point*), la molteplicità è *1-to-1*.
- *Publish-Subscribe*: nel modello *publish-subscribe* i processi *publisher* pubblicano messaggi differenziati per tipo (*publish*) e i *subscriber* possono dichiararsi interessati in base al loro tipo (*subscribe*), al fine di ricevere una notifica della presenza della coda di messaggi di interesse. In base al tipo di filtraggio che il *subscriber* può usare per specificare il tipo di messaggi che esso desidera ricevere si distinguono i meccanismi *p/s topic-based* (i messaggi si discriminano in base all'argomento), *content-based* (i messaggi si discriminano in base al contenuto), *type-based* (i messaggi sono tipati e si discriminano in base al loro tipo). Nel meccanismo di *publish-subscribe* le entità sono disaccoppiate (nello spazio, nel tempo e nel sincronismo) e l'iniziativa è *push-based* tipicamente di tipo *event-driven*. La molteplicità è tipicamente 1 ad N.

▪ *Tuplespaces*: in questo modello (detto anche *distributed shared memory*) la comunicazione tra le applicazioni avviene mediante un'astrazione costituita da uno spazio virtuale di memoria condivisa: le applicazioni possono sincronizzarsi e scambiarsi dati inserendo e prelevando strutture dati (le tuple)

dallo spazio virtuale, mediante operazioni di lettura, di scrittura e di rimozione. Per far fronte ai problemi di *fault-tolerance* e scalabilità è possibile fare riferimento a spazi di tuple distribuiti [14]. Uno spazio di tuple distribuito è un'astrazione che presenta diversi spazi di tuple organizzati adoperando due tecniche comuni dei sistemi distribuiti: la replicazione (consiste nel replicare gli spazi di tuple al fine di aumentare la *fault-tolerance*) e il partizionamento (consiste nel partizionare lo spazio di tuple in diversi insiemi disgiunti al fine di garantire scalabilità e aumento delle prestazioni). L'interazione tramite *tuplespace* garantisce disaccoppiamento spaziale e temporale e permette sia comunicazioni sincrone sia asincrone. Le entità in gioco possono essere del tutto pari per cui l'iniziativa può essere sia *pull-based* che *push-based* e sono possibili molteplicità sia 1 a 1 che 1 ad N.

In tabella 1-1 si riporta un confronto sintetico dei diversi meccanismi di interazione illustrati

	RPC	**Message Queueing**	**Publish Subscribe**	**Tuples Spaces**
Accoppiamento	spazio tempo sincronismo	Tipicamente nessuno	Tipicamente nessuno	Nessuno
Iniziativa	Pull-based	Pull-based	Push-based o ibrido	Pull-based Push-based
Molteplicità	1-1	1-1	1-1 1-N	1-1 1-N

Tabella 1-1: confronto tra i diversi meccanismi di interazione presentati, secondo i criteri di classificazione della fase di *interaction*.

1.4.5 Requisiti delle soluzioni di *service delivery* per ambienti mobili

Tenendo presente le caratteristiche dei sistemi di *mobile computing* (e *nomadic computing* in particolare), nella scelta dei meccanismi di *service delivery* per tali ambienti, non possono essere trascurati i seguenti requisiti funzionali:

Supporto alla mobilità: la mobilità introduce il problema di gestire le connessioni quando i dispositivi si muovono tra aree diverse e/o si

disconnettono in maniera volontaria e non. Le tecniche di *delivery* devono fornire un paradigma di comunicazione e i meccanismi adeguati per permettere a due entità di interagire tra loro senza che la mobilità sia un limite per la consegna dei dati.

Supporto alla dinamicità: in un ambiente di *nomadic computing* il contesto è fortemente dinamico [1]. Il processo di consegna di un servizio può essere influenzato da molteplici fattori come la variabilità della banda, l'inaffidabilità e la latenza del canale e la limitata disponibilità dei dispositivi mobili. Nel progetto delle tecniche di *delivery* è importante che siano introdotti modelli per la rappresentazione del contesto e, di conseguenza, meccanismi per reagire alle sue mutevoli condizioni.

Indipendenza all'implementazione: dato un *client agent* ed un *service agent*, un protocollo di *delivery* permette l'indipendenza (alla implementazione) quando al *client agent* è nascosta la conoscenza della particolare implementazione del *service agent*. Mantenere separate le fasi di *service access description* e *interaction* aiuta a soddisfare questo requisito.

D'altra parte nello sviluppo di un protocollo di *service delivery* è anche necessario tenere in conto i seguenti requisiti non funzionali:

Semplicità: negli ambienti di *nomadic computing* possono essere presenti dispositivi mobili leggeri con scarse capacità. E' necessario che il *software* al supporto del *data delivery* non introduca *overhead* nei dispositivi nomadi.

Eterogeneità: un ambiente di *nomadic computing* vede la presenza dei servizi più disparati; il protocollo di *delivery* deve definire un formato per descrivere l'accesso ai servizi sufficientemente generale e tale da essere adatto ad ogni tipo di servizio e a ogni tipo di interazione.

Scalabilità del meccanismo di delivery: un protocollo di *service delivery* deve adottare scelte che consentano facilmente di aumentare il fattore di

scala per gli ambienti per i quali esso è progettato. In altre parole, è indispensabile che esso non introduca un carico (in termini di traffico generato e *overhead* computazionale) eccessivo al crescere delle dimensioni del sistema.

Affidabilità: il meccanismo di *delivery* deve consentire il recupero dai malfunzionamenti dei dispositivi fissi e mobili senza provocare il fallimento dell'intero sistema. Si tenga presente che le disconnessioni non vanno considerate come un malfunzionamento, ma sono una caratteristica della mobilità.

Sicurezza: utenti maliziosi non dovrebbero avere accesso ai servizi "sicuri". Bisogna prevedere nella definizione del protocollo meccanismi per la garanzia della sicurezza.

Il supporto alla mobilità e alla dinamicità in generale può essere garantito con due approcci contrapposti: l'*application-transparent* o l'*application-aware* [11]. Nel primo caso è completa responsabilità dei meccanismi offerti dal *middleware* di supporto alle applicazioni garantire la trasparenza della mobilità e della dinamicità; nel secondo caso le applicazioni sono consapevoli della mobilità e sono in grado di decidere tra le scelte offerte dal *middleware* in merito all'interazione. Negli ambienti mobili e dinamici è opportuno rendere le applicazioni consapevoli ma è naturale che soluzioni ibride sono possibili. In ogni caso, tecniche di *delivery* che garantiscano disaccoppiamento nello spazio, nel tempo e nel sincronismo si prestano meglio al supporto della mobilità e dinamicità: *client* e *service* non hanno bisogno di avere un riferimento della localizzazione reciproca (essi possono "migrare" da un ambiente all'altro), non è necessario che siano attivi allo stesso tempo (*client* e *server* possono essere momentaneamente inattivi quando è richiesta l'interazione) e un meccanismo di comunicazione asincrono è sicuramente più generale di quello sincrono. Inoltre è bene consentire interazioni sia *push-based* che *pull-based* dato che diverse tipologie di applicazioni possono essere presenti in questi ambienti.

1.4.6 *Service delivery* e *nomadic computing*

Per quanto detto precedentemente, è evidente che i meccanismi di interazione di tipo *message-oriented* o *tuple-oriented* si prestano bene a soddisfare i requisiti funzionali e non delle tecniche di *service delivery* negli ambienti di *nomadic computing* [1], [8], [13]. C'è da osservare però che il supporto a questi tipi di interazione può costituire un carico computazionale piuttosto pesante soprattutto se si fa riferimento alla gestione distribuita e persistente dello "spazio condiviso". Siccome tale carico non può essere sostenuto dai dispositivi mobili che popolano l'ambiente, è compito del *core* dell'infrastruttura di rete a supporto del *nomadic computing* farsi responsabile di tale gestione.

CAPITOLO 2
Soluzioni di *service delivery*

2.1 Premessa

Le soluzioni di *discovery* e *delivery* dei servizi sono uno strumento indispensabile negli ambienti caratterizzati da un'elevata dinamicità e mobilità. Il loro studio sta diffondendo un nuovo paradigma di sviluppo delle applicazioni: si viene a conoscenza del contesto in cui eseguire l'elaborazione, si cerca il o i servizi richiesti, si sceglie quello adatto alla computazione, si negozia l'accesso e finalmente si avvia l'interazione con esso per ottenerne i risultati. Come si evince, il *discovery* vive con il *delivery* e viceversa. Tuttavia, anche se negli ultimi anni si sono diffuse molte soluzioni integrate di *discovery* e *delivery*, non è difficile incontrare soluzioni in cui le due cose vengono mantenute separate:

Possiamo citare soluzioni come *Jini* (di *Sun Microsystem*) e *Salutation* (del *Salutation Consortium*) quali soluzioni integrate, soluzioni come SDP-*Service Discovery Protocol* (del *Bluetooth Special Interest Group*) o SLP-*Service Location Protocol* (del *Internet Engineering Task Force*) quali soluzioni di *discovery* e soluzioni come JMS-*Java Message System* (di *Sun Microsystem*), *WirelessCORBA* (del *Object Management Group*) o *Tspaces* (di IBM) quali soluzioni di *delivery*. Non di meno sono state proposte soluzioni integrate e non anche nell'ambito della ricerca, come *Konark* (di un gruppo di ricerca della università della Florida) LIME-*Linda In Mobile Environment* (di un gruppo di ricerca della università di Washington).

Con attenzione agli aspetti di *service delivery* messi in luce in precedenza, in questo capitolo saranno presentate alcune delle soluzioni integrate per il *discovery* e *delivery* dei servizi in ambiente distribuito:

- *Jini:* la soluzione *Sun* per il *discovery* e *delivery* dei servizi [19].
- *Salutation:* implementa il servizio di *discovery* e protocolli di gestione delle sessioni [21].
- *Konark:* è un protocollo di *service discovery* e *delivery* specificamente progettato per reti *ad hoc peer to peer* [18].

Presenteremo inoltre alcune soluzioni di *service delivery*, pratica implementazione delle tecniche esposte in precedenza:

- *WSDL* e *SOAP* per il *delivery* dei servizi in ambito WEB [23], [24].
- *JMS*: è un *framework* proposto dalla *Sun* per la programmazione di applicazioni *message-oriented* [27].
- *Linda* e *JavaSpaces*: il primo è il capostipite dei sistemi *tuple-oriented*, il secondo è l'implementazione dello spazio di tuple ad oggetti proposto dalla *Sun* [28], [29], [30].

Il loro studio è un naturale punto di partenza per la definizione di una nuova soluzione più adatta agli ambienti di *nomadic computing* e volta alla interoperabilità e alla integrazione di tutti questi approcci.

2.2 *Jini*

Jini [19] è un soluzione completa per il *discovery* e il *delivery* dei servizi che si affida alla mobilità del codice con *RMI* (*Remote Method Invocation*) [20] e alla forza della *platform-indipendence* garantita dalla JVM (*Java Virtual Machine*). Le entità di *Jini* consistono di *services*, *Lookup Service* e *clients*. I *client* sono le entità che richiedono i servizi, i *lookup service* sono le entità che catalogano i servizi disponibili. La figura 2-1 mostra l'architettura *Jini.*

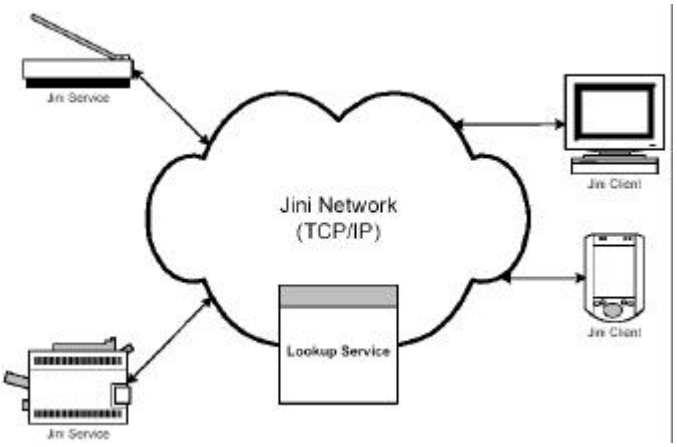

Figura 2-1: un esempio di federazione Jini.

Illustriamo brevemente il processo di *discovery* e registrazione da parte di un *client*/*service Jini*: per registrare un servizio disponibile, o ricercare un servizio, un servizio o un *client* deve prima localizzare uno o più *Lookup Service*.

Una volta che un *Lookup Service* è stato trovato, un servizio può registrarsi presso di esso inviando un *proxy* chiamato *service object*. In *Jini* ogni servizio è associato ad un *service object* per il suo controllo remoto. Il *service object* contiene la descrizione dell'interfaccia per l'utilizzo del servizio, nonché l'implementazione dei metodi che verranno utilizzati dalle applicazioni che lo richiedono.

Per usare un servizio, un *client Jini* deve prima assicurarsi una istanza del *service object* che vuole utilizzare: una volta che un *Lookup Service* è stato trovato, il *client* cerca all'interno di esso un servizio con le caratteristiche richieste ed ottiene, in caso positivo, l'istanza del *service object* di controllo remoto.

Il *service object* può essere implementato in maniera diversa secondo il tipo di servizio che si vuole offrire. Di seguito riportiamo le due scelte più comuni:

- Il *service object* è il servizio stesso: in questo caso il servizio è eseguito completamente all'interno del *client*. Ciò è utile ogni qualvolta non si ha bisogno di comunicare con il *service provider* (ad esempio perché il servizio è rappresentato da un algoritmo scientifico). In questo caso colui che offre il servizio ha solo il compito di rinnovare il *lease* per la

memorizzazione del *proxy* all'interno del *lookup service*, pena la rimozione.

- Il *service object* è un intermediario per la consegna del servizio: esso è inviato al *client* per raccogliere le invocazioni e chiamare i metodi corrispondenti lato *server* tramite RMI oppure tramite una qualsiasi soluzione proprietaria implementata all'interno di esso.

Per la gestione delle interazioni tra *client* e *service*, *Jini* può usare RMI: essa è la soluzione *Sun* per l'invocazione remota di procedure applicata ai contesti *Object-Oriented* distribuiti. Lo scopo di *RMI* è quindi quello di permettere l'invocazione dei metodi di oggetti remoti come se essi fossero locali e senza pertanto alterarne la semantica. Una volta ottenuto il *service object*, esso mette a disposizione i metodi da invocare per accedere via *RMI* alle funzionalità del *service*. Se il *service object* non usa *RMI* o è lo stesso servizio, *RMI* viene comunque utilizzato quando il *client* accede al *Lookup Service*.

Lo sviluppo di un *proxy* lato *client* (quale è il *service object*) tramite la mobilità di codice *Java* è una scelta tecnologica: la possibilità di avere un *server* che, una volta localizzato da un *client* invii il *proxy* necessario all'interazione, senz'altro aumenta la flessibilità dell'ambiente.

2.3 *Salutation*

L'architettura *Salutation* [21] è stata creata per risolvere i problemi di *service discovery* e *delivery* in ambienti pervasivi. Le soluzioni adottate sono indipendenti dal tipo di rete e dal protocollo di comunicazione utilizzato. L'architettura mette a disposizione di applicazioni, servizi e dispositivi, metodi per descrivere e rendere nota la loro presenza all'interno della rete. E' inoltre possibile cercare applicazioni, servizi e dispositivi in base ad attributi e instaurare sessioni di lavoro tra loro.

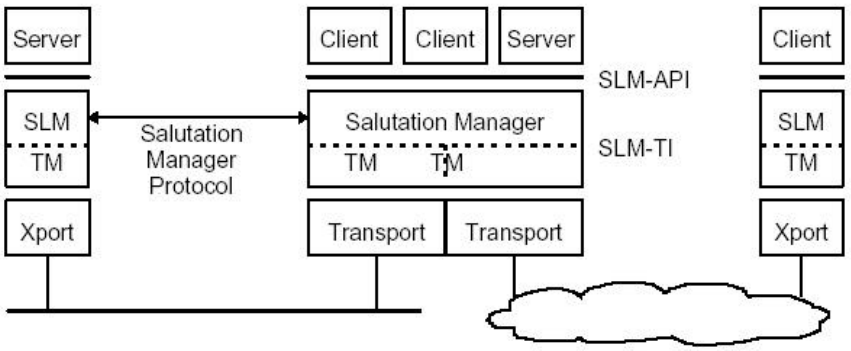

Figura 2-2: l'architettura di *Salutation.*

L'architettura di *Salutation*, riportata in figura 2-2, è composta dalle seguenti entità fondamentali:

- ***Salutation Manager*** (SLM)**:** ha la funzione di essere un *service broker* per le applicazioni. Il servizio deve registrare le sue capacità al *SLM*. Il cliente cerca il servizio e lo usa attraverso il SLM. Per far sì che l'architettura sia *platform-independent* SLM fornisce le *SLM-API.* Ogni SLM comunica con altri SLM per realizzare il suo ruolo di *service broker*.

  SLM fornisce le seguenti quattro funzionalità: *service registry* (SLM contiene un registro che conserva le informazioni sui servizi), *service discovery* (SLM può scoprire altri SLM remoti e determinare i loro servizi registrati), *service availability* (ogni SLM può periodicamente verificare la disponibilità di un servizio), *service session management* (in cui avviene l'interazione tra *client* e *service* e dove il SLM fa da intermediario). Nelle operazioni di *session management*, il SLM stabilisce una *pipe virtuale* tra *client* e *service* mediante la quale i comandi, le risposte e i dati vengono scambiati. I messaggi che viaggiano nella *pipe* hanno un formato definito e sono scambiati secondo protocolli specifici (detti *personality protocol*): esistono tre tipi di *personality protocol*:

  - *Native personality*: il SLM può configurare la *pipe* e poi passa in *background*, permettendo al *client* e al *service* di gestire il flusso dei messaggi e il formato dei dati. I messaggi scambiati "sono dati nativi

in pacchetti nativi". In questo protocollo gli SLM sono usati solo per il *service discovery*.

- *Emulated personality*: il SLM può configurare la *pipe* e gestire il flusso di messaggi, mentre il formato dei dati è scelto e controllato da *client* e *service*. I messaggi scambiati sono "dati nativi in pacchetti *Salutation*", ma il SLM non ispeziona il contenuto o la semantica dei messaggi.
- *Salutation personality*: il SLM può configurare la *pipe*, gestire il flusso di messaggi e fornire il formato dei dati per l'interazione tra *client* e *service*. Tutti i messaggi scambiati con questo protocollo sono trasportati attraverso SLMP (*SLM Protocol*, stabilito da *Salutation*) e sono "dati *Salutation* in pacchetti *Salutation*". In ogni caso il SLM non ispeziona mai il contenuto o la semantica dei messaggi.

- ***Transport Manager*** (TM)**:** i TM isolano l'implementazione degli SLM dal livello di trasporto sottostante, rendono quindi l'architettura *Salutation* indipendente dal protocollo di trasporto. Infatti i TM si occupano di interfacciare gli SLM con i protocolli specifici di ogni rete. Vi sarà un TM per ogni protocollo (TCP/IP, IrDA, *Bluetooth*) e ogni SLM può avere più *TM*.
- ***Functional Unit*** (FU)**:** dal punto di vista di una applicazione *client* una *Functional Unit* definisce un servizio. L'architettura di *Salutation* fornisce una definizione astratta delle FU. *Salutation* definisce la sintassi e la semantica per descrivere gli attributi di ogni FU attraverso l'astrazione di un *record*, chiamato FU *Description Record*. Quando un *client* o una FU vuole usare un *service*, richiede al SLM di stabilire una *service session* tra esso e la FU del *service* richiesto. E' compito del *client* specificare il protocollo di *personality* da usare.

Per rispondere a richieste di *service discovery* tra i *Salutation Manager* è definito un protocollo, detto *capability exchange*. Un *client* può anche non usare il *capability exchange* e passare direttamente alla successiva fase di *service request*

(ovvero *interaction*) se in qualche modo già possiede le informazioni del servizio che vuole usare: una volta che un *client* riceve il *service description record* di un *SLM server*, può partire la comunicazione con il *server* per usare le *FU*. La comunicazione avviene attraverso i seguenti messaggi RPC:

- *Open Service*: il messaggio *Open Service* è usato dal SLM lato *client* per aprire una *service session* tra il *client* e la FU richiesta. Il messaggio contiene informazioni quali la FU con cui stabilire la sessione, il tipo di protocollo di *personality* da usare, informazioni di autenticazione ed altro.
- *Transfer Data*: il messaggio di *Transfer Data call* (conforme alla *personality* scelta) contiene un comando o una risposta inviati tra *client* e FU *server*.
- *Close Service*: quando un *client* ha completato l'uso del servizio, richiede al suo SLM di inviare un messaggio *Close Service* all'altro capo della sessione.

RPC è la soluzione (proposta da *Sun*) per l'invocazione remota di procedure applicata ai contesti *function-oriented* distribuiti. Lo scopo di RPC è quindi quello di permettere l'invocazione di procedure situate su *server* remoti come se esse fossero *locali* e senza pertanto alterarne la semantica.

2.4 *Konark*

Konark [18] è una *service oriented architecture* progettata specificatamente per il *discovery* e il *delivery* dei servizi in reti ad *hoc*. *Konark* è basato su un modello *peer-to-peer* dove ogni *device* partecipante ha le capacità di ospitare i suoi servizi locali, di consegnare i suoi servizi, interrogare la rete per scoprire i servizi offerti ed usare tali servizi.

Gli aspetti importanti di un ambiente mobile *ad hoc* trattati dal progetto *Konark* sono:

- *configurazione automatica dei dispositivi*: in una rete ad *hoc* la partecipazione dei *device* è fortemente dinamica. Non è possibile assumere che vi sia una entità di controllo ad assegnare indirizzi ai nodi. *Konark* adotta meccanismi automatici per la configurazione dei dispositivi al fine di dotare la rete di elevata flessibilità (*autoIP*, *zeroconfIP* ecc.). L'uso di *IP* rende anche l'architettura indipendente dalla tecnologia di rete sottostante;
- *catalogazione dei servizi*: non essendo possibile fare affidamento ad un *repository* centralizzato, *Konark* usa un approccio completamente distribuito; ogni *device* avrà il suo *repository locale* per la gestione dei servizi offerti dal *device* stesso;
- *descrizione dei servizi*: quando un ambiente di rete è caratterizzato dalla presenza di dispositivi *eterogenei* è necessario fornire una descrizione dei servizi che sia indipendente dal dispositivo su cui esso è ospitato e che sia sufficientemente generale da permettere la descrizione dei servizi più disparati. *Konark* definisce un linguaggio di descrizione semplice, ma allo stesso tempo completo, basato su *XML* e ispirato a WSDL (cfr. 2.4) che permette la descrizione di un ampia gamma di servizi. I servizi sono organizzati per classe è attraverso una struttura ad albero in cui la loro classificazione è generica alla radice e più specifica verso le foglie (un nodo di alto livello potrebbe essere "*games*" da cui deriva il nodo "*chess*" che raggruppa i diversi giochi di scacchi disponibili);
- *discovery dei servizi*: *Konark* supporta sia l'*advertisement* che il *discovery* dei servizi. Chi fornisce un servizio lo può annunciare agli altri dispositivi della rete o alternativamente un *client* interessato ad un servizio può effettuarne il *discovery*;
- *delivery dei servizi*: il *delivery* dei servizi è complicato dall'eterogeneità dei dispositivi immessi continuamente dai produttori sul mercato. Per questo motivo *Konark* utilizza degli standard globalmente accettati quali *HTTP* e *SOAP* per la gestione del *delivery*. Per rispondere alle richieste di

delivery, viene fornito un *micro-HTTP server* su ogni *device* e i pacchetti scambiati tra *client* e *server* sono basate su *SOAP*.

Descritte le caratteristiche generali dell'architettura, passiamo a guardare con maggior dettaglio ciò che riguarda il *service delivery*. *Konark* separa nettamente le fasi di *service access description* ed *interaction*. Nella prima fase, *description* nel gergo *Konark*, un *client* viene a conoscenza delle capacità e proprietà del servizio (scoperto in precedenza) e nella seconda, *usage* nel gergo *Konark*, il *client* usa il servizio. Per questo motivo, ogni servizio può essere visto come composto da due componenti: un file di descrizione, ovvero un documento di testo, e un *service object*, che consiste in un *class file* o una *DLL*. Al fine di consentire la descrizione dei servizi, Konark definisce il suo *Service Description Language*. Ogni servizio può essere caratterizzato da un nome, dal tipo di servizio che rappresenta, da una serie di *keyword* (utili nella fase di *discovery*) e da un insieme di proprietà e funzioni. Le proprietà descrivono le caratteristiche del servizio, mentre le funzioni sono relative all'invocazione del servizio: ogni servizio può mettere a disposizione più funzionalità ognuna delle quali è descritta in termini di nome e parametri di scambio. I file *XML* contenenti le informazioni relative ai servizi sono conservati in un *Registry* distribuito su ogni *device* (ognuno di essi mantiene le descrizioni dei propri servizi; ciò è necessario nei sistemi *ad hoc*, data l'assenza di unità elaborative fisse) e che costituisce una parte del *SDP Manager* (*Service Discovery Protocol Manager*), il componente adoperato per il *discovery* dei servizi. Alla fine della fase di *discovery*, un *client* ha poche informazioni sul servizio scoperto: ne conosce il nome, il tipo e l'indirizzo *IP*

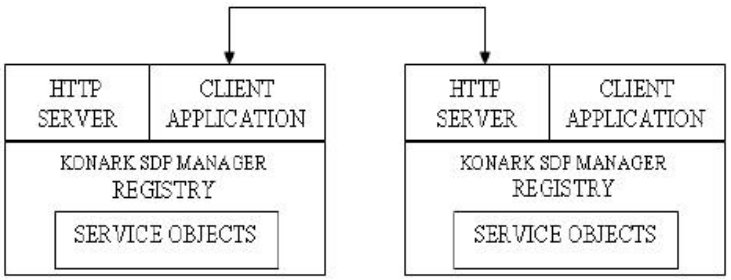

Figura 2-3: i componenti di un *Peer Konark*.

del *device* che lo offre. Basandosi su queste informazioni, il *client* accede al *registry* del *device* lato *server* per ottenere la descrizione dettagliata del servizio. Fatto ciò, il *client* può interagire col servizio invocandone una delle sue funzionalità con i giusti parametri. Le invocazioni sono impacchettate in delle *SOAP request* e spedite al *micro-HTTP server*; parimenti il risultato è ritornato al *client* attraverso delle *SOAP response*. La figura 2-3 mostra le entità di un *peer Konark* coinvolte nella fase di *delivery*.

2.5 *WSDL* e *SOAP* per il *delivery* dei servizi in ambito WEB

Il *framework* per lo sviluppo dei *web services* è diviso in tre aree [26]: protocolli di comunicazione (*bind*), descrizione dei servizi (*publish*) e *discovery* dei servizi (*find*). Ognuna di queste area vede lo sviluppo di specifiche diverse ed in particolare:

WSDL (*Web Services Description Language*) [23] è un linguaggio per consentire ai *service provider* di descrivere le specifiche per l'invocazione dei *web*

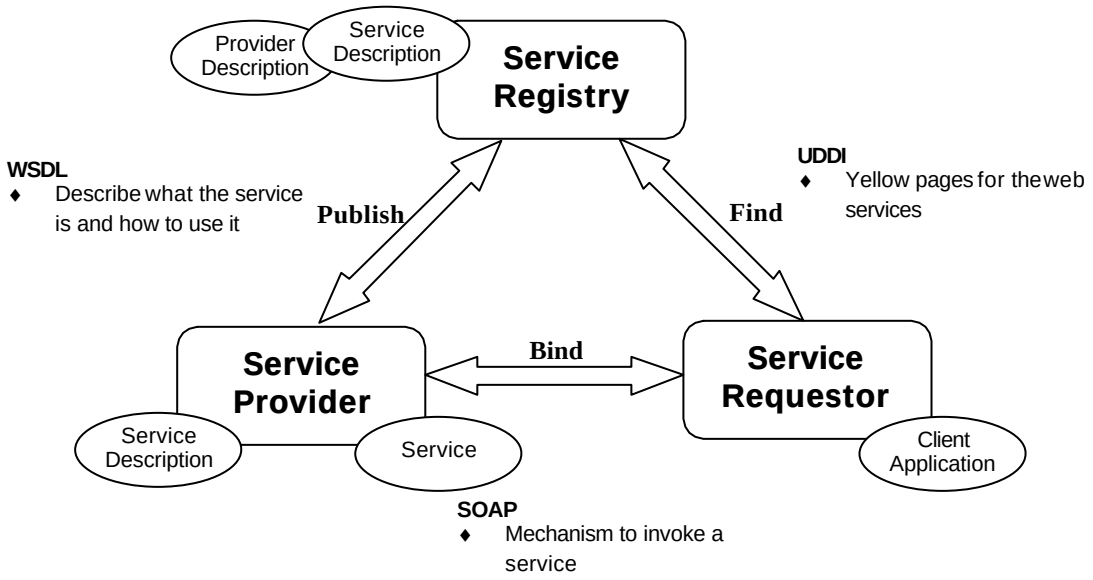

Figura 2-4: il modello per lo sviluppo dei *web services.*

service da parte dei *Service Requestor*.

SOAP (*Simple Object Access Protocol*) [24] è un protocollo per lo scambio di messaggi *XML* tra le applicazioni e fornisce un semplice meccanismo di interazione negli ambienti distribuiti.

UDDI (*Universal Discovery Description and Integration*) [25] rappresenta un *repository* dei servizi *Web*; esso permette di localizzare e conoscere i servizi offerti dai *web services* iscritti come se fossero delle pagine gialle residenti sul *Web*.

L'operazione di *service access description* fatta con *WSDL* e combinata con l'infrastruttura sottostante fornita da *SOAP*, permette il *delivery* dei servizi in ambito *WEB*. La soluzione di *discovery* di *UDDI* (*Universal Description and Discovery Integration*) combinata a sua volta con *WSDL* e *SOAP* rappresenta un *framework* di sviluppo dei *web service*, ovvero una *service oriented architecture* per il *discovery*/*delivery* delle applicazioni distribuite in *Internet* (così come riportato in figura 2-4).

Vediamo in dettaglio *WSDL* e *SOAP*:

WSDL: *WSDL* fornisce uno strumento per la descrizione formale dei servizi tale da consentire la generazione automatica di *proxy*/*stub* per l'accesso ad essi, rappresenta quindi lo strumento attraverso il quale implementare l'operazione di *service access description*.

Un documento *WSDL* descrive un servizio separando tra la descrizione della sua interfaccia (*Service Interface Description*) e la descrizione della sua implementazione (*Service Implementation Description*). La prima contiene le informazioni necessarie per invocare il servizio ed è costituita dai seguenti elementi:

- *Types*: definisce i tipi utilizzati all'interno dei messaggi del servizio.

- *Message*: esistono più sezioni *message* in un documento *WSDL*. Esse contengono le definizioni dei messaggi di *input* e *output* scambiati tra *client* e *service*.

- *PortType*: definisce le operazioni, ovvero i metodi, esposte dal *web service*. Per ogni operazione (o metodo) viene definito il messaggio di *input* ed il messaggio di *output*.

- *Binding*: contiene il collegamento tra il *portType* (cioè la definizione astratta del servizio in termini dei metodi offerti) all'*end-point* (il punto di fornitura fisico del servizio *web*). Queste sono le informazioni che indicano il protocollo da utilizzare e come ricondurre i messaggi di *input* ed *output* al protocollo utilizzato (tipicamente *SOAP*).

La seconda sezione contiene le informazioni relative alla concreta locazione del servizio e comprende gli elementi:

- *Service*: la sezione *service* descrive il *web service* come una collezione di *port* (tipicamente uno), detti anche *endpoints.*

- *Port*: contiene una breve descrizione del servizio e della sua posizione fisica (tipicamente il suo *URL*). Un *web service* può avere più punti di accesso (*ports* ovvero *endpoints*).

Le operazioni supportate dal servizio (*portType*) possono essere implementate in modo diverso:

- *one-way*: l'*endpoint* riceve un messaggio inviato dal *client*. I dati vengono scambiati sono in un senso, dal *client* verso il *service*;

- *request-response*: l'*endpoint* riceve un messaggio di richiesta, esegue le operazioni relative e fornisce al *client* una risposta inviando un messaggio correlato alla richiesta ricevuta;

- *solicit-response*: è l'opposto del precedente. Qui è l'*endpoint* che invia una notifica al *client* ed in seguito attende da questo una risposta correlata;

- *notification*: la notifica prevede l'invio da parte dell'*endpoint* di un messaggio. E' l'opposto dell'*one-way* e può essere utile per implementare sistemi *push-based*.

***SOAP*:** *SOAP* viene utilizzato per l'interazione con i servizi in ambito *Web*, ma dato che esso definisce solo la struttura del messaggio ed alcune regole per elaborare quest'ultimo, rimane ad un alto livello di astrazione e completamente indipendente dal protocollo di trasporto sottostante. Questo significa che *SOAP* può essere adottato per l'*interaction* tra due entità che non siano necessariamente *web client* e *web service*; tuttavia l'adozione di *http* (nelle sue maggiori implementazioni) come protocollo di trasporto sottostante ha fatto di questo protocollo lo standard *de facto* per la comunicazione dei *web services*.

SOAP si basa su un modello di *request/response* dei messaggi; un messaggio *SOAP* è un documento *XML* costituito dalle seguenti parti:

- *Header* (opzionale): può essere usato per trasportare informazioni ausiliarie ed è un *building block* per aggiungere nuove caratteristiche al messaggio *SOAP* (informazioni di autenticazione, controllo per le transazioni, …);

- *Body* (obbligatorio): contiene i dati passati dal richiedente al servizio;

- *Envelope* (obbligatorio): costituisce l'elemento principale e rappresenta il messaggio all'interno del quale si trova il *Body*. L'*Envelope* contiene quindi le informazioni necessarie al destinatario del messaggio e le informazioni sulla struttura e i contenuti dell'oggetto remoto richiesto

SOAP supporta due forme di interazione tra *client* e *service*: *remote invocation* (RPC) e *message oriented*: nel primo caso il corpo (*Body*) del messaggio *SOAP* contiene i parametri e i risultati delle funzionalità riportati in *XML*; nel secondo

caso il corpo del messaggio *SOAP* contiene un documento *XML* che rappresenta il contenuto del messaggio. La scelta tra un meccanismo di interazione di tipo *call/return* ed uno *publish/subscribe* è condizionata dalla scelta del protocollo di trasporto.

2.6 *JMS*

Java Message Service [27] è una parte della *J2EE* di *Sun* che fornisce un insieme standardizzato di *API* che i *client message-oriented Java* possono adoperare indipendentemente dal sistema di *messaging* sottostante, così come evidenziato nella figura 2-5.

In questo modo, non solo le applicazioni sono portabili su differenti macchine e sistemi operativi (grazie a *Java*), ma anche su differenti sistemi di *messaging*. *JMS* fornisce entrambe le modalità di comunicazione previste dai sistemi *message-oriented*, e cioè sia comunicazioni attraverso *message queueing* o *point-to-point*, sia attraverso *publish-subscribe*:

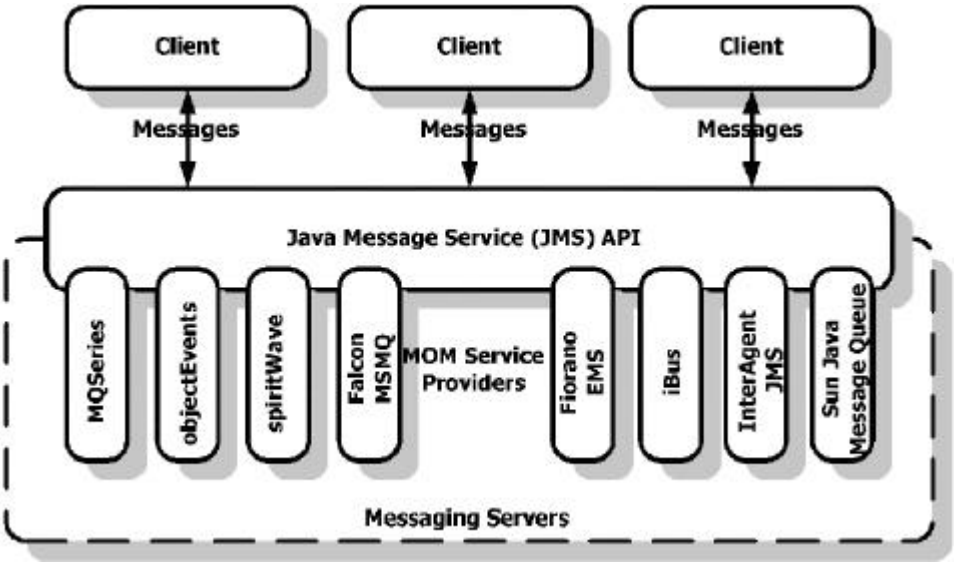

Figura 2-5: la soluzione di *messaging* proposta da *Sun.*

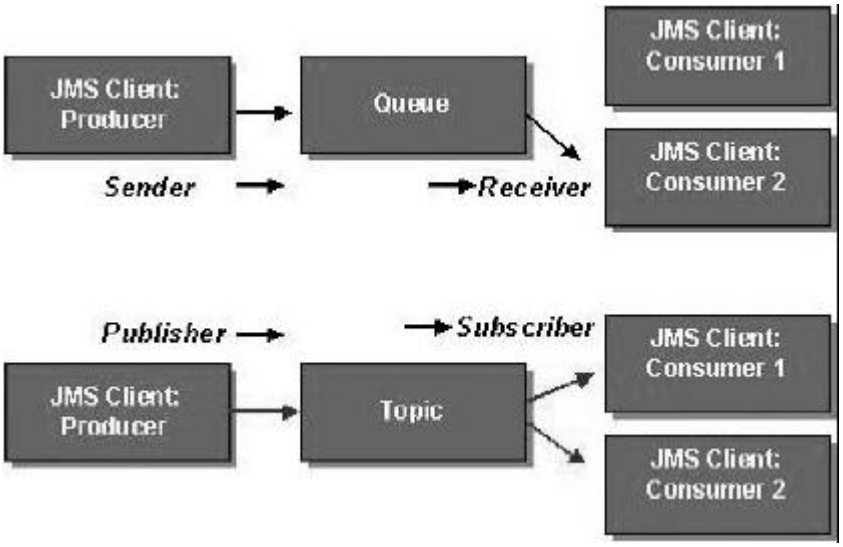

Figura 2-6: le modalità di *messaging* offerte da *JMS*.

Con riferimento alla figura 2-6, le entità interagenti in *JMS* possono essere distinte in *producer* e *consumer*. Un *producer* è l'entità che scrive un messaggio in una coda, un *consumer* è l'entità che legge un messaggio da una coda. Quando l'interazione è di tipo *point-to-point*, la coda viene creata dal *producer* e rappresenta la destinazione dei messaggi che scrive. Più *consumer* possono leggere dalla coda simultaneamente, ma nel momento in cui un messaggio è letto, questo è rimosso dalla coda. Nel caso di comunicazione di tipo *publish-subscribe*, il *producer* crea una coda di messaggi denominata *topic* e in cui scrive i propri messaggi (*publish*). I *consumer* si devono registrare (*subscribe*) ad un *topic*, che provvede alla consegna dei messaggi inviati dal *producer* a tutti i *consumer* registrati. Diversamente a quanto accade nella comunicazione *point-to-point*, nel caso di paradigma *publish-subscribe* un messaggio non viene rimosso quando letto dal *topic*. I messaggi in *JMS* sono costituiti da tre parti: un *header* per identificare il messaggio, delle proprietà utili per costruire "filtri" a livello applicativo, e un corpo che comprende il contenuto del messaggio. Il corpo può contenere sia *textmessage* che *objectmessage*. Nei primi le stringhe possono essere utili per implementare interazioni *publish-subscribe* di tipo *content-based* (i messaggi sono discriminati in base al loro contenuto); i secondi sono costituiti da oggetti *Java* (e consente quindi numerose possibilità per fare migrazione di

codice). In quest'ultimo caso i messaggi sono dotati di un tipo, per cui è possibile realizzare interazioni *publish-subscribe type-based*.

2.7 *Linda* e *Javaspaces*

In questo paragrafo ci occuperemo della descrizione di soluzioni di *delivery tuple-oriented*, *Linda* e *Javaspaces*: il primo definisce nei fatti il paradigma di comunicazione *tuple-oriented*, mentre il secondo rappresenta una implementazione orientata agli oggetti di tale paradigma.

Linda [28] fu proposto come un nuovo modello di comunicazione tra processi concorrenti, ed ha posto nella pratica le radici del paradigma di interazione basato sulle tuple. Più che una tecnologia, *Linda* rappresenta un linguaggio di *coordinazione* tra processi: i suoi promotori, *Gelernter* e *Carriero* in [29], sostengono che un modello di programmazione può essere visto come composto da due parti: il modello di *computazione* e il modello di *coordinazione*. Il primo permette ai programmatori di costruire una singola attività di calcolo. Il secondo è il collante che lega tra di loro diverse attività. Per ognuno dei due modelli vanno definiti dei linguaggi: il linguaggio di programmazione e il linguaggio di coordinazione. I due linguaggi possono essere integrati in un singolo linguaggio oppure possono essere mantenuti separati: questo secondo approccio è preferito dai sostenitori di *Linda* in quanto consente ad applicazioni scritte adoperando linguaggi di programmazione differenti, di interagire grazie alla presenza di un linguaggio di coordinazione comune come *Linda*. Il modello di interazione basato sulle tuple proposto da *Linda*, prevede che una tupla sia una struttura dati costituita da una lista di parametri tipati contenente le informazioni delle comunicazioni tra i processi. Una tupla *t* può essere aggiunta allo spazio di tuple, con l'operazione *out(t)*, può essere letta da esso, con l'operazione *rd(p)*, e può essere rimossa con l'operazione *in(p)*. Le tuple sono anonime (anche se non è difficile renderle identificabili, aggiungendo i campi mittente e destinatario) e la

loro lettura o rimozione avviene attraverso il *matching* del loro contenuto: il parametro *p* rappresenta un *template* che specifica in parte la tupla che si desidera leggere e che contiene una serie di parametri attuali e formali. I parametri attuali sono dei valori ben specificati mentre quelli formali sono da intendere come *jolly* nel *matching*. Le operazioni di lettura e rimozione sono bloccanti, ma sono anche definite operazioni non bloccanti quali la *rdp(p)* e *inp(p)*.

Implementazioni di *Linda* sono state previste per essere affiancate a diversi linguaggi di programmazione, quali *C*, *Fortran*, *Java* e altri, utilizzando precompilatori, compilatori modificati oppure librerie esterne. Esistono poi diverse soluzioni di *delivery* il cui meccanismo di scambio delle informazioni deriva direttamente da *Linda*, quali *LIME* [22] (l'adattamento di *Linda* al *computing* ad *hoc*), *Tspaces* di *IBM* e *Javaspaces* della *Sun Microsystems.*

JavaSpaces [30] è la specifica di uno spazio di tuple *object-oriented* basato si *Java* ed è implementato come servizio all'interno dell'architettura *Jini* per risolvere due problemi correlati: la persistenza delle informazioni e l'implementazione di algoritmi distribuiti. In *Javaspaces* le tuple, dette *entry*, sono esse stesse dotate di tipo, e non solo i loro campi, e ciò consente una tipizzazione molto più spinta di quella fornita da *Linda*. Infatti, le *entry* sono degli oggetti *Java* e pertanto possono inglobare in sé sia lo stato che il comportamento. Inoltre *Javaspace* consente un *matching* delle *entry* di tipo polimorfo, secondo cui una operazione di lettura o rimozione può ritornare una *entry* il cui tipo è una sottoclasse della classe specificata nel *template.* Un'altra principale differenza fra *Linda* e *JavaSpaces* riguarda i meccanismi di sincronizzazione. *JavaSpaces* utilizza prevalentemente la notifica di eventi distribuiti, mentre in *Linda* si sfrutta il fatto che le primitive di consumo possono essere bloccanti e non bloccanti. Altre differenze riguardano il numero di spazi disponibili (in genere uno solo per *Linda* e diversi per *JavaSpaces*) e il *life-time* di *entry* e tuple che può essere limitato in *JavaSpaces*, ma non in *Linda*. Come anticipato, *JavaSpaces* è orientato verso due categorie di problemi: la persistenza distribuita ed il progetto di algoritmi distribuiti. La persistenza distribuita è la capacità di memorizzare

oggetti e di renderli reperibili. Gli algoritmi distribuiti sono modellati come flussi di oggetti tra i partecipanti; un'applicazione è vista come una collezione di processi che cooperano attraverso un flusso di oggetti che entrano ed escono da uno o più spazi.

2.8 Analisi delle soluzioni di *service delivery*

In questo capitolo sono state presentate alcune soluzioni di *service delivery*. Al fine di confrontarle tra loro, la tabella 2-1 illustra per ognuna di esse le caratteristiche salienti e il rispetto dei requisiti funzionali cui dovrebbe fare riferimento una soluzione di *delivery* per ambienti di *nomadic computing*.

Come è evidente nessuna offre tutte le caratteristiche richieste o affronta in maniera diretta i requisiti di un ambiente mobile, anche se non sempre il piatto della bilancia (il "SI" o il "NO") pende chiaramente da un solo lato.

Bisogna dire, infatti, che adattando alcune di queste soluzioni ai nuovi ambienti (come indicato in [1], *JMS* [27] o *JavaSpaces* [30] ad esempio), qualche forma di supporto al *nomadic computing* può essere fornito.

Se però si pensa ad una soluzione ad *hoc*, nasce l'esigenza di definire un'architettura nuova il cui progetto sia esplicitamente mirato a soddisfare i requisiti e i vincoli di un ambiente di *nomadic computing*.

Definire una nuova piattaforma *middleware* non basta: resta il problema di interoperabilità tra le soluzioni esistenti e il *middleware* stesso. Progettare una soluzione che mira all'interoperabilità può aspirare a contenere "*la Babele delle tecnologie*", semplificando la vita degli sviluppatori e di chi usa i loro prodotti. Molti sollevano il problema, ma ancora poco è stato fatto (come osservato in [34], [35]).

L'architettura presentata nei prossimi capitoli affronta i limiti delle soluzioni appena analizzate: viene fornito un nuovo *framework* di *discovery/delivery* per applicazioni di *nomadic computing* nonché i meccanismi per un *bridging* trasparente delle tecnologie esistenti.

	Supporto mobilità	Supporto dinamicità	Indipendenza dalla Implementazione	Access Description E Interaction	Tipo Interazione	Accoppiamento
Jini	NO	NO	SI	UNITE	CALL RETURN	TEMPORALE SPAZIALE * SINCRONISMO
Salutation	NO	NO	NO	UNITE	CALL RETURN	TEMPORALE SPAZIALE SINCRONISMO
Konark	NO	NO	SI	SEPARATE	CALL RETURN	TEMPORALE SPAZIALE SINCRONISMO
WSDL SOAP	SI	NO	SI	SEPARATE	CALL RETURN + MESSAGE ORIENTED	SPAZIALE
JMS	SI	NO	SI	SOLO Interaction	MESSAGE ORIENTED	NESSUNO
Linda JavaSpaces	SI	NO	SI	SOLO Interaction	TUPLE ORIENTED	NESSUNO

Tabella 2-1: compendio delle caratteristiche delle soluzioni di *delivery* analizzate.

* Jini e JavaSpaces sono stati valutati separatamente.

Capitolo 3

Introduzione ad Esperanto

3.1 Introduzione

Esperanto è una architettura *middleware* orientata ai servizi per ambienti di *nomadic computing*. In particolare nel suo approccio innovativo al *service delivery* cerca di integrare anche le *Service Oriented Architecture* esistenti.

Il progetto di un tale tipo di architettura è un compito certamente complesso, soprattutto se è fatto nel rispetto dei vincoli presenti e dei requisiti richiesti da questi ambienti.

In questo capitolo saranno illustrati, attraverso una trattazione tipica dei *design pattern* trattati in [17] e [31], gli elementi principali e le caratteristiche salienti dell'infrastruttura di *delivery* Esperanto.

Bisogna dire che in un ambiente dove il contesto è estremamente dinamico la fase di *service delivery* è tipicamente preceduta da una fase di *service discovery*: in essa le applicazioni diventano consapevoli del contesto che le circonda e quindi delle risorse che l'ambiente o gli altri dispositivi del vicinato offrono. Esperanto, quale *Service Oriented Architecture* per ambienti mobili, rappresenta una soluzione completa per il *discovery* e per il *delivery* dei servizi.

3.2 Contesto e motivazioni

Esperanto nasce con l'intento di integrare le tecnologie di *discovery* e *delivery* esistenti in un nuovo *framework* per lo sviluppo di applicazioni per ambienti di *nomadic computing*. L'infrastruttura di rete di questo tipo di sistemi vede una parte *core*, su rete fissa, caratterizzata da un insieme di nodi con buone capacità

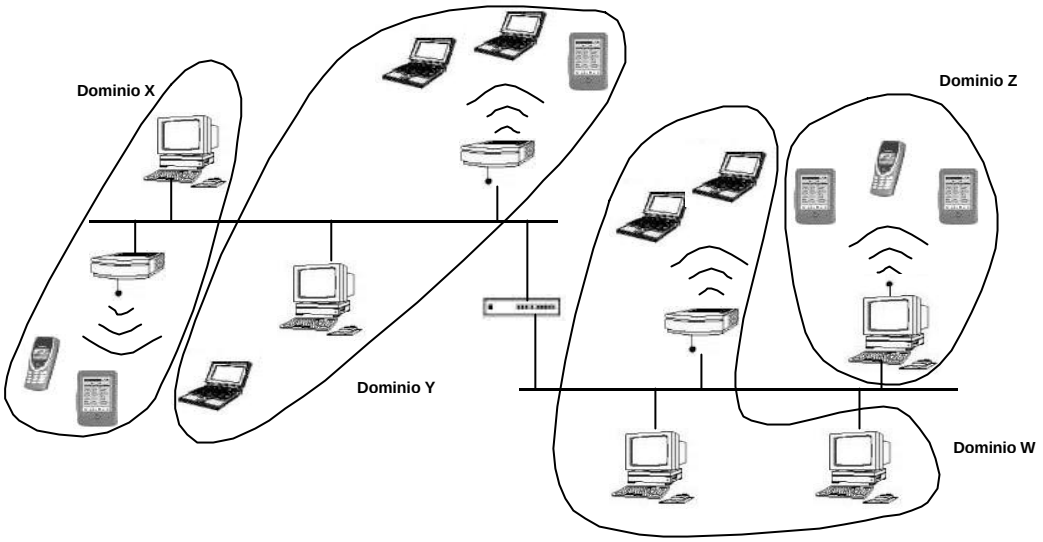

Figura 3-1: un modo di raggruppare gli elementi di un'infrastruttura di *nomadic computing* in domini distinti.

elaborative, che forniscono servizi e che ospitano i *device* mobili. I dispositivi che costituiscono questa rete possono essere raggruppati in domini distinti secondo le tecnologie di *discovery* e *delivery* che adottano. In questo contesto un *dominio* è inteso come l'insieme dei dispositivi che utilizzano una particolare tecnologia di *discovery* e che interoperano tra loro attraverso gli elementi ed i meccanismi introdotti da una comune tecnologia di *delivery* (un dominio ad esempio può essere una federazione *Jini* [19], una *scatternet Bluetooth* [34] o, con l'introduzione di Esperanto, un gruppo di Esperanto *Peer*).

In figura 3-1 è riportato un esempio di una possibile suddivisione di un'infrastruttura di *nomadic computing* in diversi domini.

Secondo la visione Esperanto, l'infrastruttura di rete può quindi essere decomposta in più domini eterogenei, tra cui sono presenti anche domini Esperanto. Ad esempio, i domini X e Z potrebbero essere domini Esperanto, mentre i domini Y e W potrebbero essere *Jini* [19] e *Web Services* [26] rispettivamente. L'architettura mira a superare le incompatibilità che nascono quando applicazioni di soluzioni distinte vogliono comunicare, integrando i domini in un'unica infrastruttura omogenea. Un tale approccio è motivato dalle seguenti considerazioni:

- nessuna delle tecnologie di *discovery*/*delivery* ha la forza necessaria per imporsi nel mercato e sulle altre tecnologie esistenti: il *bridging* sembra quindi essere la prospettiva più promettente per l'interoperabilità tra di esse (come osservato in [34], [35]);
- nessuna delle tecnologie di *discovery*/*delivery* più note nasce con l'esplicito intento di rispondere alle esigenze dei sistemi di *nomadic computing* e di rispettarne i vincoli.

La nuova architettura fornisce, da un lato, la possibilità di far interoperare in maniera trasparente due o più domini eterogenei, dall'altro, (con l'introduzione di un nuovo dominio, quello Esperanto) mira a soddisfare i requisiti e i vincoli imposti da un ambiente di *nomadic computing*.

3.3 Scenari

Nel seguito saranno illustrati, come spesso riportato in letteratura ([17]), i "*motivating scenarios*":

3.3.1 Scenario I

Dispositivi che integrano differenti tecnologie di *discovery* e di *delivery* non possono parlare tra loro in maniera nativa. Questo succede ad esempio quando un elettrodomestico *Jini* [19] (una lavatrice, ad esempio) cerca un servizio di *messaging* che risiede solo su di un palmare *Bluetooth* [34]. In questo caso la ricerca del servizio da parte dell'agente "domotico" è rivolta verso gli altri dispositivi *Jini* che si inscrivono, con i loro servizi offerti, al *lookup registry* della federazione/dominio *Jini*. L'assenza di un *Information Communication Center*, ad esempio, fa fallire il tentativo da parte della lavatrice di avvisare il suo proprietario che il ciclo di lavaggio è terminato oppure che è il momento di inserire l'ammorbidente. Se esistesse un modo per notificare alla lavatrice che il

palmare (*Bluetooth*) del suo proprietario è comunque presente nell'ambiente e che il servizio è offerto ad essa, allora per la lavatrice tutto è visto come un unico dominio *Jini*.

Questo è solo uno dei numerosi esempi, che possono venire in mente quando si pensa all'interoperabilità tra dispositivi dotati di caratteristiche di *spontaneous configurability* e *context-awareness* ma che integrano tecnologie diverse e concorrenti.

3.3.2 Scenario II

Un'esigenza sentita è senz'altro quella di far parlare tra di loro dispositivi di "lingua" diversa in maniera totalmente trasparente.

Un'altra esigenza è quella di permettere ai *device* con spiccate doti di mobilità di ospitare applicazioni che sappiano come gestire le caratteristiche e i vincoli di un ambiente in movimento.

Già da qualche anno la presenza degli *e-book* (dispositivi in grado di memorizzare un testo in formato elettronico, e presentarlo al lettore come se fosse un vero libro fatto di tanti fogli) è sempre più diffusa nel mercato dell'*Information Technology*. Immaginiamo allora di essere impiegati di un'azienda di sviluppo *software* e di voler "scaricare da *Internet*" sul nostro libro elettronico il documento di specifica dell'ultima trovata tecnologica di un consorzio di colossi dell'elettronica. Per consentire il *download* del documento è necessario dapprima che l'*e-book* trovi il servizio di *downloading* e successivamente che attenda il *delivery* del digitale. Se però avviamo il *download* prima della pausa pranzo, non vorremmo lasciare il libro in ufficio perché connesso al *server* aziendale, ma andare al parco, magari per continuare a leggere un romanzo del nostro autore preferito. Inoltre non vogliamo preoccuparci del completamento del *downloading*, desideriamo che il nostro dispositivo lo faccia per noi: quando abbandoniamo il nostro ufficio, l'*e-book* sospende il *download*; quando rientriamo nell'ambito aziendale, eventualmente anche in un'altra sede, lo completa con tutta tranquillità.

Lo scenario descritto mette in luce come sia necessaria la definizione di una nuova tecnologia che permetta la realizzazione di applicazioni capaci di funzionare anche al variare delle condizioni circostanti.

3.4 Problema

Una caratteristica comune agli ambienti di *mobile computing* è la forte dinamicità del contesto; i dispositivi mobili possono disconnettersi e riconnettersi al sistema (volontariamente e non), migrando da una posizione ad un'altra e le risorse a disposizione cambiano rapidamente. Come osservato al par. 1.4.2, le tecniche di *service delivery* classiche non sono adeguate per lo sviluppo di applicazioni in questi ambienti emergenti: il modo migliore per soddisfare requisiti e vincoli come la mobilità, la scarsa capacità elaborativa dei dispositivi, l'inaffidabilità delle connessioni, ecc. è dunque quello di progettare una soluzione nuova ed innovativa.

D'altra parte, se da un lato bisogna definire una nuova soluzione, dall'altro è necessario garantire l'interoperabilità con le soluzioni di *delivery* e *discovery* che sono adottate nei differenti domini in cui l'infrastruttura di *nomadic computing* può essere decomposta.

Un *middleware* che guardi a questi aspetti, deve quindi sia fornire gli elementi che facciano da collante tra i diversi domini, sia definire gli elementi di un nuovo dominio, quello Esperanto, per rispondere alle esigenze delle applicazioni di *nomadic computing*.

Pertanto, come si usa dire nell'ambito della *pattern community*, "*…the architecture has to balance the following forces…*" (intendendo con "forza" ogni aspetto del problema che dovrebbe essere considerato prima di risolverlo, come requisiti da adempiere, vincoli da considerare e proprietà desiderabili da conferire alla soluzione [17]), ovvero il progetto della architettura deve mitigare le seguenti forze:

- Scarse capacità dei dispositivi (*light-weight device*)
- Elevata mobilità dei dispositivi (*mobility*)
- Dinamicità del contesto di elaborazione (*dinamicity*)
- Eterogeneità delle tecniche di *discovery* e *delivery*

Facendo fronte ad esse con le seguenti caratteristiche:

- Introdurre un carico computazionale leggero (*light-weight computational load middleware*)
- Supportare un meccanismo di interazione disaccoppiato (*decoupled communication*)
- Dotare le applicazioni di consapevolezza del contesto elaborativo (*adaptation context-aware*)
- Fornire meccanismi di *bridging* delle tecnologie esistenti

In questo lavoro di tesi l'attenzione è quindi rivolta allo sviluppo di una nuova piattaforma *middleware service oriented* caratterizzata da due aspetti fondamentali:

L'interoperabilità degli approcci al *service delivery* delle SOA esistenti: un qualsiasi *client agent* di un dominio X deve poter utilizzare i servizi offerti da un qualsiasi *service agent* di un dominio Y in maniera trasparente. X ed Y possono SOA diverse da Esperanto: l'interazione deve avvenire in maniera inconsapevole alle applicazioni preesistenti all'architettura. In altre parole un *client agent* del dominio X (non Esperanto) deve poter interagire con i servizi offerti dal *provider* del dominio Y (qualsiasi) utilizzando gli stessi meccanismi di *delivery* del suo dominio, senza sopportare direttamente alcun carico aggiuntivo per la gestione della comunicazione.

L'introduzione di meccanismi di *delivery* adatti agli ambienti di *nomadic computing*: come osservato al par. 1.4.5, una soluzione di *service delivery* per ambienti di *nomadic computing* deve essere progettata nel rispetto dei seguenti requisiti funzionali:

- *Supporto alla mobilità*: la mobilità dei dispositivi non deve costituire un limite al loro utilizzo, anzi, deve aumentare l'inventiva degli sviluppatori e le possibilità delle applicazioni.
- *Supporto alla dinamicità*: le mutevoli condizioni del contesto possono ridurre il beneficio che si ottiene utilizzando un servizio remoto. Affinché un servizio sia veramente di qualità bisogna che nel *delivery* si assuma un comportamento che reagisca alla dinamicità.
- *Indipendenza all'implementazione*: in ambienti molto eterogenei, separare la specifica di un servizio dalla sua implementazione e dalle procedure di interazione aiuta ad aumentare il numero di clienti che possono utilizzarlo.

Anche i requisiti di qualità non dovrebbero essere trascurati:

- *Semplicità*
- *Eterogeneità*
- *Scalabilità*
- *Affidabilità*
- *Sicurezza*

I benefici di questo approccio innovativo al *delivery* del servizi è naturalmente offerto agli sviluppatori di applicazione tramite un insieme di *API* (*Application Program Interface*) che i *client agent* e *service agent* possono utilizzare per comunicare.

Poiché la fase di *service delivery* può essere decomposta in due fasi (cfr. 1.4.3), l'insieme di primitive è decomposto in:

- ***Service Access Description API***: un insieme di *API* che un *client agent* può invocare per ottenere la descrizione dell'accesso ad un servizio.
- ***Service Interaction API***: un insieme di *API* che *client* e *service agent* possono usare per interagire.

3.5 Soluzione

Il progetto di Esperanto è stata guidato da tutti i requisiti funzionali e parte dei requisiti non funzionali definiti al par. 3.4. Nel seguito verrà illustrato come ognuno di essi è stato affrontato e quale soluzione è stata proposta per soddisfarli.

3.5.1 Supporto alla mobilità

3.5.1.1 PARADIGMA DI COMUNICAZIONE

Perché un protocollo di *delivery* supporti la mobilità dei dispositivi deve prevedere che nelle interazioni tra due entità si adotti un paradigma di comunicazione disaccoppiato nello spazio, nel tempo e nel sincronismo. In questo modo la non disponibilità di un *device* (dovuta all'inaffidabilità della rete, alle sue scarse capacità di alimentazione oppure alla sua disconnessione volontaria dall'ambiente) non rappresenta una situazione di malfunzionamento (entro un certo tempo prestabilito).

In Esperanto si assume che le applicazioni del dominio da esso definito non comunichino direttamente tra loro, ma attraverso un intermediario residente sul *core* dell'infrastruttura di *nomadic computing*: il *Mediator*. Quando due applicazioni Esperanto vogliono interagire tra loro (l'una per offrire un servizio, l'altra per utilizzarlo), lo fanno inviando *service request* e *service response* tramite il *Mediator* del dominio Esperanto: lo scambio dei dati tra le applicazioni avviene comunicando sempre e solo attraverso il mediatore. In questo modo si garantisce il disaccoppiamento nello spazio.

Il disaccoppiamento nel tempo e nel sincronismo viene garantito assumendo che le primitive di supporto alla comunicazione con il *Mediator* siano asincrone e non bloccanti, ovvero che invocandole, una applicazione non si blocchi nell'attesa

di una risposta. Prevedendo anche primitive bloccanti, la flessibilità nella comunicazione può essere estesa con la costruzione di primitive sincrone.

Con questo paradigma di interazione due applicazioni residenti su due dispositivi mobili possono interagire senza vincoli: i dati sono localizzati indipendentemente dai movimenti di un *device*, *client* e *service* non devono essere attivi contemporaneamente e non è necessario che si sincronizzino nelle loro elaborazioni.

3.5.1.2 DISPOSITIVI MOBILI E LE MIGRAZIONI

Come osservato al par. 3.2, l'infrastruttura di rete può essere decomposta in più domini eterogenei. In questo contesto è lecito pensare che ci siano più domini Esperanto (anche distanti tra loro) e un dispositivo mobile che voglia migrare da un dominio Esperanto all'altro.

Il problema delle migrazioni si apre ogni qualvolta un ambiente viene tassellato in tanti domini (distinti dalla loro posizione fisica) in cui un'entità mobile è libera di muoversi utilizzando le risorse del dominio in cui attualmente si trova.

Quando un dispositivo migra da un dominio all'altro, vorrebbe continuare le elaborazioni in corso (l'insieme di *service request* e *service response*). Il problema della consegna dei dati al dispositivo giusto nel dominio giusto è risolto tramite l'adozione di un mediatore per ogni dominio Esperanto (connesso all'infrastruttura di rete) e tramite la cooperazione tra tutti i mediatori.

Grazie ai *Mediator* è consentita l'interazione tra applicazioni residenti in domini Esperanto distinti e interazioni tra applicazioni eseguite su dispositivi in migrazione tra dominio Esperanto distinti.

3.5.2 Supporto alla dinamicità

Le mutevoli condizioni del contesto in cui vive l'elaborazione distribuita tra due applicazioni possono incidere negativamente sulle aspettative di un utente quando egli fruisce del servizio (si pensi ad un *web browser* per PDA e al *downloading* di un'immagine: la fluttuazione delle caratteristiche del canale potrebbe essere nascosta con la richiesta di una versione degradata dell'immagine). Affinchè un protocollo di *delivery* supporti la dinamicità dell'ambiente deve fornire alle applicazioni dei meccanismi per conferire reattività durante le loro interazioni.

Molte volte, inoltre, le variazioni del contesto possono dipendere anche dal numero di dispositivi che lasciano o entrano nel sistema. Intraprendere un'interazione con un dispositivo che abbandona l'ambiente non deve costituire un malfunzionamento. L'introduzione di un paradigma disaccoppiato in cui l'interazione avviene attraverso un componente residente sul *core* della rete, aiuta a gestire situazioni anomale per i normali sistemi di elaborazione distribuiti: in un ambiente di *mobile computing* potrebbe non essere inusuale che un dispositivo si disconnetta dal sistema (volutamente o non) durante un'interazione. La presenza di un intermediario dà migliori garanzie al successo dell'interazione e consente di implementare azioni di ripristino da situazioni di malfunzionamento (ad esempio un dispositivo che voglia completare l'interazione dopo una disconnessione, può farlo interagendo con il mediatore, anche se la sua controparte non è più attiva).

3.5.3 Indipendenza dalla implementazione

Perché un protocollo di *delivery* supporti l'indipendenza dall'implementazione (di un servizio) deve prevedere che tra la definizione del servizio e la sua realizzazione ci sia un confine ben delineato; in questo modo un *client* può usare allo stesso modo più implementazioni del servizio senza che ne abbia consapevolezza. In Esperanto l'indipendenza all'implementazione viene garantita separando la fase di *delivery* in *service access description* e *interaction*: nella

prima, il *client* ottiene una descrizione delle caratteristiche del servizio nella seconda, una volta istruito, interagisce sulla base della conoscenza acquisita.

Altri elementi guida del progetto dell'architettura Esperanto sono i seguenti:

3.5.4 Interoperabilità delle soluzioni di *delivery* esistenti

Dati due domini distinti (*Jini* [19] e *Bluetooth* [34] oppure *Jini* e *Esperanto,* ad esempio), si vuole che le applicazioni sviluppate per questi domini interagiscano tra loro in maniera trasparente. Tipicamente l'interoperabilità tra tecnologie incompatibili avviene con l'uso dei *bridge*, tuttavia la loro adozione è trasparente se essi sono localizzati all'esterno di ogni dispositivo che si vuole far interoperare.

Visto che l'approccio dell'architettura non è invasivo (le applicazioni non devono sostenere alcun carico aggiuntivo per interoperare), ogni dominio viene

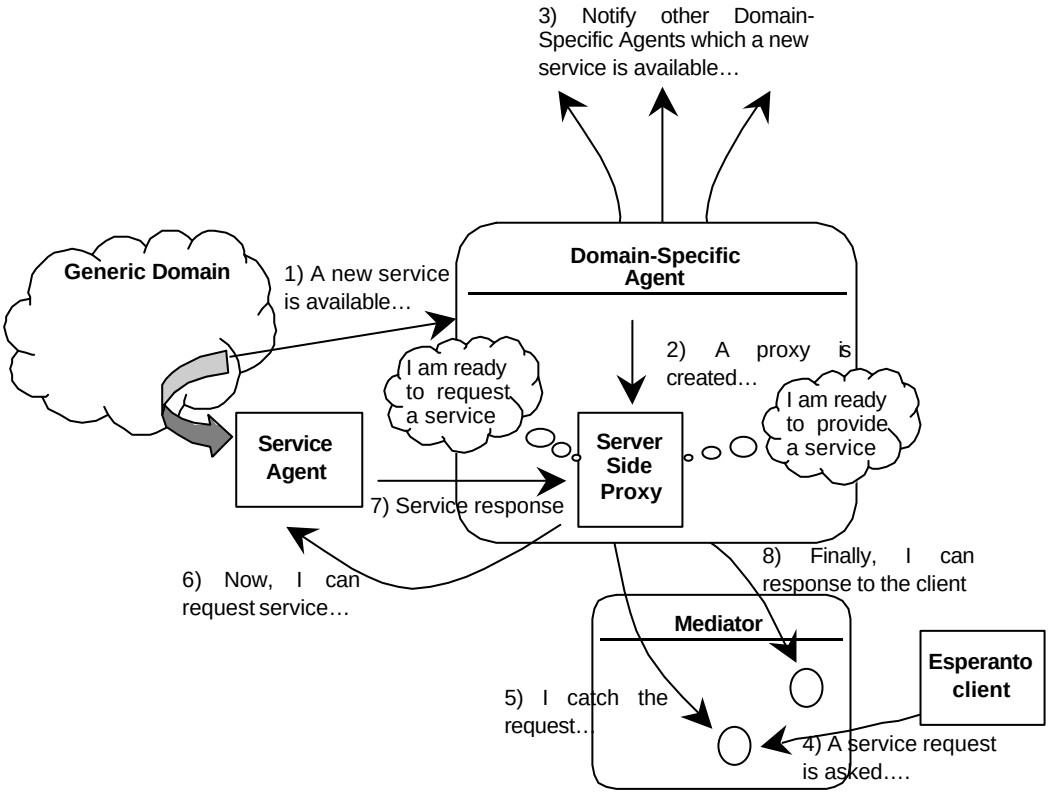

Figura 3-2: Il *Domain-Specific Agent* e il meccanismo per garantire l'interoperabilità tra un servizio del suo dominio di competenza ed un cliente Esperanto.

dotato di un componente localizzato sul *core* di rete, l'*Esperanto Agent* (per i domini Esperanto) o il *Domain-specific Agent* (per i domini generici). Questi componenti non si limitano solo ad espletare i singoli compiti di un *bridge.*

Per consentire l'interoperabilità tra i clienti Esperanto e un servizio pubblicato nel suo dominio di competenza, l'agente *Domain-specific* ha l'incarico di emulare, da un lato il comportamento di un *client* compatibile con il dominio, dall'altro, il comportamento di un servizio Esperanto (cfr. figura 3-2).

L'interoperabilità tra un cliente qualsiasi e un servizio pubblicato nel suo dominio di competenza comporta inoltre che l'agente *Domain-specific* segnali ad ogni suo pari ogni nuovo servizio pubblicato. L'agente notificato di ciò dovrà emulare (quando richiesto dal cliente), da un lato il comportamento di un *service* compatibile con il suo dominio, dall'altro il comportamento di un cliente Esperanto (cfr. figura 3-3).

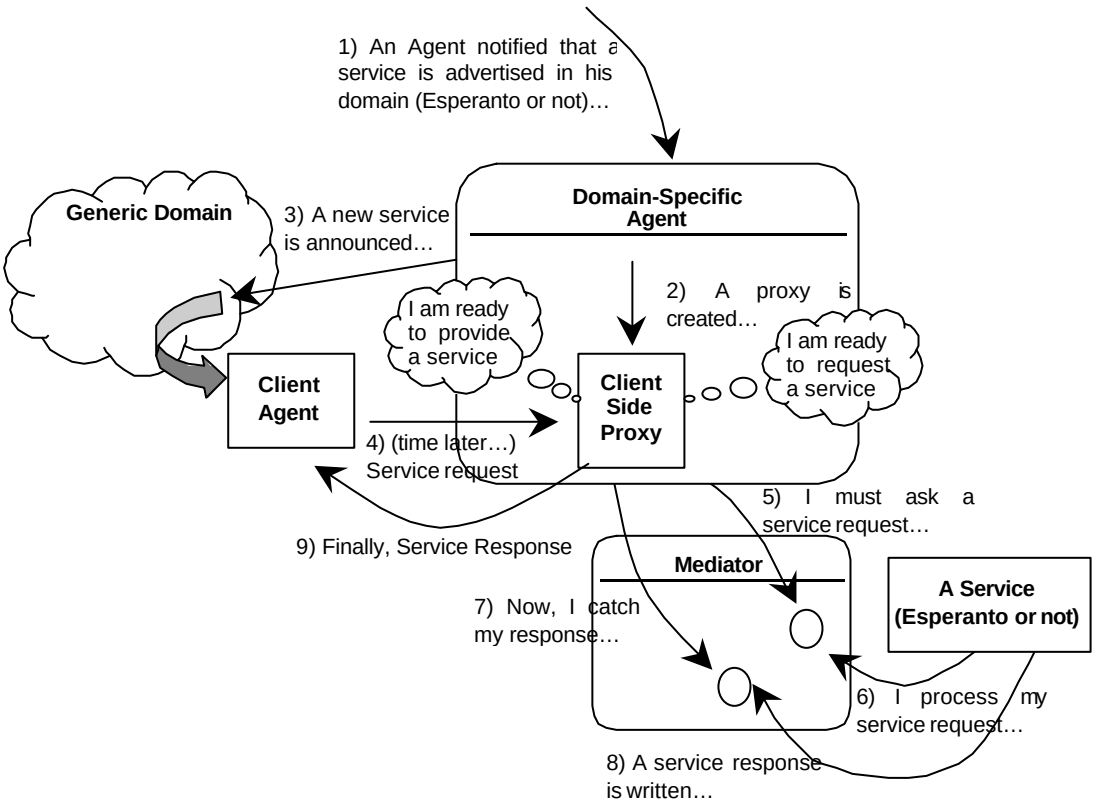

Figura 3-3: Il *Domain-Specific Agent* e il meccanismo per garantire l'uso di un servizio da parte di un cliente del suo dominio di competenza.

L'interoperabilità tra le soluzioni di *delivery* integrate nell'architettura avviene grazie all'adozione del comune modello di *delivery* Esperanto.

La compatibilità nativa delle applicazioni Esperanto con l'architettura è tale che a differenza del *Domain-specific Agent*, l'*Esperanto Agent* ha il solo compito di notificare (ad ogni agente *Domain-specific*) che un nuovo servizio è stato pubblicato, senza alcuna emulazione di clienti o servizi. In tutte le collaborazioni tra gli agenti, le informazioni (i *SADR – Service Access Descriptor Record*, i descrittori dell'accesso al servizio) vengono scambiate in un formato comune definito dalla nuova infrastruttura. Per ogni servizio importato/esportato, l'agente ha quindi il compito di effettuare una conversione di formato da Esperanto a locale (e viceversa) sempre che questa sia possibile. Esportare un servizio presso un altro dominio prima che venga richiesto è l'unico modo che una applicazione ha per interagire in maniera spontanea ed inconsapevole con un servizio incompatibile.

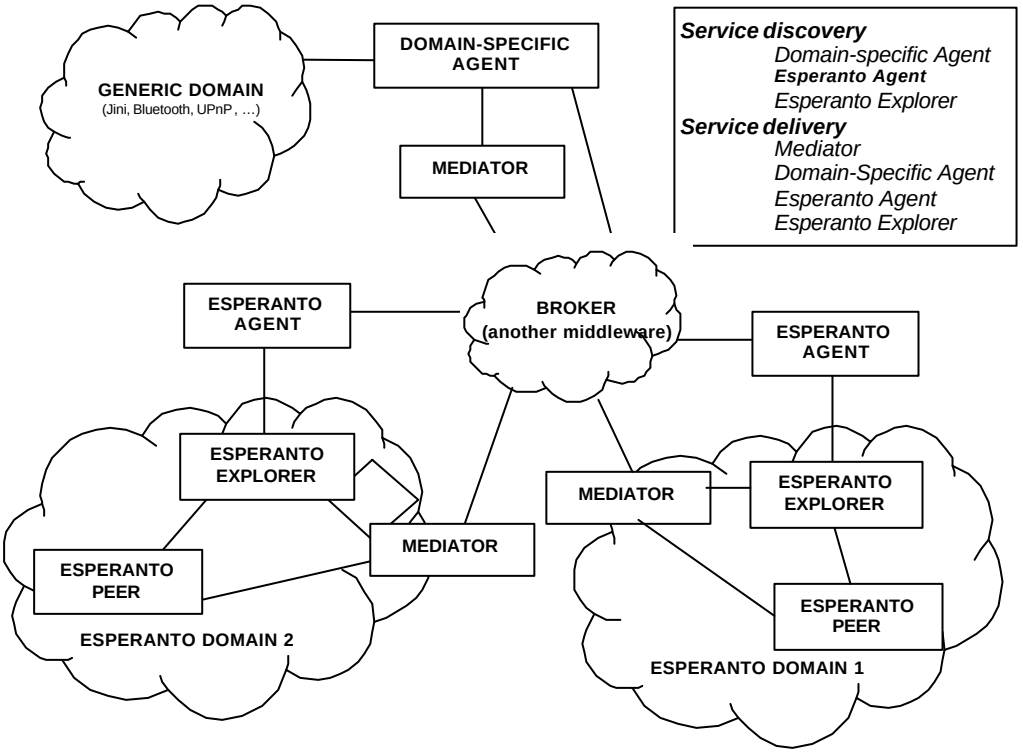

Figura 3-4: rappresentazione del modello architetturale Esperanto**.**

3.5.5 Leggerezza del carico computazionale introdotto

Gli ambienti di *nomadic computing* vedono la presenza di numerosi dispositivi con scarse capacità computazionali (*smart e mobile phone*, PDA,…). Il carico richiesto (computazionale e di memoria) a ogni *device* Esperanto deve essere dunque reso leggero sia dalla semplicità dell'architettura, sia dall'introduzione di componenti dove il pesante carico computazionale non costituisce un problema. La presenza in ogni dominio, di un componente localizzato sul *core* di rete, l'*Esperanto Explorer*, solleva i dispositivi della gestione delle usuali operazioni previste dalle *Service Oriented Architecture* come ad esempio la catalogazione dei descrittori di servizio pubblicati in un dominio.

3.6 Struttura

In figura 3-4 è riportata la rappresentazione del modello architetturale di Esperanto. E' possibile distinguere i diversi domini (*Jini* [19], *Bluetooth* [34], *web services* [26], …, *Esperanto*) interconnessi tra loro attraverso gli agenti (*Domain-specific* o *Esperanto)* ed i mediatori.

In generale è possibile prevedere anche domini fisicamente distinti, ma che condividono la stessa infrastruttura SOA (ad esempio due federazioni *Jini* non interconnesse tra loro, ma tramite Esperanto). Allo stesso modo possono esservi più domini Esperanto.

A differenza dei domini generici, il dominio Esperanto è dettagliato in tutti i suoi elementi principali, l'*Esperanto Peer* e l'*Esperanto Explorer*. Gli elementi dell'architettura comunicano tra loro attraverso un *broker* (un *middleware* ad oggetti distribuiti, *socket*, *RPC*, …) che rappresenta un grado di libertà nella definizione di dettaglio dell'architettura.

3.6.1 Esperanto *Peer*

Con il termine *Esperanto Peer* si intende quella componente *software* che implementa un *client agent* e/o un *service agent* delle *Service Oriented Architecture*. Come osservato al par. 3.1, la fase di *service discovery* è generalmente preliminare a quella di *delivery*: essa prevede che il servizio venga pubblicato prima di essere offerto e conseguentemente che venga scoperto prima di essere utilizzato.

La fase di *service delivery*, invece, prevede che prima avviare un'interazione (*interaction*), venga stabilito come questa debba avvenire (*service access description*).

L'Esperanto *Peer* deve rispettare questo modello affinché possa offrire/usare servizi. Il supporto alle fasi di *service discovery* e *service access description* è fornito al *Peer* grazie all'*Esperanto Explorer*, l'interazione avviene (come già osservato al par. 3.5.1.1) grazie al *Mediator*.

Naturalmente ogni dispositivo Esperanto può ospitare più *Esperanto Peer*.

3.6.2 Esperanto *Explorer*

Tipicamente ogni SOA prevede l'esistenza di un *locator*, ovvero un componente *software* cui spetta il compito di catalogare i servizi (pubblicati dai *service provider*) e soddisfare le *query* espresse dai *service requestor* per la ricerca dei descrittori di servizio (talvolta tale componente può coincidere con il *service broker* [17]). L'*Esperanto Explorer* è il componente di un dominio Esperanto a cui competono proprio questi compiti: esso cataloga i descrittori di servizio (SADR) offerti dagli Esperanto *Peer* (in esecuzione su dispositivi residenti all'interno del suo dominio di competenza), e si incarica, conseguentemente, di soddisfare le richieste di *service discovery* o *service access description*. Per la suddivisione in domini dell'architettura, le richieste di un SADR possono fare riferimento ad un servizio remoto (non localizzato nel dominio da cui proviene la richiesta); per questo motivo, l'*Esperanto Explorer* rivolge le *SAD request*

all'*Esperanto Agent* locale. Il descrittore richiesto verrà restituito tramite la collaborazione tra l'*Esperanto Agent* locale e l'agente a cui compete il servizio remoto.

Inoltre l'*Esperanto Explorer* è, insieme al *Mediator*, un elemento fondamentale nel supporto alla mobilità dei dispositivi: infatti, grazie alla loro "portatilità", ogni dispositivo Esperanto può spostarsi fisicamente da un dominio (Esperanto) ad un altro (Esperanto). Data la natura omogenea dei domini Esperanto, questa migrazione deve essere resa trasparente alle applicazioni: esse devono poter continuare nelle loro attività di *service discovery* e *service delivery* senza che siano influenzate direttamente. La collaborazione tra gli *Esperanto Explorer*, permette quindi ai dispositivi di accedere (nel *delivery* e *discovery*) alle risorse del dominio (essenzialmente il catalogo dei servizi) sempre allo stesso modo (interrogando l'*Explorer* locale) ed indipendentemente dalle migrazioni che possono compiere durante le loro elaborazioni.

Per questo è importante che l'*Esperanto Explorer* sia governato da politiche di accesso volte a garantire sicurezza e protezione.

3.6.3 *Mediator*

La presenza del mediatore (uno per ogni dominio Esperanto) è stata ampiamente giustificata nel par. 3.5.1. Sinteticamente esso è l'elemento a cui fanno riferimento due o più *Esperanto Peer* (o *proxy*) quando invocano *service request* oppure quando invocano *service response*. I dati scambiati durante l'interazione sono inviati, mantenuti e poi prelevati dal mediatore, che rappresenta uno spazio di memoria condiviso attraverso cui le applicazioni possono cooperare. Quindi, concettualmente, una *service request* espressa da *client agent* ha luogo in due tempi: il primo in cui si richiede l'esecuzione del servizio (si inviano al mediatore i dati necessari all'attivazione), il secondo in cui l'eventuale risultato del servizio viene prelevato dallo spazio comune offerto dal mediatore. Anche il comportamento del *service agent* evolve in due tempi: il servizio deve prima

stabilire se ci sono *service request* da soddisfare (prelevandole dal mediatore), elaborarle e poi successivamente (ad esempio se l'interazione è *request/response*) inoltrare le *service response* al mediatore. Esso avrà cura di conservare il risultato fino a che il *client agent* non accederà allo spazio. In figura 3.3, i passi 5-6-7-8 riassumono quanto descritto.

L'introduzione del *Mediator* consente quindi comunicazioni asincrone, disaccoppiate nel tempo e nello spazio, in quanto:

- *client* e *service agent* collaborano esclusivamente con il *Mediator* (disaccoppiamento nello spazio);
- le richieste (per il servizio) e le risposte (per i clienti) possono essere prese in considerazione anche se una delle due controparti non dovesse essere momentaneamente disponibile (disaccoppiamento nel tempo);
- L'uso di primitive non bloccanti per l'accesso allo spazio offerto dal mediatore, consente alle parti di proseguire nelle loro elaborazioni anche dopo l'invocazione della primitiva.

Per la flessibilità della comunicazione è possibile che le interazioni siano sia *pull-based* che *push-based*, sia 1-1 che 1-N.

Esperanto prevede dunque un approccio innovativo al *service delivery* fornendo una infrastruttura appositamente pensata per il supporto alla mobilità dei dispositivi dei domini Esperanto. L'accesso ai mediatori deve essere governato da politiche di accesso volte a garantire sicurezza e protezione.

3.6.4 *Domain-specific Agent*

Come si è visto al par. 3.5.4, il *Domain-specific Agent* è, insieme al *Mediator*, il componente attraverso cui è permessa la condivisione dei servizi tra le applicazioni *Domain-specific* e le applicazioni degli altri domini connessi all'architettura.

Tuttavia, l'approccio Esperanto volto all'integrazione prevede che l'agente *Domain-specific* nasconda alle applicazioni la decomposizione dell'architettura in tanti domini eterogenei. Tale trasparenza è garantita dal *Domain-specific Agent* con l'importazione dei servizi dai domini esterni e l'esportazione dei servizi locali verso i domini remoti. Questo significa che ogni servizio importato è visto dal dominio alla stessa stregua di un servizio locale: le applicazioni possono fare *service discovery* e *service delivery* con le consuete tecniche messe a disposizione del dominio, possono trovare il servizio e decidere successivamente di usarlo. Per questo motivo, l'agente deve interagire, tramite i suoi componenti interni, con gli elementi attivi del dominio cui esso è assegnato (*client*, *server*, *lookup registry*, …). Ad esempio, la consegna del servizio importato (e il suo utilizzo effettivo) è permessa dalla presenza dei *proxy*, i quali effettuano il *bridging* tra le tecniche di *delivery* locali e la modalità di interazione Esperanto (che prevede l'uso del mediatore).

Discorso analogo può essere fatto per ogni servizio esportato: il cliente remoto dovrà apparire al *service*, a tutti gli effetti come un cliente locale al dominio. Sarà sempre il *Domain-specific Agent*, con l'uso dei *proxy*, a garantire questa trasparenza.

3.6.5 Esperanto *Agent*

L'*Esperanto Agent* è l'agente del dominio Esperanto. Pertanto anch'esso è insieme al *Mediator* il componente attraverso cui è permessa la condivisione dei servizi tra le applicazioni *Esperanto* e le applicazioni degli altri domini connessi all'architettura. Tuttavia l'agente Esperanto è sollevato di alcuni compiti che il *Domain-specific Agent* è obbligato a fare. Le applicazioni hanno consapevolezza del modello architetturale (la decomposizione in domini) e possono quindi esplicitamente fare *service discovery* e *service delivery* sia locali che remoti. Pertanto l'agente non deve effettuare operazioni di importazione dei servizi né tantomeno creare *proxy* visto che le applicazioni Esperanto utilizzano nativamente il modello di interazione in cui è introdotto l'uso del *Mediator*. D'altra parte

l'agente deve supportare questo comportamento delle applicazioni recuperando le informazioni richieste ogni volta che è necessario (ad esempio ogni volta che viene richiesto SADR di un servizio remoto).

3.6.6 Esperanto *Agent* e *Domain-Specific Agent*

Come si è avuto modo di comprendere nei par. 3.6.4 e 3.6.5, gli agenti di dominio hanno ruoli simili, ma sono diversi.

A differenza dell'*Esperanto Agent*, il *Domain-specific Agent* deve nascondere l'architettura alle applicazioni del dominio di competenza. Tuttavia entrambi devono permettere la condivisione dei servizi, ma ognuno con un proprio approccio. Allo scopo di fare chiarezza sui motivi di questa scelta e sulle differenze dei due agenti, in questo paragrafo viene esposto un confronto tra questi due componenti dell'architettura.

Nella tabella 3-1 sono riportate le caratteristiche degli agenti ed il loro confronto.

Responsabilità	Esperanto Agent	Domain Specific Agent	Note
Esporta i servizi del suo dominio verso tutti i domini, non Esperanto, connessi all'architettura	✓	✓	Questa responsabilità è a carico degli agenti per consentire alle applicazioni dei domini generici di poter utilizzare in maniera trasparente i servizi offerti nei domini connessi all'architettura (Esperanto e non)
Importa i servizi di tutti i domini connessi all'architettura nel suo dominio di competenza		✓	Questa responsabilità non è a carico dell'Esperanto Agent perché i Peer hanno consapevolezza dell'architettura e quindi possono fare esplicita richiesta di un servizio di un dominio remoto (Esperanto o non)
Realizza tutte le operazioni di bridging (traduzione di formato, creazione dei proxy…)		✓	Le operazioni di bridging come la creazione dei proxy non è carico dell'Esperanto Agent in quanto gli Esperanto Peer adottano formati (uso di SADR Esperanto) e modelli di interazione Esperanto (uso del Mediator)
Collabora con gli agenti Esperanto per recuperare i descrittori SADR richiesti dalle applicazioni Esperanto	✓	✓	SAD request di un cliente Esperanto riferite ad un servizio remoto sono soddisfatte dagli agenti Esperanto collaborando tra loro. Se la richiesta si riferisce ad un servizio generico il DS Agent deve emulare questa responsabilità

Tabella 3-1: Confronto tra *Esperanto Agent* e *Domain Specific Agent.*

La generazione dei *proxy* e le conversioni di formato sono naturalmente compiti esclusivi dell'agente *Domain-specific*.

La scalabilità degli agenti è il motivo principale per cui l'importazione dei servizi non viene effettuata nei domini Esperanto: infatti per la sua complessità e il dispendio di risorse, essa può costituire un limite alla crescita del numero di domini e del numero di servizi. Inoltre l'importazione dei servizi è un'esigenza che nasce per la volontà di integrare applicazioni eterogenee in maniera trasparente. In Esperanto, il problema dell'integrazione è superato consentendo ai *Peer* di accedere ai descrittori dei servizi remoti (tramite l'agente) a seguito di una esplicita operazione di *service discovery* o di *service access description request* inter-dominio. Questo secondo approccio è quindi maggiormente scalabile dato che i servizi sono usati solo su richiesta e senza alcuna operazione di importazione.

Per effetto di queste scelte di progettazione, i *Domain-specific Agent* devono assecondare il comportamento reattivo dei loro pari Esperanto, provvedendo a soddisfare le loro richieste di *service discovery/service access description*. D'altra parte anche gli *Esperanto Agent* esportano i servizi verso i domini generici per assecondare le esigenze dei *Domain-specific Agent*. In una architettura dove solo domini Esperanto sono connessi, l'import/export dei servizi è un'operazione che non avrebbe luogo, serve solo a garantire l'interoperabilità tra i diversi approcci delle SOA integrate in Esperanto.

3.7 Comportamento dinamico

In questa sezione è presentato il comportamento dinamico dei componenti ovvero le interazioni tra loro nei vari casi d'uso della piattaforma *middleware* Esperanto. Si osservi che un'applicazione può interagire con un servizio dopo che lo ha scoperto nella fase di *discovery*. Negli scenari di interazione si fa riferimento ad

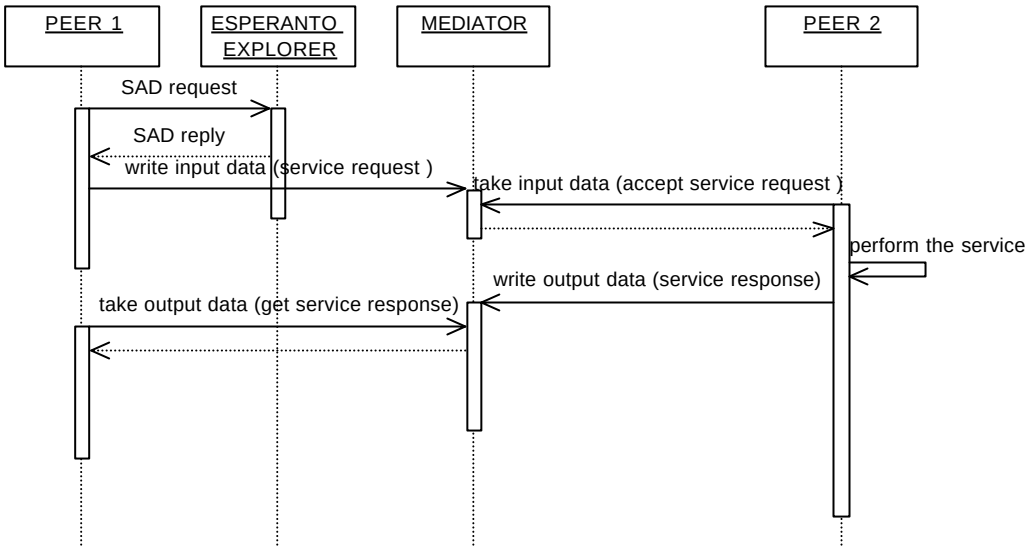

Figura 3.5: *service delivery* tra *Peer* residenti nello stesso dominio Esperanto

interaction di tipo *request/response*. Scenari in cui le interazioni sono di tipo *notify*, *solicit/response*, *one-way*, sono altrettanto possibili.

3.7.1 SCENARIO I

Service delivery tra Esperanto Peer residenti su device localizzati nello stesso dominio Esperanto (figura 3.5).

1. Il *Peer* 1 richiede all'*Explorer* la descrizione dell'accesso ad un servizio offerto dal *Peer* 2 (*SAD request*)
2. Il servizio richiesto è residente all'interno del dominio da cui proviene la richiesta. L'*Explorer* recupera il SADR associato al Peer 2 e lo restituisce al *Peer* 1 (*SAD reply*)
3. Il *Peer* 1 inizia l'interazione inviando la *service request* al mediatore
4. La *service request* è elaborata dal *Peer* 2 il quale risponde con una *service response*. La *service response* può successivamente essere prelevata dal *Peer* 1

3.7.2 SCENARIO II

Service delivery tra Esperanto Peer residenti su device localizzati in domini distinti (figura 3.6).

1. Il *Peer* 1 richiede all'*Explorer* la descrizione dell'accesso ad un servizio offerto dal *Peer* 2 (*SAD request*)
2. Il servizio (*Peer 2*) è remoto, l'*Esperanto Explorer* 1 non trova il SADR nel suo catalogo e lo richiede all'agente di dominio (*Esperanto Agent* 1)
3. L'*Esperanto Agent* 1, stabilisce a quale agente richiedere il descrittore (*Esperanto Agent 2*) e gli inoltra la richiesta
4. *Esperanto Agent* 2 recupera il SADR dall'*Explorer* (*Esperanto Explorer 2*) di dominio dove è localizzato il servizio e lo restituisce all'*Agent* 1
5. Il SADR viene restituito al *Peer* 1 che può interagire con il servizio: invia la *service request* al *Mediator* locale (*Mediator* 1)
6. Dato che il servizio è remoto, il *Mediator* 1 determina a quale dominio compete il servizio e invia i dati al mediatore associato (*Mediator* 2)
7. La *service request* è elaborata dal *Peer* 2 il quale risponde con una *service response*
8. L'accesso al mediatore locale e l'inoltro della risposta al mediatore 1 avviene come descritto al punto 6. Infine il *Peer* 1 può prelevare la risposta del servizio

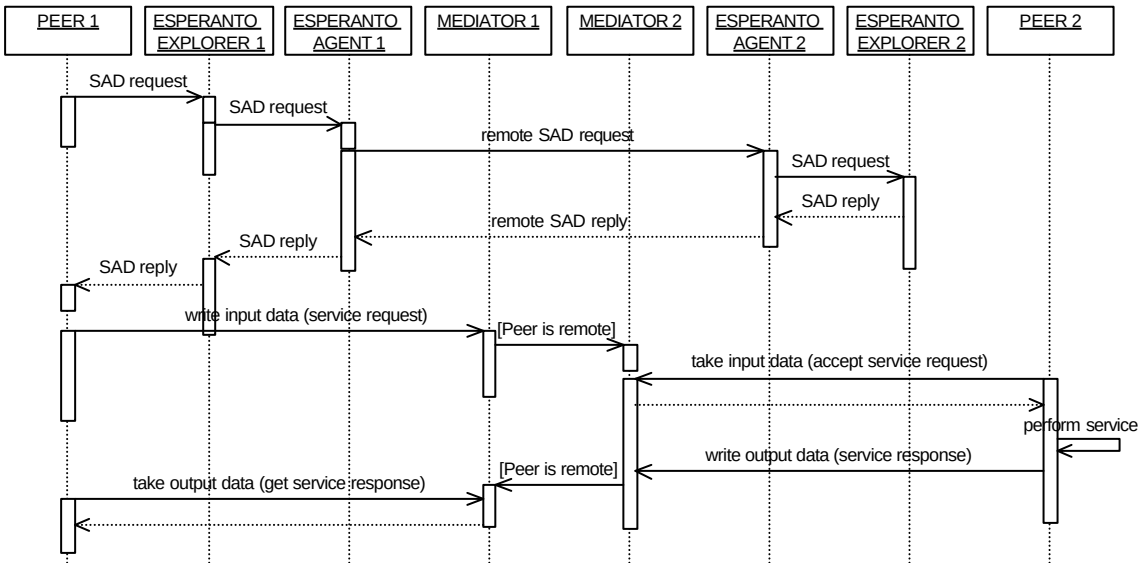

Figura 3.6: *service delivery* tra *Peer* residenti in due domini Esperanto distinti.

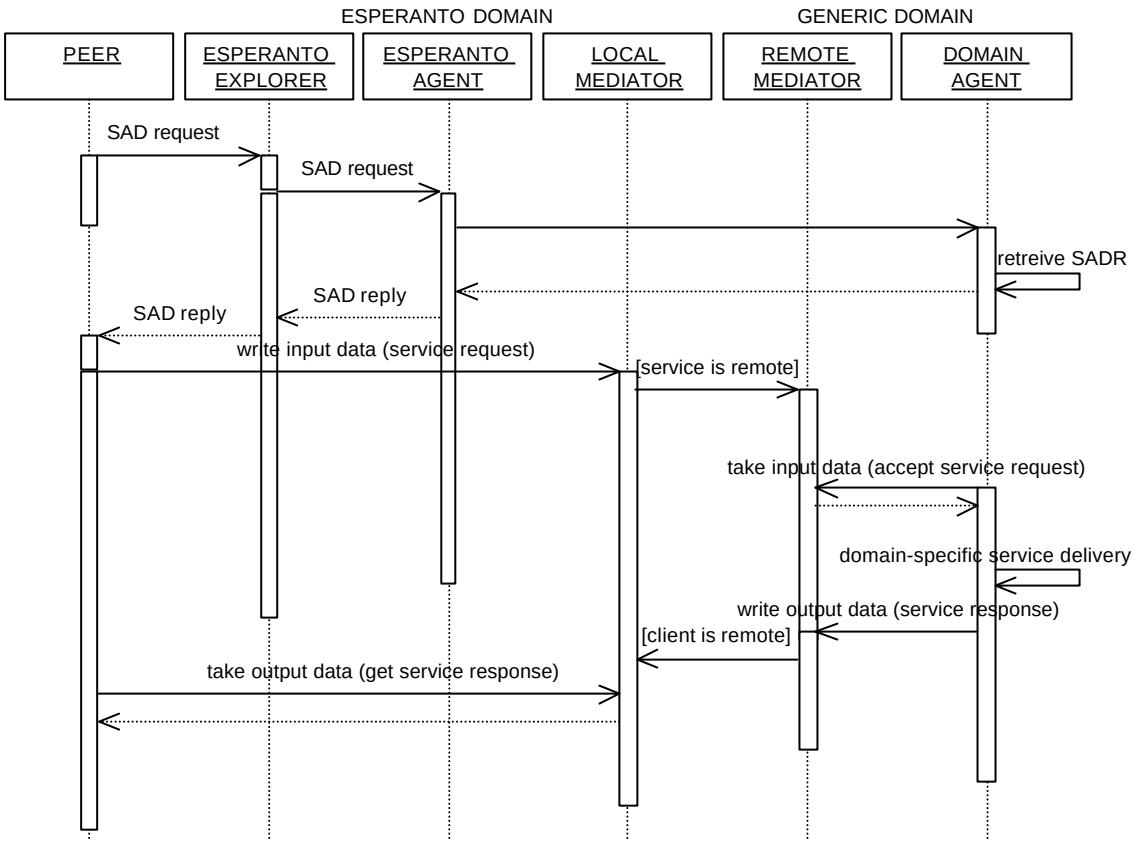

Figura 3.7: *service delivery* tra un Esperanto *Peer* ed un servizio offerto in un dominio generico.

3.7.3 SCENARIO III

Service delivery tra un Esperanto Peer ed un'applicazione di un dominio generico (figura 3.7).

1. L'Esperanto *Peer* richiede all'*Explorer* la descrizione dell'accesso ad un servizio (*SAD request*)
2. Il servizio è remoto, l'*Esperanto Explorer* non trova il SADR nel suo catalogo e lo richiede all'agente di dominio (*Esperanto Agent*)
3. L'*Esperanto Agent,* stabilisce a quale agente richiedere il descrittore (*Domain Agent*) e gli inoltra la richiesta
4. Il *Domain-specific Agent* ottiene il descrittore di servizio e lo invia all'agente Esperanto che lo restituisce al *Peer* richiedente

5. L'interazione segue i passi 5-6-7-8 dello scenario II (cfr. par. 3.7.2). In particolare il reale servizio viene attivato dal *proxy* lato *server* una volta ottenuta la *service request*

3.7.4 SCENARIO IV

Un'applicazione di un dominio generico utilizza un servizio offerto da un Esperanto Peer (figura 3.8). In questo scenario (così come in quello delineato al punto 3.7.5) non c'è una fase di descrizione dell'accesso al servizio. Se prevista dal dominio di competenza, questa fase viene implementata dal proxy lato client prodotto dal Domain-specific Agent all'atto dell'importazione del servizio.

1. L'agente di un dominio generico riceve una richiesta di *delivery* di un servizio (richiesta che perviene ad *proxy* lato *client*): esso risiede in un dominio Esperanto
2. L'agente (in particolare il *proxy* lato *client*) invia la *service request* al mediatore locale che provvede a smistarla al mediatore opportuno visto che fa riferimento ad un servizio remoto
3. L'Esperanto *Peer* accetta la richiesta di *service request*, prelevando i dati dal *Mediator*, effettua l'elaborazione associata al servizio e restituisce i risultati
4. L'agente *Domain-specific* ottiene la *service response* e la restituisce all'applicazione

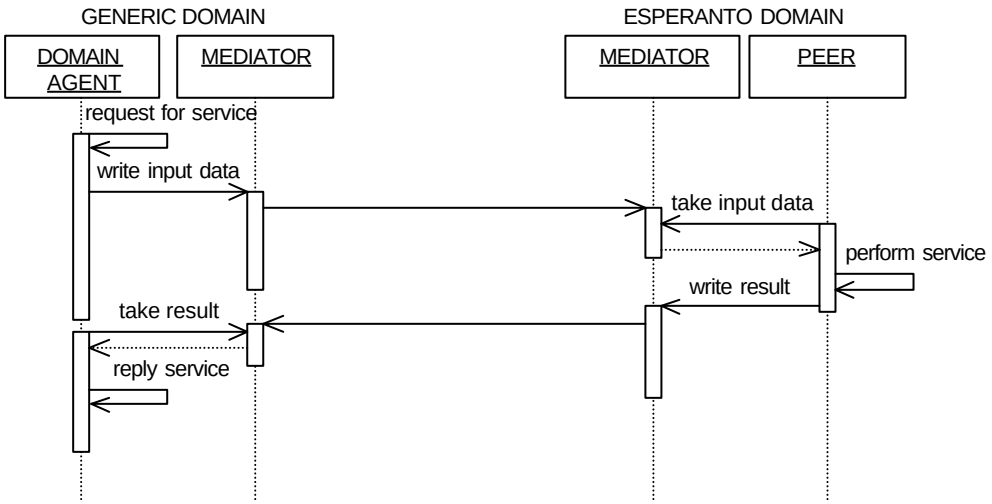

Figura 3.8: *service delivery* tra un cliente di un dominio generico ed un servizio offerto da un Esperanto *Peer*.

che ne ha fatto richiesta

5. L'interazione tra servizio Esperanto e *proxy* lato *client* evolve allo stesso modo dello scenario II (cfr. 3.7.2)

3.7.5 SCENARIO V

Un'applicazione di un dominio generico utilizza un servizio offerto da un'applicazione di un dominio generico (figura 3.9).

1. L'agente di un dominio generico riceve una richiesta di *delivery* di un servizio che risiede in un altro dominio generico (richiesta che perviene al *client-side proxy*)
2. L'agente 1 (in particolare il *proxy* lato *client*) invia la *service request* al mediatore 1 che provvede a smistarla al mediatore opportuno (mediatore 2) visto che fa riferimento ad un servizio remoto
3. L'agente 2 (il *proxy* lato *server*) accetta la *service request* prelevando i dati dal *Mediator* 2 e invoca il servizio effettivo. Una volta ottenuti i dati di *output* fornisce la *service response* al mediatore 2 che li invia al mediatore 1.
4. L'agente *Domain-specific* preleva la *service response* dal *Mediator 1* e lo restituisce all'applicazione che ne ha fatto richiesta

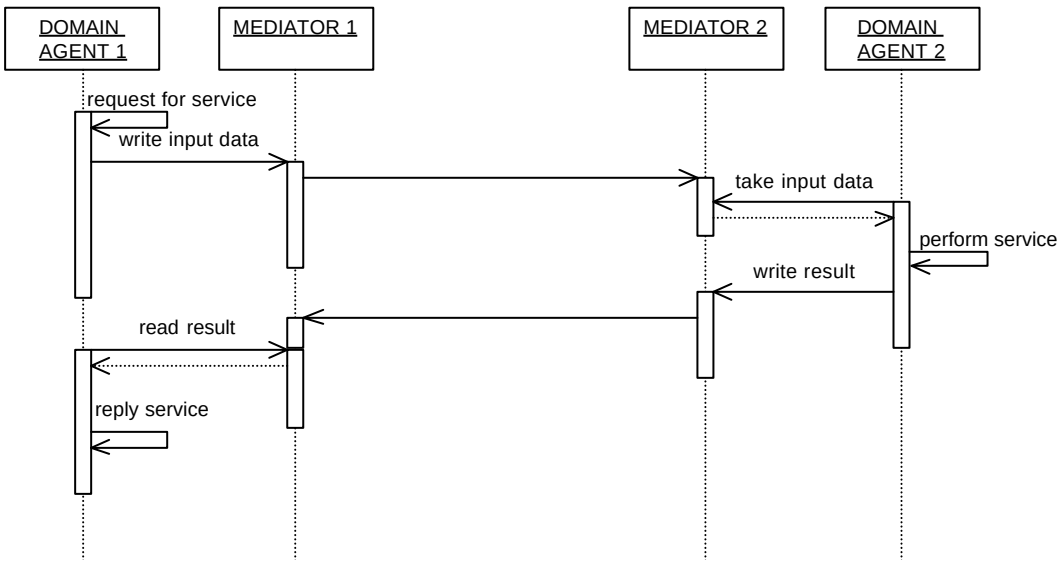

Figura 3.9: *service delivery* tra un cliente di un dominio generico ed un servizio offerto da un dominio generico.

5. L'interazione tra i *proxy* evolve allo stesso modo dello scenario II (cfr. 3.7.2)

3.8 Considerazioni sulla soluzione proposta

3.8.1 Supporto alla mobilità e meccanismi di interazione

L'architettura Esperanto prevede la definizione di una nuova soluzione di *service delivery* che adotta un paradigma di comunicazione disaccoppiato. Questo è particolarmente vantaggioso negli ambienti di *nomadic computing* perché consente di risolvere alcune tra le principali problematiche derivanti dalla mobilità dei dispositivi e la dinamicità dell'ambiente. Tuttavia l'integrazione con SOA in cui è previsto per le applicazioni un meccanismo di comunicazione fortemente accoppiato (per esempi, cfr. 2.8) può essere critica. Si pensi ad esempio all'interazione tra un *client agent* che supporta solo RPC e un *service agent agent* Esperanto: l'interazione tra il cliente ed il *proxy* ha luogo con successo (il *proxy* è sempre pronto ad accettare/inoltrare richieste e a recuperare/fornire risposte), ma il *service agent* potrebbe essere non attivo. Secondo il modello di interazione Esperanto questo non rappresenta un malfunzionamento (entro certi limiti), tuttavia il *middleware* lato RPC potrebbe far scattare dei *timeout* nella attesa di una risposta che non perverrà in tempo.

Questo problema può essere mitigato aderendo al concetto di *boundary* temporale espresso al par. 1.4.3.1 e facendo in modo che l'agente *Domain-specific* importi solo i servizi i cui meccanismi di interazione sono compatibili con i clienti del suo dominio di competenza.

3.8.2 Interoperabilità con altre soluzioni di *discovery/delivery*

In merito alle operazioni di *bridging* tra le diverse tecnologie di *delivery*, si osservi che la scelta adottata dall'architettura Esperanto si presenta essere molto

vantaggiosa. Infatti, se si vogliono far comunicare applicazioni sviluppate adoperando N tecnologie diverse, è necessario in linea di principio costruire N(N-1) *bridge* distinti in modo da portare in conto tutte le possibili interazioni. L'avvento di una nuova tecnologia implica la costruzione di N nuovi *bridge* verso le tecnologie esistenti.

La scelta adottata in Esperanto è invece quella di eleggere una nuova tecnologia (Esperanto) come tecnologia di *bridging* intermediaria. In questo modo, bisogna realizzare N-1 *bridge* tra le diverse tecnologie esistenti che si desidera integrare implicando la costruzione di un solo *bridge* per ogni nuova tecnologia che si presenta sul mercato.

3.8.3 Esperanto *Explorer*

Come riportato al par. 3.6.2, l'*Esperanto Explorer* rappresenta il componente dell'architettura cui compete la responsabilità di catalogazione dei descrittori di servizio. Quando bisogna progettare un componente di questo tipo, diversi approcci sono possibili:

1) Ogni dispositivo dell'ambiente dispone di un proprio catalogo di servizi offerti e/o trovati disponibili nell'ambiente: nelle richieste dei descrittori deve quindi "costruirsi un vicinato di dispositivi" a cui rivolgere le interrogazioni.
2) Ogni dispositivo dell'ambiente dispone di un unico catalogo (indipendente) dei servizi offerti nell'ambiente: nelle richieste dei descrittori ogni dispositivo può fare riferimento ad esso per soddisfare le proprie interrogazioni.
3) Nell'ambiente sono presenti più di un catalogo di servizi, ma sempre indipendenti dai dispositivi. Questa soluzione è analoga alla precedente, semplicemente il catalogo è distribuito.

L'approccio 1) è tipico degli ambienti *ad hoc*, ma risulta una soluzione poco scalabile a causa dell'elevato traffico che si genera (per la ricerca dei descrittori) al crescere del numero dei dispositivi. La seconda soluzione è tipica degli ambienti fissi ma anch'essa non è scalabile a causa dell'elevato numero di

interrogazioni da soddisfare al crescere della numerosità dei dispositivi. L'approccio 3) tende a migliorare la scalabilità grazie all'uso della distribuzione. Inoltre non si richiedono grosse risorse ai dispositivi, quando si fa riferimento ad un catalogo esterno (non bisogna avere capacità di memorizzazione o di elaborazione di protocolli complessi).

Il progetto dell'*Esperanto Explorer* segue gli approcci 2) e 3): è centralizzato se si pensa al singolo dominio, distribuito se si pensa all'intera architettura.

3.8.4 Verifica dei requisiti

Allo scopo di valutare il protocollo di *delivery* definito dall'architettura Esperanto, si riporta in tabella 3-2 come e se sono stati soddisfatti i requisiti definiti al par. 3.4. Si osservi che la *fault tolerance* non è stata presa in considerazione; le problematiche sollevate da un tale requisito esulano dagli scopi di questa tesi. In merito alla sicurezza, è previsto che l'accesso alle entità dell'architettura sia controllato, ma non viene specificato nel dettaglio in che maniera. Sicurezza e *fault tolerance* rappresentano pertanto delle *open issues* per la definizione completa dell'architettura Esperanto.

Requisito	Soddisfatto	Note
Supporto alla mobilità	SI	Il paradigma di comunicazione disaccoppiato consente di gestire migrazioni e disconnessioni
Supporto alla dinamicità	SI	Sono forniti i meccanismi necessari per reagire alla variazione del contesto elaborativo delle applicazioni
Indipendenza alla implementazione	SI	La separazione tra la descrizione dell'accesso al servizio e l'interazione aiuta l'indipendenza dall'implementazione
Semplicità	SI	Su ogni dispositivo bisogna solo implementare l'accesso agli elementi fissi dell'architettura, che provvedono a fare il resto
Scalabilità	SI	La distribuzione dei mediatori sui vari domini aiuta a dominare la complessità al crescere del numero di entità interagenti
Eterogeneità	SI	Attraverso le proprietà del SADR, un servizio può essere descritto in tutte le sue caratteristiche
Tolleranza ai guasti	NO	Non sono previste delle tecniche volte a garantirla, ma è sempre possibile estendere l'architettura
Sicurezza	SI	E' previsto che l'accesso ai *Mediator* e agli *Explorer* avvenga nel rispetto di opportune politiche di accesso. In ogni caso la problematica non è affrontata nel dettaglio

Tabella 3-2: *check list* dei requisiti soddisfatti dal protocollo di *service delivery* realizzato.

Capitolo 4

Progettazione dell'infrastruttura di *delivery Esperanto*

4.1 Introduzione

Come si è avuto modo di comprendere, lo studio dell'architettura di Esperanto è nato da un'analisi attenta dei requisiti e vincoli che un'infrastruttura di *delivery* deve soddisfare per rispondere alle esigenze del *nomadic computing*. Nel capitolo precedente è stata presentata una descrizione generale di Esperanto; si è visto come essa affronta gli obiettivi di progetto e come i suoi componenti mirano a soddisfarli. Nel seguito saranno descritti gli aspetti di maggiore interesse dell'infrastruttura di *delivery*:

- ***La mobilità dei dispositivi***: un dispositivo può muoversi durante l'interazione con il servizio, può lasciare l'ambiente o può migrare da un dominio all'altro. Prevedere che questo non comporti alcun malfunzionamento, rappresenta uno degli obiettivi di progetto più ambiziosi.
- ***La reazione al contesto***: in ambienti altamente dinamici, l'assenza di una qualsiasi forma di reazione alle variazioni del contesto elaborativo può rendere estremamente scadente la qualità di interazione con un servizio. Offrire dei meccanismi di *context-awareness* alle applicazioni è il primo passo per rendere veramente efficace il paradigma *client/server* in infrastrutture di *nomadic computing*.
- ***L'interoperabilità con le soluzioni di discovery/delivery***: Esperanto deve il suo nome allo sforzo di rappresentare il naturale ambiente dove fondere le soluzioni di *discovery/delivery* esistenti e future. Pensare ad una

soluzione che non sia solo un altro tentativo di risposta ai bisogni degli sviluppatori è forse l'elemento di maggior interesse di Esperanto.

4.2 Esperanto: l'approccio alla mobilità

4.2.1 Paradigma di comunicazione

In uno scenario di *nomadic computing* è lecito immaginare che un'applicazione di un dispositivo mobile cerchi i servizi che ha intorno prima di poterli utilizzare. Infatti, in un contesto in cui gli utenti liberamente lasciano o accedono all'ambiente, i servizi presenti un istante prima possono essere assenti un attimo dopo. Tuttavia, mettere a disposizione meccanismi di *discovery/delivery* che supportino solo questa dinamicità non basta. Assumere che, una volta trovato il servizio giusto, il cliente possa interagire con esso in maniera simile a come succede nei sistemi distribuiti tradizionali è un'ipotesi errata. *Client* e *server* possono essere entrambi localizzati su dispositivi mobili: possono spostarsi perché l'utente sta fruendo di un servizio mentre è in movimento (ad esempio è in una galleria d'arte ed ascolta i commenti relativi alle opere che osserva) o disconnettersi durante l'interazione (ad esempio si esauriscono le batterie del dispositivo, oppure si entra in una zona d'ombra, oppure semplicemente si vuole completare l'interazione in un secondo momento).

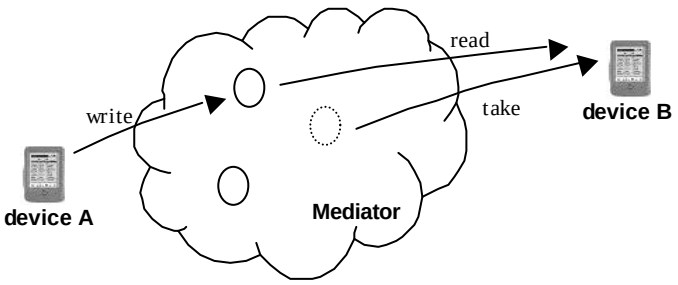

Figura 4-1: comunicazione tra due dispositivi residenti nello stesso dominio Esperanto.

Esperanto supporta questi scenari, adottando uno schema di comunicazione che si ispira al modello *tuple-oriented*. All'interno di un dominio (Esperanto) lo spazio di memoria virtuale condiviso è rappresentato dal *Mediator*: ogni applicazione, che vuole interagire con un suo pari, scrive e legge i dati dallo spazio di tuple inviandoli e richiedendoli al mediatore locale al dominio in cui si trova. A differenza delle implementazioni classiche *tuple-oriented*, come ad esempio *Linda* [28], [29], in Esperanto lo spazio è distribuito in quanto ogni dominio ha un proprio mediatore locale: le applicazioni leggono e scrivono dal *Mediator* del dominio in cui sono ospitate; dato che nell'architettura possono esserci più domini Esperanto (anche fisicamente distanti tra loro), è compito del mediatore capire se le tuple inviate ad esso vanno inoltrate al dominio dove si trova attualmente il dispositivo oppure no. In figura 4-1, ad esempio, i dispositivi A e B risiedono nello stesso dominio perché il mediatore non provvede ad inoltrare alcuna tupla.

Questo comportamento del *Mediator* è possibile se le tuple trasferite allo

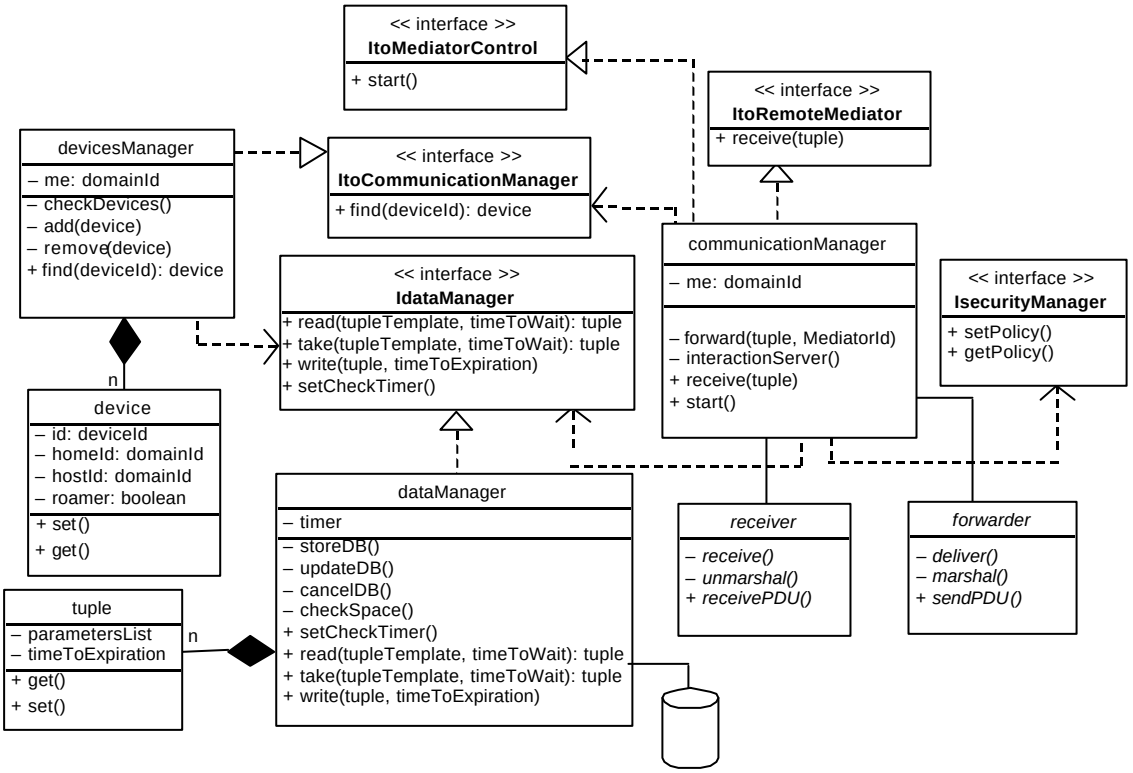

Figura 4-2: una vista del diagramma delle classi del *Mediator*.

spazio consentano in qualche modo di localizzare il dispositivo interessato alla comunicazione, e solo se il mediatore tiene traccia dei dispositivi attualmente presenti nel suo dominio di competenza. In figura 4-2 si rappresenta una vista del suo diagramma delle classi; di seguito si descrivono gli aspetti salienti:

Il *Communication Manager* è il componente attivo che interagisce con i *Peer* residenti sui dispositivi Esperanto locali (attraverso *forwarder/receiver pattern* [17]): le richieste di accesso allo spazio (in lettura/scrittura) sono accettate e processate tramite il metodo *interactionServer()*, il quale effettua le operazioni richieste (*read, write, take*) attraverso la collaborazione con il *Data Manager* e il *Devices Manager*. Come visto nel par. 1.4.4, le operazioni previste in questo paradigma di comunicazione sono:

- scrittura (*write*): attraverso il metodo *write()* un'applicazione memorizza le tuple di interesse all'interno del mediatore. Ad ogni tupla dello spazio deve essere associato un tempo massimo trascorso il quale il *Data Manager* provvederà a rimuoverla (mediante l'invocazione del metodo *checkSpace()*). Questo tempo esprime i vincoli temporali legati al *delivery* del servizio ed il suo utilizzo è previsto in accordo al concetto di *temporal boundary* espresso al par. 1.4.3.1. Se le tuple non sono destinate ad un pari che risiede nello stesso dominio di chi le scrive, esse sono inoltrate (con il metodo *forward()*) al mediatore del dominio remoto a cui fa riferimento il destinatario. In questo caso l'inoltro delle tuple è indispensabile perché i Esperanto *Peer* fanno sempre e solo riferimento al mediatore locale. La destinazione delle tuple è pertanto stabilita utilizzando l'interfaccia *ItoCommunicationManager* esportata dal *Devices Manager.*
- lettura (*read, take*): con i metodi *read()* e *take()* le applicazioni forniscono un *template* della tupla che desiderano rispettivamente leggere e rimuovere dallo spazio condiviso, coerentemente al modello di Gelernter [28]. Ogni primitiva può essere bloccante oppure no: se *timeToWait* indica un valore nullo allora le primitive sono non bloccanti; in caso contrario l'applicazione attenderà per il

tempo indicato che qualche tupla soddisfi il *match* con il *template* fornito. Scaduto il tempo, il controllo è ritornato all'applicazione con o senza tupla in lettura.

Il *Data Manager* rende possibile la persistenza delle tuple all'interno della memoria virtuale condivisa, mentre l'accesso è controllato dal *Security Manager*: ogni dispositivo può avere le sue *policy* di accesso al mediatore (in linea generale solo i dispositivi ospitati dal dominio possono accedervi) e ogni applicazione può accedere allo spazio in maniera opportuna (ad esempio un'applicazione può fare solo operazioni di *read* ma non di *take*).

Il *class diagram* riportato in figura 4-2 è incompleto degli elementi di gestione delle procedure di *handover* e saranno ripresi successivamente (cfr. par. 4.2.3). Per comprendere meglio le responsabilità del mediatore illustriamo il suo comportamento dinamico attraverso i seguenti diagrammi di interazione illustrati in figura 4-3 e 4-4.

SCENARIO: *lo scenario in figura 4-3 riporta le azioni compiute dagli elementi interni al mediatore durante l'interazione tra due Peer residenti su dispositivi ospitati nello stesso dominio. E' trascurata la gestione persistente dell'operazione di scrittura all'interno dello spazio condiviso. L'interazione prevista è one-way.*

1. Per usare il servizio offerto dal *Peer 2*, il *Peer 1* invia i dati di *input* al mediatore locale
2. Il *Communication Manager* controlla se il dispositivo può accedere allo spazio condiviso e con quali diritti l'applicazione può scrivere la sua tupla. Tali *policy* vengono configurate nelle fasi di *service discovery/service access description* (quando il *Peer* richiede la descrizione di un servizio scoperto)
3. In caso positivo, il *Communication Manager* determina il mediatore che ospita il dispositivo destinazione e quindi se la tupla deve essere inoltrata o registrata nello spazio locale: il dispositivo è ospitato dal suo dominio di competenza, la tupla è memorizzata

4. L'accesso del Peer 2 avviene con azioni analoghe

SCENARIO: *lo scenario in figura 4-4 riporta le azioni compiute dagli elementi interni al mediatore durante l'interazione tra Peer residenti su dispositivi ospitati da domini Esperanto diversi. Per una maggiore leggibilità è stato nascosto il Delivery Agent del Peer 1, la gestione persistente dei dati e le azioni compiute dal Peer servente.*

1. Per usare il servizio offerto dal *Peer 2*, il *Peer 1* invia i dati di *input* al mediatore locale
2. Il *Communication Manager* controlla se il dispositivo può accedere allo spazio condiviso e con quali diritti l'applicazione può scrivere la sua tupla. Tali *policy* vengono configurate nelle fasi di *service discovery/service access description* (quando il *Peer* richiede la descrizione di un servizio scoperto)
3. In caso positivo, il *Communication Manager* determina il mediatore che ospita il dispositivo destinazione e quindi se la tupla deve essere inoltrata o registrata nello spazio locale: il dispositivo è ospitato in un dominio diverso da quello di sua competenza, la tupla è inoltrata al *Communication Manager* destinazione
4. La tupla è inoltrata al *Communication Manager* destinazione; esso controlla se la tupla è diretta ad un *device* ospitato nel suo dominio di competenza e la registra in caso positivo; quando il *Peer 2* accederà allo spazio saranno applicati i controlli per verificare i diritti di accesso così come al punto 2

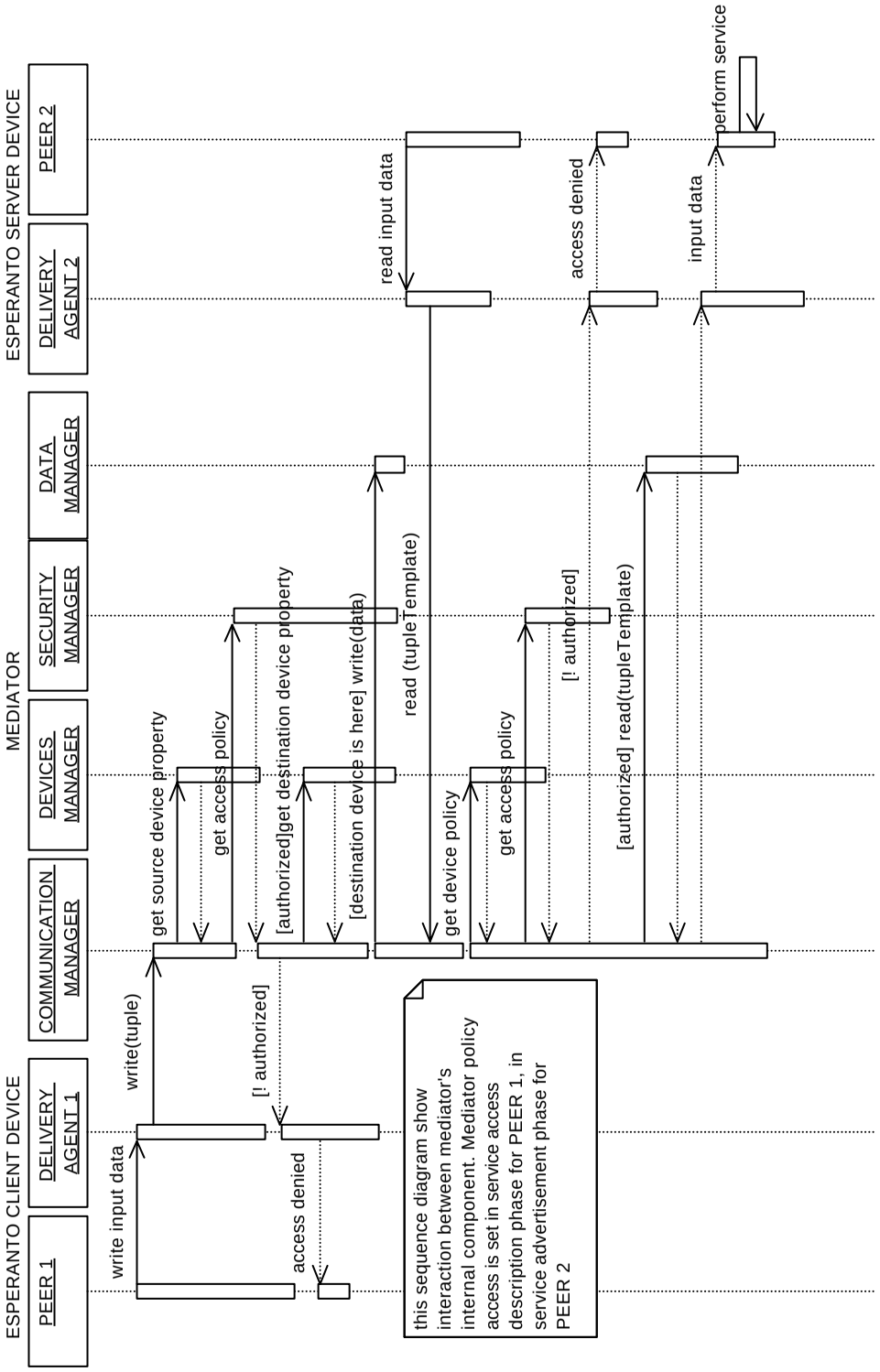

Figura 4-3: *service delivery* tra Esperanto *Peer* residenti nello stesso dominio.

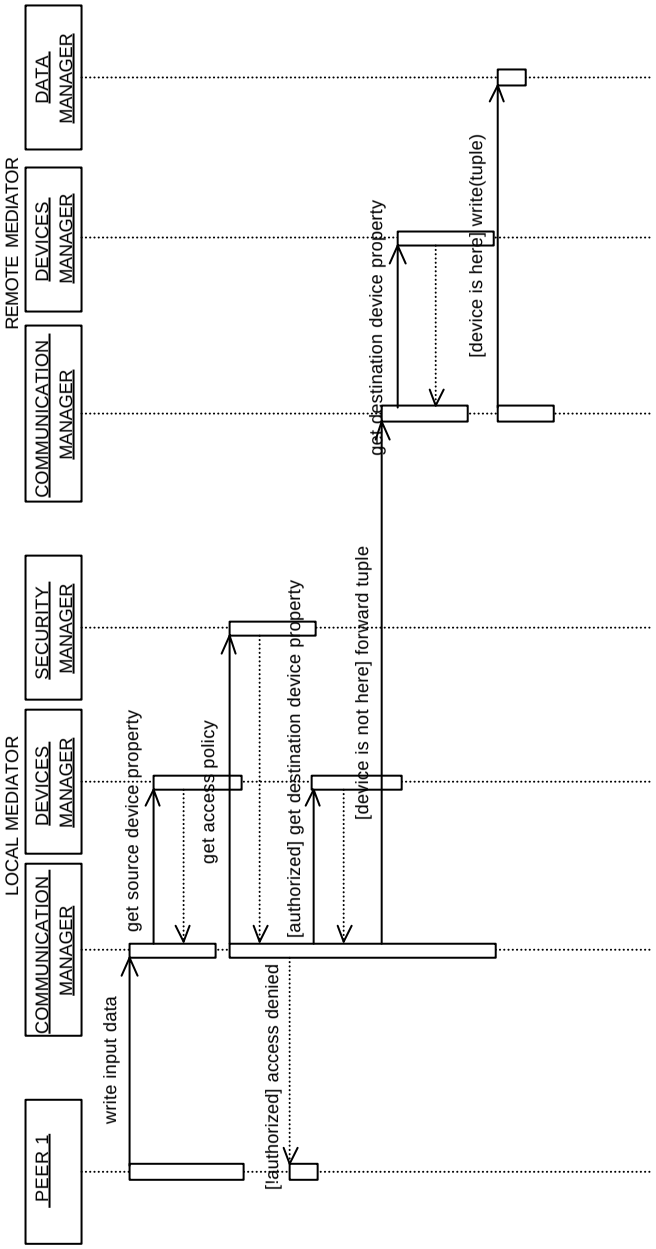

Figura 4-4: *service delivery* tra Esperanto *Peer* residenti in due domini Esperanto distinti.

4.2.2 Protocollo di *delivery* Esperanto

Le disparate caratteristiche dei dispositivi mobili fanno del *nomadic computing* un ambiente in cui è molto difficile implementare un'unica versione di cliente e servizio per tutti i dispositivi possibili. Ogni dispositivo può avere particolari capacità elaborative, le proprie periferiche di *input/output*, avere supporti di memoria di massa ecc. E' importante dunque, riprendendo le osservazioni esposte al par. 1.4.5, che un protocollo di *delivery* per questi ambienti separi la fase di *service delivery* in *service access description* e *interaction*. Nella prima viene specificato "il contratto" che cliente e servizio si impegnano a rispettare durante la fase di interazione; nella seconda avviene lo scambio delle informazioni relative all'utilizzo del servizio.

Esperanto separa le due fasi del *service delivery*. Nel seguito si analizza in dettaglio il modello di un servizio Esperanto.

4.2.2.1 MODELLO DI UN SERVIZIO ESPERANTO

In Esperanto un servizio consiste in:

- ***Funzionalità***: un servizio Esperanto è visto come un insieme di funzionalità, ovvero l'insieme dei metodi che offre l'oggetto che eroga il servizio.
  - Ogni funzionalità è completamente caratterizzata da:
    - *Un insieme di parametri di ingresso*: nell'attivazione di un metodo può essere necessario specificare uno o più parametri di ingresso; ogni parametro ha un nome, un tipo ed un eventuale valore di *default*.
    - *Un insieme di parametri di uscita*: ogni metodo, una volta attivato, può ritornare uno o più parametri di uscita; ogni parametro di uscita ha un nome ed un tipo.
    - *La modalità di interazione. Essa può essere dei seguenti tipi:*
      - ***request/response***: il cliente richiede l'esecuzione di una funzionalità attendendo una risposta in cambio;

- ***solicit/response***: il cliente è sollecitato ad inviare una risposta a valle dell'esecuzione di una funzionalità;
- ***one-way***: il cliente richiede l'esecuzione di una funzionalità, ma non necessita di alcuna risposta in cambio;
- ***notify***: il cliente è notificato con una risposta quando la funzionalità è stata eseguita.

La lista dei parametri di scambio (rispettivamente di ingresso e di uscita) per ogni funzionalità viene modellata come una tupla da inviare allo spazio condiviso. In Esperanto, il modello di un servizio si rappresenta con il *Service Access Description Record* (*SADR*), una lista di attributi che caratterizza le informazioni illustrate nell'elenco precedente. Il *SADR* viene memorizzato dal *Peer* servente all'interno dell'*Esperanto Explorer* all'atto della pubblicazione del servizio (*service advertisement*, una fase del *service discovery*). Dall'*Esperanto Explorer* è reperibile da chiunque ne faccia richiesta. In figura 4-5 si riporta la rappresentazione del *SADR* in formato *XML*:

```
<sadr>
   <id> Service id </id>
   <tuples>
      <tuple>
         <name> a name for tuple </name>
            <parameters>
               <parameter>
                  <name> </name>
                  <type> </type>
                  <default> </default> ? <-- zero or one -->
               </parameter> + <-- one or more -->
            </parameters>
            <timeToExpiration> temporal boundary </timeToExpiration>
      </tuple> +
   </tuples>
   <functionalities>
      <functionality>
         <name> functionality identifier </name>
         <description> human-readable description </description>
         <action name="nmtoken" tuple="nmtoken"
                   source="id" destination="id" timeout="nmtoken"/> +
      </functionality> +
   </functionalities>
</sadr>
```

Figura 4-5: SADR di un servizio Esperanto.

- ***Sezione Tuples***: per ogni tupla si specifica la lista dei parametri (nome, tipo e valore di *default*) e il tempo trascorso il quale la tupla viene rimossa dal mediatore.

- ***Sezione Functionalities***: l'invocazione di ogni funzionalità si specifica tramite le azioni devono essere intraprese dal cliente (*write*, *read* e *take*) e su quali tuple definite (nella sezione *Tuples*). Il *timeout* indica se l'azione deve essere bloccante oppure no: tipicamente *timeout* è nullo per le azioni di *write*, diverso da zero per le azioni di *read* e *take* (cfr. par. 4.2.1).

  - Le funzionalità *one-way* si ottengono con il seguente frammento di descrittore:

```
<-- T is defined in previous section -->
. . .
<functionality>
        <name>foo</name>
        <description> foo is one-way operation </description>
        <action name="write" tuple="T" source="CLIENT_ID"
                                        destination="SERVER_ID" timeout="0"/>
</functionality>
```

  - Le funzionalità *request/response* si ottengono con il seguente frammento di descrittore:

```
<-- Tin is for service, Tout is for client -->
. . .
<functionality>
        <name>foo</name>
        <description> foo is call/back operation </description>
        <action name="write" tuple="Tin" source="CLIENT_ID"
                                destination="SERVER_ID" timeout ="0"/>
        <action name="take" tuple="Tout" source="SERVER_ID"
                        destination="CLIENT_ID" timeout ="timeToWait"/>
</functionality>
```

  - Le funzionalità *solicit/response* si ottengono con il seguente frammento di descrittore:

```
<-- Tin is for service, Tout is for client -->
. . .
<functionality>
        <name>foo</name>
        <description> solicit/response interaction </description>
        <action name="read" tuple="Tout" source="SERVER_ID"
                        destination="CLIENT_ID" timeout =" timeToWait"/>
        <action name="write" tuple="Tin" source="CLIENT_ID"
                        destination="SERVER_ID" timeout ="0"/>
</functionality>
```

    L'interazione *solicit/response* 1-1 si ottiene specificando come *action take* anziché *read*.

- Le funzionalità *notify* si ottengono con il seguente frammento di descrittore:

```
<-- Tout is defined in previous section -->
. . .
<functionality>
        <name>foo</name>
        <description>foo is call/back operation</description>
        <action name="read" tuple="Tout" source="SERVER_ID"
                        destination="CLIENT_ID" timeout ="timeToWait"/>
</functionality>
```

  L'interazione *notify* 1-1 si ottiene specificando come *action take* anziché *read*.

Come stabilito nel *SADR* il formato della tupla è definito dai seguenti parametri:

- *Peer mittente*: alcune indicazioni sulle caratteristiche degli identificatori dei *Peer* sono fornite al par. 4.2.3.3;
- *Peer destinatario*;
- *lista di parametri*: una tupla può corrispondere alla lista dei parametri di ingresso (uscita) di una funzionalità;
- *tempo di persistenza* all'interno dello spazio condiviso, scaduto il quale la tupla è rimossa. Questa indicazione temporale è legata alle dinamiche del servizio e rappresenta un vincolo per il *delivery* delle tuple.

4.2.2.2 FASE DI *SERVICE ACCESS DESCRIPTION*

Durante questa fase un *Peer* cliente richiede il *SADR* associato ad un servizio in modo da poter conoscere quali funzionalità offre, e quali sono i meccanismi e le informazioni necessarie per potervi accedere. Come osservato precedentemente, un *SADR* è memorizzato all'interno del *repository* dell'*Esperanto Explorer*. Data la possibile suddivisione dell'architettura in diversi domini Esperanto e data la mobilità dei dispositivi, un *Peer* cliente potrebbe fare riferimento ad un *SADR* memorizzato in un *repository* remoto. Per chiarire le regole del protocollo distinguiamo i seguenti casi:

1) *SADR associato ad un servizio locale al dominio da cui proviene la richiesta*: tutti i *SADR* dei servizi pubblicati da dispositivi residenti all'interno del

dominio sono memorizzati presso l'*Explorer* locale (come mostrato meglio in seguito, anche i *SADR* dei dispositivi in *roaming* presso il dominio sono all'interno dell'*Explorer* locale). Ogni *SAD request* proveniente da un Esperanto *Peer* viene pertanto soddisfatta dall'*Explorer* con una *SAD reply*.

2) *SADR associato ad un servizio remoto al dominio da cui proviene la richiesta*: per soddisfare la richiesta l'*Explorer* chiede il *SADR* all'*Explorer* del dominio *home* del dispositivo (se non è esso stesso); altrimenti la richiesta viene inoltrata al dominio che ospita il dispositivo (il concetto di dominio *home* sarà illustrato al par. 4.2.3). La comunicazione tra gli *Explorer* avviene tramite gli *Esperanto Agent*. Ogni *SAD request* proveniente da un Esperanto *Peer* viene pertanto soddisfatta dall'*Explorer* con una *SAD reply* una volta che l'agente ha ottenuto il descrittore.

I *sequence diagram* di figura 4-6 e 4-7 illustrano graficamente i concetti espressi ai punti 1 e 2.

SCENARIO: *lo scenario di figura 4-6 illustra le regole del protocollo quando un Peer effettua una SAD request di un servizio presente nel dominio dove esso è localizzato. Sono riportate le interazioni degli elementi interni all'Explorer.*

1. Il *Peer* effettua una *SAD request* per ottenere un SADR di uno specifico servizio. L'*Explorer* verifica l'autenticità della richiesta prima di soddisfarla: per questo determina le politiche di accesso tramite il *Security Manager*
2. l'*Explorer* rifiuta la richiesta se il *Peer* non è autorizzato. In caso positivo, viene cercato il descrittore all'interno del *repository*, impostate le politiche di accesso al *Mediator* e restituito il SADR richiesto

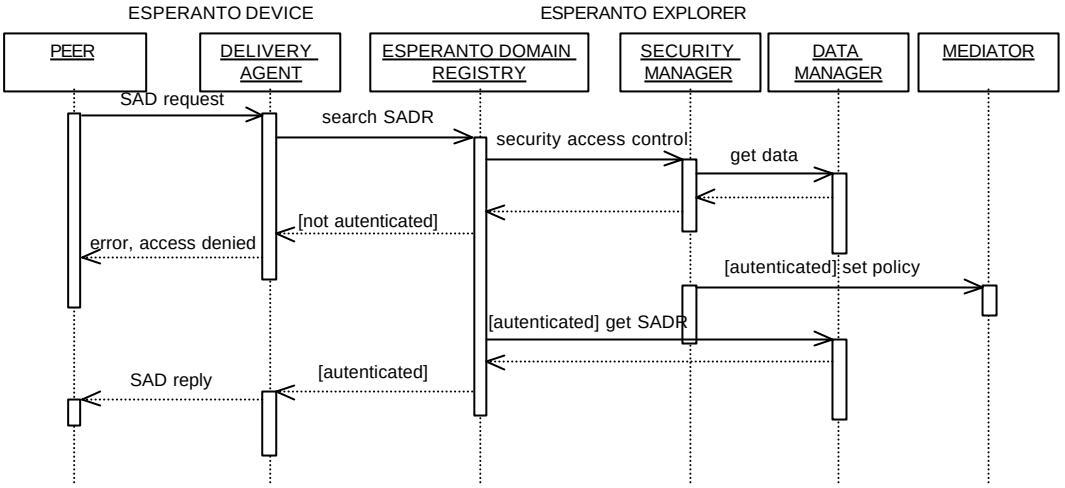

Figura 4-6: *SAD request/reply* relative ad un descrittore locale all'*Explorer*.

SCENARIO: *lo scenario di figura 4-7 illustra le regole del protocollo quando un Peer effettua una SAD request di un servizio remoto. Il servizio è residente su un dispositivo remoto localizzato all'interno del suo dominio home. Per maggiore leggibilità è stato omesso il mediatore*

1. Il *Peer* effettua una *SAD request* per ottenere un SADR di uno specifico servizio
2. l'*Explorer* rifiuta la richiesta se il *Peer* non è autorizzato; se l'autorizzazione è concessa si cerca il SADR all'interno del *repository*: dato che il *record* non è presente (è associato ad un *device* remoto non in *roaming* presso il dominio), la richiesta è inviata all'*Esperanto Agent.* Esso provvederà ad inoltrarla verso l'agente remoto del dominio *home* del dispositivo su cui è localizzato il servizio
3. Dato che il dispositivo è localizzato all'interno del suo dominio *home*, il SADR è recuperato e inviato all'agente che ne ha fatto richiesta

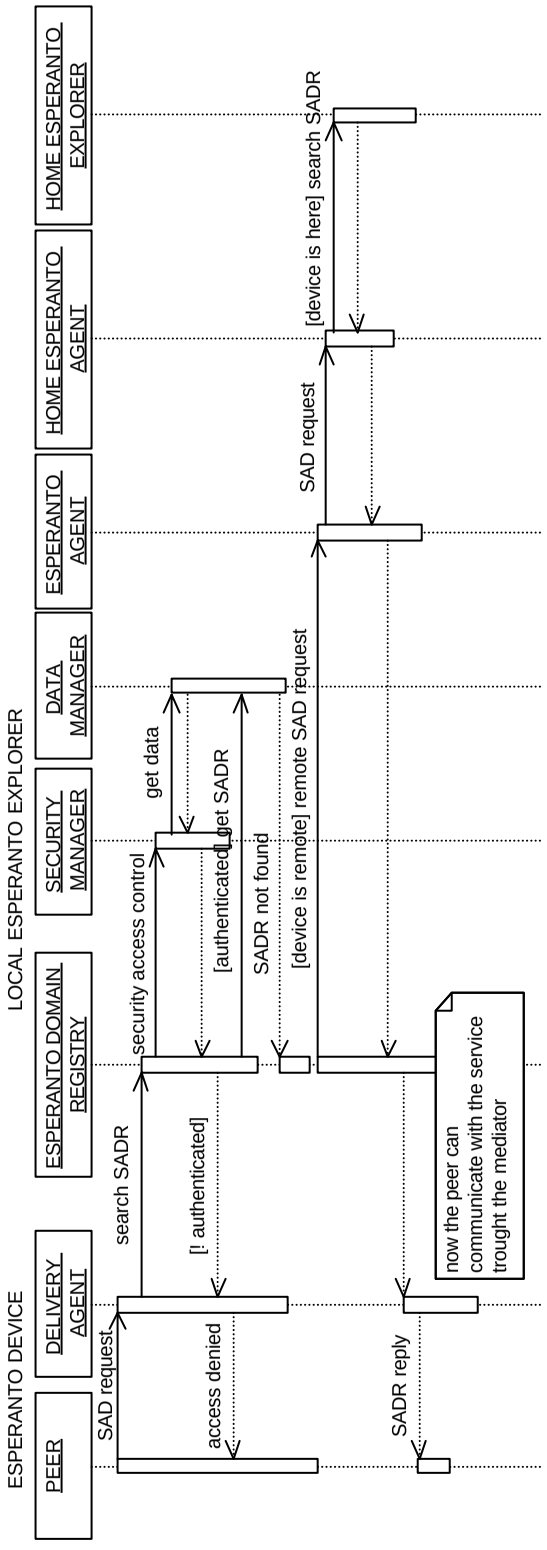

Figura 4-7: *SAD request/reply* relative ad un descrittore remoto all'*Explorer* a cui è pervenuta la richiesta.

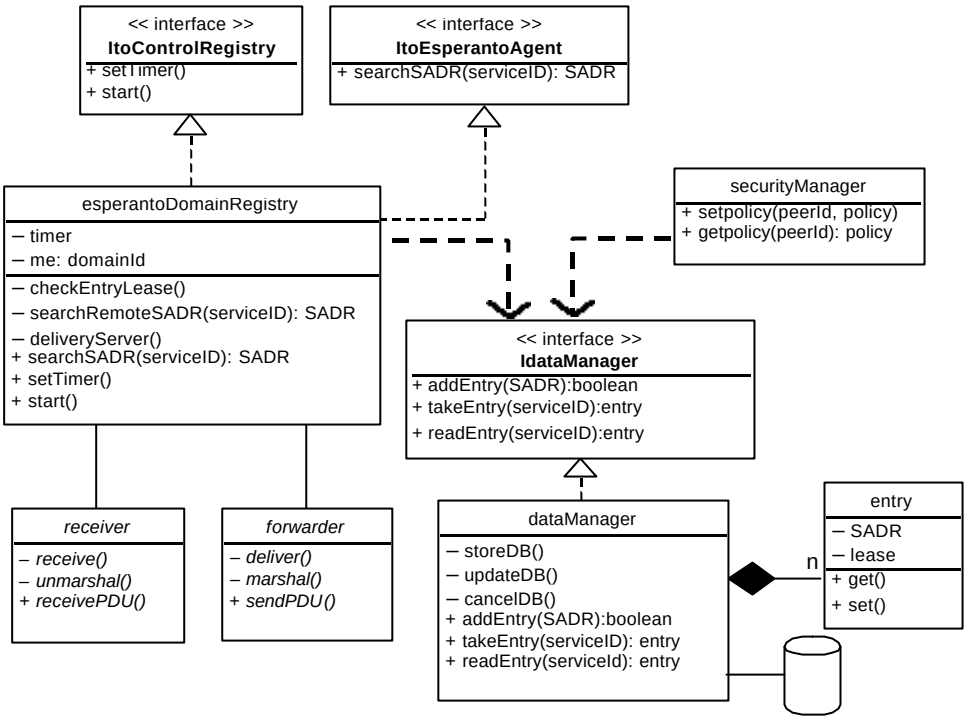

Figura 4-8: *Esperanto Explorer*. Sono riportati i componenti responsabili della gestione del *service access description*.

Come riportato nei *sequence diagram*, l'*Esperanto Explorer* è un componente caratterizzato da una complessa struttura interna. In figura 4-8 è fornita una formalizzazione di alto livello.

In particolare, in figura si riportano solo i componenti dell'*Explorer* cui spettano i compiti di gestione della fase di *service access description*: essa, inoltre, è incompleta dei servizi del protocollo di *discovery*. Con il metodo *start()* vengono attivati i servizi di gestione del protocollo di *delivery*: periodicamente il metodo *checkEntryLease()* stabilisce se ci sono *SADR* associati a servizi scaduti, mentre con *deliveryServer()* si soddisfano le richieste di *service access description*; in particolare con *searchSADR()* si avvia la ricerca di un *SADR* all'interno del *repository* locale; il fallimento di tale ricerca attiva *searchRemoteSADR()*, spostando il raggio d'azione della richiesta sul livello inter-dominio. Il metodo (*searchSADR()*) è esportato verso gli *Esperanto Agent* per consentirgli di soddisfare *SAD request* provienti dai loro pari. Il componente relativo alla gestione della sicurezza non è stato affrontato in maniera profonda:

esso rappresenta un problema aperto nel progetto completo dell'architettura. Lo scambio delle unità di protocollo della fase di *service access description* verso i *Peer* avviene tramite la coppia di *forwarder/receiver*; lo scambio delle unità di protocollo tra gli agenti può avvenire ad esempio tramite l'uso di un ORB.

4.2.2.3 FASE DI *INTERACTION*

L'interazione è la fase in cui due Esperanto *Peer* interagiscono, l'uno per fornire il servizio, l'altro per utilizzarlo. Come osservato nel par. 4.2.1 e nel par. 4.2.2.1, l'interazione avviene tramite scambio di tuple: cliente e servente leggono e scrivono le tuple nello spazio condiviso (rappresentato dal mediatore) del dominio in cui sono localizzati. I dispositivi possono trovarsi nello stesso dominio oppure essere localizzati in domini diversi (anche diversi dai domini *home*): ai *Peer* basta specificare correttamente il formato della tupla (mittente, destinatario, parametri) e fare fede al contratto rappresentato dal *SADR* (in cui sono indicate le azioni da compiere sullo spazio condiviso), ad esempio rispettando i tempi di espirazione delle tuple. L'architettura baderà a fare tutto il resto.

4.2.2.4 ESPERANTO *PEER*

L'Esperanto *Peer* rappresenta la singola entità che offre o utilizza un servizio; più Esperanto *Peer* possono coesistere su uno stesso dispositivo mobile. Un *Peer* utilizza il protocollo di *service delivery* attraverso due componenti presenti su ogni dispositivo Esperanto: il *Subscribe Agent* e il *Delivery Agent*. La struttura del legame tra *Peer* e agenti è riportata in figura 4-9.

L'interfaccia *Isubscribe Agent* consente le seguenti operazioni:

- *assegnare un ID*: ogni *Peer* ha bisogno di essere individuato univocamente all'interno di tutti i domini dell'architettura (per chiarimenti sullo spazio degli identificatori cfr. par. 4.2.3.3);

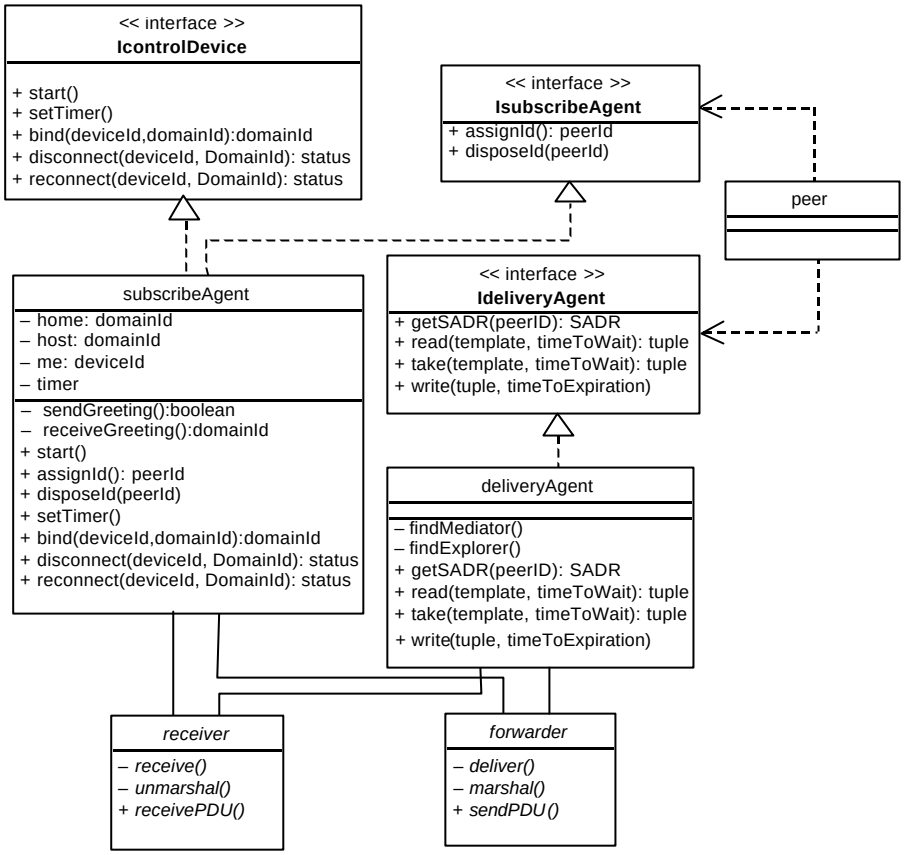

Figura 4-9: Struttura interna di un dispositivo Esperanto

L'interfaccia *IcontrolDevice* permette lo sviluppo di *Peer* di sistema che può controllare le seguenti operazioni:

- *effettuare un processo di hand-off*: un dispositivo Esperanto può disconnettersi volontariamente dall'ambiente e conseguentemente riconnettersi al momento desiderato;
- *cambiare il dominio home del dispositivo Esperanto*: questi aspetti saranno meglio illustrati al par. 4.2.3.2.

Tramite *Idelivery Agent* un *Peer* può compiere le seguenti operazioni:

- *effettuare richieste di descrizione dell'accesso ad un servizio*: quando un descrittore *SADR* è richiesto da un *Peer* (*getSADR()*), il *Delivery Agent* stabilisce una connessione con l'*Esperanto Explorer* (*findExplorer()*) locale e

invoca una *SAD request*. Una volta ottenuto una *SAD reply* ritorna il risultato al *Peer*;

- *interagire con il Peer servente*: una volta ottenuto il *SADR*, il *Peer* cliente conosce le operazioni da compiere sullo spazio condiviso per utilizzare il servizio: quando i metodi *write*, *read* e *take* sono invocati sul *Delivery Agent*, una connessione al *Mediator* locale è stabilita (*findMediator()*) e le singole operazioni compiute.

Nella richiesta di descrizione dell'accesso al servizio, l'ID è noto grazie alle precedenti operazioni di *service discovery*. Il *template* fornito nelle operazioni di *read()* e *take()* consente di prelevare dallo spazio condiviso la tupla che maggiormente le "assomiglia".

La presenza degli agenti (*Subscribe* e *Delivery*) nasconde i dettagli della comunicazione tra il *Peer* e i componenti dell'architettura (*Esperanto Explorer* e *Mediator*). Gli agenti sono resi indipendenti dai meccanismi di trasporto tramite un *forwarder/receiver pattern* [17]; inoltre la coppia di *forwarder* e *receiver* aiuta a ridurre il carico computazionale che ogni dispositivo Esperanto deve sostenere per comunicare con i componenti dell'architettura (ovvero *Esperanto Explorer* e *Mediator*), in quanto non si ricorre a soluzioni più pesanti come *middleware* ad oggetti distribuiti (detti anche ORB – *Object Request Broker*).

4.2.3 Supporto alle migrazioni e disconnessioni

Il concetto di *roaming* nasce con la telefonia mobile: l'idea di tassellare lo spazio è un buon metodo per consentire il riuso delle frequenze, ma d'altro canto introduce il problema dell'*handover*, ovvero l'insieme delle procedure da compiere quando il telefono mobile migra da una cella ad un'altra. Tali procedure sono necessarie per garantire la continuità della comunicazione durante il passaggio del confine che delimita le celle. Con la diffusione delle reti dati *wireless* lo scenario è diventato ancora più complesso: all'*handover* tra celle di

una stessa rete *wireless*, detto orizzontale, si affianca quello verticale, ovvero il processo di *handover* tra celle di reti *wireless* differenti [42].

Tuttavia l'*handover* cui abbiamo appena fatto riferimento, è relativo ad una gestione di basso livello della comunicazione tra i dispositivi mobili. Quando si vuole applicare il concetto di cella ed *handover* in un contesto più ampio quale quello del *mobile computing*, le semplici operazioni previste nei modelli di *handover* per reti di telefonia mobile iniziano ad essere insufficienti. Immaginiamo, infatti, il seguente scenario: lo sviluppo di un servizio di informazione del traffico di una grande città. Assumendo che il centro urbano venga tassellato in tante celle, ogni cella potrebbe (tramite una *base station* di copertura radio) offrire ad un utente segnalazioni sul traffico utili nella scelta del suo tragitto [43]. Tuttavia le procedure di *handover* intraprese quando l'utente cambia *base station* non bastano per fornire un buon servizio. L'utente vorrebbe avere la possibilità di fare delle richieste al centro di controllo del traffico, farsi monitorare il percorso del suo tragitto o potrebbe avere dei privilegi di accesso. Per gestire queste situazioni è importante che nella migrazione da una cella all'altra non si prendano in considerazione solo le informazioni per garantire la continuità della comunicazione tra mobile e stazione; devono essere considerate anche le informazioni relative al contesto in cui avviene l'*handover*.

Il problema dell'*hand over* in Esperanto coinvolge i domini, i dispositivi mobili e i descrittori di servizio. Quando un dispositivo migra da un dominio ad un altro, bisogna prevedere come gestire l'*handover* al fine di garantire il *delivery* dei servizi in corso di esecuzione. D'altra parte un dispositivo mobile potrebbe disconnettersi temporaneamente dal sistema: anche in questo caso vanno intraprese opportune azioni volte a gestire il cosiddetto *handoff*. In poche parole bisogna stabilire quali responsabilità deve avere l'architettura per permettere ad un dispositivo di abbandonare temporaneamente un dominio o di migrare da un dominio ad un altro.

4.2.3.1 GESTIONE DELL'*HANDOVER*

In Esperanto l'*handover* (orizzontale) è un problema pervasivo, nel senso che investe i componenti responsabili del *service discovery*, quelli responsabili del *service delivery* e gli stessi dispositivi mobili. Anche se la gestione delle procedure di *handover* avviene in maniera trasparente al dispositivo Esperanto, esso deve rispettare alcuni compiti:

- memorizzare l'identificatore del suo dominio *home* e del suo dominio *host*: ogni dispositivo Esperanto è assegnato ad un dominio, detto *home* (nell'ambito della telefonia cellulare la cella *home* è introdotta per memorizzare le informazioni di *accounting* del mobile, Esperanto eredita questo concetto per motivi che saranno chiariti in seguito). Il dominio che lo ospita durante la sua permanenza del sistema è detto *host*. Il dominio *home* è introdotto per gestire in maniera efficiente le operazioni di *discovery/delivery* dei servizi inter-dominio;
- segnalare la sua presenza all'*Explorer* di dominio: similmente a come avviene nella telefonia mobile GSM [42] il dispositivo Esperanto invia un messaggio di "*greeting*" alla *base station* di dominio (l'*Esperanto Explorer*) segnalando il suo dominio *home*, il suo dominio *host* e l'identificatore di dispositivo. Il messaggio può essere inviato con due strategie:
  - *Explorer-polling*: il messaggio è inviato su richiesta dell'*Explorer*;
  - *device-polling*: il messaggio è inviato periodicamente dal dispositivo.

  Il *polling* è utilizzato per segnalare la presenza continua del dispositivo nell'ambiente. La soluzione adottata è stata quella *device-polling*.

Ognuno di questi compiti sono assolti dal *Subscribe Agent* di ogni dispositivo; in questo modo l'*Esperanto Explorer* riesce a stabilire se il *device* vuole intraprendere una procedura di *handover* (il dominio *host* conservato dal dispositivo è diverso da quello dell'*Explorer*) e se è in *roaming* (il dominio *home* è diverso da quello *host*). Vediamo i due casi singolarmente:

1. **Procedure di *handover***: durante il processo di *handover*, bisogna in generale prevedere che: il dispositivo *roamer* provenga da un dominio *host* diverso da quello *home,* abbia pubblicato descrittori di servizio e stia interagendo con altri servizi/clienti. L'*handover* consiste nel trasferire le informazioni di stato del dispositivo da un dominio all'altro senza che le operazioni di *discovery* e *delivery* già intraprese siano influenzate. Sono possibili diverse soluzioni:

   - Vengono trasferite tutte le informazioni di stato del dispositivo dal dominio *host* al nuovo dominio *host* (descrittori di servizio dal vecchio *registry*, tuple dal vecchio spazio condiviso ecc.); si segnalano tutti i componenti dell'architettura responsabili del *discovery* e del *delivery* inter-dominio (mediatori e agenti di altri domini) che il dispositivo ha cambiato locazione.

   - Vengono trasferite tutte le informazioni di stato del dispositivo dal dominio *host* al nuovo dominio *host* (descrittori di servizio dal vecchio *registry*, tuple dal vecchio spazio condiviso ecc.); tutti i componenti dell'architettura responsabili del *discovery* e del *delivery* inter-dominio (mediatori e agenti di altri domini) fanno sempre riferimento al dominio

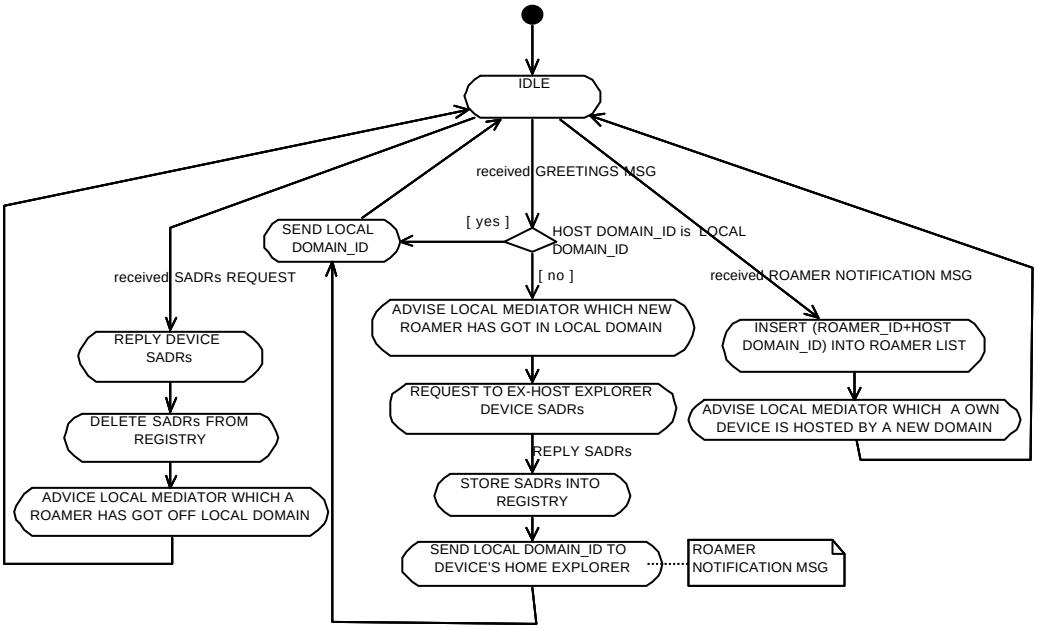

Figura 4-10: *activity diagram* che illustra le operazioni intraprese dall'*Esperanto Explorer* per la gestione dell'*handover* dei dispositivi.

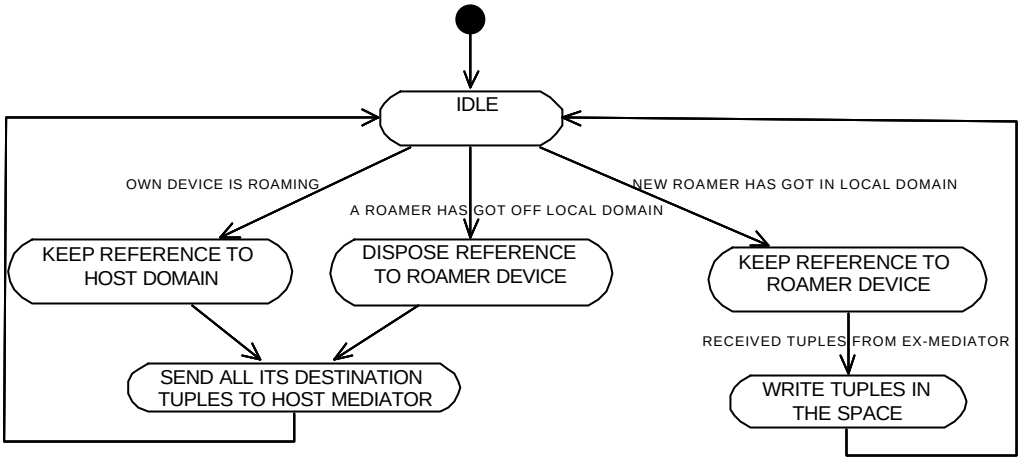

Figura 4-11: *activity diagram* che illustra le operazioni intraprese dal *Mediator* per la gestione dell'*handover* dei dispositivi.

home per soddisfare le loro richieste di scoperta/accesso/interazione remote inoltrandole ad esso. Il dominio *home* tiene traccia del dominio che attualmente ospita il suo *roamer* e si comporta da *rely* quando provengono richieste inter-dominio di interesse per il *roamer*.

Ovviamente la seconda soluzione è da preferire per la maggiore scalabilità e i ridotti tempi di gestione dell'*handover*. L'*activity diagram* di figura 4-10 riassume le azioni compiute dall'*Explorer* (da entrambi i punti di vista: *home* e *host*) ogni volta che bisogna gestire un *handover* di un dispositivo.

Come riportato, il nuovo *host Explorer* deve assolvere i seguenti compiti:

- informare il *Mediator* locale della presenza di un nuovo *roamer*;
- richiedere i descrittori di servizio del *roamer* al vecchio *host Explorer* (in modo da conservarli all'interno del *registry* locale);
- notificare il dominio *home* del dispositivo della sua nuova locazione.

Alla richiesta della lista dei descrittori di un dispositivo, il vecchio *host Explorer* deve soddisfare la richiesta con la seguente procedura:

- inviare la lista all'*Explorer* che ne ha fatto richiesta;
- rimuovere dal *repository* locale i descrittori del dispositivo che ha abbandonato il dominio;

- informare il *Mediator* locale che un *roamer* ha lasciato il dominio.

Quando un *Esperanto Explorer home* viene notificato della nuova locazione di un suo dispositivo (*roamer notification message*) dovrà intraprendere le seguenti azioni:

- aggiornare la lista dei dispositivi in *roaming* con la nuova locazione del dispositivo;
- informare il *Mediator* locale della nuova locazione del *roamer*.

In questo modo può inoltrare correttamente le future richieste di *service access description* rivolte ad esso.

Le azioni intraprese dal *Mediator* durante la fase di *handover* possono essere invece riassunte nell'*activity* di figura 4-11. Come illustrato, le attività svolte dal nuovo mediatore ospite sono le seguenti:

- tenere conto che il dispositivo *roamer* è presso il dominio di sua competenza (quindi non dovrà inoltrare presso alcun dominio le tuple ad esso destinate);
- quando ricevute, memorizzare le tuple che il dispositivo non ha ancora prelevato dal vecchio spazio ospite.

Il vecchio mediatore ospite deve invece svolgere le seguenti azioni:

- cancellare il riferimento al dispositivo *roamer*;
- inviare le tuple destinate al dispositivo (e che non ha ancora prelevato) al nuovo mediatore ospite.

I compiti del mediatore *home* sono invece i seguenti:

- tenere traccia del dominio che ospita il dispositivo ad esso associato;
- inviare le tuple destinate al dispositivo (e che non ha ancora prelevato) al nuovo mediatore ospite;

In questo modo può inoltrare correttamente le tuple che saranno indirizzate ad esso ma per il dispositivo che è in *roaming*.

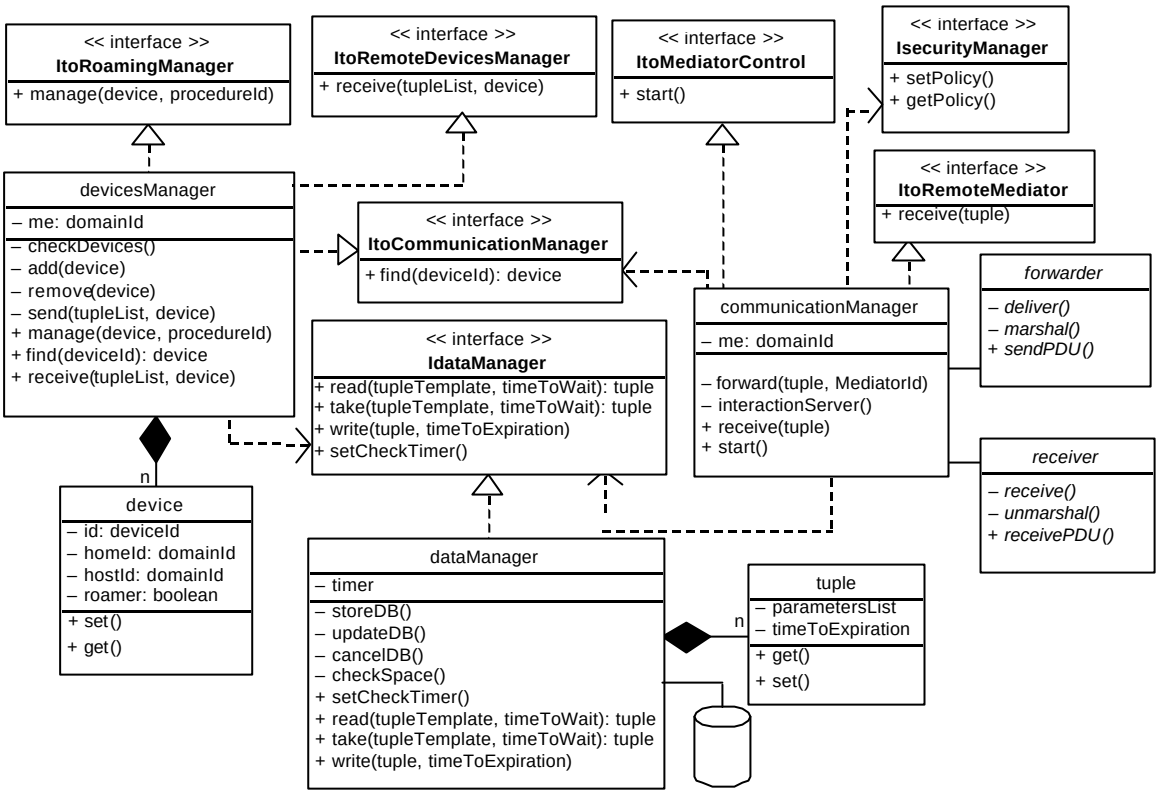

Figura 4-12: *Mediator*. Diagramma delle classi completo

Per comprendere meglio le azioni compiute dal mediatore, in figura 4-12 è riportato il suo diagramma delle classi completo. L'elemento incaricato della gestione delle procedure di *handover* all'interno del mediatore è il *Devices Manager*. L'invocazione del metodo *manage(device,procedureId)* è il meccanismo attraverso il quale l'*Explorer* notifica al *Mediator* locale la procedura di *handover* da compiere (uno dei singoli cammini dell'*activity diagram* 4-11). I metodi *add()* e *remove()* servono a tenere/rimuovere i riferimenti dei dispositivi presenti nel dominio e in *roaming*. Il metodo *send()* (*receive()*) è invocato per inviare (ricevere) le tuple al (dal) nuovo (vecchio) mediatore *host*.

2. **Dispositivi in *roaming***: per illustrare il comportamento dei componenti dell'architettura, distinguiamo i due punti di vista:

   - *Punto di vista del dispositivo roamer*: il dispositivo *roamer* accede alle risorse del dominio (*Explorer* e *Mediator*) come se fossero quelle *home.*

Può quindi accedere al mediatore per scrivere o leggere le tuple per gli altri e dirette ad esso; può rimuovere servizi vecchi o pubblicarne dei nuovi. La gestione del *roamer* è del tutto trasparente ad esso, dopo le operazioni di *handover* esso viene trattato alla stesso modo di un dispositivo *home* all'interno del suo dominio di casa.

- *Punto di vista di un dispositivo qualsiasi*: un dispositivo può richiedere la descrizione di accesso di un servizio residente su un *roamer*. Come sappiamo le richieste di *service access description* possono avere un raggio di azione di due tipi:
  - *intra-dominio*: se il servizio è residente su un dispositivo locale al dominio;
  - *inter-dominio*: se il servizio è residente su un dispositivo remoto.

  Le prime sono soddisfatte dall'*Explorer* cercando tra i descrittori memorizzati nel *repository* locale. Le seconde sono soddisfatte dall'*Esperanto Agent* su richiesta dell'*Explorer*: l'agente inoltra la richiesta ad un pari che possa soddisfargli la richiesta.

  Le richieste intra-dominio possono essere rivolte a dispositivi *home* locali al dominio, oppure a dispositivi *roamer* in *roaming* presso il dominio (da cui proviene la richiesta). La richiesta è dunque soddisfatta recuperando il descrittore dal *repository* dell'*Explorer* e inviandolo al *Peer* richiedente.

  I dispositivi oggetto delle richieste inter-dominio possono essere remoti per due motivi: sono dispositivi *home* in *roaming* (in un dominio remoto), oppure semplicemente dispositivi remoti. Nel primo caso la richiesta è inoltrata all'*Explorer* ospite, che si incaricherà di soddisfare la richiesta. Nel secondo caso la richiesta è inoltrata all'*Explorer home* del dispositivo. La comunicazione tra gli *Explorer* avviene sempre passando tra gli agenti di dominio.

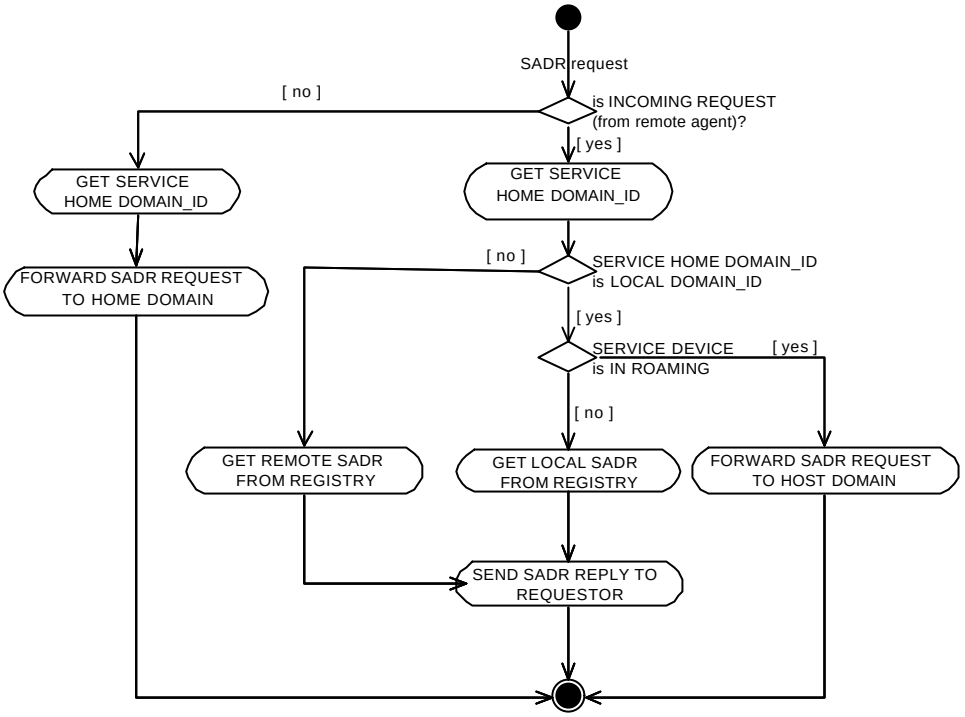

Figura 4-13: il comportamento assunto dall'*Esperanto Agent* nel protocollo di *service access description.* E' prevista anche la gestione dei dispositivi in *roaming*.

L'*activity diagram* 4-13 riassume il comportamento dell'*Esperanto Agent*, mentre il *sequence diagram* di figura 4-14 riporta le interazioni dovute ad una richiesta di *service access description* di un servizio di un dispositivo in *roaming*.

SCENARIO: *nello scenario di figura 4-14 un Peer richiede la descrizione dell'accesso ad un servizio localizzato su un device remoto in roaming in un dominio diverso da quello da cui proviene la richiesta.*

1. Il *Peer* invia una SAD *request* (*Service Access Description request*) all'*Esperanto Explorer* locale
2. L'*Explorer* non trova il servizio nel suo *repository* ed invia la richiesta all'*Esperanto Agent* locale
3. L'agente locale inoltra la richiesta al pari del dominio *home* del servizio
4. Il dispositivo su cui è localizzato il servizio è in *roaming*: l'agente del dominio *home* inoltra la richiesta al pari del dominio *host* in cui risiede attualmente il dispositivo
5. L'agente del dominio *host* risponde con il SADR richiesto

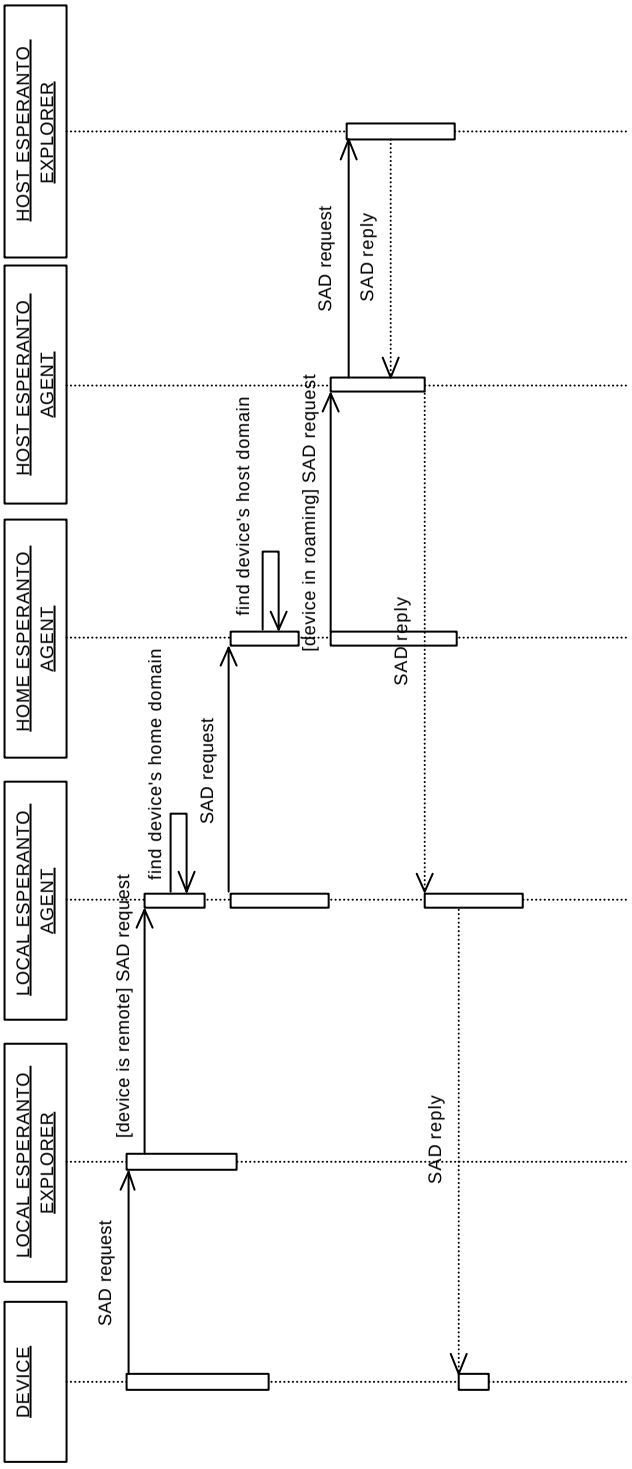

Figura 4-14: *SAD request/reply* relative ad un servizio residente su un dispositivo in *roaming*..

Delle operazioni di *interaction* (ovvero scambio di tuple tra *client* e *service*), l'*handover* influenza solo la gestione delle tuple indirizzate ai dispositivi *roamer*:

- in generale il mediatore lascia le tuple nel suo spazio condiviso se il destinatario è locale al suo dominio di competenza;
- se il dispositivo destinatario delle tuple è un *home* in *roaming*, allora il mediatore deve inoltrare le sue tuple al dominio *host* presso cui il dispositivo è ospitato;
- se il dispositivo destinatario delle tuple non è *home* e non è in *roaming* presso di sé, allora il mediatore deve inoltrare le tuple verso il mediatore *home* del dispositivo.

Grazie alle responsabilità del *Devices Manager*, il *Communication Manager* può stabilire come inoltrare le tuple quando un dispositivo accede in scrittura al suo spazio condiviso. L'*activity diagram* di figura 4-15 e il *sequence diagram* di figura 4-16 aiutano a comprendere il comportamento del *Mediator* quando una

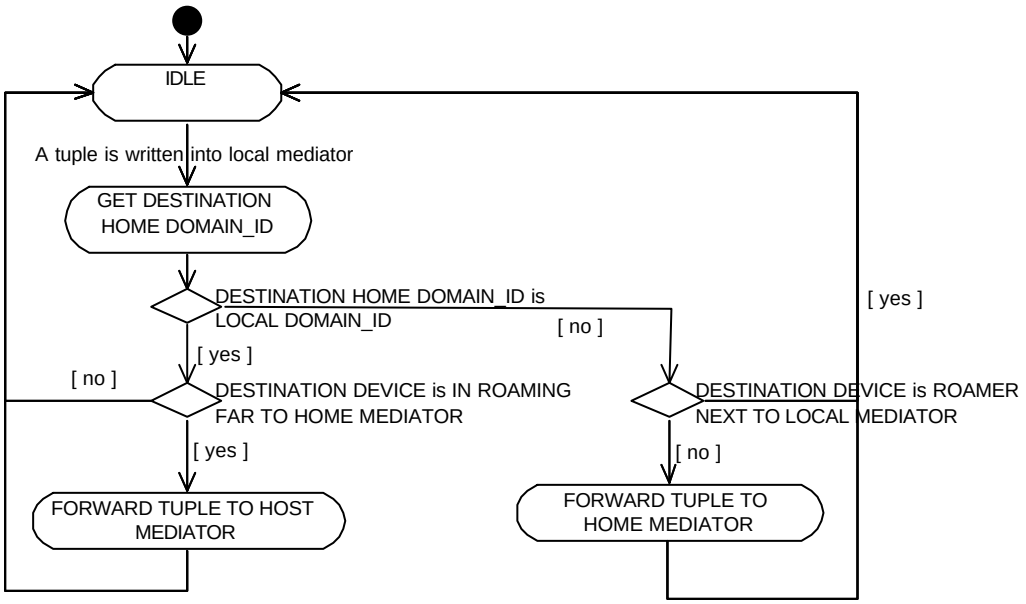

Figura 4-15: *activity diagram* delle operazioni intraprese dal *Mediator* quando deve stabilire il suo pari destinatario delle tuple. E' prevista la gestione dei dispositivi in *roaming*.

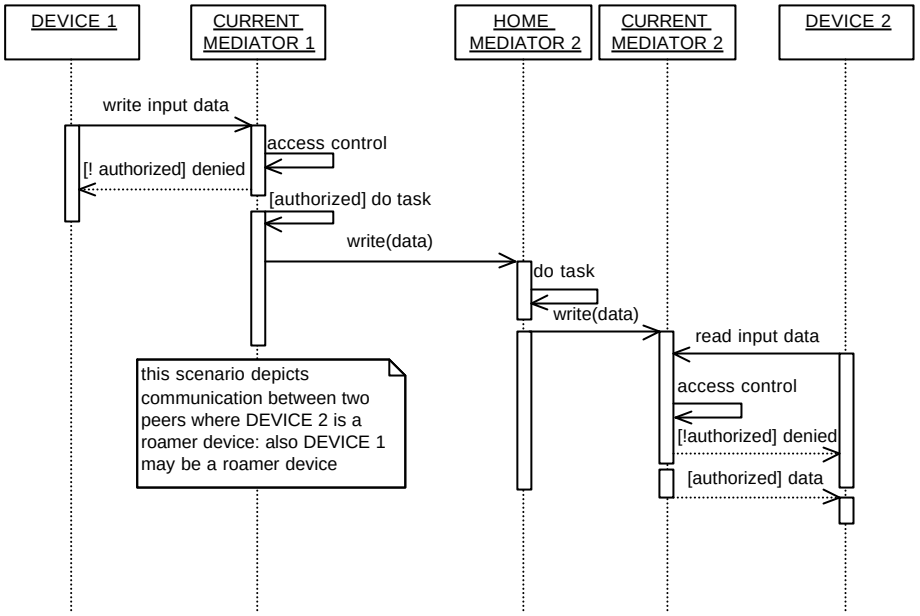

Figura 4-16: *interaction* tra due *Peer* Esperanto di cui uno residente su un dispositivo in *roaming*.

nuova tupla è scritta nel suo spazio condiviso.

SCENARIO: *lo scenario illustrato in figura 4-16 riporta l'interazione tra due Esperanto Peer. Peer 1 su device 1 è nel suo dominio home; Peer 2 su device 2 è in roaming presso un dominio diverso da quello di Peer 1*.

1. Un *Peer (*sul *device* 1*)* accede ad un servizio remoto: scrive le tuple di *input* nel suo *current Mediator*
2. Il mediatore determina che il dispositivo 2 non appartiene al dominio e non è *roamer* presso di sé, per cui inoltra i dati verso il mediatore *home* del dispositivo 2
3. Il dispositivo 2 è in *roaming.* Il mediatore *home* del dispositivo 2 risale al dominio *host* dello stesso ed inoltra i dati verso tale dominio

4.2.3.2 GESTIONE DELL'*HANDOFF*

Con il termine *handoff* si intende il processo di disconnessione volontaria di un dispositivo mobile da un dominio Esperanto. Anche se spesso il termine *handoff* è confuso con quello di *handover* (in Nord America l'*handoff* è inteso come *handover* orizzontale), in questo contesto avranno significati diversi.

Mentre l'*handover* è necessario a garantire la comunicazione senza interruzioni nel passaggio da una cella ad un'altra, l'*handoff* è utile in tutte quelle situazioni dove non è necessario che il dispositivo sia connesso al sistema (ad esempio per preservare le batterie, per motivi di costo, ecc.). La procedura di *handoff* prevede che, prima di perdere ogni riferimento al terminale mobile, venga salvato un insieme minimo delle sue informazioni di stato indispensabili alla successiva riconnessione al sistema. Dato che l'iniziativa per l'attivazione del processo è del terminale mobile, Esperanto offre le seguenti primitive per la disconnessione e la riconnessione volontaria:

disconnect(deviceId, DomainId): status
reconnect(deviceId, DomainId): status

Con l'invocazione di queste primitive, il dispositivo interagisce con l'*Explorer* di dominio. La *disconnect()* prevede che vengano compiute le seguenti azioni:

- il dispositivo memorizzi le informazioni del messaggio di *greeting*: identificativo di *device*, di dominio *home* e di dominio ospite (ovvero dove avviene l'*handoff*);
- indichi all'*Explorer* il suo identificativo e il suo dominio *home.*

Il parametro di uscita della primitiva indica se la richiesta di disconnessione è stata accettata.

- L'*Explorer* ha compiti diversi in funzione se esso è quello *home* oppure *host*:
  - *Home Esperanto Explorer*: se l'*handoff* avviene nel dominio *home*, bisogna solo tenere traccia che il dispositivo è stato disconnesso. *Explorer* e *Mediator* (sotto notifica dell'*Explorer*) dovranno poi conservare rispettivamente descrittori di servizio e tuple del dispositivo fino alla loro scadenza (come avviene di consueto).
  - *Host Esperanto Explorer*: se l'*handoff* avviene in un dominio *host*, tutte le informazioni relative al dispositivo (descrittori di servizio, tuple in cui esso è mittente o destinatario) dovranno ritornare nel dominio *home*, dove si terrà traccia che il dispositivo è stato disconnesso. Pertanto l'*Explorer*

rispedirà queste informazioni al suo pari, segnalando anche al mediatore di fare lo stesso.

La primitiva *reconnect()* può essere invocata quando il dispositivo è nel suo dominio *home*, o in generale in un dominio *host* qualsiasi (il parametro *domainId* della *reconnect()* indica il dominio di casa del dispositivo):

- dominio *home*: con la primitiva *reconnect()* il dispositivo ritorna attivo nell'ambiente; lo stato dei descrittori e delle tuple dipende dal tempo trascorso dall'ultimo *handoff*;
- dominio *host*: in questo caso la primitiva *reconnect()* consiste in una procedura di *handover* di un dispositivo che sta migrando dal suo dominio *home* verso un dominio *host* qualsiasi.

Il parametro di uscita della primitiva indica se la richiesta di riconnessione è stata accettata.

In figura 4-17 riportiamo i componenti dell'*Esperanto Explorer* responsabili della gestione delle procedure di *handoff* ed *handover*.

Con il metodo *start()* vengono attivati i servizi di gestione delle procedure di *handover* ed *handoff*: attraverso *greetingServer()* vengono ricevuti/inviati periodicamente i messaggi di *greeting* provenienti dai dispositivi Esperanto; con *bindServer()* vengono soddisfatte le richieste di *binding*, disconnessione e riconnessione ad un dominio da parte di un dispositivo.

In base al contenuto del messaggio *greeting*, il *manager* è in grado di stabilire se il dispositivo mittente è in *roaming* oppure no. La procedura di *handover* è applicata dal metodo omonimo e prevede che:

- per dispositivo *roamer* che entra (esce) nel (dal) dominio bisogna:
  - richiedere (inviare) i descrittori ad esso associati al vecchio (nuovo) dominio ospite: i descrittori sono richiesti (inviati) tramite un *entryRequest()* (*entryReply()*); essi vengono registrati nel (rimossi dal) *repository* locale utilizzando l'interfaccia *IdataManager*.

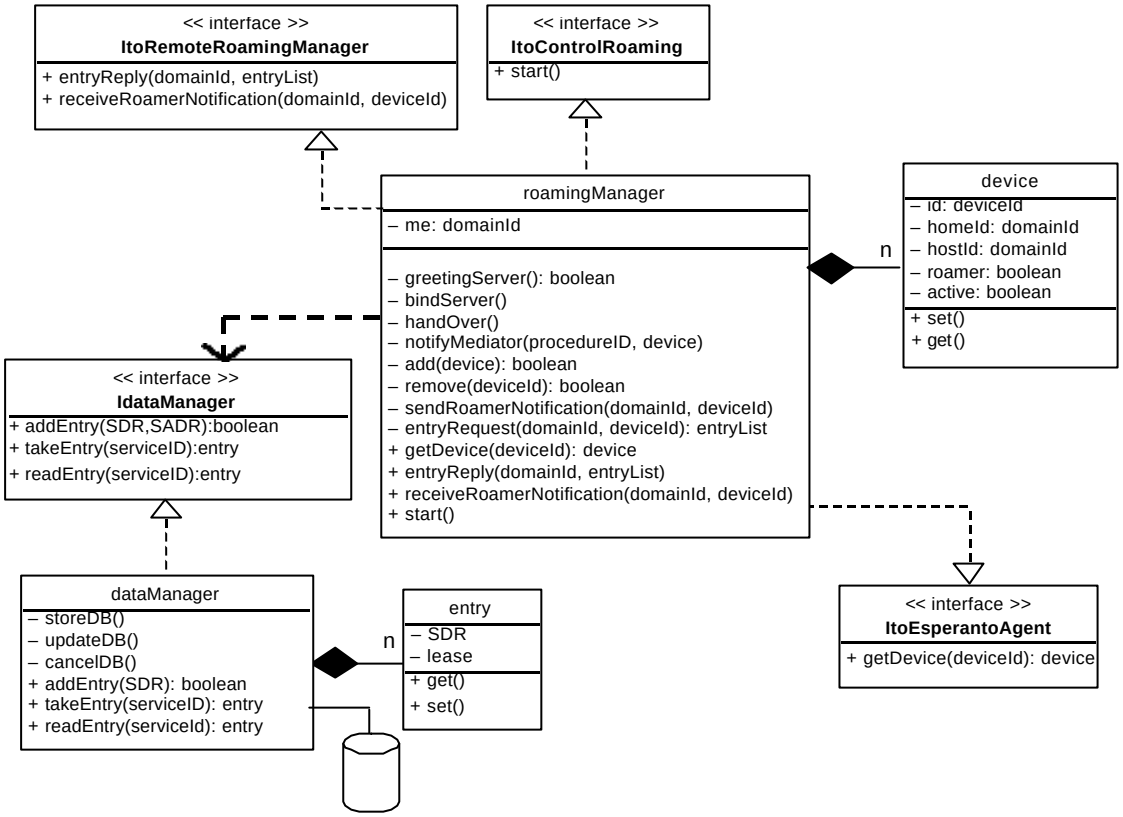

Figura 4-17: I componenti dell'*Esperanto Explorer* responsabili delle procedure di *handover* ed *handoff*.

- notificare il mediatore locale che un dispositivo è giunto nel (ha lasciato il) dominio: un codice di questa situazione è utilizzato in *notifyMediator()*
- notificare il cambio di locazione al suo dominio *home: sendRoamerNotification()*

- alla notifica del cambio di locazione di un proprio dispositivo *home* bisogna:
  - informare il mediatore locale mediante un messaggio di notifica (*notifyMediator()*)
  - tenere traccia del dominio ospite in cui si trova il dispositivo

I metodi *add()* e *remove()* servono a supporto delle procedure di *handover* per la gestione della lista dei dispositivi presenti nel dominio (*roamer* e *home*) e dei dispositivi *home* in *roaming* nei domini remoti. Il metoto *getDevice()* è esportato

all'*Esperanto Agent* per consentirgli di soddisfare richieste di *service access description* inter-dominio (vedi *activity diagram* in figura 4-13).

4.2.3.3 SPAZIO DEGLI IDENTIFICATORI

In generale nella definizione di un'infrastruttura *software* distribuita è necessario stabilire come due componenti dell'architettura devono individuarsi reciprocamente. Molte soluzioni sono possibili: ad esempio, disponendo di un *directory* che memorizza le associazioni *(nome, locazione)*, è possibile fare riferimento ad esso per cercare l'elemento con cui si è interessati ad interagire. Altre soluzioni prevedono che è lo stesso identificatore dell'elemento a stabilire come localizzarlo.

Per semplificare la gestione dei dispositivi nei singoli domini dell'architettura, almeno ad alto livello, Esperanto si orienta alla seconda soluzione descritta. L'identificatore di ogni componente del sistema (dispositivi, *Mediator*, *Agent* e *Explorer*) deve rispettare le seguenti caratteristiche:

- ogni agente o mediatore deve avere un proprio identificatore: attraverso questo *id* è possibile individuare l'agente o il mediatore di un particolare dominio Esperanto. Lo spazio degli identificatori di dominio (ovvero di agente e mediatore) può essere generato in un modo qualsiasi, nel rispetto della proprietà suddetta;
- ogni dispositivo deve avere un proprio identificatore: attraverso questo *id* è possibile individuare il dispositivo indipendentemente dal dominio in cui esso è localizzato. Lo spazio degli identificatori di dispositivo può essere generato in un modo qualsiasi, nel rispetto della proprietà suddetta;
- ogni *Peer* deve avere un proprio identificatore: attraverso questo *id* deve essere possibile individuare il *Peer* all'interno dell'architettura. Lo spazio degli identificatori dei *Peer* deve essere generato in modo da prevedere che dall'*id* di *Peer* si possa risalire al dispositivo su cui risiede e al suo dominio *home.*

PEER INDEX	DEVICE ID	HOME DOMAIN ID

Tabella 4-1: l'identificatore di un *Esperanto Peer*.

Per rispondere ai requisiti richiesti, il formato dell'identificatore di un *Peer* è quello riportato in tabella 4-1. Il *PeerIndex* consente di individuare il *Peer* sul dispositivo *DeviceId* il cui dominio *home* è *HomeDomainId*. L'esigenza di un tale identificatore è giustificata dai seguenti motivi:

- bisogna individuare il *Peer* tra tutti quelli residenti sul singolo dispositivo;
- bisogna individuare il *device* tra tutti quelli localizzati nel singolo dominio;
- nelle operazioni di *service delivery* è importante individuare rapidamente il dominio *home* dei dispositivi coinvolti nell'interazione.

Alla luce di queste considerazioni, osservando gli *activity diagram* riportati in precedenza si evince la necessità di definire un identificatore di *Peer* così come presentato.

4.2.3.4 INFORMAZIONI DI STATO DI UN DISPOSITIVO

Le informazioni dello stato di un dispositivo possono essere suddivise in:

- Statiche:
  - il suo dominio *home*
  - l'identificatore di dispositivo
- Dinamiche:
  - i descrittori del servizio (quelli utili al *service discovery* e al *service access description*)
  - le tuple indirizzate/inviate ad/da esso

- il suo attuale dominio *host* ovvero la posizione fisica del dispositivo (tale locazione fisica va intesa in termini di dominio Esperanto)
- lo stato del dispositivo (connesso, disconnesso)

Le informazioni di stato statiche sono conservate presso il dispositivo; quelle dinamiche, presso i componenti dell'architettura (*Mediator*, *Explorer*), dove sono memorizzate nelle liste gestite rispettivamente da *Devices Manager* e *Roaming Manager*. Durante le procedure di *handover* e di *handoff* le informazioni di stato statiche sono necessarie per il trasferimento delle informazioni di stato dinamiche.

4.3 Supporto alla dinamicità del contesto

In letteratura esistono diverse definizioni di contesto; nell'ambito di questo lavoro, esso verrà inteso come l'insieme delle informazioni legate alle risorse interne ed esterne ad un dispositivo, in grado di influenzare lo stato e dunque il comportamento di un'applicazione [39].

In ambienti mobili la dinamicità del contesto rende poco efficace l'interazione tra due entità se le tecniche di *delivery* non prevedono opportune strategie di adattamento. L'*adaptation* è la capacità con cui un'entità si adatta ai cambiamenti dei contesti in cui si trova ad evolvere. Le strategie di *adaptation* possono essere di due tipologie:

Application-transparent: il sistema ha completa responsabilità di adattarsi al contesto, provvedendo inoltre alla gestione delle risorse. Questo approccio è attraente in quanto rende possibile l'utilizzo delle applicazioni "*legacy*" negli ambienti di elaborazione emergenti; tuttavia le decisioni intraprese dal sistema, quando l'adattamento è richiesto, possono essere controproducenti o inadeguate agli scopi delle applicazioni [37].

Application-aware: con questa strategia l'adattamento ha luogo grazie alla collaborazione tra le applicazioni e il sistema. Questo approccio permette alle

applicazioni di essere autonome nel processo di adattamento, ma la responsabilità di monitoraggio delle risorse e notifica dei loro cambiamenti è lasciata al sistema [37].

Nell'ambito degli ambienti mobili possono esserci diverse tipologie di applicazioni (dalle applicazioni tradizionali alle nuove applicazioni *location-aware*); è importante, quindi, che le procedure di adattamento siano stabilite dalle applicazioni.

Anche la soluzione di *delivery* Esperanto si orienta alle strategie di *adaptation application-aware*, identificando (nel progetto iniziale) come informazioni caratteristiche del contesto, le seguenti risorse:

- la connettività di rete offerta ad un dispositivo;
- lo stato dell'alimentazione di un dispositivo;
- la locazione fisica di un dispositivo.

4.3.1 Modello di *adaptation application-aware*

In generale, il processo di *adaptation* è guidato dalle seguenti attività [39]:

1. ***Recupero e monitoraggio delle risorse del contesto***: per stabilire l'adattamento alla variazione di una o più risorse del contesto, è necessario definire come monitorare ed ottenere lo stato delle risorse che lo caratterizzano. Allo scopo di stabilire le tecniche di monitoraggio e recupero delle risorse è necessario definire rigorosamente i seguenti elementi:
   a) *un modello di rappresentazione del contesto*: bisogna individuare quali sono le caratteristiche di contesto in grado di influenzare le applicazioni e come rappresentarle. A titolo esemplificativo, se per contesto viene inteso solo la connettività di rete, il modello dovrà formalizzare quali informazioni scegliere nella caratterizzazione della risorsa (larghezza di banda, latenza, affidabilità, …) e come rappresentarle (rispettivamente *bit per seconds*, *msec*, *bit error rate*, ad esempio).

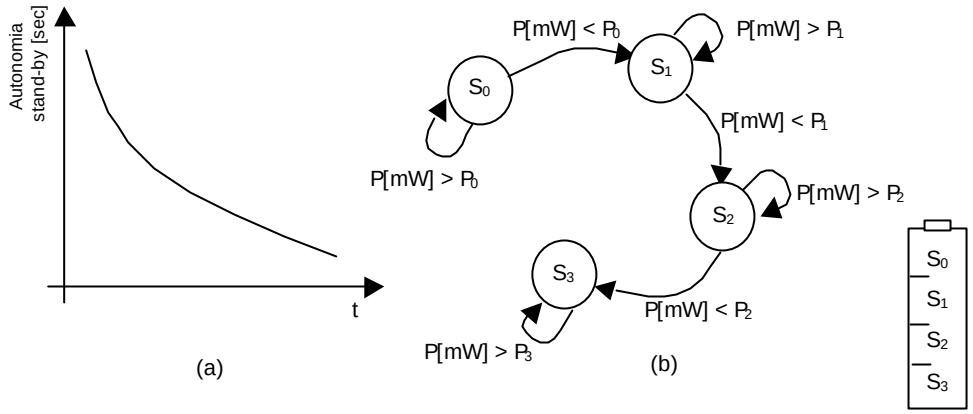

Figura 4-18: modelli di evoluzione dell'autonomia in *stand-by* di un dispositivo; a) evoluzione continua, b) evoluzione discreta.

b) *un modello per l'evoluzione del contesto*: il modello al punto a) definisce una rappresentazione statica del contesto; il suo comportamento dinamico è descritto quando per ogni sua risorsa viene stabilito come caratterizzare la sua evoluzione nel tempo. Ad esempio, l'alimentazione di un dispositivo può essere caratterizzata da diversi elementi: il tempo di autonomia residua in *stand-by* o il tempo di autonomia residua in attività. L'evoluzione di queste grandezze (ad esempio l'autonomia in *stand-by*) può essere rappresentata con modelli continui nel tempo (figura 4-18 (a): *nel tempo l'autonomia del dispositivo si riduce secondo il degrado delle batterie*) oppure con modelli discreti in cui s'identificano un numero finito di stati della risorsa e gli eventi che fanno transitare la risorsa da uno stato all'altro (figura 4-18 (b): *l'autonomia diminuisce a causa del consumo di potenza: dallo stato di max autonomia si passa alla autonomia medio-alta quando la potenza scende al di sotto della soglia P_0*).

2. ***Decisione delle regole di adattamento***: le applicazioni devono cambiare la loro configurazione in funzione dello stato corrente del contesto. Vanno dunque stabilite quali decisioni intraprendere per effettuare tale cambiamento. In questo caso bisogna definire come e chi deve decidere l'adattamento al contesto. In una strategia *application-aware* bisogna fornire alle applicazioni un modo per formalizzare le regole da applicare per reagire alle mutate condizioni del contesto.

3. ***Applicazione delle regole individuate***: una volta stabilite le regole per adattarsi al contesto, tali regole devono essere applicate per modificare il comportamento del sistema o dell'applicazione (in funzione del tipo di strategia). Nel caso di *application-aware adaptation*, bisogna stabilire come applicare tali regole di adattamento al comportamento dell'applicazione.

Nel rispetto di questo modello, in Esperanto si adottano i due approcci descritti di seguito:

- ***Application-driven***: le applicazioni possono monitorare direttamente lo stato della risorsa cui sono interessate e reagire in maniera proattiva al suo cambiamento. Il sistema si occupa di monitorare la risorsa e fornire i meccanismi per accedere alla sua rappresentazione (nel caso della connettività di rete, ad esempio, il sistema si fa carico di stimare parametri come latenza, *throughput*, affidabilità, e poi di presentarli all'applicazione ad un elevato livello di astrazione sotto forma di indice dello stato del canale).
- ***Event-driven***: le applicazioni devono stabilire le regole da applicare a fronte del cambiamento della risorsa cui sono interessate e renderle disponibili al sistema (approccio reattivo). Il sistema deve poi farsi carico di monitorare la risorsa del contesto implicata dalle regole di adattamento stabilite dall'applicazione, e attivarle all'atto del mutamento di contesto. In questo caso, ad esempio, l'adattamento alla connettività di rete avviene secondo una modalità più complessa: l'applicazione segnala al sistema l'interesse per una notifica quando l'indice di stato del canale non rispetta opportuni vincoli (ad esempio scende sotto una determinata soglia); produce le regole da applicare a fronte del cambiamento (ad esempio sotto forma di *handler*, un segmento di codice, che modifica i parametri della connessione influenzati dal contesto) ed inizia a compiere le elaborazioni previste. Quando le condizioni specificate dall'applicazione sono verificate, il sistema attiva l'*handler* di gestione, che modifica l'elaborazione prevista dall'applicazione coerentemente al mutamento avvenuto.

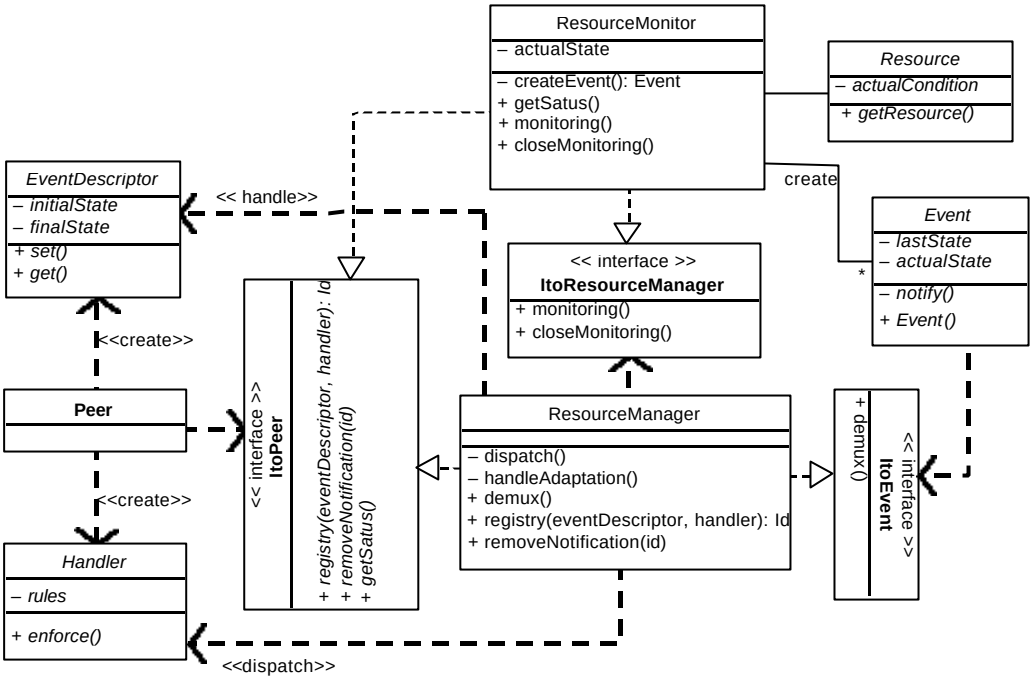

Figura 4-19: *Pattern* per le strategie di *adaptation application-aware.*

In figura 4-19 si riporta il *pattern* della soluzione proposta in Esperanto: un *Peer* può essere consapevole dello stato di una particolare risorsa, *Resource* (connettività di rete, locazione del dispositivo, …). La strategia di adattamento è *application-driven* se il *Peer* usa i servizi del *Resource Monitor*, è *event-driven* se usa i servizi del *Resource Manager*. Analizziamo i due approcci separatamente:

Application driven: una classe *Resource Monitor* per ogni risorsa (*Resource*) rappresentata nel contesto, permette il suo "*recupero e monitoraggio*". In generale ogni risorsa è descritta da diversi parametri che evolvono nel tempo (*actualCondition*). Il compito del *Resource Monitor* è quello di recuperare queste informazioni e presentarle alle applicazioni ad un elevato livello di astrazione ogni volta che ne fanno richiesta (attraverso il metodo *getStatus()*). L'astrazione è ottenuta codificando le combinazioni dei parametri della risorsa con un'informazione che ne rappresenta sinteticamente lo stato. A valle dell'invocazione del metodo *getStatus()*, i *Peer* sono consapevoli dello stato della risorsa e conseguentemente possono procedere con l'adattamento.

Event driven: la classe *Resource Manager* controlla lo stato della risorsa attraverso l'ausilio del suo *Resource Monitor*. L'interesse per la notifica del cambiamento dello stato di una risorsa, avviene da parte delle applicazioni attraverso l'operazione di *registry()*. Nell'operazione di *registry* al *Resource Manager*, i *Peer* forniscono un descrittore con cui si esprime il tipo di cambiamento cui si è interessati (*EventDescriptor*), e l'*handler* che il *Resource Manager* deve attivare all'atto della verifica del cambiamento dello stato della risorsa. Compilando l'*EventDescriptor* in tutti i suoi attributi, i *Peer* manifestano il loro interesse alla notifica dell'evento in cui la risorsa transita dallo stato *initialState* (condizione necessaria affinché accada l'evento) allo stato *finalState* (condizione creata a causa dell'evento). Qualora lo stato iniziale (finale) non dovesse essere importante, perché si è interessati a tutte le transizioni da (in) uno stato iniziale (finale) qualsiasi ad (da) uno specifico stato finale (iniziale) (dunque non ad un evento ma ad una famiglia di eventi), allora lo stato iniziale (finale) può essere omesso. Il metodo *registry()* attiva (all'atto della prima registrazione) l'operazione di monitoraggio della risorsa, compiuta dal *Resource Monitor* attraverso il

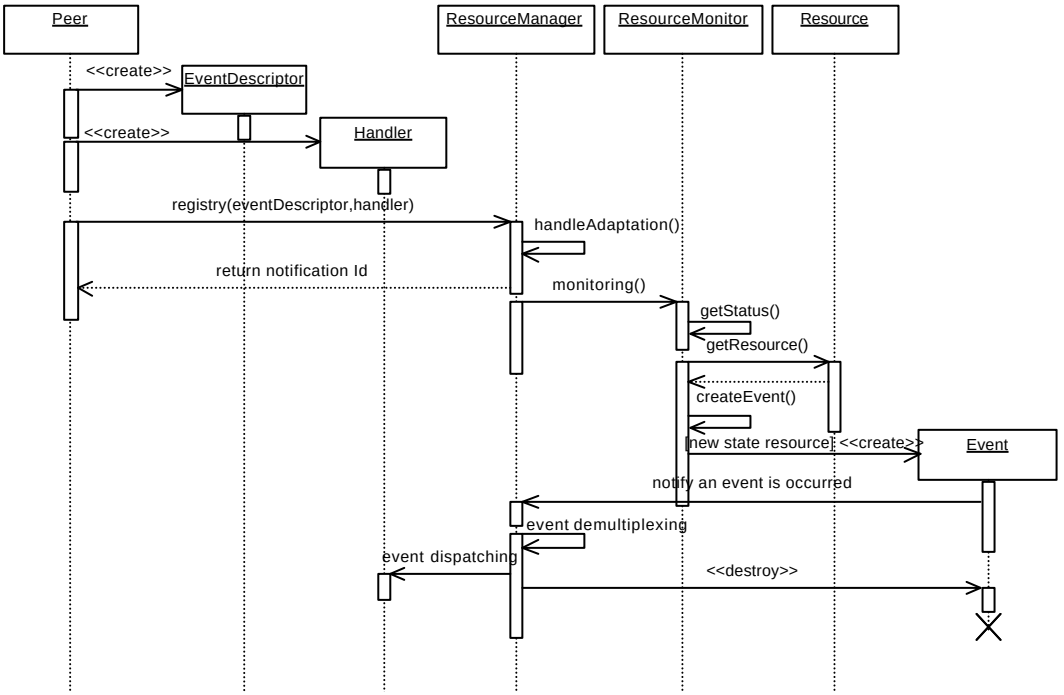

Figura 4-20: l'approccio *event-driven* nel supporto alla dinamicità.

metodo *monitoring()*. Il cambiamento di stato della risorsa durante la sua evoluzione implica l'occorrenza di un evento; pertanto il *Resource Monitor* crea un oggetto *Event* e memorizza al suo interno due informazioni utili al *demultiplexing* da parte del *Resource Manager*: lo stato da cui l'evento ha avuto luogo e lo stato in cui la risorsa si trova a seguito della sua occorrenza. Una volta che l'evento viene notificato al *Resource Manager* (tramite il metodo *demux()*), gli *handler* per il suo trattamento saranno attivati (attraverso il metodo *dispatch()*) se esistono *Peer* interessati alla transizione occorsa. Ogni *handler* codifica le regole di adattamento al cambiamento della risorsa, pertanto l'*adaptation* è guidata dall'evento. Come già anticipato, è possibile prevedere che le regole di adattamento modifichino solo i parametri usati nell'elaborazione e influenzati dal cambiamento. In figura 4-20 è riportato un *sequence diagram* per descrivere graficamente i concetti riportati.

Nei paragrafi successivi sono fornite alcune indicazioni su come rappresentare le risorse del contesto definito in Esperanto, coerentemente al modello espresso dal *pattern* riportato in figura 4-19.

4.3.2 *Network adaptation*

Quando un dispositivo mobile entra in un ambiente rumoroso o in una zona d'ombra potrebbe essere utile che venga notificato della variazione delle caratteristiche della connettività di rete (*network adaptation*). Ad esempio, un'applicazione di *streaming* video può adattarsi a questa condizione riducendo il tasso di spedizione delle *frame*; un *Web browser*, invece, potrebbe richiedere l'invio di versioni degradate delle immagini di una pagina web [38].

La connettività di rete è tipicamente descritta dai tre seguenti parametri:

a) *delay*: è un parametro legato alla latenza della rete;

b) *delay variation*: è legato alla varianza del *delay*;

c) *information lossy*: rappresenta l'affidabilità del canale.

In base a questi tre parametri si può risalire ad uno stato della risorsa "rete": ad esempio con a), b), e c) molto grandi sicuramente è possibile dire che la rete si trova in uno stato "*worst*". Viceversa è possibile dire che lo stato di salute è "*good*".

Il ***recupero*** della risorsa, si traduce pertanto nella realizzazione di una classe *Resource*, per la connettività di rete, la cui responsabilità è quella di stimare le grandezze a), b), e c) (ogni volta che è richiesto tramite il metodo *getResource()*).

Il suo ***monitoraggio*** avviene tramite il *Resource Monitor* associato: ad esso spetta il compito di codificare lo stato dei parametri a), b) e c) in un'informazione più sintetica e comprensibile ai *Peer*; ad esempio, indicazioni come "*worst*", "*worse*", "*bad*", "*normal*", "*good*", "*better*", "*best*", indicano lo stato complessivo della connettività di rete in maniera senz'altro più semplice rispetto una sequenza di numeri. Con questa soluzione, nella creazione degli *EventDescriptor* relativi alla risorsa "connettività di rete", i *Peer* dovranno specificare la transizione tra gli stati che deve far scattare la notifica e l'attivazione dell'*handler*.

4.3.3 *Energy adaptation*

L'energia è una risorsa vitale per il *mobile computing*. I recenti vantaggi nella tecnologia delle batterie e nel progetto di circuiti *low-power* non possono da soli andare incontro al risparmio energetico dei dispositivi mobili: anche le applicazioni devono essere coinvolte. Avere consapevolezza dello stato attuale di alimentazione (*energy adaptation*) di un dispositivo può avere diversi vantaggi: come dimostrato in [40], un degradamento della qualità nel *delivery* di dati multimediali può consentire una vita più lunga alle batterie durante il loro funzionamento; d'altra parte essere a conoscenza del loro stato può aiutare inoltre nell'intervento tempestivo di procedure di *handoff*.

Lo stato di alimentazione di un dispositivo può essere descritto da diversi parametri, tra i quali è utile menzionare i seguenti:

a) *potenza residua della batteria*: data la potenza massima nominale della batteria del dispositivo, questo parametro dà una indicazione della frazione di risorsa rimasta per alimentarlo;

b) *autonomia residua in stand-by*: rappresenta il tempo d'autonomia che il dispositivo dispone al minimo dei consumi di potenza;

c) *autonomia residua in attività*: rappresenta il tempo d'autonomia che il dispositivo dispone al massimo dei consumi di potenza.

Analogamente al caso della *network adaptation*, anche in questo caso è possibile risalire ad uno stato più o meno sintetico della risorsa "energia": ad esempio con la misura di a) avere una frazione 1/1 della potenza residua della batteria può sicuramente corrispondere ad uno stato "*full*" dello stato di alimentazione. Quindi, l'operazione di ***recupero*** della risorsa "energia" prevede che nell'implementazione della classe *Resource* siano stimate una delle tre grandezze a), b), o c) (la scelta del tipo di grandezza dipende dai meccanismi sottostanti offerti dal dispositivo). Nell'operazione di ***monitoraggio*** compiuta dal suo *Resource Monitor*, esso codifica lo stato di questi parametri in un'informazione più sintetica e comprensibile ai *Peer*. Ad esempio, disponendo di una misura della potenza residua del dispositivo e conoscendo la potenza nominale della batteria, è possibile indicare lo stato dell'energia con la frazione *(potenza residua*/*potenza nominale)*⨯*100*. L'indicazione diventa ancora più sintetica, se si approssima questa frazione con un numero finito di stati quali 25%, 50%, 75%, 100%.

Con questa soluzione, nella creazione degli *EventDescriptor* relativi alla risorsa "energia", i *Peer* dovranno specificare la transizione tra gli stati dell'alimentazione che deve far scattare la notifica e l'attivazione dell'*handler*. Se lo stato dell'alimentazione iniziale deve essere quello attuale, il *Peer* può ottenerlo tramite il metodo *getStatus()* offerto dal *Resource Monitor* assegnato alla risorsa.

Con questo tipo di *adaptation*, durante il processo di *delivery* le applicazioni possono intraprendere opportune strategie in funzione dell'autonomia rimasta al dispositivo.

4.3.4 *Location adaptation*

Il concetto di *location-aware computing* è preso in considerazione ogni qualvolta che il *delivery* di un servizio viene automaticamente adattato alla localizzazione spaziale di un dispositivo [41]. Immaginiamo un centro commerciale "intelligente": un servizio di pubblicità promozionale può inviare, una volta individuato l'utente abbonato, messaggi pubblicitari idonei alla posizione in cui si trova il suo dispositivo mobile. Fornire il supporto allo sviluppo di servizi *location-aware* è un problema vasto è complesso. Ad esempio dal punto di vista tecnologico vanno risolti problemi di eterogeneità delle tecniche di *positioning outdoor* ed *indoor* (*GPS*, *WI-FI, Bluetooth*…): bisogna stabilire come far interoperare le diverse soluzioni (un utente può lasciare un posto al chiuso servito con *WI-FI* e andare per strada dove è utilizzato il *GPS*) e capire i limiti dell'accuratezza nella localizzazione (ad esempio non è utile individuare un utente al chiuso se l'accuratezza è dell'ordine della decina di metri) [36].

Gli algoritmi e i meccanismi per il *positioning* dei dispositivi vanno incapsulati all'interno del *Resource Monitor* e nel *wrapper Resource*. Vista la suddivisione dell'architettura in domini Esperanto, come primo approccio è possibile prevedere una localizzazione di un dispositivo a livello di dominio.

L'adattamento alla localizzazione va però affrontato in maniera leggermente differente da quanto fatto precedentemente, dato che la risorsa da monitorare non è più una grandezza continua ma, essendo la posizione corrente del dispositivo (il dominio *host*), una grandezza discreta.

L'operazione di ***recupero*** della risorsa prevede quindi che nel metodo *getResource()* (membro della classe che rappresenta la risorsa "locazione") si implementi la comunicazione con uno dei componenti del *core* di rete, al fine di ottenere l'identificatore del dominio *host* in cui attualmente si trova il dispositivo. Nel ***monitoraggio***, il *Resource Monitor* può presentare questa informazione ai *Peer* in maniera più comprensibile, risalendo da essa alla localizzazione fisica del

dominio (ad esempio, ottenendo dal mediatore l'indirizzo dove è localizzata la macchina che lo ospita e il raggio con cui si estende il dominio da esso servito). Con questa soluzione, nella creazione degli *EventDescriptor* relativi alla risorsa "locazione", i *Peer* dovranno specificare, nello stato, un indirizzo di località fisica. Distinguiamo, pertanto, le seguenti modalità di compilazione dell'*Event Descriptor*:

si specificano sia lo stato iniziale, sia lo stato finale: in questo modo l'evento di cui vuole notifica l'applicazione è la migrazione del dispositivo da un dominio di partenza ad un dominio di arrivo;

si specificano solo lo stato iniziale: in questo modo l'evento di cui vuole notifica l'applicazione è l'abbandono del dominio specificato;

si specificano solo lo stato finale: in questo modo l'evento di cui si vuole notifica è l'arrivo in un dominio specificato dallo stato finale.

A differenza dei meccanismi di *adaptation* precedenti, la compilazione dell'*Event Descriptor* è più difficile in quanto bisogna specificare un indirizzo di località fisica di un dominio che talvolta non è conosciuto a priori. Per risolvere questo problema, nella stessa filosofia delle *Service Oriented Architecture*, il servizio di *location adaptation* può essere esteso con un servizio di *directory* dei domini per la conservazione di tutte le corrispondenze (*domainId, località fisica, attributi generali*). In questo modo, prima di compilare un *Event Descriptor*, un *Peer* può interrogare il servizio di *directory* dei domini per ottenere prima le informazioni richieste e poi effettuare la registrazione al *Resource Manager*.

4.4 Interoperabilità con le altre soluzioni di *discovery/delivery*

Oltre ad essere una soluzione nuova per il *delivery* dei servizi, Esperanto mira anche a garantire la compatibilità tra soluzioni di *discovery* e *delivery* eterogenee esistenti e future. Questo è un obiettivo molto importante in quanto non fa di

Esperanto una "*yet another architecture*" per soddisfare requisiti e vincoli dei sistemi di elaborazione emergenti. Permettendo inoltre di far interoperare in maniera trasparente le applicazioni sviluppate su altre infrastrutture (*software*), Esperanto rappresenta l'ambiente naturale dove far coesistere tutti i servizi e i clienti che gli scenari di *nomadic computing* possono offrire.

Per queste motivazioni, in Esperanto è fondamentale il concetto di dominio: un ambito contraddistinto da una particolare soluzione di *discovery/delivery* e connesso all'architettura tramite il *Domain-specific Agent*. E' naturale che un dominio Esperanto dovrà essere connesso all'architettura tramite un agente con delle responsabilità minori. Per consentire alle applicazioni del proprio dominio di offrire/richiedere servizi a/da tutti i domini dell'architettura, il *Domain-specific Agent* deve assumere le seguenti responsabilità:

- **esportare i servizi locali**: per ogni servizio pubblicato nel proprio dominio di competenza, l'agente deve esportarne i descrittori verso tutti i domini (distinti da Esperanto) connessi all'architettura. Per ogni servizio pubblicato, inoltre, esso deve creare un *proxy* (lato *server*) per l'utilizzo del servizio da parte di applicazioni remote. Questo *proxy* emula il comportamento di un cliente compatibile con il dominio guardando al servizio, emula un Esperanto *Peer* guardando al *Mediator*.
- **importare i servizi remoti**: per ogni servizio pubblicato nei domini remoti (compreso i domini Esperanto) l'agente deve importarne i descrittori. Per ogni servizio importato, inoltre, esso deve creare un *proxy* (lato *client*) per l'utilizzo del servizio remoto da parte di applicazioni locali. Questo *proxy* emula il comportamento di un *server* compatibile con il dominio guardando al cliente, emula un Esperanto *Peer* guardando al *Mediator*.

L'esportazione (importazione) dei servizi locali (remoti) implica l'interazione dell'agente con i componenti attivi dei protocolli di *discovery/delivery* che il dominio di competenza supporta (*registry*, *client*, *service*, ecc.). L'interoperabilità trasparente dei protocolli di *discovery* è garantita con il meccanismo di *import/export*, quella dei protocolli di *delivery* con l'uso dei *proxy* e del *Mediator*.

Tuttavia non è detto che tale processo vada sempre a buon fine. Infatti bisogna osservare che all'architettura possono connettersi domini di diversa natura. Questo significa che:

- ogni agente prima di esportare i servizi deve tradurre i loro descrittori in un formato comprensibile a tutti, i descrittori Esperanto. La fase di importazione richiede una conversione di formato inversa. Non è detto che entrambi le conversioni siano sempre possibili;
- per ogni coppia di domini eterogenei (e diversi da Esperanto) deve essere generata una coppia di *proxy* per interfacciare *client* e *server* al *Mediator*. Non è detto che entrambi i *proxy* possano essere generati.

In generale però è lecito affermare che il successo dell'operazione di:

- esportazione garantisce l'utilizzo dei servizi da parte di clienti Esperanto: basta la traduzione dei descrittori locali in descrittori Esperanto e la sola generazione del *proxy* lato *server* a garantire che un Esperanto *Peer* possa accedere al servizio pubblicato.
- importazione garantisce l'utilizzo di servizi Esperanto da parte dei clienti del dominio: basta la traduzione dei descrittori di servizi Esperanto in descrittori locali e la sola generazione del *proxy* lato *client* a garantire che un Esperanto *Peer* possa offrire il servizio pubblicato.

Diamo ora alcune linee guida sul processo di generazione dei *proxy*.

4.4.1 Generazione del *Server-side Proxy*

Un servizio è definito *usabile in Esperanto* se sono verificate le seguenti condizioni:

- ***il servizio è visibile nel dominio Esperanto***: il servizio è *visibile* quando è possibile tradurre il *SDR* locale nel *SDR* Esperanto.

- ***E' possibile astrarre dalla sua implementazione le funzionalità offerte***: in particolare devono essere ottenute le seguenti informazioni:
  - *Identificatore della funzionalità.*
  - *Parametri di scambio*: per ogni funzionalità vanno determinati parametri di ingresso e uscita.
  - *Iniziativa*: per ogni funzionalità del servizio va stabilita l'iniziativa (*one-way*, *request*/*response*, *solicit*/*response* e *notify*).
- ***Compatibilità dei tipi***: per ogni parametro di scambio, il tipo ad esso associato deve essere compatibile con i tipi definiti in Esperanto.

Con queste informazioni la compilazione del *SADR* è definita in tutte le sue parti:

- La sezione *tuples* è compilata a partire dai parametri di scambio (individuati per ogni funzionalità): è possibile ottenere una tupla per ogni n-pla di parametri di ingresso o di uscita. Il tempo di espirazione delle tuple va definito in funzione delle dinamiche del servizio.
- La sezione *functionalities* può essere compilata a partire dalle caratteristiche delle singole funzionalità: il nome della funzione, i parametri di scambio e l'iniziativa permettono di generare la sequenza di azioni necessaria alla corretta interazione con il servizio.

Ad esempio, se un *service* "*ServiceId*" offre solo una funzionalità *foo (in1: char, in2: byte, out: integer)* con iniziativa *call/back*, si otterrà il *SADR* Esperanto riportato in figura 4-21.

A partire dal *SADR* Esperanto, il *Server-side Proxy* è generato guardando la specifica della sezione *functionality.* Ad esempio, alle azioni specificate nella sezione *functionality* riportata in 4-21 sono associate le seguenti azioni del *proxy*:

1) Azione di *take(in_tuple: template, maximum: timeToWait)* sul mediatore locale
2) Invocazione di *foo (in1: char, in2: byte, out: int)* del *server* locale
3) Azione di *write(out_tuple: template, ZERO: timeToWait)* sul mediatore locale

In generale, in base al tipo di interazione, il comportamento *server-side proxy* è modellato in figura 4-22. Le parti in grigio indicano azioni che dipendono dalla tecnologia locale.

```
<sadr>
   <id> ServiceId </id>
   <tuples>
        <tuple>
           <name> in_tuple </name>
           <parameters>
               <parameter> <name tag=in1 /> <type tag=char /> </parameter>
               <parameter> <name tag=in2 /> <type tag=byte /> </parameter>
           </parameters>
           <timeToExpiration> maximum </timeToExpiration>
        </tuple>
        <tuple>
           <name> out_tuple </name>
           <parameters>
             <parameter><name tag=out /><type tag=integer /></parameter>
           </parameters>
           <timeToExpiration> timeout </timeToExpiration>
        </tuple>
   </tuples>
   <functionalities>
        <functionality>
            <name> foo </name>
            <description> </description>
            <action name="write" tuple="in_tuple"
                source="id" destination="ServiceId" timeout="0"/>
            <action name="take" tuple="out_tuple"
                source="ServiceIS" destination="id" timeout="maximum"/>
        </functionality>
    </functionalities>
</sadr>
```

Figura 4-21: esempio di SADR associato ad un servizio che offre una sola funzionalità *call/back.*

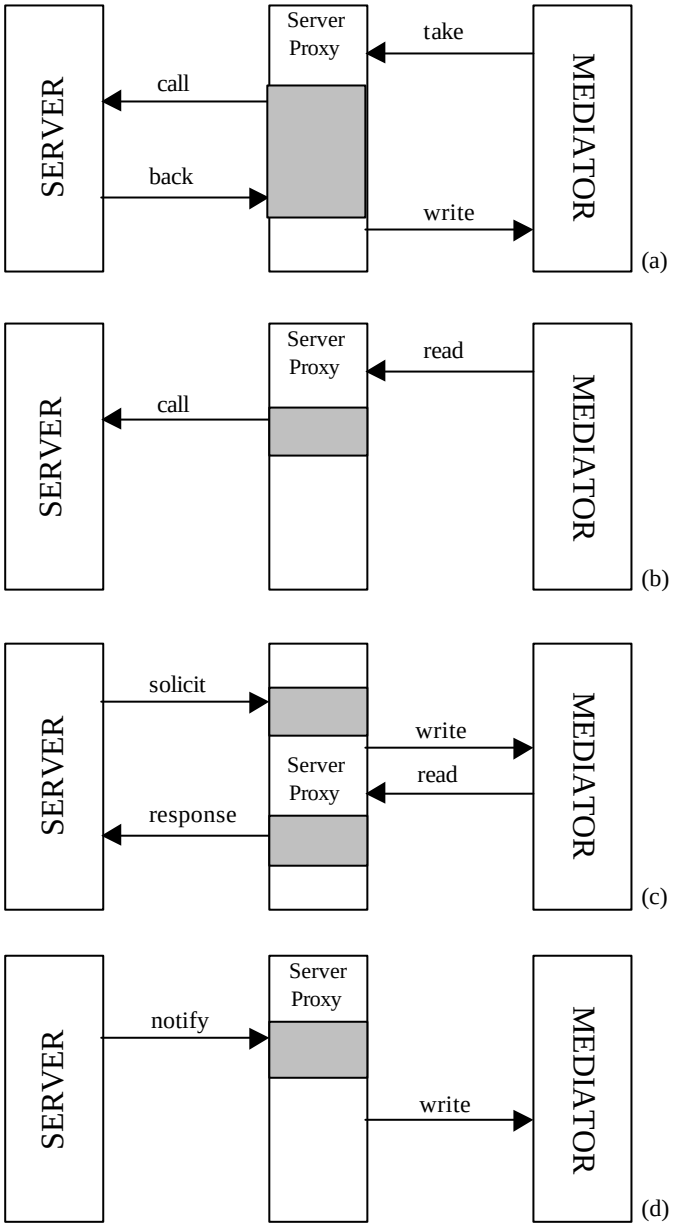

Figura 4-22: il comportamento del *server-side proxy* al variare delle possibili interazioni previste dalle funzionalità del servizio: (a) *request/response*; (b) *one-way*; (c) *solicit/response*; (d) *notification*.

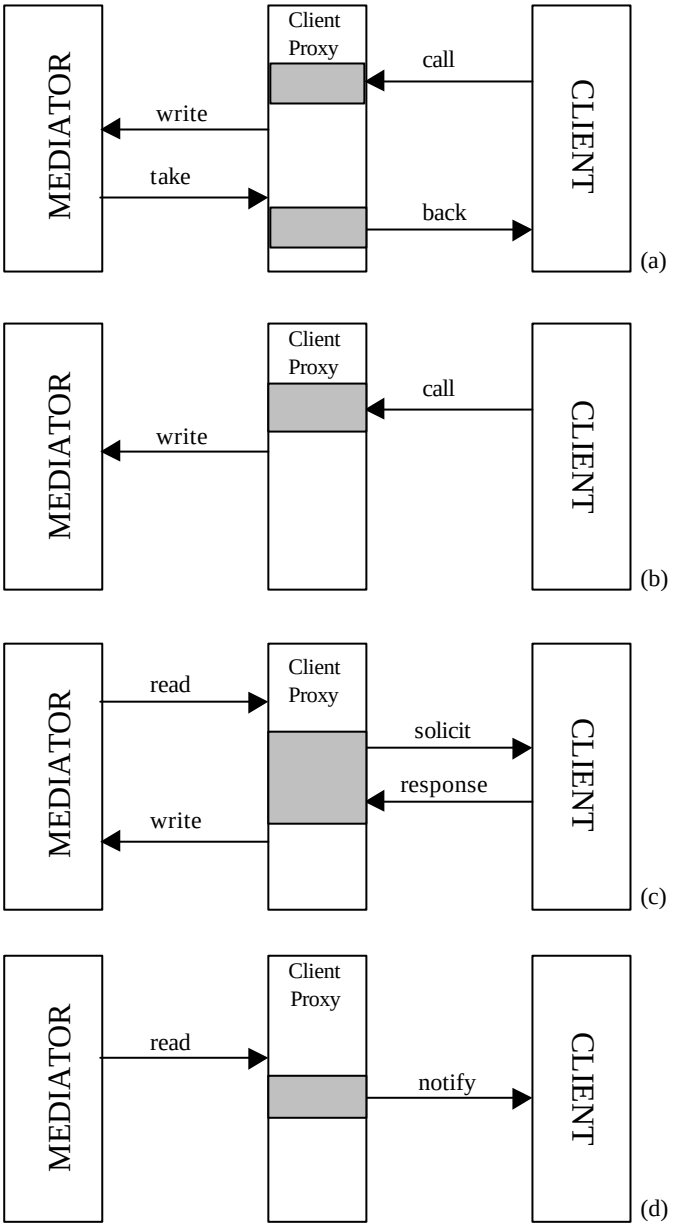

Figura 4-23: il comportamento del *client-side proxy* al variare delle possibili interazioni previste dalle funzionalità del servizio: (a) *request/response*; (b) *one-way*; (c) *solicit/response*; (d) *notification*.

4.4.2 Generazione del *Client-side Proxy*

Un servizio viene definito *usabile in un dominio X* se sono verificate le seguenti condizioni:

1) ***Il servizio è visibile nel dominio X***: il servizio è *visibile* quando è possibile tradurre il *SDR* Esperanto nel *SDR* espresso nella tecnologia locale.
2) ***Compatibilità dei meccanismi di interazione***: per ogni funzionalità espressa nel *SADR* Esperanto, il meccanismo di interazione definito (*call*/*back*, *notify*, *solicit*/*response*, *one-way*) deve essere supportato dalle tecnologie del dominio locale.
3) ***Compatibilità dei tipi***: tutti i tipi dei parametri di scambio delle funzionalità espresse nel *SADR* devono essere supportati dalla tecnologia locale.

La generazione del *SADR* può essere guidata dalla costruzione del *client-side proxy*: in altre parole, sempre che il concetto di *SADR* sia previsto, il descrittore all'accesso del servizio può essere generato con le tecniche offerte dal dominio in cui si vuole importare. Ad esempio, in *Jini* [19], il *client-side proxy* può corrispondere al *service object* (cfr. 2.2), e dunque la generazione del SADR non è prevista.

Il comportamento del *client-side proxy* può essere classificato in base ai diversi meccanismi di interazione così come riportato in figura 4-23. Le parti in grigio indicano azioni che dipendono dalla tecnologia locale.

4.4.3 *Domain-Specific Agent*

Il ruolo fondamentale del *Domain-specific Agent* è quello di importare/esportare i servizi nel/dal dominio di sua competenza. A supporto di queste operazioni, il modulo dell'agente atto alla creazione/distruzione dei *client-side proxy* e *server-side proxy* è il *Service Access Manager.* Come è riportato nello scenario di figura 4-24, una volta che la coppia di *proxy* è attiva, la comunicazione ha luogo non appena il *client* o il *service* del dominio locale interagisce con il singolo *proxy.*

SCENARIO: *in figura 4-24 è riportata l'interazione tra un client agent di un dominio generico e un service agent di in un altro dominio generico. L'interazione è iniziata dal cliente ed è di tipo call/return; altri scenari sono possibili*

1. Un cliente di un dominio interagisce con il *client-side proxy* fornendo i dati necessari per attivare il servizio. L'invocazione avviene con i meccanismi e i formati proprietari del dominio
2. Il *client-side proxy* fornisce i dati al *server-side proxy* attraverso il *Mediator*
3. Il *server-side proxy* è in attesa di tuple dal mediatore; appena disponibili invoca il servizio effettivo e scrive i risultati del servizio
4. Il *client-side proxy* preleva il risultato e lo fornisce al cliente che lo ha richiesto

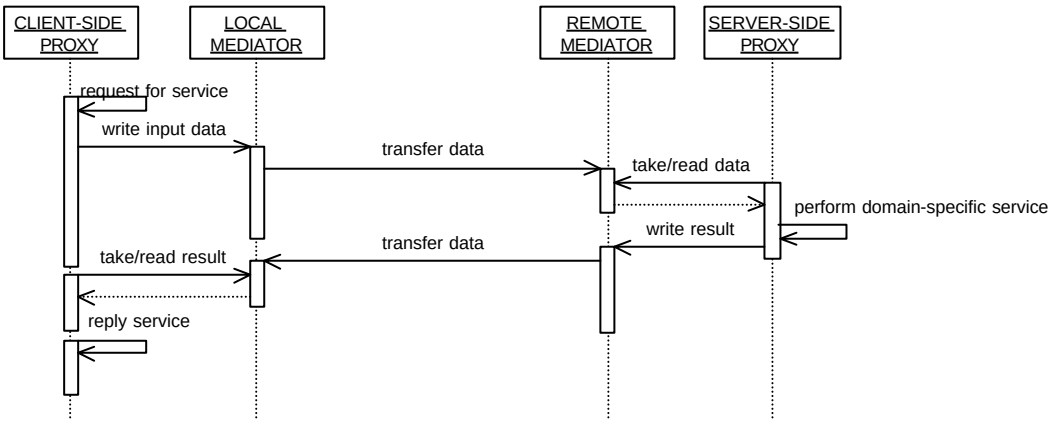

Figura 4-24: *interaction* tra *client agent* e *service agent* di domini generici attraverso l'uso dei *proxy.*

Bisogna osservare però che il modello di *service delivery* introdotto in Esperanto prevede una fase preliminare dove l'*Esperanto Peer* ottiene la descrizione dell'accesso al servizio. In particolare, quando il servizio è remoto (ovvero risiede in un dominio diverso da quello da cui proviene la richiesta) questa descrizione è fornita dall'*Esperanto Agent* su richiesta dell'*Esperanto Explorer*. Questo comportamento del protocollo va emulato dal *Domain-specific Agent*, che ad ogni

proxy lato *server*, dovrà associare un descrittore di accesso al servizio (*SADR*) e i meccanismi per soddisfare *SAD request* provenienti dagli agenti Esperanto remoti. In figura 4-25 si riporta un possibile scenario di interazione tra un *Peer* cliente ed un servizio associato ad un *proxy*. In figura 4-26 è invece riportata la situazione opposta.

SCENARIO: *nello scenario di figura 4-25 un Esperanto Peer effettua una SAD request di un servizio localizzato in un dominio generico. L'interazione è di tipo request/response*

1. L'*Esperanto Peer* invia una *SAD request* associato ad un servizio scoperto nella fase di *service discovery*. La richiesta è rivolta al suo *Esperanto Explorer* locale
2. La richiesta deve essere soddisfatta tramite l'*Esperanto Agent* in quanto il servizio è remoto
3. L'*Esperanto Agent* invia la richiesta al *Domain-specific Agent* (in particolare la richiesta è rivolta al *Service Access Manager*, il gestore dei *proxy*) dove è localizzato il servizio. Non è previsto supporto per le migrazioni dei dispositivi dei domini generici. Il dominio a cui si inoltra la richiesta è certamente quello individuato dall'identificatore del servizio.

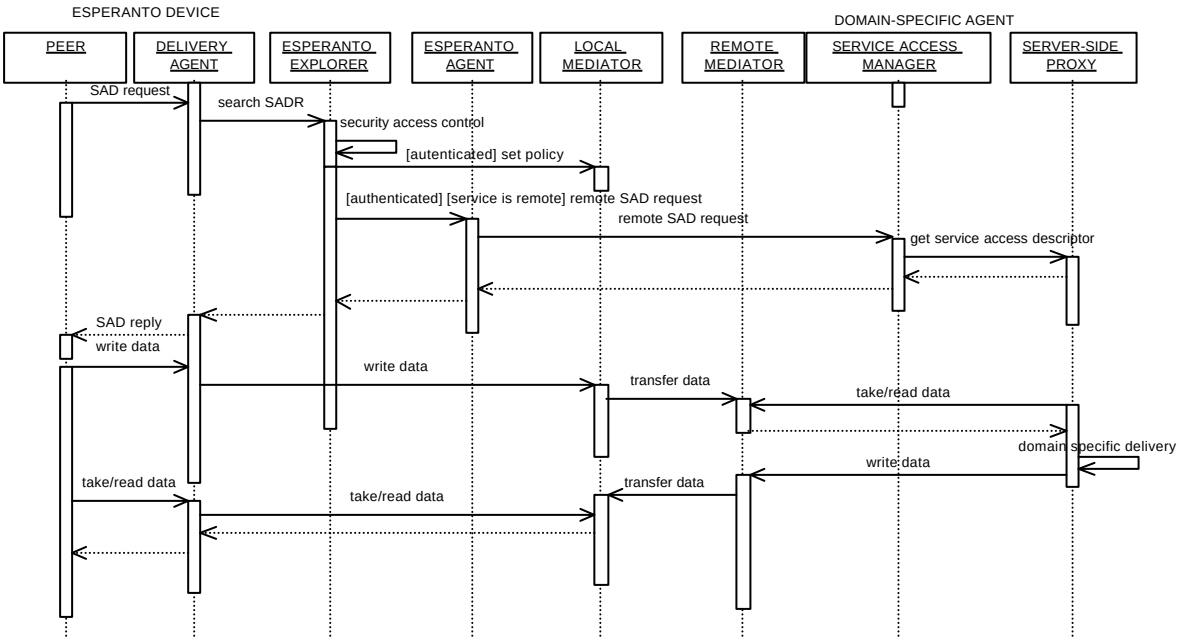

Figura 4-25: *service delivery* tra un Esperanto *Peer* ed un servizio localizzato in un dominio generico.

4. Il *Service Access Manager* ottiene il descrittore e lo ritorna all'agente richiedente; questo (tramite l'*Explorer*) provvede a consegnarlo al *Peer* che ne ha fatto richiesta
5. Ottenuto il SADR, il *Peer* accede al servizio. Il *Server-side Proxy* preleva i dati dal mediatore, invoca il servizio effettivo e scrive i risultati nello spazio assegnato. L'interazione ha luogo nel consueto modo previsto in Esperanto

SCENARIO: *nello scenario di figura 4-26, un client agent di un dominio generico richiede di interagire con un servizio offerto da un Esperanto Peer. In questo caso la fase di service access description manca perché il cliente si rivolge al proxy utilizzando i meccanismi proprietari al dominio.*

1. Un cliente del dominio generico invoca un servizio fornendo al *client-side proxy* i suoi dati di *input.* L'invocazione avviene con i meccanismi e i formati proprietari del dominio
2. Il *proxy* fornisce i dati al *Peer* che li preleva dal mediatore, effettua la computazione associata al servizio e scrive nello spazio assegnato i dati di *output*
3. Il *client-side proxy* preleva l'eventuale risultato dal mediatore e lo fornisce al cliente che lo ha richiesto

Il *class diagram* di figura 4-27 riassume tutte le responsabilità del *Service Access*

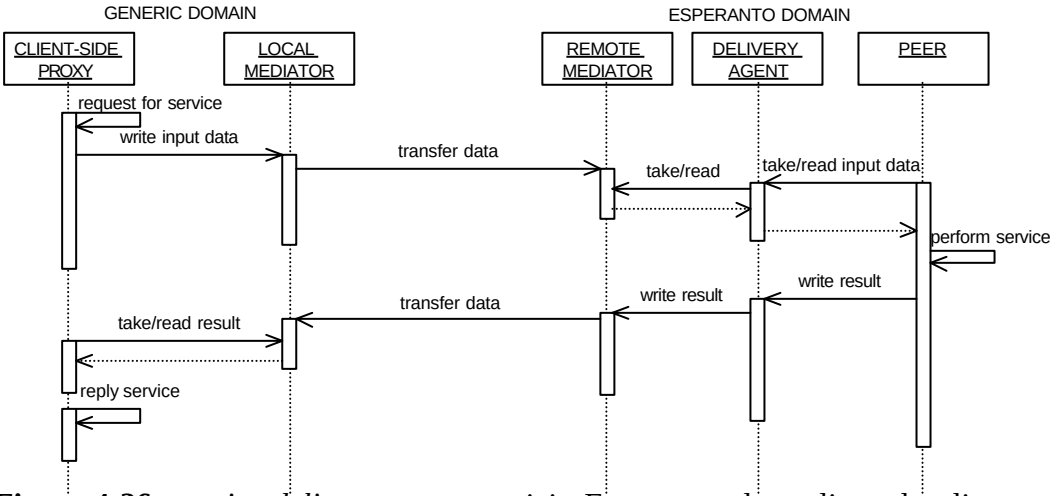

Figura 4-26: *service delivery* tra un servizio Esperanto ed un cliente localizzato in un dominio generico.

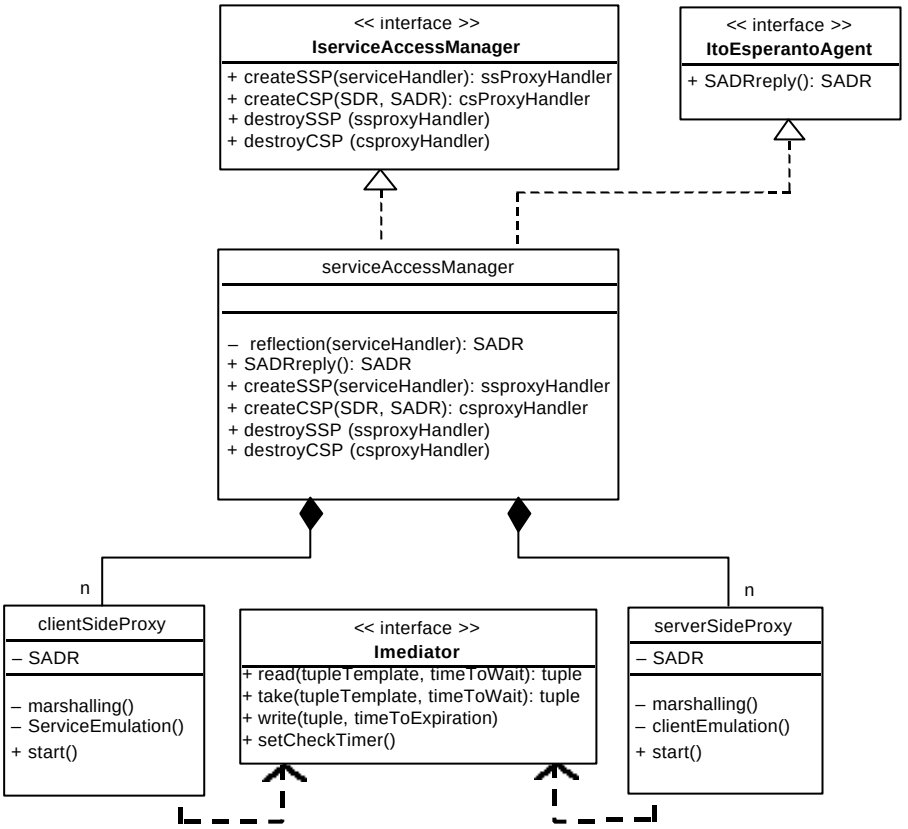

Figura 4-27: Il *Service Access Manager*

Manager. La generazione del *proxy* lato *server* può dipendere fortemente dall'architettura a cui l'agente si affaccia. Essa potrebbe essere guidata da un descrittore dell'accesso al servizio (come in *WSDL* [23]) oppure ottenuta con meccanismi di ispezione del codice (*reflection*). La strada per la generazione del *proxy* lato *client* è invece spianata dai descrittori Esperanto.

4.4.4 Compatibilità di un servizio

Dalle definizioni date ai par. 4.4.1 e 4.4.2, è evidente che la ricerca di un cliente di un dominio generico può restituire:

L'insieme dei servizi nativi del dominio: la lista dei servizi pubblicati all'interno del dominio

L'insieme dei servizi usabili nel dominio: la lista dei servizi visibili di cui è stato possibile generare il *client-side proxy*

L'insieme dei servizi non usabili nel dominio: la lista dei servizi visibili di cui non è stato possibile generare il *client-side proxy*

Osserviamo che la condizione di usabilità non è sufficiente a garantire il reale utilizzo di un servizio. La diversa natura dell'accoppiamento temporale tra *client* e *service* può causare il fallimento dell'interazione.

4.5 *API* offerte dall'infrastruttura di *delivery* Esperanto

L'infrastruttura di *delivery* Esperanto mette a disposizione degli sviluppatori di applicazione le primitive riportate in tabella 4-2.

Primitiva	Note	Offerta da
read (tupleTemplate, timeToWait): tuple	Lettura di una tupla dal mediatore locale	*Delivery Agent*
take (tupleTemplate, timeToWait): tuple	Prelevamento di una tupla dal mediatore locale	*Delivery Agent*
write (tuple, timeToExpiration)	Scrittura di una tupla nel mediatore locale	*Delivery Agent*
getSADR (peerID): SADR	Lettura di un descrittore di accesso ad un servizio	*Delivery Agent*
assignId (): peerId / disposeId(peerId)	Assegnazione/rimozione di un id all'entità (cliente o servente)	*Subscribe Agent*
bind (deviceId,domainId):domainId	Assegnazione di un dispositivo ad un dominio (dominio *home*)	***Subscribe Agent***
disconnect (deviceId, DomainId): status	Disconnessione volontaria di un dispositivo	***Subscribe Agent***
reconnect (deviceId, DomainId): status	Riconnessione di un dispositivo	***Subscribe Agent***
getStatus (): State	Lettura di uno stato di una risorsa monitorata dal sistema	*Resource Monitor*
registry (eventDescriptor,handler): notifyId	Notifica del cambiamento di una risorsa monitorata dal sistema	*Resource Manager*
removeNotification(notifyId)	Rimuove la notifica del cambiamento di una risorsa monitorata dal sistema	*Resource Manager*

Tabella 4-2: schema riassuntivo delle primitive offerte dall'infrastruttura di *delivery*. Le primitive associate al *Subscribe Agent* in grassetto sono concesse solo a *Peer* di sistema.

Naturalmente altre primitive sono messe a disposizione: ad esempio la gestione e la compilazione dei *Service Access Descriptor Record*, la gestione e la compilazione delle tuple. Osserviamo inoltre che queste primitive vanno affiancate a quelle offerte dall'infrastruttura di *discovery* Esperanto.

4.6 Approcci all'implementazione

Con gli scenari presentati in questo capitolo, si è cercato di descrivere i possibili casi d'uso dell'architettura e i meccanismi sottesi dal protocollo di *delivery*. Dei tanti aspetti che caratterizzano l'architettura, a nostro avviso, quelli che nell'ambito di questa tesi meritano di essere approfonditi nel progetto di basso livello sono il supporto alla mobilità dei dispositivi Esperanto (con la definizione del *Mediator* e del suo comportamento dinamico) e il supporto alla dinamicità di contesto (previsto con l'uso dei *pattern* di *adaptation* del par. 4.3).

Nel prossimo capitolo alcuni di questi elementi saranno studiati con maggiore dettaglio, saranno presentati alcuni elementi di basso livello e i dettagli realizzativi.

Capitolo 5

Approcci realizzativi all'infrastruttura di *delivery*

5.1 Introduzione

Nel capitolo 3 e nel capitolo 4 sono state presentate rispettivamente un'introduzione all'architettura Esperanto ed una progettazione di alto livello dell'infrastruttura di *delivery*.

In questo capitolo saranno presentati gli aspetti di basso livello del progetto dell'infrastruttura di *delivery* Esperanto. Il supporto alla mobilità garantito dall'introduzione del *Mediator* è ulteriormente dettagliato nei meccanismi di trasferimento delle tuple allo spazio da parte degli Esperanto *Peer*. Saranno inoltre affrontati gli aspetti realizzativi dei meccanismi di monitoraggio e notifica applicati rispettivamente dal *Resource Monitor* e dal *Resource Manager* per il supporto alla dinamicità del contesto. In entrambi i casi, l'approccio realizzativo è guidato dai requisiti di qualità come l'efficienza e le prestazioni garantite.

5.2 Studio del paradigma di comunicazione

In Esperanto, ogni *Peer* cliente invia le richieste di servizio (*service request*) al *Peer* servente indirettamente tramite il suo *Mediator* di dominio. Se i due *Peer* dovessero risiedere in domini distinti, il *Mediator* deve inoltrare la richiesta al pari dove il servente è attualmente localizzato. L'interazione avviene sempre secondo questo semplice meccanismo, sia per le *service request* che per le *service response*.

Per avere garanzie nel *service delivery*, i *Peer* possono specificare il tempo di persistenza delle tuple all'interno dello spazio (*boundary temporale*): scaduto il termine non ha più senso portare a compimento il *delivery* del servizio.

Per questi motivi è importante che nel progetto di basso livello del mediatore si adottino meccanismi volti a garantire efficienza e prestazioni, soprattutto quando la comunicazione coinvolge due entità residenti in domini distinti.

5.2.1 Scenario

La progettazione di basso livello dei mediatori permette di studiare come caso d'uso dell'architettura, la fase di *interaction* tra due Esperanto *Peer*.

Nel processo di *service delivery* la fase di *interaction* costituisce la componente variabile: infatti, una volta ottenuto il descrittore, è possibile che tra cliente e servizio vi sia un'interazione costituita da molteplici richieste e risposte di servizio. Pertanto l'implementazione del *Mediator* rappresenta uno dei punti cruciali della definizione dell'infrastruttura, in quanto dalle sue *performance* è possibile stabilire le prestazioni del modello di interazione complessivo.

Inoltre, lo studio dei suoi componenti e l'analisi dei tempi di trasferimento dati nelle *service request/service response* tra un cliente ed un servizio, può fornire importanti indicazioni anche alle eventuali direzioni da seguire per il miglioramento delle prestazioni e il dimensionamento di tutti gli altri componenti dell'architettura.

5.2.3 Mediatori e dispositivi Esperanto

Gli elementi dell'architettura coinvolti nella fase di interazione sono i dispositivi Esperanto, che ospitano rispettivamente *Peer* cliente e servente, e i mediatori assegnati ai domini in cui i dispositivi sono correntemente localizzati. Nella realizzazione di un caso di studio, si può immaginare che i dispositivi non siano in *roaming* e che la comunicazione avvenga tra i *Peer* solo tramite i loro diretti mediatori *home.*

Si osservi inoltre che non è necessario che tutti gli elementi del singolo componente siano sviluppati. Ad esempio nel dispositivo Esperanto la presenza

del *Subscribe Agent* è superflua, basta solo che il mediatore sappia che il dispositivo è presente all'interno del suo dominio di competenza.

A fronte di questa semplificazione, in figura 5-1 e 5-2 si presentano i diagrammi delle classi del mediatore e del dispositivo Esperanto cui si può fare effettivamente riferimento nello sviluppo del caso di studio.

Il caso di studio prevede naturalmente che i dispositivi Esperanto comunichino con il mediatore (per leggere o scrivere le tuple), e che i mediatori comunichino tra loro (collaborando per lo scambio delle tuple). Se nel primo caso la comunicazione è garantita attraverso il *forwarder* e *receiver*, nel secondo caso essa è garantita da un *broker*.

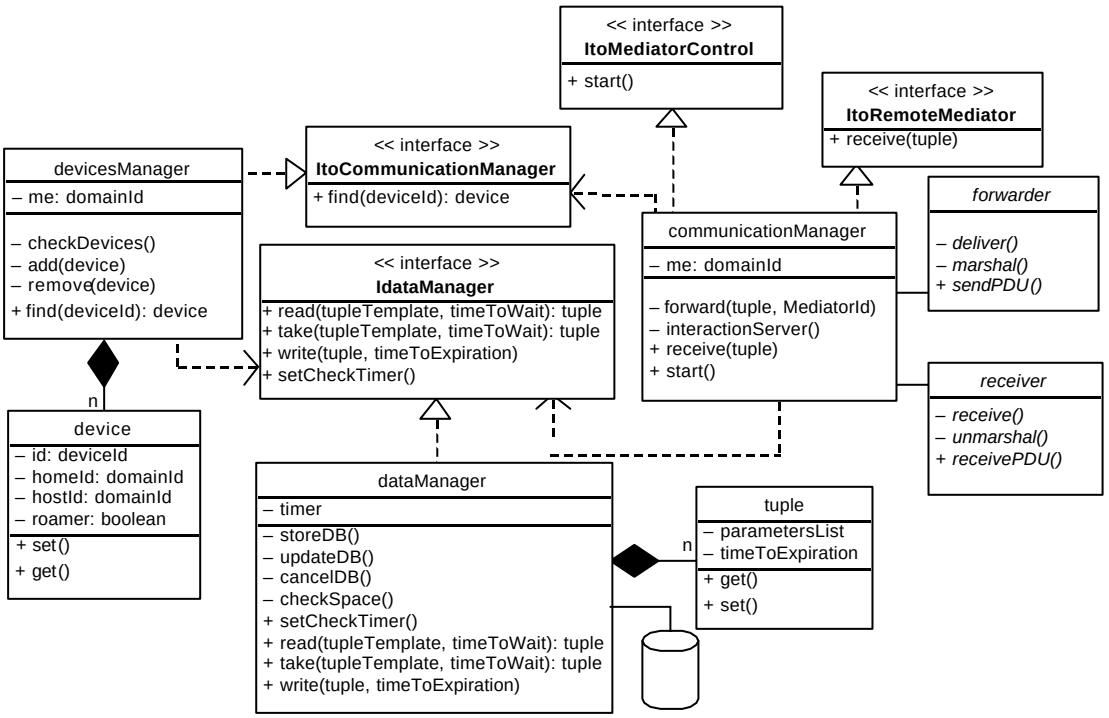

Figura 5-1: *Mediator*. Il diagramma delle classi di riferimento per il caso di studio.

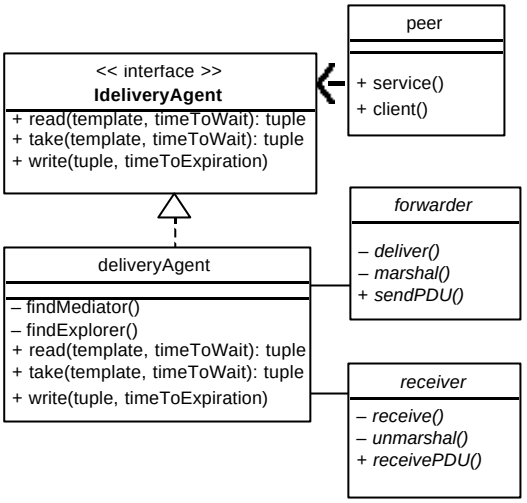

Figura 5-2: Struttura interna di un dispositivo Esperanto di riferimento per il caso di studio.

L'introduzione del *forwarder/receiver pattern* è dovuta all'esigenza di ridurre la complessità delle procedure di comunicazione necessarie a stabilire "un canale" tra dispositivo e *Mediator*, garantendo comunque l'indipendenza dalla piattaforma e dai protocolli di trasporto. Dato che lo standard *de facto* nei protocolli di trasporto/rete è TCP/IP, *forwarder* e *receiver* sono gli elementi del dispositivo Esperanto che fanno ricorso ai servizi di trasporto offerti da TCP per la gestione della comunicazione con il *Mediator* (ad esempio attraverso *socket* o librerie equivalenti).

La comunicazione tra i mediatori può essere realizzata rispettando il modello architetturale ad oggetti distribuiti (cfr. figura 3.4) e ricorrendo pertanto ad un ORB (JavaRMI, CORBA, …). Bisogna osservare, come già anticipato, che il mediatore ha delle responsabilità piuttosto critiche per le operazioni di *service delivery*. Infatti i descrittori del servizio (SADR) prevedono per le tuple un vincolo temporale espresso dal parametro "*TimeToExpiration*" e legato ai tempi di servizio. Esso indica la validità temporale di una tupla all'interno dello spazio condiviso. Pertanto, consegnare la tupla in tempi confrontabili a *TimeToExiration* ne impedisce in pratica l'utilizzo.

L'uso di un *middleware* che non dia garanzie di *performance* o di predicibilità al processo di trasferimento delle tuple (da un mediatore ad un altro) può quindi incidere negativamente sui tempi e sul successo dell'interazione tra cliente e

servizio. E' necessario ricorrere a soluzioni che permettono di applicare al comportamento dei mediatori il concetto di QoS (*Quality of Service*).

Al fine di fare chiarezza sulle scelte adottate nell'implementazione del mediatore, nei successivi paragrafi saranno presentate alcune soluzioni per lo sviluppo di *software* distribuito con garanzie di qualità del servizio.

5.2.4 *Real Time* CORBA

CORBA (*Common Object Request Broker Architecture* [45]) è un *middleware* ad oggetti distribuiti.

Come indicato in [46], le implementazioni convenzionali di CORBA non prevedono alcun meccanismo per la garanzia di QoS. Esse non sono utilizzabili nello sviluppo di applicazioni *real-time* per i seguenti motivi:

- *Mancanza di interfacce per la specifica di QoS*: nelle implementazioni convenzionali di CORBA (ad esempio quelle che aderiscono alla specifica 2.x [45]), non è data alcuna possibilità ad un *client*, ad esempio, di specificare all'ORB le priorità delle sue richieste.
- *Mancanza di applicazione della QoS*: gli ORB convenzionali non forniscono garanzie di QoS *end-to-end*, ci si affida al comportamento *best-effort* dei sistemi operativi o della rete.
- *Mancanza di ottimizzazione delle prestazioni*: ad esempio, le strategie di *demultiplexing* e *dispatching* degli ORB convenzionali non sono ottimizzate per applicazioni *real-time.*

Per questi motivi, le specifiche *Real-Time* CORBA (RT-CORBA [47]) definiscono un insieme di caratteristiche che gli ORB *real-time* devono possedere per assicurare QoS e predicibilità alle attività di *client* e *server*. Di seguito ne riportiamo alcune [48]:

- Gestione delle risorse e dell'infrastruttura di comunicazione: un ORB *real-time* deve gestire le politiche e i meccanismi per il supporto alla qualità del

servizio delle infrastrutture di comunicazione sottostanti (ad esempio un ORB deve poter scegliere che tipo di connessione adottare per una particolare invocazione di un metodo).

- Gestione dei meccanismi di *scheduling* dei sistemi operativi: un ORB *real-time* deve poter utilizzare i meccanismi di *scheduling* offerti dai sistemi operativi al fine di specificare le politiche di *scheduling* e le priorità stabilite dalle applicazioni.
- Interfaccia *real-time*: un ORB *real-time* deve fornire un'interfaccia standard per permettere alle applicazioni di specificare i loro requisiti di QoS. Le specifiche OMG permettono di configurare le risorse dell'ORB come le priorità dei *thread*, la dimensione dei *buffer* per le code dei messaggi, le connessioni a livello di trasporto, ecc.
- Servizi *real-time*: avere un ORB *real-time* risolve il problema della QoS solo in parte. Ad esempio un servizio CORBA di *scheduling* globale può essere usato per gestire e schedulare le risorse distribuite. In questo modo tale servizio può interagire con l'ORB per fornire meccanismi di supporto alla QoS specificati da applicazioni ad un elevato livello di astrazione e nella stesso modello di programmazione CORBA.

RT-CORBA standardizza interfaccia e *policy* QoS per permettere alle applicazioni di configurare e controllare le risorse quali processore (ad esempio, i meccanismi di priorità dei *thread*), rete (ad es. le proprietà delle connessioni) e memoria (ad es. la dimensione delle code delle richieste – le specifiche, però, trattano quest'ultimo aspetto in maniera piuttosto superficiale).

Queste configurazioni possono essere stabilite con tre modalità diverse [47]:

- Usando direttamente l'interfaccia esportata dall'ORB;
- Specificando le *policy* QoS (oltre le *policy* standard) quando si crea l'*Object Adapter* (ovvero il POA, *Portable Object Adapter*), solo nel caso delle applicazioni lato servente;

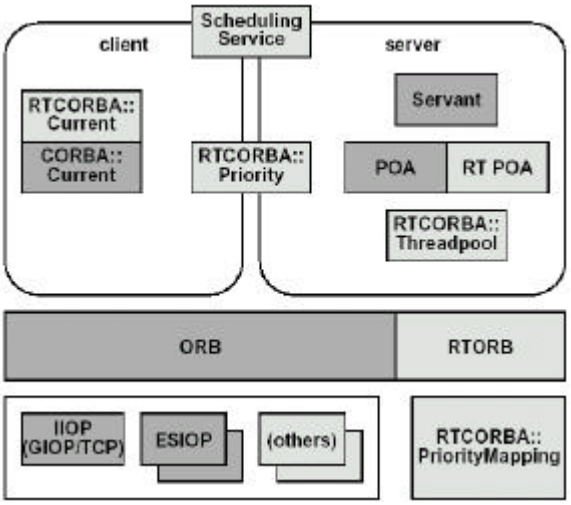

Figura 5-3: l'architettura RT-CORBA definita dall'OMG.

- Usando lo *Scheduling Service* definito dalle specifiche: questo servizio CORBA facilita l'applicazione delle politiche di *scheduling* delle risorse astraendo i meccanismi di basso livello messi a disposizione dall'ORB.

In figura 5-3 si riporta l'architettura di RT-CORBA. Come si vede, agli elementi già esistenti nel modello di riferimento classico si affiancano un *real-time* ORB per la nuova interfaccia orientata alla QoS dell'ORB, un modulo di *mapping* sui meccanismi di priorità sottostanti (*RTCORBA::PriorityMapping*) e i moduli per la gestione della QoS offerta alle applicazioni (*Scheduling Service*, *RTCORBA::Current*, *RTCORBA::Priority* e *RTCORBA::ThreadPool*).

5.2.5 TAO e RT-CORBA

TAO [46] e un'implementazione *open-source* dello standard RT-CORBA che consente di gestire la qualità del servizio in maniera efficiente, prevedibile e scalabile [7]. Visto che l'OMG definisce solo una specifica di RT-CORBA, lo studio di TAO consente di capire come le esigenze realizzative vanno incontro alle specifiche standard. Uno degli elementi di maggior interesse di TAO è il *Real-time Scheduling Service* [49].

5.2.5.1 RTSS – REAL TIME SCHEDULING SERVICE

Questo servizio CORBA permette alle applicazioni di specificare i loro requisiti di *scheduling* (CPU) in maniera molto intuitiva. L'oggetto CORBA, si farà poi carico di allocare le risorse del sistema, per andare incontro ai bisogni di QoS delle applicazioni che condividono l'infrastruttura ORB.

Il modello di specifica della QoS TAO prevede che un'applicazione registri attraverso una interfaccia dello *Scheduling Service* le operazioni che devono essere sottoposte a QoS (*RT_Operation*) e con quali attributi (*RT_Info*).

Se tutte le operazioni possono essere schedulate secondo i requisiti imposti, lo *Scheduling Service* assegna una priorità ad ogni operazione. A tempo di esecuzione il *Run-time Scheduler* di TAO (un componente dell'ORB [46]) effettuerà, per ogni operazione del *servant* richiesta dai *client* (preventivamente schedulata), la traduzione delle priorità assegnate (dal RTSS) nelle priorità comprensibili dal *dispatcher* del sistema operativo sottostante. In figura 5-4 si riportano gli elementi più interessanti dell'interfaccia dello *Scheduling Service* TAO per la specifica della QoS. Il descrittore dei parametri di QoS è la *struct RT_Info,* descriviamo alcuni suoi attributi:

- ***Worst-case execution time***: è il massimo tempo di esecuzione che una *RT_Operation* richiede;
- ***Typical execution time***: è il tempo di esecuzione che una *RT_Operation* tipicamente richiede;
- ***Importance***: il RTSS usa questo parametro per stabilire l'ordine di esecuzione delle *RT_Operation* a parità di altri fattori discriminanti;
- ***Criticality***: il RTSS in alcune strategie di *scheduling* considera questo parametro come primario nelle assegnazioni delle priorità alle *RT_Operation*.

Attraverso l'interfaccia *Scheduler*, un'applicazione può chiedere QoS per le sue operazioni (*create()* per la creazione di un *RT_Info* e *set()* per la compilazione) e invocare lo *scheduling off-line* con *compute_scheduling().*

Il RTSS è un servizio per lo *sheduling* QoS della risorsa CPU. Nello schema IDL non è dunque riportato alcun elemento per stabilire QoS per le risorse di rete (ad esempio specificando i privilegi di una connessione). Per queste esigenze, la QoS va specificata interagendo direttamente con l'ORB.

```
module RT_Scheduler {
  interface RT_Operation {
    typedef TimeBase::TimeT Time;           // 100 nanoseconds
    typedef Time Quantum;
    typedef long Period;                    // 100 nanoseconds

// Defines the importance of the operation, which can be used by
// the Scheduler as a "tie-breaker".
    enum Importance {
        VERY_LOW_IMPORTANCE, LOW_IMPORTANCE, MEDIUM_IMPORTANCE,
        HIGH_IMPORTANCE, VERY_HIGH_IMPORTANCE
    };
// Certain sheduling strategies consider criticality as the primary
// distinction between operations when assigning priority
    enum Criticality {
        VERY_LOW_CRITICALITY, LOW_ CRITICALITY, MEDIUM_ CRITICALITY,
        HIGH_ CRITICALITY, VERY_HIGH_ CRITICALITY
    };
        ...

// Describes the QoS for an "RT_Operation". The CPU requirements and QoS
// for each "entity" implementing an application operation is described
// by the following information.
    struct RT_Info {
// string that uniquely identifies the operation.
      string entry_point_;
      Time worstcase_execution_time_;
      Time typical_execution_time_;
      Time cached_execution_time_;
      Period period_;
      Importance importance_;
      Criticality criticality_;
// For time-slicing (for BACKGROUND operations only).
      Quantum quantum_;
// The number of internal threads contained by the operation.
      long threads_;
       ...
    };
  };
// This class holds all the RT_Info's  for a single application.
  interface Scheduler {
       ...
// Creates a new RT_Info entry for the function identifier "entry_point"
    handle_t create (in string entry_point) raises (DUPLICATE_NAME);
       ...
// Set the attributes of an RT_Info.
    void set (in handle_t handle, in Time time, in Time typical_time,
          in Time cached_time, ...) raises (UNKNOWN_TASK);
       ...
// Computes the scheduling priorities
    void compute_scheduling (...) raises (UTILIZATION_BOUND_EXCEEDED);
       ...
  };
};
```

Figura 5-4: una vista dell'interfaccia IDL del RTSS.

5.2.5.2 GESTIONE DELLA COMUNICAZIONE INTER-ORB

Per permettere alle applicazioni di controllare i protocolli di comunicazione sottostanti, le specifiche di RT-CORBA, e conseguentemente TAO, definiscono un'interfaccia standard che può essere usata per i seguenti scopi:

- *selezionare e configurare le proprietà di un opportuno protocollo di rete*: l'interfaccia esportata dall'ORB permette alle applicazioni di specificare le proprietà dei protocolli inter-ORB e di trasporto sia dal lato *client*, sia dal lato *server*;
- *configurare le proprietà del protocollo lato server*: con la interfaccia *RTCORBA::ServerProtocolPolicy*, il *server* può selezionare quale protocollo configurare nella creazione del POA e gli attributi della connessioni verso i suoi clienti;
- *configurare proprietà del protocollo lato client*: quando un cliente ottiene un *binding* ad un oggetto, può configurare le proprietà della connessione con la interfaccia *RTCORBA::ClientProtocolPolicy*. Ad esempio, nel caso di *Internet* e il protocollo TCP, tramite *RTCORBA::TCPProtocolProperties* (figura 5-5), un cliente può fissare gli attributi della connessione TCP *end-to-end*: con *keep_alive* il modulo TCP è configurato in modo che invii un segnale di *probe* su una connessione inattiva per verificarne la validità, con *no_delay* viene disabilitato l'algoritmo di Nagle per evitare il *buffering* dei segmenti TCP ecc.;

```
module RTCORBA {
  interface TCPProtocolProperties : ProtocolProperties {
    attribute long send_buffer_size;
    attribute long recv_buffer_size;
    attribute boolean keep_alive;
    attribute boolean dont_route;
    attribute boolean no_delay;
  };
...
};
```

Figura 5-5: proprietà per la specifica di QoS in TCP.

- esplicitare il *binding* all'oggetto lato servente descrivendo le proprietà della connessione: nelle specifiche 2.x, CORBA supporta solo il *binding* implicito. In questo modello, le risorse per l'attivazione delle operazioni tra *client* e *server* sono stabilite all'atto della prima invocazione di un metodo del *servant*. Il *binding* implicito è inadeguato per le applicazioni a prestazioni garantite con requisiti di QoS, perché aumenta latenza e *jitter* [48]. Per ovviare a questi problemi, il *binding* all'oggetto viene reso esplicito. Con il *binding* esplicito il *client* può stabilire preventivamente una connessione (tramite l'operazione *CORBA::Object::validate_connection*) e può controllare come le sue richieste sono servite dall'ORB (attraverso *RTCORBA::ClientProtocolPolicy*). Ad esempio un *client* potrebbe richiedere all'ORB che la connessione al *server* avvenga preliminarmente alle sue richieste e che esse siano inoltrate su canali senza *multiplexing*.

Bisogna sottolineare che i benefici ottenuti da quest'approccio alla QoS impediscono nella pratica la trasparenza ai dettagli della comunicazione sottostante. Quest'aspetto è comunque in linea con l'approccio *application-awareness*.

5.2.6 Mediatore

Da quanto appreso nei paragrafi 5.2.4 e 5.2.5 e dalle considerazioni del par. 5.2.3, nell'implementazione dei mediatori si può fare affidamento sui meccanismi di QoS garantiti da TAO.

Prima però di poter realizzare i singoli componenti interni, bisogna stabilire i requisiti di QoS richiesti e successivamente tradurli nel modello di qualità del servizio offerto da TAO.

5.2.6.1 REQUISITI DI QOS NELLO SVILUPPO DEI MEDIATORI

Nel modello di qualità del servizio proposto da RT-CORBA e TAO è prevista la possibilità di specificare QoS in termini di risorse di processo, di risorse di rete e

risorse di memoria. Al fine di stabilire quali requisiti di QoS sono necessari per l'efficienza e la predicibilità del comportamento del *Mediator* sono stati compiuti i seguenti passi:

1) **individuazione di una gerarchia di *task* critici**: tra i componenti presenti all'interno del mediatore possiamo distinguere ad esempio il *Communication Manager* e il *Devices Manager*. Il primo è responsabile della gestione delle tuple, il secondo della gestione dei dispositivi. Di conseguenza, le operazioni del CM sono più critiche di quelle del DM;

2) **individuazione di una gerarchia delle politiche di utilizzo dei protocolli di trasporto**: ad esempio, quando un *Communication Manager* deve inoltrare le tuple verso un nuovo dominio, è importante che questo avvenga su una connessione già stabilita. Inoltre è utile che la operazione di *forwarding* delle tuple venga discriminato in base al *TimeToExpiration*: a tuple con vincoli temporali più stringenti bisogna assegnare risorse maggiori (ad esempio canali non multiplati);

3) **definizione del livello di distribuzione del mediatore**: le specifiche di CORBA hanno da sempre avuto il pregio di fornire meccanismi di trasparenza alla locazione, al sistema operativo, al linguaggio ecc. Con RT-CORBA è introdotto un ulteriore meccanismo di trasparenza: quello al controllo della priorità (associate ai processi e/o ai *thread*). Utilizzando i meccanismi di priorità offerti da CORBA *real-time* si conferisce maggiore portabilità alle applicazioni, che non devono perciò fare riferimento a quelli offerti dal sistema operativo.

   Prevedendo di mettere in comunicazione attraverso l'ORB *real-time* tutti i componenti di uno stesso mediatore (*Communication Manager*, *Devices Manager*, *Data Manager*, ecc.), anziché solo quelli preposti alla comunicazione con i mediatori remoti (*Devices Manager* e *Communication Manager*), permette benefici sia grazie alle garanzie di QoS sia grazie alla trasparenza ai meccanismi di priorità dei sistemi operativi.

Sulla base delle considerazioni fatte ai punti 1 e 3 è possibile stilare la gerarchia di priorità delle operazioni critiche, come riportato in tabella 5-1.

RT_OPERATION NAME	CRITICALITY	IMPORTANCE
start()	MEDIUM_CRITICALITY	VERY_HIGH_IMPORTANCE
receivePDU()/sendPDU()	HIGH_CRITICALITY	VERY_HIGH_IMPORTANCE
marshal()/unmarshal()	VERY_HIGH_CRITICALITY	VERY_HIGH_IMPORTANCE
interactionServer()	HIGH_CRITICALITY	VERY_HIGH_IMPORTANCE
forward()/receive()	VERY_HIGH_CRITICALITY	VERY_HIGH_IMPORTANCE
find()	HIGH_CRITICALITY	HIGH_IMPORTANCE
read()/take()	HIGH_CRITICALITY	HIGH_IMPORTANCE
add()	MEDIUM_CRITICALITY	HIGH_IMPORTANCE
write()	LOW_CRITICALITY	MEDIUM_IMPORTANCE
checkDevices()/checkSpace()	VERY_LOW_CRITICALITY	MEDIUM_IMPORTANCE
setCheckTimer()	VERY_LOW_CRITICALITY	VERY_LOW_IMPORTANCE
remove()	VERY_LOW_CRITICALITY	LOW_IMPORTANCE
storeDB()	VERY_LOW_CRITICALITY	VERY_LOW_IMPORTANCE
updateDB()	VERY_LOW_CRITICALITY	VERY_LOW_IMPORTANCE
cancelDB()	VERY_LOW_CRITICALITY	VERY_LOW_IMPORTANCE

Tabella 5-1: i criteri di *scheduling* delle operazioni dei componenti interni al *Mediator*.

Tuttavia con la tabella, sono stati fissati solo i requisiti di QoS per la risorsa CPU. In base alle considerazioni del punto 2 è necessario stabilire le politiche per la gestione del trasferimento delle tuple da un *Mediator* all'altro.

Il *forwarding* delle tuple deve essere tanto più efficiente e rapido quanto minore è il tempo di espirazione delle stesse. Quindi si può pensare si suddividere il livello di criticità della loro spedizione in tre fasce a priorità e caratteristiche di qualità decrescenti: le tuple con un *TimeToExpiration* molto basso sono assegnate ad una priorità alta e quindi inoltrate al mediatore di destinazione attraverso una connessione di alta qualità, le tuple con un *TimeToExpiration* molto alto sono invece assegnate ad una priorità bassa e quindi inoltrate al mediatore di destinazione attraverso una connessione di bassa qualità. Possiamo formalizzare i requisiti di QoS per le risorse di rete come segue:

- Le tuple in cui *TimeToExpiration* è confrontabile con i tempi di trasferimento della tupla stessa, sono trasferite attraverso una connessione di qualità massima. Una tupla va inoltrata su una connessione di questo tipo quando sussiste la seguente relazione:

$$TimeToExpiration \leq x\,\% \times TimeToDelivery$$

Dove *TimeToDelivery* e $x>1$ sono determinati opportunamente.

- Le tuple in cui *TimeToExpiration* è circa confrontabile con i tempi di trasferimento della tupla stessa, sono trasferite attraverso una connessione di qualità media. Una tupla va inoltrata su una connessione di questo tipo quando sussiste la seguente relazione:

$$x\,\% \times TimeToDelivery \leq TimeToExpiration \leq y\,\% \times TimeToDelivery$$

Con $1 < x < y$, interi e *TimeToDelivery* calcolato opportunamente.

- Le tuple in cui *TimeToExpiration* è sufficientemente maggiore dei tempi di trasferimento della tupla stessa, sono trasferite attraverso una connessione di bassa qualità. Una tupla va inoltrata su una connessione di questo tipo quando sussiste la seguente relazione

$$TimeToExpiration \geq y\,\% \times TimeToDelivery$$

I parametri *x* ed *y* possono essere stabiliti a priori oppure in maniera adattativa, imponendo che le tre connessioni siano equamente sfruttate. Le proprietà delle singole connessioni dipendono dai meccanismi di trasporto sottostante; ad esempio, nel caso di *Internet* e TCP, è possibile prevedere per le connessioni i parametri di qualità riportati in tabella 5-2 e già descritti in TAO (cfr. 5.2.5.2).

	Massima Priorità	Media Priorità	Bassa priorità	Note
send_buffer_size	**Max**	**Med**	**Min**	Buffer di invio assegnato alla socket
recv_buffer_size	**Max**	**Med**	**Min**	Buffer di ricezione assegnato alla socket
keep_alive	´	´	´	Controlla se la connessione è attiva
dont_route	´			Fissa il percorso verso l'end-point
no_delay	´	´	´	Non c'è bufferizzazione
privateConnection	´	´		Connessione non multiplata
explicit binding	´			La connessione avviene esplicitamente

Tabella 5-2: i parametri di QoS assegnati alle connessioni tra due *Mediator*.

5.2.6.2 APPLICAZIONE DEI REQUISITI DI QOS AL MODELLO OFFERTO DA TAO

Nel modello di QoS previsto da TAO, la specifica di qualità del servizio nell'uso delle risorse computazionali va espressa tramite l'uso del RTSS. La

determinazione di uno *scheduling* ottimale deve seguire i passi della procedura riportata in basso:

1) Individuare le operazioni da sottoporre a QoS: individuare ovvero le *RT_Operation*;
2) Per ogni operazione individuata, caratterizzare i parametri di qualità del servizio: costruire ovvero gli *RT_Info* per ogni *RT_Operation*;
3) Registrare nel *Real-Time Scheduling Service* le operazioni da sottoporre a QoS;
4) Attivare l'elaborazione dello *scheduling off-line* per la costruzione delle strutture dati utilizzate nella schedulazione a tempo di esecuzione da parte del *run-time Scheduler* dell'ORB.

In tabella 5-1 sono individuate le *RT_Operation* dei componenti del mediatore e per ognuna di esse i parametri per la compilazione della struttura *RT_Info.*

La specifica di qualità del servizio nell'uso delle risorse di rete va, invece, espressa tramite l'ORB da entrambi i lati dell'elaborazione (cliente o servente) o tramite la interfaccia POA (solo dal lato servente). Data la criticità dell'operazione di inoltro delle tuple, è più adatto prevedere che le *policy* di utilizzo delle risorse di comunicazione siano assegnate dal lato servente all'atto della creazione del POA. Dal lato cliente, invece, è opportuno che si specifichi il *binding* esplicito (come previsto in tabella 5-2) all'atto dell'inizializzazione del sistema. Per questo, la configurazione dei requisiti di QoS avviene con la seguente procedura:

1) configurazione delle politiche (standard e di QoS) del POA in funzione dei serventi definiti;
2) configurazione dei *binding* all'inizializzazione del sistema.

In figura 5-6 si riporta una versione semplificata dell'interfaccia IDL del modulo Mediatore. Per ogni interfaccia presente all'interno del modulo va implementato un *servant*. Le interfacce *receiveMaxPriority, receiveMedPriority, receiveLowPriority* esportano i metodi per il trasferimento delle tuple da un

mediatore ad un altro. I *servant* che realizzeranno queste interfacce saranno correlati ai POA con le politiche definite anche sulla base della tabella 5-2.

L'implementazione completa del *Mediator* prevederà naturalmente tutti i passi previsti nel ciclo di sviluppo di un'applicazione CORBA [7].

```
module Mediator {
  typedef ... TupleType; // a sequence describing a tuple
  typedef ... Pdu; // Protocol Data Unit exchanged with an Esperanto device
  typedef TimeBase::TimeT TimetoWait;
  typedef TimeBase::TimeT TimetoExpiration;
       ...
  interface ReceiveMaxPriority {
    void receiveMax (in TupleType tuple);
  };
  interface ReceiveMedPriority {
    void receiveMed (in TupleType tuple);
  };
  interface ReceiveLowPriority {
    void receiveLow (in TupleType tuple);
  };
  interface CommunicationManager: ReceiveMaxPriority,
    ReceiveMedPriority, ReceiveLowPriority {
    void start(void);
  };
  interface DevicesManager {
    typedef ... Device; // a struct describing device property
    Device find(in Device.deviceId id);
  };
  interface DataManager {
    TupleType read(in TupleType template, in TimeToWait timeOut);
    TupleType take(in TupleType template, in TimeToWait timeOut);
    void write(in TupleType tuple, in TimeToExpiration timeToLeave);
    void setCheckTimer(in TimeBase::TimeT timer);
  };
  interface Forwarder {
    boolean sendPDU(in Pdu dataUnit);
  };
  interface Receiver {
    boolean receivePDU(out Pdu dataUnit);
  };
       ...
};
```

Figura 5-6: interfaccia IDL del *Mediator*.

5.3 Studio del supporto alla dinamicità

Il supporto alla dinamicità è espresso sinteticamente con il termine di *adaptation*. In particolare, per gli ambienti altamente dinamici come quelli di *mobile computing*, la *application-aware adaptation* è l'unico modo per consentire un'interazione efficace [37]. Bisogna osservare, però, che qualsiasi approccio

application-aware porta con sé una complessità elaborativa che va sostenuta dalle risorse *hardware* e *software*. Allo scopo ridurre questo carico aggiuntivo è importante che, anche nell'implementazione dei *pattern* adottati per il supporto alla dinamicità, si tenga conto dell'efficienza e delle *performance* sia in termini di *footprint* (la quantità di memoria occupata) che di velocità di esecuzione.

Prima di presentare gli aspetti del progetto di basso livello del *pattern* riportato al paragrafo 4.3.1, illustriamo ACE [50] (*Adaptive Communication Environment*), un *framework* che mette a disposizione una notevole quantità di componenti e *pattern*, realizzati in C++, per lo sviluppo di *software* per sistemi distribuiti a prestazioni garantite.

5.3.1 ACE

ACE [51] include molti *pattern* che semplificano lo sviluppo di software distribuito, conferendogli flessibilità, efficienza, affidabilità e portabilità. Il *framework* offre, tra le numerose caratteristiche, componenti per la gestione di:

- *Inter-process communication* (IPC);
- *event demultiplexing* e *handler dispatching*;
- *connection establishment* e *service inizialization*.

Come è evidente dalla figura 5-7, l'architettura di ACE è organizzata in livelli:

- *OS adaptation layer*: il livello di adattamento al sistema operativo è uno sottile strato di codice C++ interposto tra il livello nativo delle API di sistema operativo e il resto dell'architettura ACE. Questo livello ha lo scopo di nascondere al *framework* ACE tutti i dettagli e le dipendenze dai meccanismi sottostanti offerti dai sistemi operativi. In questo modo sviluppare codice con i componenti ACE garantisce facile portabilità da una piattaforma all'altra.
- *C++ Wrapper Façade layer*: questo livello comprende tutte le classi C++ che possono essere usate per costruire applicazioni in C++ in maniera facile, veloce e affidabile. Queste classi offrono funzionalità come:

- Gestione della concorrenza e della sincronizzazione: in questo modo si astraggono le API per il *multi-threading* e il *multiprocessing* (semafori, *lock*, ecc.);
- Gestione dei meccanismi di IPC: in questo modo si astraggono le API per l'*Inter-Process Communication* (*socket*, *pipe*, *Unix FIFO*, ecc.);
- Gestione dei segnali: in questo modo si astraggono le API per la gestione dei segnali.

- *ACE framework component*: questo livello è quello al maggiore livello di astrazione; in esso sono disponibili *building block* alcuni dei quali progettati per specifici domini applicativi. Componenti molto utilizzati di questo livello sono:
  - *Reactor*: il *Reactor* è un componente per la gestione dell'*event-demultiplexing* e *dispatching*, il suo uso è quindi prevalentemente destinato ad applicazioni orientate agli eventi. In quest'approccio, il codice è decomposto in tre sezioni logiche: la gestione dell'occorrenza degli eventi, il loro riconoscimento/attivazione degli *handler* atti a gestirli, e le

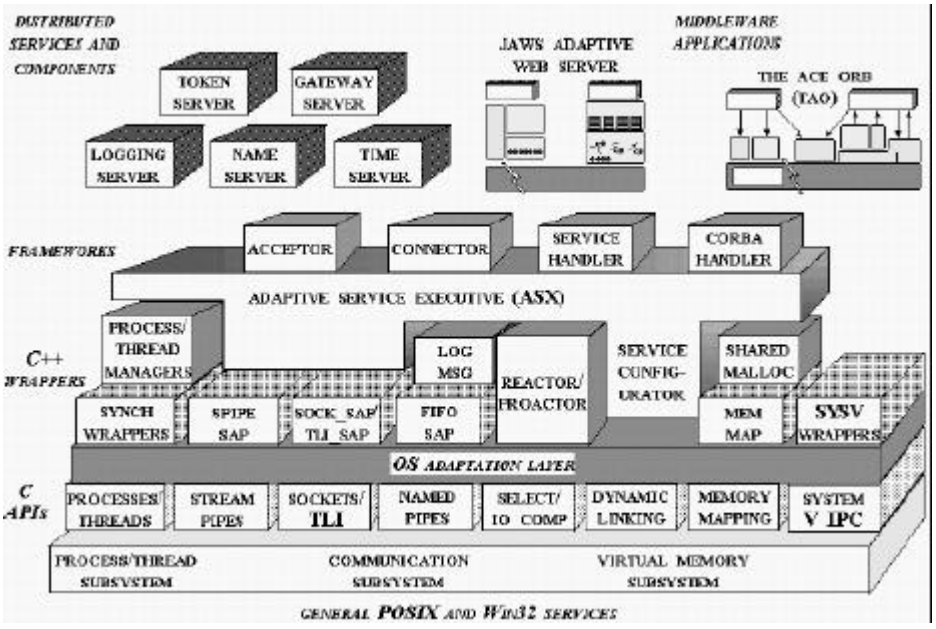

Figura 5-7: l'architettura di ACE.

procedure di gestione dell'occorrenza del singolo evento. Quando un evento occorre, esso viene riconosciuto e inviato al giusto *handler* che si occupa del suo trattamento. Questo comportamento è ad esempio alla base di tutte le interfacce grafiche offerte dai sistemi operativi. Il *Reactor* è un *pattern* molto diffuso in letteratura e nell'ambito del *design Object-Oriented*. Vista la somiglianza con il *Resource Manager* (cfr. 4.3.1) esso sarà approfondito al paragrafo seguente.

- *Connector* e *Acceptor*: sono due componenti utili quando si vuole sviluppare *software* di rete. Il loro uso aiuta a disaccoppiare la fase di inizializzazione di una connessione da quella di scambio dati tra le applicazioni dopo che essa è stabilita.

5.3.1.1 ACE *REACTOR*

Attraverso l'utilizzo del *Reactor* [52] è possibile gestire attraverso un'interfaccia semplice ed omogenea un numeroso insieme di eventi come i *timer-based, signal-based, I/O port-based* nonché eventi definiti dall'utente. L'idea alla base del suo funzionamento è molto semplice: il *Reactor* "ascolta" continuamente l'interfaccia di *event-demultiplexing* del sistema operativo; quando un evento ha luogo (un segnale inviato da un processo, un *timer* scaduto, ecc.), il *Reactor* lo riconosce e attiva l'*handler* interessato all'evento preventivamente registrato presso di sé. Inoltre esso può attivare *handler* anche quando hanno luogo eventi definiti dall'utente. Nell'utilizzo del *Reactor*, in generale lo sviluppatore deve quindi:

- creare l'*Event-Handler* per gestire l'occorrenza di un particolare tipo di evento;
- registrare l'*Event-Handler* al *Reactor*.

Dal canto suo, il *Reactor* è progettato per:

- gestire strutture dati per mantenere l'associazione tra tutti gli *Event-Handler* in esso registrati e le rispettive corrispondenze ai tipi di eventi a cui essi sono interessati;

- attivare l'*Event-Handler* appropriato quando un evento occorre.

In figura 5-8 è riportato un esempio di utilizzo del *Reactor* per la gestione degli eventi definiti dall'utente. Nel caso riportato in figura, l'evento è la pressione del tasto <invio>: quando un utente preme <invio> una notifica è inviata al *Reactor* il quale attiverà l' *handler* di gestione dell'evento preventivamente registrato. Dato che il *Reactor* sa manipolare solo eventi gestiti dal sistema operativo, la notifica definita dall'utente in pratica forza l'occorrenza dell'evento a cui l'*handler* si è inscritto. Altri elementi importanti che è possibile evidenziare nel frammento di codice riportato sono:

1) L'implementazione della classe e dei metodi per la gestione dell'evento: la classe deve ereditare *ACE_Event_Handler* e ridefinirne il/i metodi per la gestione degli eventi cui si è interessati;
2) La registrazione della classe all'interno del *Reactor* con il metodo *register_handler()*;
3) L'attivazione del *Reactor* in un ciclo perenne per il *demultiplexing* degli eventi ed il *dispatching* degli *handler.*

```
#include <signal.h>
#include <stdio.h>
#include ”ace/Reactor.h”
#include ”ace/Event_Handler.h”
#include “ace/ace_os.h”
#DEFINE EVENT 0x0D “carriage return”
// Since that this particular event handler is going to be
// using I/O event we only overload the handle_input method
class MyEventHandler: public ACE_Event_Handler {
  int handle_input(int) {
    ACE_DEBUG((LM_DEBUG, ”Event occurred\n”));
    return 0;
  }
};
int main(int argc, char *argv[]) {
// instantiate the handler: eh handles user event
  MyEventHandler *eh = new MyEventHandler;
// Register the handler asking to call back when user event occurs
(*) ACE_Reactor::instance()->register_handler(ACE_Event_Handler::READ_MASK,eh);
  char c;
  switch (ACE_OS::fork()) {
    case –1:
      ACE_ERROR_RETURN ((LM_ERROR, "error with fork"), 1);
    break;
    //Start the reactors event loop
    case 0:
      while(1) ACE_Reactor::instance()->handle_events();
    break;
    // someone can generate events
    default:
      while (1) {
        c = getchar();
        // event generated: a CARRIAGE RETURN is typed
        if (c == EVENT)
          ACE_Reactor::instance()->notify(ACE_Event_Handler::READ_MASK)
      }
    break;
  }
  return 0;
}
```

Figura 5-8: esempio di utilizzo del *Reactor*.

5.3.2 Modulo di *adaptation*

Come è stato ampiamente illustrato al par. 4.3.1, il modello di *adaptation* Esperanto fornisce due meccanismi di adattamento:

application-driven: l'adattamento ai mutamenti della particolare risorsa, avviene monitorandone attivamente lo stato. In questa strategia, gli Esperanto *Peer* utilizzano il monitor ad essa associato per ottenere la sua rappresentazione dello stato ad un elevato livello di astrazione. L'adattamento dell'applicazione è basato sul *polling* attivo (delle condizioni attuali della risorsa) compiuto dal *Resource Monitor*.

event-driven: l'adattamento ai mutamenti della particolare risorsa avviene all'atto di una sua transizione di stato, tramite l'attivazione di una *routine*

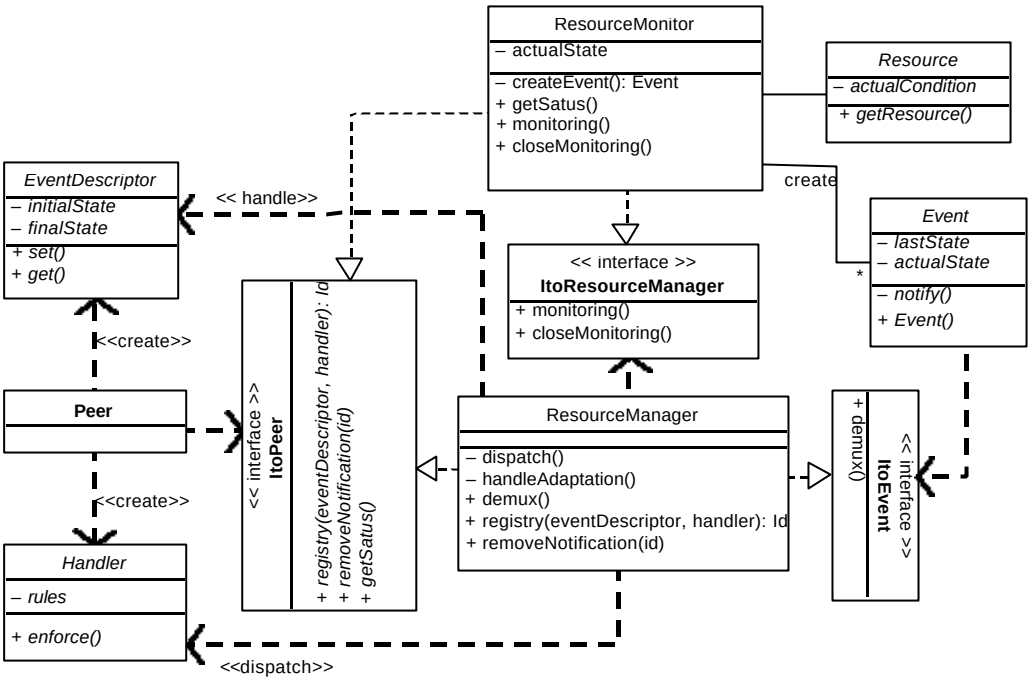

Figura 5-9: *Pattern* per le strategie di *adaptation application-aware.*

definita dall'applicazione. In questa strategia di adattamento, gli Esperanto *Peer* descrivono l'evento associato alla risorsa cui sono interessati tramite un descrittore, ed applicano l'adattamento all'interno di un *handler*. Queste due informazioni sono poi passate, attraverso un'operazione di registrazione, al *Resource Manager*, il quale si fa carico di attivare l'*handler* non appena l'evento occorre. Sinteticamente, il *Resource Manager* è un *event demultiplexer* ed *handler dispatcher*: classifica l'evento che occorre e se c'è qualche *handler* interessato alla sua gestione, lo attiva. Il suo comportamento è definito pertanto da un *Reactor pattern* [52].

Il *pattern* di figura 4-19, che qui è nuovamente riportato (figura 5-9), fa riferimento ad uno scenario in cui un'applicazione desidera essere consapevole di una sola risorsa di contesto. Ad ogni risorsa è dunque associata una classe *wrapper* (*Resource*) per stimare la sua condizione, una classe monitor (*Resource Monitor*) per monitorare lo stato ed i suoi cambiamenti, una classe manager (*Resource Manager*) per la notifica dei cambiamenti di stato.

Gli eventi ad essa associati sono modellati con delle transizioni di stato (i dati membro della classe *Event*) e l'interesse verso l'occorrenza di un particolare

evento o di una particolare famiglia di eventi viene rappresentata con un descrittore delle transizioni di stato (i dati membro della classe *EventDescriptor*). In un contesto più ampio, fatto da più risorse e più applicazioni, è necessario definire un *pattern* più generale nel cui progetto sono tenute in conto le seguenti esigenze:

a) *Ogni risorsa fisica che caratterizza il contesto deve essere associata ad una ed una sola classe wrapper*. Coerentemente al punto a) del par. 4.3.1 un modello del contesto prevede che esso sia composto da tante risorse (in Esperanto la connettività di rete, l'energia e la locazione dei dispositivi). La stima delle risorse è un'operazione costosa (ad esempio, le misure di rete sono tipicamente intrusive) e che "ruba" risorse computazionali. Il modulo di *adaptation* destinato ad ogni dispositivo Esperanto deve pertanto prevedere che, per ogni risorsa sottoposta a monitoraggio, debba esserci una ed una sola classe *wrapper*.

b) *Ogni entità applicativa deve disporre di uno ed un solo manager (monitor) per ogni risorsa cui è interessata.* Caratteristica desiderabile della soluzione è quella di essere semplice sotto l'aspetto computazionale. Tuttavia manager e monitor sono componenti complessi e prevedere che ci sia un unico manager (ed un monitor) per ogni risorsa in tutto il modulo di *adaptation* è una soluzione poco scalabile. Imporre un manager ed un monitor (assegnati ad ogni risorsa) per ogni *Peer* che ne fa richiesta rappresenta un giusto compromesso.

c) *Bisogna astrarre quanto possibile le caratteristiche comuni dei manager, dei monitor, delle classi wrapper, ecc. al fine di rendere il pattern quanto più riusabile ed estendibile.* Tutti i manager, ad esempio, condividono la medesima struttura (con i metodi *registry()* e *removeNotification()*), ma i descrittori degli eventi variano in funzione del tipo con cui si rappresenta lo stato della particolare risorsa a cui fanno riferimento (cfr. 4.3.2, 4.3.3, 4.3.4). Allo stesso modo anche le classi *wrapper* condividono la stessa struttura (ognuna di esse possiede un metodo per stimare le condizioni della risorsa e

renderle disponibili ai propri monitor), ma ogni risorsa è caratterizzata in un modo a sé (cfr. 4.3.2, 4.3.3, 4.3.4). Queste motivazioni rendono difficile l'uso delle consuete tecniche *Object-Oriented* come ereditarietà e polimorfismo, e bisogna pertanto ricorrere anche all'uso della genericità e dei *templates*.

5.3.2.1 *PATTERN* DELLA SOLUZIONE

Un *pattern* che tenga conto delle considerazioni esposte al paragrafo precedente è illustrato in figura 5-10. Gli elementi responsabili delle politiche di allocazione delle classi, indicate ai punti a) e b), sono i *factory pattern*.

L'***Abstract Resource Factory***, e di conseguenza il *Resource Factory*, ha il compito di creare, per ogni tipo di risorsa che caratterizza il contesto, un'unica istanza della sua classe *wrapper* in tutto il modulo di *adaptation*. Diversamente dal *factory pattern* riportato in letteratura ([31] e [17]), il prodotto "della fabbrica di risorse" non eredita una interfaccia astratta, ma una classe generica parametrica rispetto a *Tcondition*, il tipo con cui (un oggetto, una struttura, un tipo semplice,…) si rappresenta la condizione attuale della risorsa. La gerarchia è utile ad estendere le proprietà della classe *Resource*: oltre al metodo *getResource()* (che ritorna la condizione corrente della risorsa) deve essere previsto un metodo per la stima della particolare risorsa associata alla classe *wrapper*. In figura 5-11 si riporta, a scopo esemplificativo, uno dei possibili modi per ottenere la classe *wrapper* della risorsa "connettività di rete" derivando la classe *Resource*, base della gerarchia.

Il *ResourceFactory* collabora con i monitor di risorsa assegnati ad ogni *Peer*, per fornire il riferimento all'unica istanza della classe *wrapper* che ad essi compete. Pertanto lo specifico monitor di risorsa A otterrà l'istanza della risorsa A invocando sul *factory* il metodo corrispondente *getResourceARef()*.

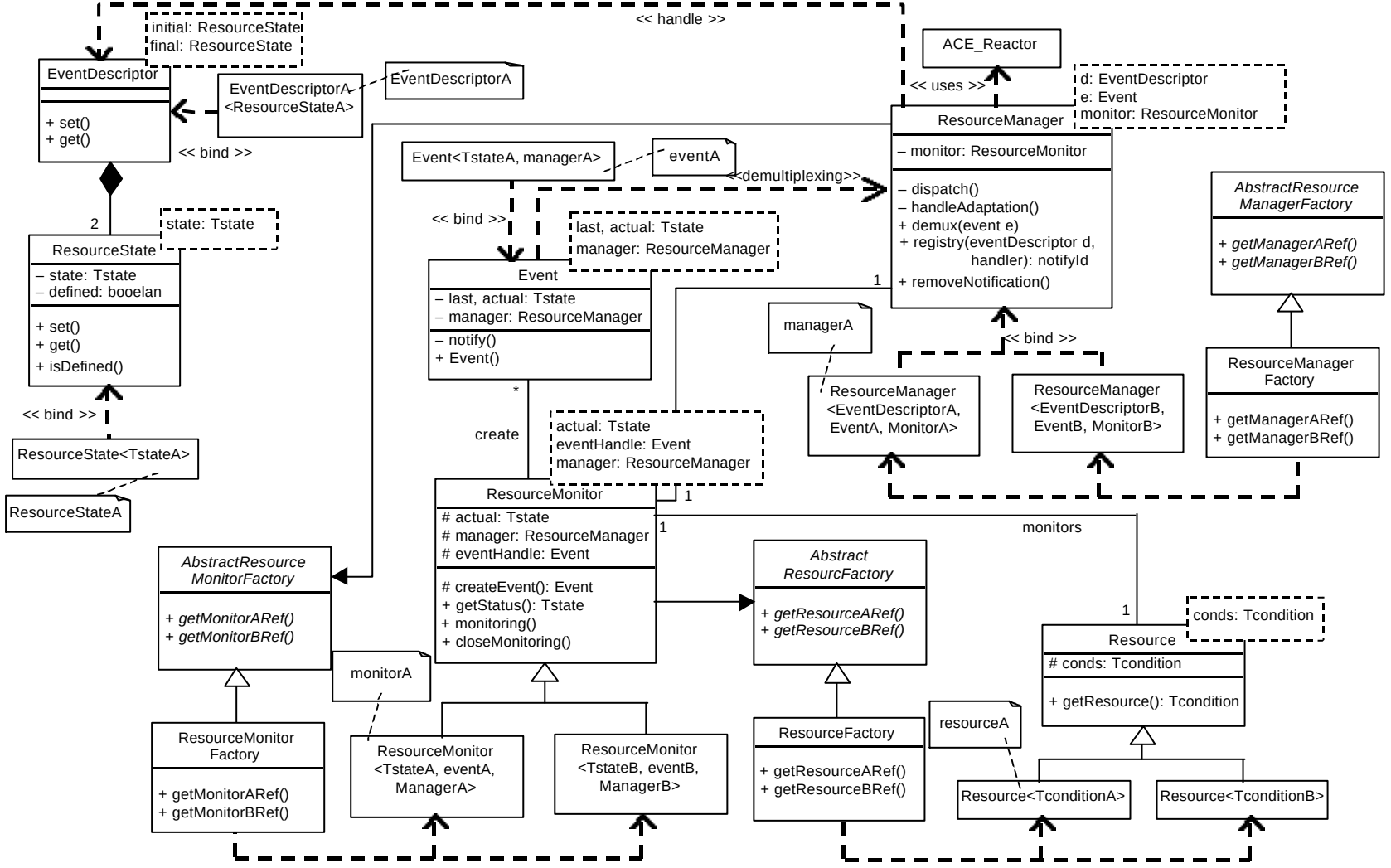

Figura 5-10: il modulo di *adaptation*.

```
// a way to represent network condition
typedef struct {
  double delay;
  double delayVariation;
  double informationLoss;
} networkConditionType;

template <class resourceConditionType> class resource {
  protected:
    resourceConditionType condition;
  public:
    resourceConditionType getCondition() const {
      return condition;
    };
}; // resource

class networkResource: public resource<networkConditionType> {
  protected:
    networkConditionType estimate(); // evaluate network condition
  public:
    networkConditionType getCondition() {
      condition = estimate();                    // update condition
      return resource<networkConditionType>::getCondition();
    };
}; // networkResource
```

Figura 5-11: rappresentazione in C++ della classe *Resource* riportata nel *pattern* di figura 5-10 ed una sua estensione *networkResource*.

L'***Abstract Resource Monitor Factory***, e di conseguenza il *Resource Monitor Factory*, ha il compito di creare, per ogni *Peer* (su esplicita richiesta nella strategia *application-driven* o indirettamente nella strategia *event-driven*), uno ed un solo monitor per ogni risorsa del contesto. Come nel caso del *factory pattern* utilizzato per le risorse, allo stesso modo anche il *factory* dei monitor prevede una classe generica alla base della gerarchia dei "prodotti": essa è parametrica rispetto lo stato (la rappresentazione di alto livello offerta ai *Peer*) della risorsa (che può essere codificato con tipi semplici, strutturati o definiti dall'utente), rispetto al tipo di manager e rispetto al tipo di eventi che deve generare. La gerarchia è necessaria per specializzare metodi come *getStatus()*, le cui proprietà dipendono dalla risorsa e dal tipo di stato adottato. Altri metodi, come *monitoring()* o *createEvent()* sono sufficientemente generali per essere definiti nella classe base e riutilizzati nelle classi derivate. In figura 5-12 si riporta il manager della risorsa "connettività di rete". Insieme alla dichiarazione del monitor si riporta anche la classe degli eventi da esso generati. Un oggetto *Event* è creato dal monitor all'atto della transizione di stato della risorsa; i suoi attributi membro memorizzano lo stato da cui si è scatenato l'evento e lo stato in cui la risorsa si è portata a seguito dell'evento. La sua responsabilità consiste nella notifica al manager corrispondente, al fine di

```
// a way to represent an high-level state of network resource
enum networkChannel { WORST, WORSE, BAD, NORMAL, GOOD, BETTER, BEST };

// event treated by resource monitor
template <class state, class managerRef> class event {
   private:
     state last;
     state actual;
     void notify();
   public:
     event(const state, const state, const managerRef);
}; // event

template <class Tstate, class managerRef, class eventRef>
class resourceMonitor {
  protected:
    Tstate actual;
    managerRef man;
    eventRef event;
    eventRef createEvent(const managerRef, const Tstate, const Tstate);
  public:
    Tstate getStatus();
    void monitoring (const managerRef);
    void closeMonitoring();
}; // resourceMonitor

// event occured when network changes its state
typedef event<networkChannel, networkManagerRef> networkEvent;

class networkMonitor:
    public resourceMonitor<networkChannel, networkManagerRef, networkEvent> {
  public:
    // redefinition of getStatus() method
}; // networkMonitor
```

Figura 5-12: rappresentazione in C++ delle classi *resourceMonitor* ed *Event* riportate nel *pattern* di figura 5-10 con le estensioni relative alla risorsa "connettività di rete".

effettuare *demultiplexing* ed *handler dispatching*. Per la sufficiente generalità del suo comportamento, non è prevista alcuna gerarchia, ma una semplice specializzazione dei parametri *template* (lo stereotipo UML <<*bind*>> sta ad indicare proprio questo).

Il *Resource Monitor Factory* collabora con i manager di risorsa assegnati ad ogni *Peer*, o con gli stessi *Peer*, per fornire il riferimento all'istanza del monitor della risorsa cui si è interessati. Nel primo caso, il manager di una risorsa ottiene un riferimento al monitor ad esso corrispondente, allo scopo di implementare i meccanismi di notifica degli eventi; nel secondo caso, il *Peer* ottiene il riferimento allo specifico monitor di risorsa per ottenere il suo stato corrente. In entrambi i casi, si otterrà l'istanza del monitor A, se si invoca sul *factory* il metodo corrispondente *getMonitorARef()*.

```
enum boolean { FALSE, TRUE };

template <class Tstate> class resourceState {
  private:
    boolean defined;
    Tstate state;
  public:
    boolean isDefined() const;
    void set(const Tstate);
    Tstate get() const;
}; // resourceState

template <class resourceState> class eventDescriptor {
   private:
     resourceState initial;
     resourceState final;
   public:
     void setInitial(const resourceState);
     void setFinal(const resourceState);
     resourceState getInitial() const;
     resourceState getFinal() const;
}; // eventDescritor

typedef unsigned long notifyId;

template <class eventDescriptor, class event, class monitorRef>
class resourceManager {
  private:
    monitorRef mon;
    ACE_Reactor reactor;
    // any abstract data type (such as hash table...)
    // to store eventDescriptor and handler objects
    void handleAdaptation();
    void dispatch();
  public:
    void demux(const event);
    notifyId registry(const eventDescriptor, const handler);
    void removeNotification(const notifyId);
}; // resourceManager


enum networkChannel { WORST, WORSE, BAD, NORMAL, GOOD, BETTER, BEST };

typedef resourceState<networkChannel> networkState;
typedef eventDescriptor<networkState> networkEventDescriptor;

typedef resourceManager<networkEventDescriptor, networkEvent, networkMonitorRef>
networkManager;
```

Figura 5-13: rappresentazione in C++ delle classi *resourceManager* ed *event Descriptor* riportate nel *pattern* di figura 5-10 con le estensioni relative alla risorsa "connettività di rete".

L'***Abstract Resource Manager Factory***, e di conseguenza il *Resource Manager Factory*, ha il compito di creare, per ogni *Peer*, uno ed un solo manager per ogni risorsa del contesto a cui si vuole applicare la strategia di adattamento *event-driven*. Come per le classi *wrapper* ed i *Resource Monitor*, il *factory pattern* crea le istanze di un manager *template*, parametrico rispetto al tipo di monitor, al tipo del descrittore di evento e all'evento che deve trattare. A differenza della classe *Resource Monitor*, la classe *Resource Manager* è sufficientemente generale da

non prevedere alcuna gerarchia per specializzare il suo comportamento. La definizione di un *template* per il manager, da specializzare in base ai tipi degli oggetti con cui deve trattare, consente di utilizzarlo per ogni risorsa che compone il contesto.

Per le buone caratteristiche di performance, affidabilità ed efficienza dei *pattern* del *framework* ACE, l'implementazione del manager può essere fatta utilizzando i servizi del *Reactor*, riusando pertanto codice già testato, efficiente e portabile. In figura 5-13 si riporta la dichiarazione della classe modello *Resource Manager*. In figura, si riporta anche la classe *EventDescriptor*: secondo le indicazioni del paragrafo 4.3.1, per una data risorsa, un descrittore di un evento deve prevedere due attributi membro per memorizzare gli stati coinvolti nella operazione di notifica. Poiché non tutte le astrazioni degli stati di una risorsa prevedono un valore convenzionale "NULL" (il tipo enumerativo *networkChannel*, ad esempio, non possiede alcun valore che possa indicare uno stato indefinito) si incapsula lo stato astratto della risorsa all'interno di una classe, *ResourceState*, che associa ad esso un valore booleano. Conseguentemente anch'essa è una classe modello.

Il *Resource Manager Factory* collabora con ogni *Peer*, per fornire il riferimento all'istanza del manager della risorsa cui essi sono interessati.

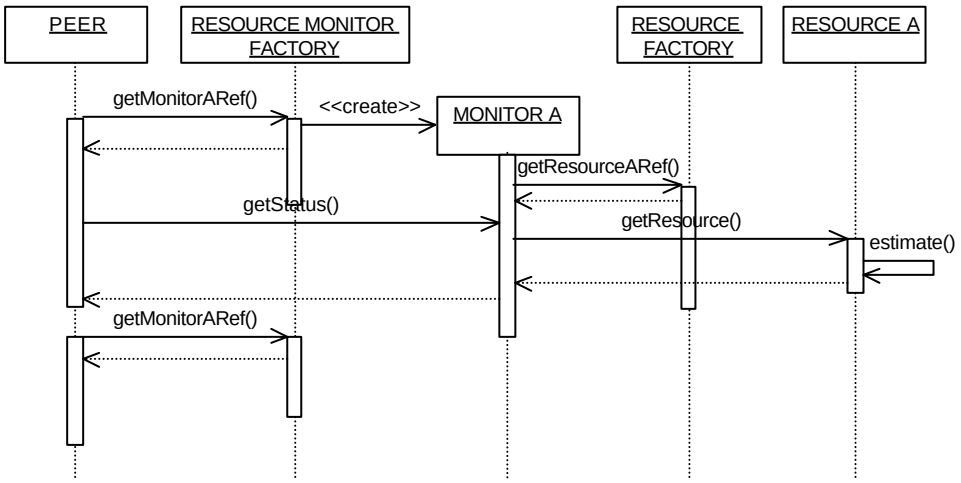

Figura 5-14: le operazioni compiute da un *Peer* per ottenere i servizi dal modulo di *adaptation* nel caso di strategia *application-driven*.

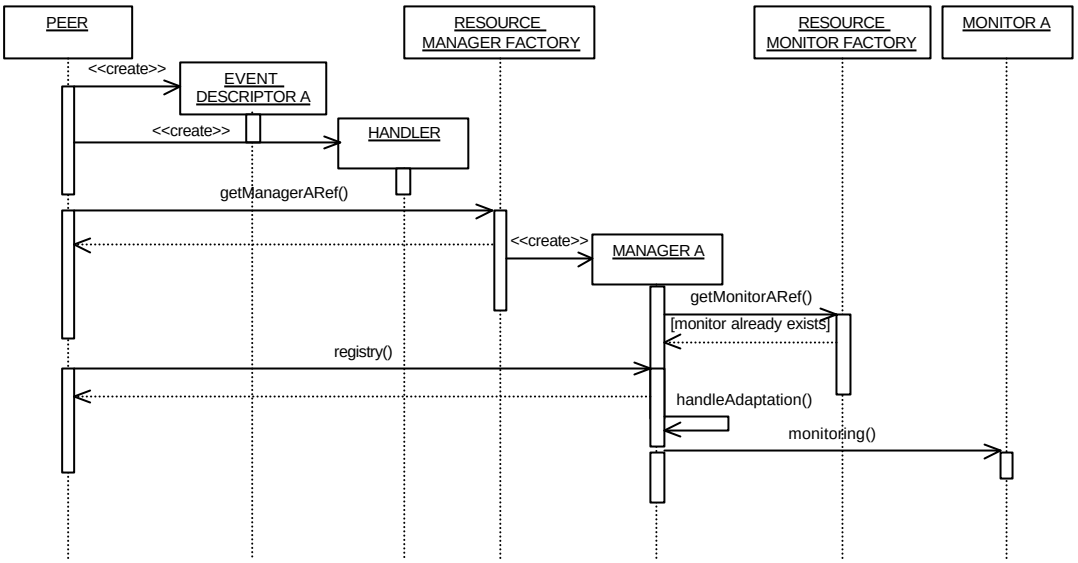

Figura 5-15: le operazioni compiute da un *Peer* per ottenere i servizi dal modulo di *adaptation* nel caso di strategia *event-driven*.

In figura 5-14 è riportato il comportamento dinamico di un *Peer* per accedere ai servizi offerti da un *Resource Monitor* di una particolare risorsa (nell'esempio, la risorsa A). E' evidente che una volta ottenuto il riferimento allo specifico monitor, una nuova richiesta al *factory* comporta la restituzione del riferimento al monitor già allocato. Il monitor, a sua volta, all'atto della sua inizializzazione, richiede al *factory* delle risorse il riferimento alla classe *wrapper* che gli compete: se la classe non è disponibile è attivata prima di restituire il suo riferimento (nel diagramma questa sequenza non è riportata).

In figura 5-15 è riportato invece il comportamento del *Peer* per accedere ai servizi offerti dal *Resource Manager*. Come nel caso precedente, la creazione del manager e del monitor della risorsa avviene solo alla prima richiesta rivolta ai rispettivi *factory*. Anche nel caso in cui il *Peer* abbia già creato il monitor corrispondente e non lo abbia esplicitamente distrutto, la richiesta da parte del manager comporta la restituzione del riferimento al monitor senza ulteriore dispendio di risorse.

5.3.2.2 ASPETTI REALIZZATIVI

Diversamente dalla strategia di *adaptation application-driven*, la strategia *event-driven* rende più delicata la progettazione delle applicazioni *context-aware*, a causa dell'impredicibilità degli eventi e della complessità introdotta dallo sviluppo dei loro *handler* di gestione.

Da un punto di vista puramente concettuale un'applicazione *context-aware* può essere decomposta in tre sezioni logicamente distinte:

1) **Procedura di elaborazione**: questa sezione comprende le consuete operazioni compiute dall'applicazione (ad esempio l'insieme delle operazioni per espletare un servizio), e sviluppate sulla base della *business logic* del programma. Naturalmente tale procedura, nelle applicazioni *context-aware* è influenzata dal contesto in cui esse sono eseguite.

2) **Procedura di adattamento**: questa sezione comprende le operazioni compiute dall'applicazione per adattarsi alla variazione del contesto. Essa dipende dalla *business logic* del programma, ma soprattutto dal tipo di contesto a cui si vuole essere consapevoli.

3) **Stato dell'elaborazione**: ogni applicazione, in ogni istante della sua esecuzione, è caratterizzata da uno stato (l'insieme di variabili, le risorse assegnate all'applicazione, le connessioni attive, i *files* aperti ecc.). Il mutamento del contesto influenza il comportamento delle applicazioni *context-aware* alterando indirettamente (o direttamente) il loro stato di elaborazione: a fronte delle variazioni del contesto, le applicazioni *context-aware* adattano il loro stato di elaborazione, attraverso le ***procedure di adattamento***, per applicare efficacemente le ***procedure di elaborazione***.

Nel caso di *adaptation application-driven*, il momento in cui applicare l'adattamento è stabilito dallo sviluppatore all'atto del progetto dell'applicazione; conseguentemente egli sa, a priori, come coordinare le procedure di elaborazione e le procedure di adattamento (può decidere di eseguirle in sequenza, oppure in concorrenza). Nel caso di *adaptation event-driven*, invece, le procedure di

adattamento possono intervenire in maniera imprevedibile. Pertanto per la loro coordinazione con le procedure di elaborazione bisogna ricorrere necessariamente ai meccanismi di sincronizzazione e/o di protezione di informazioni condivise.

In altre parole, nel primo caso, il ricorso ai principi della programmazione concorrente (nell'esecuzione delle procedure di adattamento e di elaborazione) è motivato dalle performance; nel secondo caso è invece un'esigenza.

Definire un modello di programmazione generale da adottare nello sviluppo di applicazioni *context-aware*, è reso estremamente complesso dalle particolari esigenze del dominio applicativo. Tuttavia, al fine di comprendere le considerazioni appena esposte e per illustrare con dettaglio il funzionamento dei componenti del *pattern* proposto, in questo paragrafo saranno messe a confronto le due strategie di adattamento previste nel supporto alla dinamicità di Esperanto.

In figura 5-16 e 5-17 sono riportati i frammenti di codice di un'applicazione che usa i servizi del modulo di *adaptation*: si vuole notificare il numero di transizioni di stato di una particolare risorsa (nell'esempio, le variazioni di stato della connettività di rete da *GOOD* a *WORSE*). L'adattamento dell'applicazione alle variazioni della risorsa consiste dunque nel conteggio di tali transizioni di stato; il valore del contatore è restituito ai clienti tramite un'operazione *one-way server-push* (*notification*). In figura 5-16, l'applicazione implementa la *business logic* adottando una strategia di adattamento *application-driven*; in figura 5-17, la strategia adottata è quella *event-driven*. Commentiamole singolarmente:

I. L'adattamento riportato in figura 5-16 avviene su esplicita volontà dell'applicazione: la procedura di elaborazione *notifyCount()* e la procedura di adattamento *adaptation()* sono eseguite in sequenza, modificando il valore del contatore alternativamente. L'applicazione è stata quindi progettata in modo che l'adattamento e la consueta elaborazione avvengano in tempi distinti.

```
class incrementService {
  private:
    peerId myId;
    notifyId nid;
    tuple t;
    networkMonitor* mon;
    networkChannel networkS1, networkS2;     // state
    int count;
       ...
    void advertiseService();         // advertising service to Esperanto Explorer
    void unregistryService();        // unregistry service to Esperanto Explorer
    void activateService() {         // obtain monitor reference
          // init monitor factory...
      mon = monitorFactory.getNetworkMonitorRef();
       ...
    };
    void adaptation() {                      // adaptation procedure
      networkS2 = mon->getStatus();
      if ((networkS1 == GOOD) && (networkS2 == WORSE)) {
       count++;
       networkS1 = networkS2;
      } // if
    };
    void getCount() {                        // computing procedure
      t.data.set(count);
      deliveryAgent->write(t, max_expiration);
    };
    void notifyCount() {                     // server-push operation
      t.destination.set(all);
      t.source.set(myId);
      while (1) {
       adaptation();
       getCount();
         // sleep
      } // while
    };
  public:
    incrementService() {                           // service start up
      ...
      myId = subscribeAgent->assignId();
      activateService();
      advertiseService();
      count = 0;
      networkS1 = networkS2 = mon->getStatus();
      notifyCount();
    };
    ~incrementService() {                          // destroy service
      monitorFactory.destroyNetworkMonitor();
      unregistryService();
      subscribeAgent->disposeId();
       ...
   };
};
```

Figura 5-16: esempio di un'applicazione *context-aware* che applica una strategia di adattamento *application-driven*.

L'adattamento avviene grazie all'uso del monitor associato alla risorsa "connettività di rete", il quale restituisce lo stato della risorsa ogni volta che viene richiesto. Il riferimento ad esso è ottenuto tramite il *factory* dei monitor.

```
class handler {
  private:
    ACE_Event_Handler* p;
  public:
    handler(): p(0) {};
    handler(ACE_Event_Handler* h): p(h) {};
};

class incrementService: public ACE_Event_Handler {
  private:
    pthread_mutex_t mutexCount;      // state
    int count;
    peerId myId;
    notifyId nid;
    tuple t;
    networkManager* man;
    void advertiseService();         // advertising service to Esperanto Explorer
    void unregistryService();        // unregistry service to Esperanto Explorer
    void activateService() {         // register event notification
      handler HANDLER(this);
      networkState s1(GOOD), s2(WORSE);
      networkEventDescriptor event;
      event.setInitial(s1); event.setFinal(s2);
       // init manager factory...
      man = managerFactory.getNetworkManagerRef();
      nid = man->registry(event, HANDLER);
        ...
    };
    void getCount () {                     // computing procedure
      pthread_mutex_lock (&mutexCount);
        t.data.set(count);
      pthread_mutex_unlock (&mutexCount);
      deliveryAgent->write(t, max_expiration);
    };
    void notifyCount() {                   // server-push operation
      t.destination.set(all);
      t.source.set(myId);
      while (1) {
       getCount();
        // sleep
      } // while
    };
  public:
    incrementService() {     // service start up
       ...
      myId = subscribeAgent->assignId();
      activateService(); advertiseService();
      count = 0;
      pthread_mutex_init(&mutexCount, NULL);
      notifyCount();

    };
    virtual int handle_input(int) {         // adaptation procedure
      pthread_mutex_lock (&mutexCount);
       count++;
      pthread_mutex_unlock (&mutexCount);
      return 0;
    };
    ~incrementService() {
      man->removeNotification(nid);
      managerFactory.destroyNetworkManager();
      unregistryService();
      subscribeAgent->disposeId();
       ...
      pthread_mutex_destroy(&mutexCount);
      pthread_exit(NULL);
    };
};
```

Figura 5-17: esempio di un'applicazione *context-aware* che applica una strategia di adattamento *event-driven*.

II. L'adattamento riportato in figura 5-17 è invece causato dall'evento e può intervenire in qualsiasi momento. Per questo motivo, l'*handler* di gestione (il metodo virtuale *handle_input()* ereditato dalla classe *ACE_Event_Handler* e ridefinito) deve essere attivato dal manager della risorsa rete in concorrenza alla consueta procedura di elaborazione compiuta dall'applicazione. L'aggiornamento della variabile *count* può quindi capitare anche quando si invoca *getCount()* ed è necessario proteggerla con una variabile di mutua esclusione.

L'efficienza del secondo approccio porta con sé una maggiore complessità del modello di programmazione. Infatti, oltre a gestire la sincronizzazione delle operazioni e l'accesso in mutua esclusione sullo stato dell'applicazione, è necessario (secondo l'approccio reattivo) creare il descrittore dell'evento, sviluppare l'*handler*, registrarsi e deregistrarsi al manager della risorsa. Per comprendere meglio i passi compiuti nella fase di attivazione, facciamo luce sulle azioni eseguite nella procedura *activateService()*:

inizializzazione della classe *handler*: essa incapsula al suo interno un puntatore ad una classe *ACE_Event_Handler*. Utilizzando il *Reactor* nell'implementazione di un *Resource Manager*, il modello di applicazione delle regole di adattamento (cfr. 4.3.1) è definito sulla base di quello adottato in ACE. In ACE, ogni *handler* di gestione di un evento deve ereditare da *ACE_Event_Handler*, che prevede diversi metodi virtuali puri associati alla gestione di particolari eventi. Nell'esempio di fig. 5-8, per gestire l'evento simulato "lettura da un *device* di I/O" (sia esso una *pipe*, una *socket* o un *file*), nella classe *MyEventHandler* è stato ridefinito il metodo virtuale *handle_input()*. La definizione della nuova classe, *handler*, permette di non alterare il modello di applicazione delle regole definito in ACE, ma allo stesso tempo consente di mantenere uniforme il modello di *adaptation* in Esperanto: per produrre un *handler* di gestione di un evento descritto da *eventDescriptor* (dopo che esso è stato debitamente compilato), si deriva quindi una classe da *ACE_Event_Handler*, si ridefinisce il metodo che deve gestire l'evento, e si

incapsula il puntatore all'*ACE_Event_Handler* all'interno di un oggetto della classe *handler*. L'invocazione del metodo *registry(eventDescriptor, handler)* del *ResourceManager* realizza nella pratica la stessa semantica dell'operazione (*) dell'esempio 5-8. In figura 5-17, il servizio eredita direttamente da *ACE_Event_Handler* e conseguentemente è esso stesso l'*handler* di gestione dell'evento.

Compilazione del descrittore di evento: in figura è riportato l'uso del descrittore di evento associato alla risorsa "connettività di rete" (*networkEventDescriptor*). Questa classe incapsula due oggetti per memorizzare gli stati della risorsa coinvolti nella operazione di notifica, di tipo *networkState.* La classe *networkState* serve ad associare alla rappresentazione di alto livello dello stato della risorsa (espresso attraverso un qualsiasi tipo definito dall'utente – nel caso indicato il tipo enumerativo *networkChannel*) un valore booleano atto ad indicare se l'istanza della variabile (di tipo *networkChannel*) contiene un valore significativo oppure no. Questo è utile ai fini del *demultiplexing* degli eventi perché aiuta a stabilire se l'applicazione specifica nel descrittore l'interesse verso un singolo evento o una famiglia di eventi.

Con l'operazione di *registry(),* il manager registra gli *handler* all'interno delle tabelle gestite dal *Reactor* tramite l'operazione *register_handler()*: più in dettaglio, quando un *Peer* invoca questa operazione, il manager compie le seguenti azioni:

- costruisce una tabella di corrispondenza per l'associazione (*EventDescriptor*, *ACE_Event_Handler::MASK*, *handler*) ed assegna ad ogni *entry* un ID, che restituisce al *Peer*. Ogni *entry* della tabella è costituita dal descrittore dell'evento, l'*handler* ed una maschera associata all'evento;
- registra l'*handler* all'interno del *Reactor* per la gestione degli eventi associati alla *ACE_Event_Handler::MASK*. L'uso del *Reactor* nello

sviluppo del manager lo solleva dalla gestione delle strutture dati usate per la registrazione e l'attivazione degli *handler*.

Il monitor monitora la risorsa con il metodo *monitoring()*. Quando la risorsa cambia stato esso genera un evento *Event* che attiva il manager corrispondente (il costruttore della classe, con il metodo *notify()*, invia il messaggio *demux()* al manager passando come parametro l'oggetto stesso). Il manager stabilisce se ci sono uno o più *Peer* interessati all'evento occorso ed in caso positivo invoca la notifica *user-defined* sul *Reactor* (all'interno del metodo *dispatch()* si invoca il metodo di notifica del *Reactor*) che si farà carico di attivare gli *handler*. In ogni caso l'oggetto creato dal monitor viene distrutto.

Conclusioni

La volontà dell'Ingegneria del Software di rispondere alle esigenze degli sviluppatori quali la riusabilità, la flessibilità, l'adattabilità, l'estensibilità, si è spinta fino all'affermazione delle *Service Oriented Architecture*. Tante entità in grado di offrire e usare servizi costituiscono il contesto d'elaborazione delle applicazioni *service oriented*. Servizi più semplici possono costituirne di più complessi tramite il loro assemblaggio. Grazie all'adozione di un modello di interazione *loose coupled* dove le interfacce dei servizi sono ben distinte dalle loro implementazioni, sono localizzate e gestite da entità individuate ad hoc (i cosiddetti *service broker*, *service registry*, *service locator*…) che implementano opportune strategie volte alla flessibilità e adattabilità nello sviluppo delle applicazioni, questi ambienti ben si prestano alla realizzazione degli scenari del prossimo futuro in cui le telecomunicazioni e tecnologie *wireless* saranno sempre più tangibili nell'utilizzo di *wearable computer*, *personal digital assistant* e *smart phone* come nodi di tante *personal network* individuali.

Ai principi delle *Service Oriented Architecture*, vanno quindi affiancati i concetti come la mobilità e la dinamicità implicate dall'uso di questi dispositivi. Durante il *delivery* di un servizio, la libertà di movimento è più una regola che l'eccezione, e progettare soluzioni in cui è prevista la gestione delle migrazioni e delle disconnessioni (dei dispositivi), i meccanismi per reagire ai cambiamenti del contesto che circonda l'elaborazione e la consapevolezza dello stato delle risorse che lo caratterizzano è un requisito fondamentale nello sviluppo di nuove piattaforme *middleware service oriented*.

In questo lavoro di tesi è stato affrontato il progetto di una nuova piattaforma *middleware* per lo sviluppo di applicazioni distribuite per ambienti mobili orientate ai servizi, ma non basta. Una nuova piattaforma non è mai una risposta efficace se non guarda ai bisogni del passato e alla volontà di rappresentare un'evoluzione graduale dai vecchi sistemi di elaborazione ai nuovi sistemi emergenti.

In Esperanto ciò si è concretizzato nell'adozione delle infrastrutture di *nomadic computing*, infrastrutture di rete ibride *wireless* e *wired* che permettono la coesistenza di vecchie e nuove soluzioni. Attraverso le caratteristiche offerte da questo tipo di infrastrutture è possibile quindi rispettare i vincoli presenti nell'ambito del *mobile computing* e inglobare all'interno di un unico ambiente omogeneo anche le soluzioni tecnologiche passate. Il modello architetturale di Esperanto si basa proprio sul *core* di rete fissa dell'infrastruttura di *nomadic computing*. Elementi come il *Mediator*, il *Domain-specific/Esperanto Agent* e l'*Esperanto Explorer*, nascono con l'intento di rispondere efficacemente ai requisiti e ai vincoli degli ambienti mobili. Il *Mediator* consente lo sviluppo di interazioni disaccoppiate nel tempo, nello spazio e nel sincronismo, garantendo di conseguenza un efficiente paradigma di comunicazione per le applicazioni mobili. L'*Esperanto Explorer* solleva i dispositivi mobili della gestione e della complessità di tutte le operazioni relative all'approccio *service oriented*. Inoltre, grazie al concetto di dominio previsto nella visione Esperanto, è anche possibile gestire le migrazioni dei dispositivi. Gli agenti nascono per integrare tutti i domini connessi all'infrastruttura di *nomadic computing* in un unico ambiente omogeneo: ogni cliente/servizio di un dominio può utilizzare servizi di altri domini senza alterare il proprio modello di interazione e senza avere consapevolezza dei meccanismi di *bridging* sottostanti.

Le strategie di *adaptation* sono fornite come strumenti da utilizzare nelle operazioni di *service delivery* con un approccio *context-aware*, in cui le applicazioni hanno consapevolezza dei mutamenti del contesto e si adattano ad essi coerentemente alle loro esigenze.

In questo lavoro sono stati progettati tutti questi elementi. E' stata fatta luce sugli aspetti del *service delivery*, sono stati individuati i requisiti e vincoli fondamentali per una soluzione di *service delivery* nell'ambito degli ambienti mobili, e sono stati presi in considerazione nel progetto Esperanto. Inoltre, esso deve il nome alla volontà di integrare tecnologie eterogenee. Per questo è stato

necessario uno studio attento dei punti di forza e debolezza delle soluzioni esistenti.

Data la complessità dell'infrastruttura, il progetto è andato in profondità solo per alcuni elementi della piattaforma. Durante lo sviluppo del progetto il mediatore è stato il componente cui è stata posta la maggiore attenzione: per il suo ruolo cruciale nel paradigma di interazione sono stati individuati requisiti di QoS e garanzie nel servizio nella sua implementazione. L'esigenza di un *broker real time* per la comunicazione tra i mediatori si è quindi spinta all'adozione di *RealTime CORBA* ed in particolare di TAO, una sua implementazione *open-source*.

Anche altri aspetti sono stati presi in considerazione. E' stata fatta luce sui meccanismi per garantire la *context-awareness* alle applicazioni attraverso l'implementazione del modulo di *adaptation* da destinare ad ogni dispositivo Esperanto. Sono stati forniti anche i principi dell'*adaptation application-aware* attraverso la formalizzazione di un modello del contesto, delle risorse e delle applicazioni *context-aware*.

Gli immediati sviluppi di questo progetto possono investigare le prestazioni del comportamento dei mediatori nel trasferimento delle tuple, anche allo scopo di migliorare le performance. Inoltre, sfruttando i *pattern* per la definizione dei meccanismi di *adaptation*, è possibile estendere il contesto Esperanto, ora previsto solo come insieme delle risorse connettività di rete, alimentazione e locazione fisica dei dispositivi.

Naturalmente, ci sono problemi aperti o trattati solo in parte. Uno di essi è la tolleranza ai guasti (degli elementi dell'infrastruttura di rete fissa quali mediatori, *explorer* e agenti).

I meccanismi di sicurezza e protezione nell'accesso alle risorse offerte dagli elementi dell'architettura, o nell'accesso ai domini, invece, sono stati previsti solo parzialmente. Questi possono senz'altro essere considerati gli sviluppi futuri dell'infrastruttura di *delivery* Esperanto.

Bibliografia

[1] Mascolo, Capra, Emmerich. *Mobile Computing Middleware*. Dept. of computer Science, University College London. 2002

[2] Lyytinen, Yoo. *Issues and challenges in Ubiquitous Computing*. Communication of ACM. 2002

[3] Kindberg, Fox. *System software for Ubiquitous Computing*. IEEE computer society press. 2002

[4] Henricksen, Indulska, Rakotonirainy. *Infrastructure for pervasive computing: Challenges*. School of computer Science, University of Queensland. 2000

[5] Kleinrock. *Nomadicity: anytime, anywhere in a disconnected world*. J. C. Baltzer AG Science Publishers. 1996

[6] Saha, Mukhrjee. *Pervasive Computing: A paradigm for the 21st Century*. IEEE computer society press. 2003

[7] Russo, Savy, Cotroneo, Sergio. *Introduzione a CORBA*. The McGraw-Hill Companies. 2002

[8] Jung, Paek, Kim. *Design of mobile MOM: Message Oriented Middleware service for mobile computing.* Dept. of Computer Science, Korea University. 1998

[9] Sandland, Israni. *An Architectural Analisys of Message Oriented Middleware*. Dept. of Computer Science, University of Illinois. 2001

[10] Eugster, Felber, Guerraoui, Kermarrec. *The many faces of Publish/Subscribe*. ACM Computer Surveys. 2003

[11] Satyanarayanan. *Mobile information Access*. IEEE personal communications. 1996

[12] Franklin, Zdonik. *A framework for scalable dissemination-based systems.* ACM press. 1997

[13] Johanson, Fox. *Tuplespaces as Coordination Infrastructure for Interactive Workspaces*. Stanford University. 2001

[14] Maraikar, Ranasinghe. *From Linda to Javaspaces – A review of the Tuple Space paradigm*. University of Colombo School of Computing. 2002

[15] *Discovering device in Home Networks.* IBM Corporation. 1999

[16] Jing, Helal, Elmagarmid. *Client-Server Computing in Mobile Environments.* ACM Computer Surveys. 1999

[17] Buschmann, Meunier, Rohnert, Sommerlad, Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley and Sons. 1996

[18] Helal, Desai, Verma, Lee. *Konark – A Service Discovery and Delivery Protocol for Ad-Hoc Networks.* University of Florida. 2002

[19] Sun Microsystems. *Jini Network Tecnology*, http://www.sun.com/jini

[20] Sun Microsystems. *Java Remote Method Invocation home page:* http://java.sun.com/products/sdk/rmi

[21] The Salutation Consortium. *Salutation Architecture Specification – part 1.* http://www.salutation.org. 1999

[22] Picco, Murphy, Roman. *LIME: Linda meets mobility.* Dept. of Computer Science, Washington University. 1998

[23] The World Wide Web Consortium. *Web Services Description Language (WSDL) 1.1*. http://www.w3.org/TR/wsdl.http. 2001

[24] The World Wide Web Consortium. *Simple Object Access Protocol (SOAP) 1.1*. http://www.w3.org/TR/SOAP. 2000

[25] *Universal Description, Discovery and Integration: UDDI.org white papers*. http://www.uddi.org

[26] Snell. *Implementing Web Services*. IBM Web Services Toolkit, documentation – tutorial

[27] *The Java Message System home page:* http://java.sun.org/products/jms

[28] Gelernter. *Generative Communication in Linda*. ACM transactions. 1985

[29] Gelernter, Carriero. *Coordination Languages and their Significance*. Communications of the ACM. 1992

[30] Sun Microsystem. *The Javaspace Service Specification*. http://wwws.sun.com/software/jini/specs/

[31] Gamma, Helm, Johnson, Vlissides. *Design Pattern – Elements of Reusable Object-Oriented Software.* Addison-Wesley. 1995

[32] Weiser, Brown. *The coming age of Calm Technology*. Xerox Park. 1996

[33] The Object Management Group. *Wireless Access and Terminal Mobility in CORBA*. 2003

[34] G. G. Richard III. *Service Advertisement and Discovery: Enabling Unversal Device Cooperation.* University of New Orleans. 2002

[35] Helal. *Standard for Service Discovery and Delivery*. IEEE Computer Society. 2001

[36] Duri, Cole. *An Approach to providing a Seamless End-User Experience for Location-Aware Applications*. ACM Conference. 2001

[37] Noble, Satyanarayanan. *Agile Application-Aware Adaptation for Mobility.* ACM Symposium on Operative System Principles. 1997

[38] Noble, Satyanarayanan. *A Programming Interface for Application-Aware Adaptation in Mobile Computing*. USENIX Symposium on Mobile and Location-Indipendent Computing. 1995

[39] Houssos, Alonistioti. *Advanced Adaptability and Profile Management Framework for support of flexible Mobile Service Provision.* IEEE Wireless Communications. 2003

[40] Flinn, Satyanarayanan. *Energy-Aware Adaptation for mobile applications*. ACM Symposium on System Principles. 1999

[41] Banerjee, Agarwal. *Rover Technology: Enabling Scalable Location-Aware Computing*. UMIACS conference. 2001

[42] Lebre, Titmuss, Smyth. *Handover between heterogeneous IP networks for mobile multimedia services*. ACM Symposium on Mobile Communication. 1999

[43] Endler, Okuda, Silva. *RDP: A Result Delivery Protocol for Mobile Computing.* SIDAM Project supported by FAPESP. 2000

[44] Cotroneo, di Flora, Russo. *An Enhanced Service Oriented Architecture for developing Web-base application.* Journal of Web Engineering. 2003

[45] Object Management Group. *The Common Object Request Broker: Architecture and Specification, 2.2 ed.* 1998

[46] Schmidt, Levine. *The Design of TAO Real-Time Object Request Broker*. Computer Comunications Elsivier Science. 1998

[47] Object Management Group. *Real Time CORBA Joint Revised Submission.* 1999

[48] Schmidt, Kuhns. *An Overview of the Real Time Corba Specification.* IEEE Computer Special Issue on OODComp. 2000

[49] Gill, Schmidt, Levine. *The Design and Performance of Real-Time CORBA Scheduling Service*. International Journal of Time-Critical Computing System. 1999

[50] Schmidt. *The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software*. Sun User Group Conferences. 1993

[51] The Hughes Network System. *A tutorial introduction to ADAPTIVE Communication Environment (ACE).* International Journal of Time-Critical Computing System. 1998

[52] Schmidt, Pyarali. *The Design and Use of the ACE Reactor: an Object Oriented framework for event demultiplexing*. Dept. of Computer Science, Washington University, St. Louis. 1999

[53] A.W. Brown. *Large-Scale, Component-Based Development*. Prentice Hall. 2000